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

by

Steven Woods

A thesis

presented to the University of Waterloo

in fulfilment of the

thesis requirement for the degree of

Doctor of Philosophy

in

Computer Science

Waterloo, Ontario, Canada, 1996

I hereby declare that I am the sole author of this thesis.

I authorize the University of Waterloo to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize the University of Waterloo to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

The University of Waterloo requires the signatures of all persons using or photocopying this thesis. Please sign below, and give address and date.

Abstract

The process of understanding a source code in a high-level programming language is a complex cognitive task. The provision of helpful decision aid subsystems would be of great benefit to software maintainers. Given a library of program plan templates, generating a partial understanding of a piece of software source code can be shown to correspond to the construction of mappings between segments of the source code and particular program plans represented in a library of domain source programs (plans). These mappings can be used as part of the larger task of *reverse engineering* source code, to facilitate many software engineering tasks such as *software reuse*, and for program *maintenance*.

We present a novel model of program understanding using constraint satisfaction. The model composes a partial global picture of source program code by transforming knowledge about the problem domain and the program structure into constraints. These constraints facilitate the efficient construction of mappings between code and library knowledge. Under this representational framework, earlier heuristic approaches to program understanding may be unified, contrasted, and compared.

We describe an empirical study which demonstrates the feasibility of our model in the program understanding subtask of *partial local explanation*. In addition, we incorporate this local model into the larger task of combining these local explanations into a coherent global picture of the code. While many heuristic global models are possible, we describe an encompassing structure and demonstrate, through a careful depiction of the algorithm and several domain examples, how constraint satisfaction offers a rich paradigm for exploiting both library and program structural constraint information. One primary advantage of the constraint satisfaction paradigm (CSP) is its generality; many previous program understanding efforts can be more easily compared. Another advantage is the improvement in search efficiency using various heuristic techniques in CSP.

Acknowledgements

As with any work of this size, many people must be acknowledged for their concern, interest and efforts on my behalf. Much of the work presented in this thesis has resulted from collaborations with Dr Qiang Yang (University of Waterloo) and Dr Alex Quilici (University of Hawaii). In particular I would like to thank Qiang for literally hundreds of hours of conversations on many topics, several of which actually included program understanding! As well, thank you Qiang for your continuous mentoring and encouragement without which this thesis would never have been completed. Thanks Alex for your interest in the application of some of my ideas to your previous work, and for encouraging a very successful, ongoing collaborative effort to extend the possibilities of program understanding.

To my Waterloo Ph.D. committee members, Drs Robin Cohen, Grant Weddell and Rick Kazman, thank you for frequently finding time to discuss related aspects of your work and for many helpful suggestions and criticisms of my wanderings. Thank you to my external examiners, Drs Rudolph Seviora (University of Waterloo, Computer Engineering), and Hausi Müller (University of Victoria) for taking the time to read my thesis and offer specific and valuable observations. Drs Peter van Beek (University of Alberta), Josh Tenenberg (University of Indiana), Jim Ning (Arthur Andersen Consulting) and Premkumar Devanbu (AT&T Bell Labs) all receive my thanks for their discerning comments, pointers, and references which I have undoubtedly incorporated without citation into this thesis. Thank you to Drs Martha Pollack (University of Pittsburg), Sandra Carberry (University of Delaware), and Robert Holte (University of Ottawa) for offering their valuable insight and encouragement on early aspects of my work.

During the course of my (second) tenure as a graduate student I have received substantial financial and equipment support from a variety of groups, all of whom I would like to acknowledge. In particular, thanks to the Natural Sciences and Engineering Research Council of Canada (NSERC) for two years of valuable support. In addition, thank you for continuous monetary and administrative support from the University of Waterloo Department of Computer Science and the Information Technology Research Centre (ITRC, Waterloo). Special mention to the Intelligent Software Group (ISG) of Simon Fraser University, and the Department of Electrical Engineering of the University of Hawaii for providing me with both computer support and space to work during my visits.

Portions of the work in this thesis evolved out of related work I undertook while employed with the Commonwealth Scientific and Industrial Research Organization (CSIRO), Division of Information Technology in Canberra, Australia and the Department of National Defence, Defence Research Establishment Valcartier (DREV) in Québec, Canada. Thank you to the Spatial Information System and Tactical Information Fusion research groups respectively for providing interesting and dynamic work environments.

The University of Waterloo is blessed with tremendous administrative staff who take an unprecedented interest in helping graduate students with a never-ending stream of difficulties. I owe debts to them all, however, I would like to especially thank Wendy Rush for going far above and beyond in helping me with the complicated task of submitting and getting approvals for various phases of this thesis, and Jane Prime for helping me untangle the web of administration actually required to graduate.

Several members of the Logic Programming and Artificial Intelligence Group (LPAIG) from the University of Waterloo merit special mention for their willingness to talk at all hours of the day and help with the inevitable, arcane and unsolvable problems - technical and otherwise. Thanks for all your help Stephanie and Toby - I guess all of LPAIG owes a special thank you to the folks at the Grand China for so many wonderful meals! Several fantastic friends put up with me (and in fact, put me up) during my often hectic Ph.D. pursuit. Thank you for everything ... especially Ruth, Megan, Kelly, Wendy and Bart.

My parents Dorothy and Frank have been supportive through all of my endeavours and I can't begin to thank them enough for all of their patience, love, and encouragement over all the years. Last and most importantly, I owe a tremendous debt to my closest friend and co-conspirator - my fiancée, Kirsten Wehner. Thank you for not only reading, editing, listening, and offering insightful suggestions in response to hours of mostly unintelligible and always confusing techno-babble, but also for retaining both your sanity and mine during what has been an exciting but also difficult and trying adventure!

It should be clearly noted that portions of this work have been published by the author and collaborators prior to the delivery of this dissertation. While it is impossible to localize this material specifically within the thesis, the following publications cover most of Chapters 6, 7, and preliminary work extended in Chapters 2, 3, and 8: [Woods and Yang, 1995b], [Woods and Yang, 1995a], [Woods and Yang, 1996a], [Woods and Quilici, 1996a], [Woods and Quilici, 1996c], [Quilici *et al.*, 1996] and [Woods and Quilici, 1996b].

Dedication

... Software entities are more complex for their size than perhaps any other human construct, because no two parts are alike ... [Brooks, 1995, p. 182]

Man with Cuboid, M.C. Escher, 1958.

... Yesterday's complexity is tomorrow's order. The complexity of molecular disorder gave way to the kinetic theory of gases and the three laws of thermodynamics. Now software may not ever reveal those kinds of ordering principles, but the burden is on you to explain why not. I believe that someday the "complexity" of software will be understood in terms of some higher order notations ... Steve Lukasik, [Brooks, 1995, p. 211]

For Kirsten, in all your complexity ...

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Software Engineering and Program Understanding Goals

Software is an artifact created by human experts as a means to encode knowledge about a specific domain. Once created, software will be repeatedly accessed by other experts as part of routine software upgrades, maintenance and debugging. Those constructing software are, however, seldom those who maintain it afterwards, with the consequence that the software itself constitutes highly specific modes of communication between the constructors of software and those accessing it throughout its lifespan. The encoding of knowledge of various domains in software and the lack of direct communication between constructors and maintainers means that the processes by which maintainers *understand* the knowledge encoded in software are central to continued use of legacy software. Legacy software is typically thought of as a constantly evolving corporate asset, critical to corporate goals, and which needs to be maintained against depreciation. Indeed, studies of software maintenance indicate that as much as 80% of software maintenance costs are directly attributable to time software maintainers spend attempting to understand what is being conveyed in particular software.

The process by which software maintainers understand a software system may be best viewed as a primarily conceptual process, wherein the software expert develops a *mapping* between his/her knowledge of both conventions of software construction and the domain knowledge and the given source. A successful understanding is thus the successful construction of a mapping between some portion of the expert's store of relevant knowledge and the structures and components inherent in the source code. Figure 1.1 illustrates such a mapping.

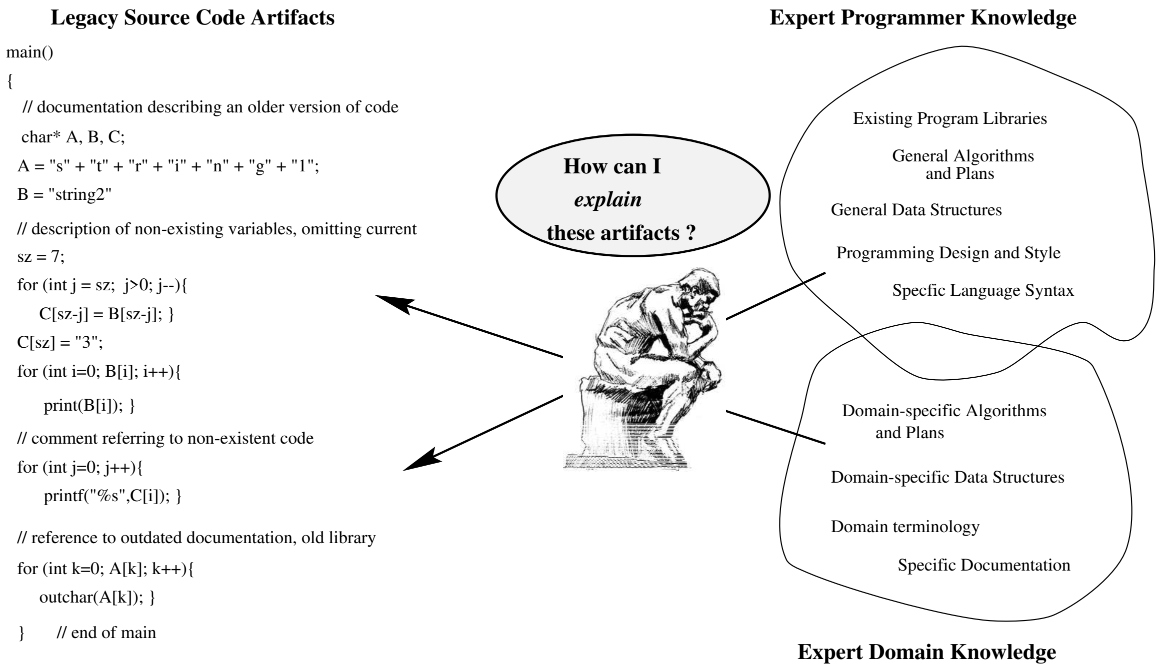


Figure 1.1: Conceptualizing source with expert knowledge

The task of *program understanding*, that is, the process of successful mapping between the expert's knowledge and the source, may be assisted by both physical representations of relevant knowledge and by automated decision support tools. Software systems are inherently extremely complex [Brooks, 1995]. Software is highly multi-dimensional in terms of causation and effect, and there is little necessary correlation between physical representation and program execution behaviour in large scale software constructs. Consequently, visualization and physical representation of software is a difficult task. Nevertheless, commercial software developers are seeking to create packages of software in the form of software libraries and object sets that provide both general computational and specific domain functionality. Software development firms themselves are seeking to extend the efficiency of their software divisions through highly organized programs of *software reuse*. Although humans are adept at interpreting material representations of knowledge created by others, the sheer complexity and volume of software often hinders this process. The usefulness of such visualization tools for understanding software systems depends upon creators and perceivers of these representations sharing an understanding of the context of both its production and reception. In order to be useful to a programmer, any visualization tool must deal with complicated issues in separating and integrating various code views, trade-offs in information hiding and complexity brought on by viewing code abstractly or in greater detail.

In addition to visualization tools, other decision support tools have been proposed to assist in the program understanding process. These include configurable pattern matching systems and case-based reasoning systems for retrieval of software appropriate to a particular task. Both of these approaches are directed at increasing the effectiveness of constructing mappings between software and existing bases of software knowledge or libraries. The central focus of this thesis is the development of automated tools that can *assist* the expert software maintainer in the task of program understanding. In particular, a tool that can help in constructing mappings between existing knowledge sources and highly complex but also highly structured source code is developed. This interactive tool should be seen as a part of a larger expert-driven toolset which includes effective software visualization tools.

The interactive tool described in this thesis is based on algorithmic methods for identifying portions of code structure that correspond to known program plans or sets of program plans. These recognized portions are intended to supplement and complement the knowledge of the reverse engineer during the process of working with the code. In particular, the algorithms discussed are designed with the intention of supporting interactive interpretation of the code. Consequently, expert-supplied knowledge can be utilized during recognition to reduce the overall complexity of the problem, and to generate "views" on the code that the expert has had a role in constructing. The power of the algorithms lie in their ability to propagate small pieces of understanding through the space of all interpretations in order to limit the set of possible interpretations. In this way the software expert can take advantage of the recognition capability of such algorithms in very noisy and complicated situations, and yet not be overly limited by the problems with the inevitable inability of an understanding algorithm to explain an entire program.

The construction of mappings between existing knowledge, either embodied in an expert or represented physically, and source code permits the software expert to make inferences about the source program's possible higher-level goals. This process of inference permits, through abstraction, perceptions of the source as actual code statements to be re-conceptualized at the more general level of the existing representation (or language of expression) of the domain knowledge. This abstract understanding may be exploited in many ways, including: (1) as part of a process of translating the program into the source code of another programming language, (2) recognizing errors in the code (or design/requirements) and assisting in debugging the system at the more abstract level, and (3) replacing understood code portions with generic application code or calls to other code libraries. In addition, in many real-world circumstances, a reduction in the size of an existing source code through adoption of standard code libraries or reduction of redundancy by only a small percentage can result in a substantial reduction of the ongoing maintenance cost, simplify future extensions to code, and reduce the probability of introducing errors through such modifications and extensions. Consequently, creating a mapping (even a partial one) between existing domain knowledge and a particular source code offers a possible lever for the software expert to employ.

Within the broader context of software reverse and re-engineering, program understanding is a sub-task based upon one of the primary goals of software engineers. This goal is to provide a solid and clear shared context for communication between a software creator and a software maintainer. The construction of this solid medium of readable, understandable software is based upon standard software engineering principles such as information hiding which is embedded in object-oriented methodologies. Another such principle for shared context standardization is represented in focused efforts at standardizing software re-use. The attempt to implement rigid, shared standards for system and code design, documentation and coding is based upon the need to maintain a coherent shared context of presentation for the entire group of future expert maintainers. In essence, this shared standardization forms the basis for any partial automatic understanding of code. As a medium of communication, software has the potential to be highly structured and regular, especially when compared with more general and flexible communication forms such as natural language.

1.2 A Brief Look at Program Understanding

In Artificial Intelligence research, the problem of program understanding has been approached indirectly from the perspective of plan recognition. In Section 2.2 I discuss in some detail how much of this plan recognition work fails to meet the requirements of the program understanding task. In other research more closely related to the software engineering community, a more direct approach to program understanding has been undertaken in which an explicit library of program plan templates and concepts is constructed, and various top-down and bottom-up search strategies are utilized to implement a mapping process between them. These approaches are introduced in detail in Section 2.2.2.

As I have introduced, program understanding methodologies typically attempt to construct formal mappings between knowledge sources and code. For example, in Figure 1.2 a subset of expert knowledge about a particular application domain is represented in a fragment of a hierarchical library of program templates. One possible mapping is shown between a plan template from the library and a specific source fragment, in this case a single source statement. The existence of such a mapping essentially *explains* the presence of the low-level source statement at a higher level of abstraction, in this case as an instance of the plan template **copy-character** specified in the library.

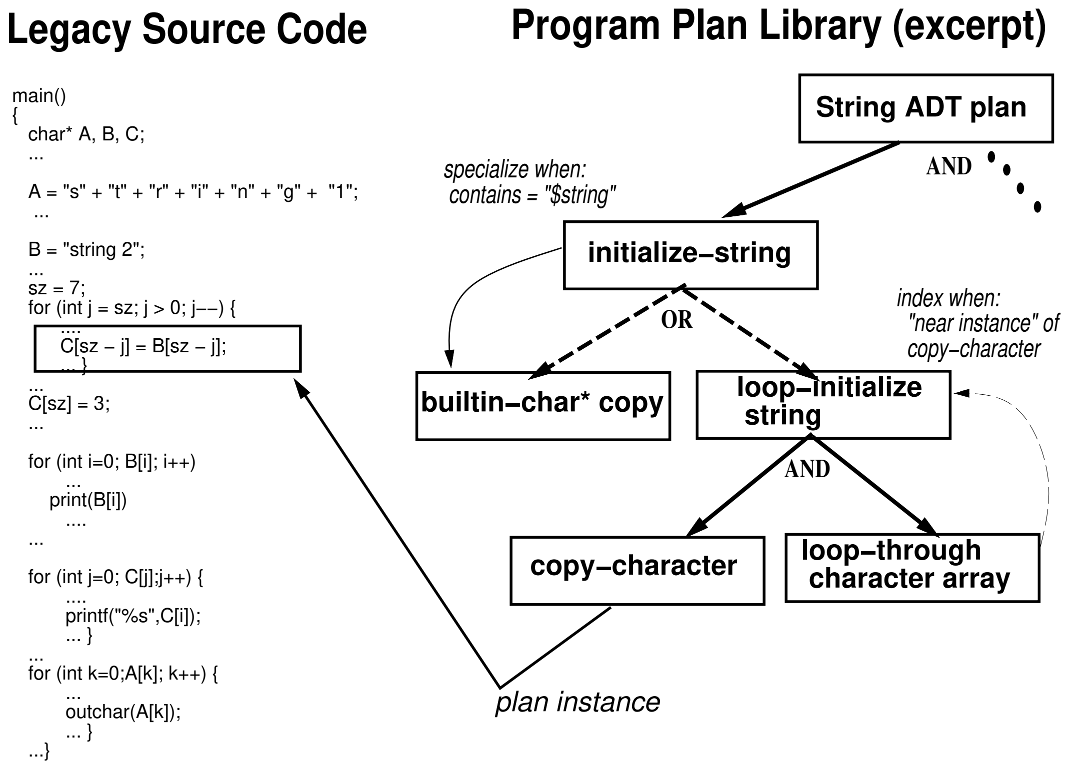


Figure 1.2: Conceptualizing source with a plan library

Much of the previous program understanding work has failed to demonstrate heuristic adequacy in even partially generating "understanding" of large problems. Specifically, many recognition algorithms presented may be viewed as partially disjoint collections of heuristic tricks. Some of the landmark efforts are described in Section 2.2.2. Such heuristic construction makes it difficult for one to perform a systematic analysis of different search methods, or to understand how the addition or deletion of certain types of domain-specific knowledge may affect performance. I am unaware of concrete examples or experiments which might suggest that these approaches might scale up for specific uses in large sources. However, both [Wills, 1992], and [Quilici, 1994] present empirical results promising in identifying partial mappings from sources of up to 1,000 lines to a small library of program plans.

1.3 My Approach to Modeling Program Understanding

This work is part of a research effort structured towards: (1) unifying previous heuristic approaches to program understanding with a single model capable of representing both structural knowledge and control knowledge in a single framework, and (2) demonstrating that an effective approach to automatic partial program understanding is possible with large code examples. Specifically, I intend to clearly categorize the circumstances in which this use is possible, and the preconditions which must first be met in terms of representation and application of domain knowledge. In response to these two primary goals, I present a generalized representation of program understanding as a *Constraint Satisfaction Problem (CSP)* [Mackworth, 1977], and represent previous program understanding algorithms within the CSP framework.

In this CSP approach, a large source code is decomposed into a series of functionally related source "blocks" or "components" which may then be partially explained independently. These independent explanations may be used to locally reduce the range of explanation of logical neighbour components. This divide-and-conquer approach to program understanding relies on several new algorithms which I present in this thesis: program plan recognition algorithms, hierarchical constraint satisfaction algorithms and local constraint propagation methods.

I represent program understanding in two parts, and in two corresponding CSPs. The goal of the first part is to identify local instances of program plan templates in the source. These local instances may be thought of as partial local explanations of the code in which they reside. The CSP representation of the first part is known as MAP-CSP.[1] The goal of the second part, or PU-CSP, is to integrate these local partial solutions into a coherent global view. Ongoing work such as has been reported in [Quilici, 1994], is motivated by cognitive studies [Pennington, 1987a] which support this two-phased approach.

There are at least two advantages in the constraint-based approach. The first is its **generality**; most of the previous recognition methods and heuristics can now be unified under the constraint-based view. Another 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 may be applied. It is possible to now perform a systematic study of different search heuristics, including both top-down and bottom-up as well as many other hybrids, in order to discover their applicability to a particular source code.

[1]The term "MAP" derives from the process of constructing a **MAP**ping between a known library plan and an arbitrary piece of course code.

1.4 Program Understanding and Artificial Intelligence

I have identified AI problem-solving approaches as applicable to the program understanding problem domain in at least three primary ways.

1. First, *plan recognition* is a sub-domain of AI in which the plans and goals of an agent are interpreted based on a set of perceptions of that agent's actions and a library of (hierarchical) actions of which that agent is capable. In Chapter 3 I identify the program understanding problem as a special case of plan recognition in which software reverse engineering algorithms have been designed to address the restricted plan recognition domain. In particular, these algorithms are able to exploit specific restrictive problem features to empirical advantage.

2. The constraint satisfaction problem (CSP) sub-domain of artificial intelligence has recently received a lot of attention as a standard approach to modeling and solving hard problem instances. The application domain of software reverse engineering and program understanding is identified in this thesis as a rich testbed for CSP representation and solution schemes. Through a "bridging" thesis such as this, I have provided the opportunity for application of local, global and hierarchical work in constraint satisfaction to the program understanding domain. Similarly, program understanding researchers are provided with the opportunity to see the value of formally representing problems in the CSP framework in terms of increased scalability and standardization of heuristic representations.

3. Finally, through working in the software engineering world with the CSP modeling paradigm, a novel algorithm for propagating consistency in a constraint graph is presented in Chapter 8 which significantly advances the current state of the art in CSP. In particular, this algorithm is intended to accommodate domain values

situated in a hierarchical structure consisting of both *is-a* (inheritance) and *is-part-of* (composition) relationships whereas previous work accommodated only *is-a* relations. This work is easily generalizable to other problems in which domain values can be hierarchically structured.

1.5 Primary Research Assumptions and Context

1.5.1 A Tool-based Vision of Program Understanding

Program understanding may be considered as an integral sub-task of software reverse and re-engineering. Within this context, large-scale program understanding is typically most profitably undertaken with the assistance of a software visualization toolset such as provided by RIGI [Müller *et al.*, 1994]. I understand a visualization tool of this kind to form the basis of a code understanding decision support system in which automated program understanding tools may be embedded.

1.5.2 Software Repository

In accordance with the work upon which I build [Wills, 1992, Quilici, 1994, Kozaczynski and Ning, 1994], the existence of a software repository is assumed from which program plans of a domain-specific or domain-independent nature are situated. Such a repository could be populated through the use of existing commercial class and template libraries in languages such as C++, or possibly through the utilization of generic application libraries constructed by many companies as part of software reuse. While other work is concerned with the population or abstraction of such a repository, I restrict myself to its application.

1.5.3 Procedural Software and Plans

During the course of this research I have focused on procedural languages and plans which correspond to algorithms in the procedural sense. This choice was motivated by the constant desire to be relevant to the community sporting the largest set of deployed code - COBOL. That said, I regard the relationship between procedural plans and functional implementations (as might be evidenced in PROLOG for example) as an open research issue. It may well be that abstracted procedural plan representations have a poor canonicalization of functional implementations.

1.5.4 Program Plan Definition

The abstract program plans (clichés or idioms) utilized for program understanding have been primarily defined by understanding researchers attempting to generate *canonical* representations of particular algorithms. The language of representation tends to be pseudo-code-like and capable of mapping to many programming languages. Standard algorithmic devices such as "loops" are represented abstractly in terms of fundamental concepts such as "loop-begin", "loop-end" and "loop-until" and matching methodologies are devised to instantiate these abstract terms to particular instantiations.

The acquisition of domain and generic program knowledge is clearly a labor-intensive task, roughly analogous to the work performed by systems analysts in learning about a particular domain. Wills[Wills, 1992, pp. 19-58] devotes an entire chapter of her Ph.D. thesis to describing a cliché library and experiences in obtaining it. In Wills' work, effort was expended both bottom-up in finding more than one implementation of a particular cliché before an attempt was made to generate an "abstract" representation, and top-down in mapping deliberately abstract descriptions of algorithms (i.e. from textbooks) to the cliché representation. Other subsequent work [Quilici, 1994] is pushing forward this definitional phase through an attempt to create an interactive, visual tool which will allow an expert to select a piece of code, abstract it partly through as yet undetermined methods of automated canonicalization, and support the expert in editing these representations.

As a result of the impossibility of defining "perfect" algorithmic abstractions, any automated understanding support tool which attempts to assist an expert in understanding source code *must* be able to accommodate partial and local explanations, and be able to successfully interact with the expert during interaction.

1.5.5 Software Structure-Analysis Tools

It is a recognized advantage in software understanding to have a range of possible source code views with which to examine a given source code. In particular, code visualizers discussed in Section 2.2.3 provide ways of both abstracting out and elaborating detail depending on the desire of the analyst. Views of software based on data-flow or control-flow tend to be related, but each provides different information, and only together does their individual power become apparent to an expert attempting to understand software. Views based purely on the use of given data structures and executable traces provide examples of a range of other possible code examination techniques. Each of these software views provides a unique set of constraint information about the construction of a given source code. Throughout this work I assume the existence of the tools which provide this constraint information. In particular, I assume the existence of control and data-flow analyzers such as GEN++ [Devanbu and Eaves, 1994] which annotates an abstract syntax tree view of software. In the absence of such a tool[2], I approximate such constraint information through direct reasoning with source code examples. In verifying a constraint without explicit data and control-flow information, it is necessary to analyze the source

[2]Due to licensing difficulties, an annotating parser could not be obtained for nearly a year. In 1996 this license was obtained, and current research is underway incorporating the annotated ASTs.

program directly for the possible existence of such a data-flow. This type of as-required checking is more expensive computationally than a simple check for a pre-identified data-flow; however, since I am primarily measuring computation in terms of constraint-checks required, this assumption does not skew the check-counting results.

1.5.6 Phases of Analysis

1.5.6.1 Empirical Analysis

While I examine the specific structure of experiments performed in Chapter 7, it is important to note the assumptions that lead us to examine the MAP-CSP closely. I assume that local partial explanations are a crucial sub-task in program understanding, and further, one which could benefit from a constraint-based tool assisting an expert in very large program analysis. Partial explanations of this type are utilized by at least Quilici, Wills, Ning and Kozacyznski albeit in varying representational forms. All of these works have attempted to work towards models that will have the potential to scale to usefully sized code instances.

1.5.7 Global Analysis

It is difficult to conduct an empirical study of the effectiveness of structure-identification tools in the absence of an integrated program analysis tool. Such a tool would utilize a combination of program visualization, program structure extraction, a hierarchically structured program plan library *and* the constraint-based local and global explanation assistance tools. I anticipate that such combination and study will occur in the future, and the goal of this aspect of my research is primarily to ascertain that constraint technology is adequate for the representation and solution of this problem in this context. The discussion in this thesis of specifically extended constraint algorithms is a step towards this combination.

1.6 Thesis Outline

The remainder of this thesis is divided into five primary parts.

Part I, **Foundations**, presents necessary background material to acquaint or re-acquaint the reader with the tools that I will use to build my thesis arguments. I present a short introduction to the problem of program understanding in software engineering, outline how semi-automatic analysis of structured software typically proceeds, and provide an overview of crucial previous work in program understanding and plan recognition.

Part II, **Framework**, describes the modeling paradigm I have selected to represent the program understanding problem, and outlines related algorithms used in the solution of problems represented in this way. I introduce the conception of the two primary stages of program understanding: partial local, and global explanation. The description of these two sub-tasks within the context of my chosen modeling framework forms the background for the next two thesis parts.

Part III, **Local Explanation**, details the first stage of my program understanding model. Analysis of problem complexity and description of experimental empirical results are presented along with an analysis of how local explanations are mirrored in earlier program understanding work.

Part IV, **Global Explanation**, completes my program understanding representation with a complete description of an integrative model for partial local explanations. This model does not represent the only possible way in which local explanations may be integrated into a more comprehensive view. Like other integrative approaches, it exploits both the presence of inter-component relationships and partial knowledge about program plan content. Unlike other approaches, my approach makes the explicit use and value of program plan knowledge and structural constraint information in program understanding clear. In addition to a second complexity analysis of the integration problem, I present a set of illustrative examples and carefully describe the algorithmic approach of the global strategy in terms of a cooperative interactive system. The integration of constraint propagation (limited inferencing) with expert interaction is a crucial component of any useful decision support tool.

Part V, **Conclusions**, summarizes the finding of my research. In addition, I position myself with respect to other closely related work in program understanding, and outline the future direction of this project which in many ways is only now commencing.

Part I

Foundations

Chapter 2

The Understanding Process

2.1 Software Engineering and Program Understanding

The main task of program understanding fits within the context of both software engineering and reverse engineering. Several important questions need to be posed and answered before one can discuss program understanding cogently. First, what do the often-used but seldom defined terms *software engineering* and *reverse engineering* mean? What underlying process of software development requires that a software expert approach an existing piece of software and attempt to understand its function? How does program understanding fit into the larger world of software development?

2.1.1 Software Engineering: an attempt at definition

A great range of definitions of software engineering have been presented in a variety of texts and articles. Software embodies the controlling logic governing the application of a computer to solving a given domain problem. Software encodes domain knowledge, domain methodologies, and domain data into a particular language designed to be compiled into machine-executable instructions. As an embodiment of logic, software is a very precise specification of a domain process. The specification itself, however, is only as accurate a representation of the true system as the particular specifier's understanding of the process itself allows. In addition, errors can be made during specification in the encoding process itself. Consequently it is more accurate to say that any piece of software is only an approximation of the process it represents and models. This approximation has two dimensions. First, the software is based on an approximate understanding of the process itself. Second, the software is only an approximately-correct representation of this understanding.

The process of constructing this model or software has traditionally been referred to as "writing" software, reflecting the perception that software construction is more about the manipulation of concepts rather than materials. However, as attempts have been made to instill software with a higher degree of both accuracy to the modeled process and intended logic, the process of software development has become known as *software engineering*.

> Software engineering is concerned with the theories, methods, and tools which are needed to develop the software for these computers. In most cases, the software systems which must be developed are large and complex systems. They are also abstract in that they do not have any physical form. Software engineering is therefore different from other engineering disciplines. It is not constrained by materials, governed by physical laws, or by manufacturing processes.
>
> Software engineers model parts of the real world in software. These models are large, abstract, and complex so they must be made visible in documents such as system designs, user manuals, and so on. Producing these documents is as much part of the software engineering process as programming. As the

> real world which is modeled changes, so too must the software. Therefore, software engineering is also concerned with evolving these models to meet changing needs and requirements. [Sommerville, 1996, p. 4]

As Sommerville points out, the correctness and reliability of software is of *more* than critical importance.

> In all industrialized countries, ... computer systems are economically critical. More and more products incorporate computers in some form. Educational, administrative and health care systems are dependent on computer systems. The software in these systems represents a large and increasing proportion of the total system costs. The effective functioning of modern economic and political systems therefore depends on our ability to produce software in a cost-effective way. [Sommerville, 1996, p. 4]

Not only is software only an approximate specification of a real process, but it must necessarily be an evolving specification as well. These software models are "large" in the sense that they can easily total several million lines of coded instructions, and "complex" in the sense that software entities are "...more complex for their size than perhaps any other human construct..."[Brooks, 1995]. This complexity is attributable at least partly to the fact that no two parts of a software model are alike in a semantic sense. The engineering of computers differs from the engineering of buildings, automobiles and almost any physical construction in this important respect.

Complexity hinders software in significant ways. Complex software has more opportunity to vary from the modeled process. Complex software is by definition hard for a third party to understand after it has been written. Complex software is dangerous to change for fear of adverse or unintended effects of a given change. Large, complex software is difficult to write and maintain since it will need to be segmented across members of a team, introducing problems of communication, coordination and mutual understanding.

Brooks identifies three other primary inherent difficulties with the engineering of software in addition to size and complexity. First, software must *conform* to pre-existing specifications and interfaces that are arbitrary when viewed with respect to the task at hand. The result of forcing a given software model to conform to this set of arbitrary constraints is to increase software complexity even further. Second, software is highly *changeable* since it is "embedded in a cultural matrix of applications, users, laws, and machine vehicles" which themselves change continuously. Third, software is difficult to *visualize* since it is not embedded in space and consequently has no ready geometric representation in the vein of other engineering applications such as, say, circuit diagrams. Graph-based representation models of software fail in their dimensionality since software has a multiplicity of parallel "views" such as control-flow, data-flow, dependency graphs, time sequence charts, name-space relationships and the like.

Software is very large, incredibly complex, changeable (and frequently changed), forced to conform to often arbitrary structures, and highly difficult to visualize spatio-temporally. One is tempted then to re-define software engineering as simply the attempt to define control logic with a maximal reduction in complexity of software. However, Brooks' further observation that "The complexity of software is an essential property, not an accidental one." helps dissuade us from this simple definition.

What then is software engineering? The answer is that there is no single answer. Many techniques have been proposed as "software engineering" over the years. Software design and development methodologies such as structured analysis and design, rapid prototyping, waterfall-design and a myriad of other approaches have been and continue to be hailed as the next and best method for software development. The quest for Brooks' famed but mythical "silver bullet" for instituting a massive jump in software development productivity continues. Software engineering is an accumulation of the processes of analyzing an existing system and a set of user requirements, designing an appropriate and exacting model of the existing system with careful emphasis on how an automated set of tasks would fit within the existing system, design of software logic for these automated tasks, implementation of the software required to perform these automated tasks while minimizing complexity and still satisfying conformability constraints, and attempting to minimize the number and effect of changes required after-delivery while still satisfying functional requirements.

2.1.2 Reverse and re-engineering

If software engineering covers such a wide range of tasks, it is necessary to segment it in order to focus the scope of interest more precisely! Sommerville [Sommerville, 1996] proposes a convenient breakdown as follows:

- *Forward engineering* is the conventional development of a software product from domain analysis through design to software production. This process typically includes the delivery of a software product and accompanying documentation and manuals.
- *Re-engineering* is the systematic re-implementation of an *existing* software system so as to make it more maintainable.
- *Reverse engineering* is the derivation of a design or specification of a system from its source code.

Sommerville further summarizes each of these methodologies in terms of the primary tasks and sub-products involved in each as shown in Figure 2.1.

During the "automated analysis" phase of reverse engineering, automated (or partially automated) tools collect information about the *structure* of the system. As Sommerville

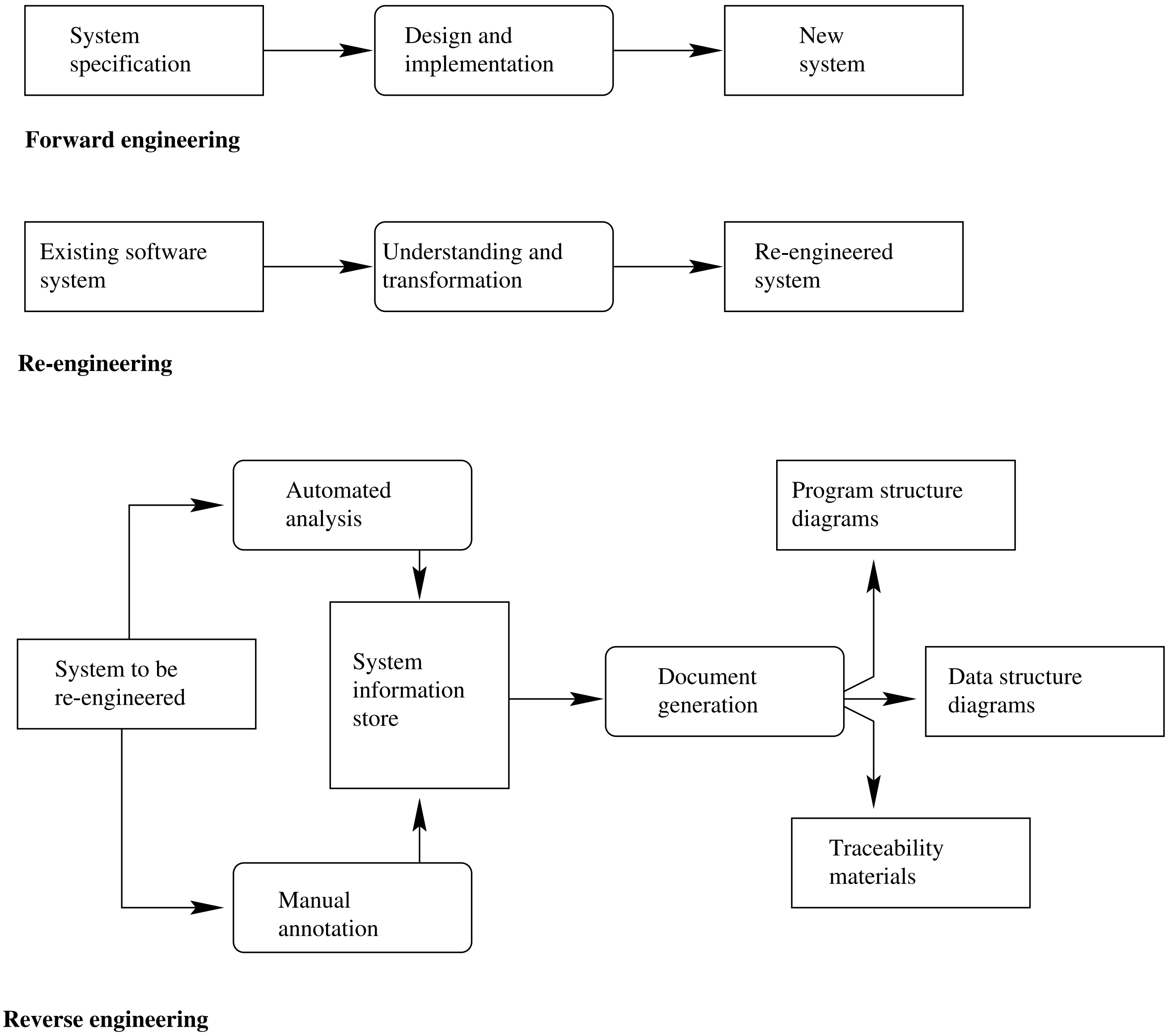


Figure 2.1: Sommerville's software engineering world

points out, the structural information collected is insufficient to recover the system design itself. Rather, engineers "understand" information from the system, add this to the source code and the structural model(s), to produce an "information store", typically represented as some kind of directed graph or set of directed graphs of varying types. Further analysis can be undertaken using various source and graphical browser combinations in the subsequent annotation of the model of the source system. While these tools are aids for program understanding they are primarily intended as navigational and view tools intended to aid the expert in his/her correlative task between source code and internalized, expected abstracted programming concept or plan representations. It is important to note that the understanding tools cited as relevant by Sommerville [Sommerville, 1996, p. 712] do not include those proposed and discussed elsewhere [Wills, 1992, Quilici, 1994, Kozaczynski and Ning, 1994] which take a more correlative focus between source and some type of abstract conceptual representations.

Sommerville importantly notes that the process of analysis and document generation is necessarily an iterative one. That is, the process of the analysis is one in which the system in question is analyzed to determine not only its structure, but rather that the structure itself is further used as a framework on which the expert may append knowledge about what semantic tasks the system achieves. This structure may be viewed as evidence of a programmer's (or programmers') plan for achieving the task encoded in the software. Gamma [Gamma *et al.*, 1993] and others support this plan-based view of software.

> Studies of expert programmers for conventional languages ... have shown that knowledge is not organized simply around syntax, but in larger conceptual structures such as algorithms, data structures and idioms, and also plans that indicate steps necessary to fulfill a particular goal [Gamma *et al.*, 1993, p. 407]

While reverse engineering seeks to delineate the original system structure from source code, re-engineering intends to "improve" the system structure, generate new system documentation, and make the system easier to understand. The benefits of this task are seen directly during subsequent software maintenance and extension. The re-engineering task mirrors the reverse engineering task, however, in at least one very important area: understanding. In both cases, an expert or a team of experts must develop a clear understanding of an existing software system.

Sommerville outlines the factors that directly affect the cost (and success) of a re-engineering process as:

- *Existing quality* of the software and documentation.
- *Tool support* for re-engineering. In particular, "re-engineering software is not normally cost-effective unless some automated tool support can be deployed to support the process."[Sommerville, 1996, p. 702].
- Extent of *Data Conversion* required.
- Availability of *Expert Staff*. In particular, the availability of the staff who maintain the system is critical as these people can greatly reduce the amount of time spent by system re-engineers in *understanding* the software system of focus.

Software re-engineering is instantiated in both the task of *software translation* and *software restructuring*. Translation is typically a largely (although not frequently totally) automated process involving the mapping between structured representations of logic in one language to **identical** representations in another language. Much of this process is potentially one-to-one if the target language has functionality both mirroring and extending that of the source. In addition, tools such as REFINE (Software Refinery)

[Burn, 1992] and TXL [Cordy *et al.*, 1991] support basic syntactic pattern matching and program transformations for the translation process.

Ongoing maintenance of software systems tends to destroy program structure and further complicate any understanding attempt. Control restructuring is intended to simplify the overall program structure so that conditionals and loop constructs are "parsed" to provide a more readable form such as that offered through a series of embedded if-then-else statements. Other restructuring such as program modularization is typically undertaken manually. During this process, constraints and relationships between program components are identified and the function of components deduced. Sommerville [Sommerville, 1996] states that "it is therefore impossible to automate this process, although some experimental systems have been produced to provide some computer-aided assistance for modularization." These systems rely on an analysis of references to data and procedure call structures to infer which program components may be part of the same module. Once well or tightly-coupled modules are identified, it may be possible to totally or partially explain their interconnection structure and data-flow with reference to other templates of similar structure. In fact, in well-written programs that already provide tightly-coupled modules with shared data structures, the understanding task is much simplified.

In Figure 2.2 Sommerville's view of the understanding process is extended somewhat to focus more on the interactive aspect of understanding, and on the tool support Sommerville identifies as critical. This extended model accommodates both reverse and re-engineering subtasks of expert understanding. The automated analytical tools are utilized by a software expert to the degree desired, in conjunction with manual explanatory techniques, and domain and programming knowledge. Most importantly, this new model acknowledges the fact that *a particular* expert is dealing with a specific subset of knowledge about domain programs and structure and about general programming techniques, and clichés [Wills, 1992]. If one considers that some subset of this domain-specific and domain-independent knowledge may be formally representable (or represented) in the form of in-company or generic software re-use libraries (for instance), then portions of the perceived structure in the software system can be mapped to these existing software structures.

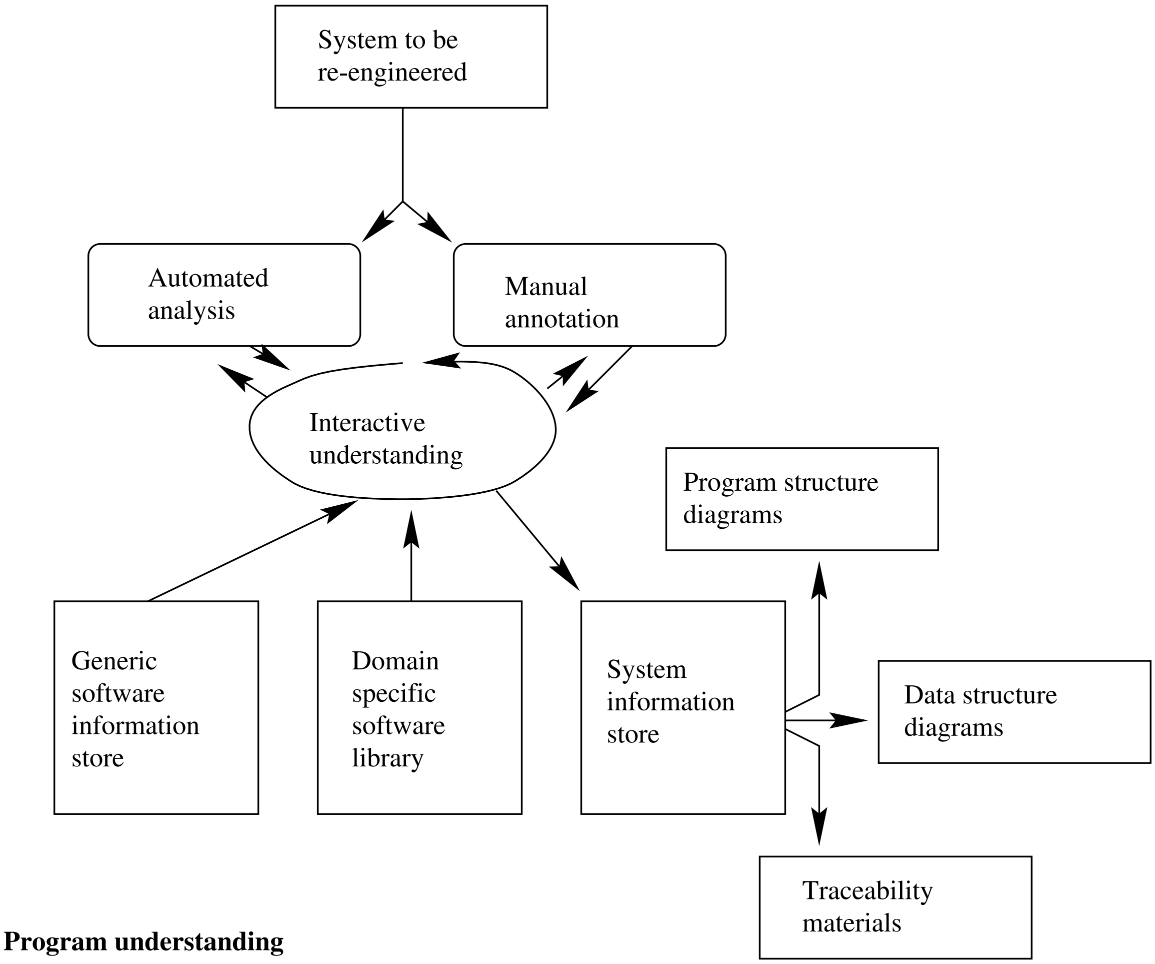


Figure 2.2: Program understanding in software engineering

This extended representation supports the view that complexity management can be addressed, at least partially, through information hiding paradigms such as that offered by object oriented technologies. Indeed, Brooks argues that information hiding is "the only way of raising the level of software design" [Brooks, 1995, p. 272]. In the extended model, the identification of repetitious (or clichéd) functionality could be extracted and replaced through calls to the existing library, thus effectively *hiding* the details underlying that call from the system of focus.

Brooks further states that, "The most obvious way to [attack the essential difficulties of complex conceptual constructs] is to recognize that programs are made up of conceptual chunks much larger than the individual high-level language statement— subroutines, or modules, or classes." The understanding process of Figure 2.2 can be thought to express this conceptual chunking through the identification of clichéd chunks of source code. While it is difficult (today) to imagine that such a breakdown of a software system might be undertaken entirely automatically, it is much easier to envisage an automated support system for the expert undertaking this task.

It is often observed that well-chunked code is easy to understand and poorly-chunked code is difficult. The implication of this statement is that we are primarily interested in automating the difficult portions (as opposed to the easy), and yet these are in fact the hardest to consider even partially automating. Brooks says "If we can limit design and building so that we only do the putting together and parameterization of such chunks from pre-built collections, we have radically raised the conceptual level, and eliminated the vast amounts of work and the copious opportunities for error that dwell at the individual statement level." Clearly, automated analysis of code built entirely from the program plans or clichés in a known software library will be much easier to undertake than the general instance in which only few clichés will be present. However, it is possible to reduce the realm of explanation of an entire chunk of source code through the identification of even some clichés. Consequently, an iterative process in which partial local explanations *contributes* to an expert understanding of the larger context can still be highly useful.

2.1.3 Understanding

Software engineering methodologies imply that there exist program parts or modules which have been formulated at a much higher level of abstraction than mere program statements. Any perception of this higher-level definition or view of source programs must have originated through an identification or understanding of shared conceptual chunks between the creator (original software developer) and receptor (the expert undertaking understanding). It has been seen how it is reasonable to assume that some subset of these potentially sharable software concepts may be resident in, for example, re-use libraries. In object technology, these shared conceptions are encapsulated as objects in object libraries often structured hierarchically. In this work I examine previous approaches at specifying and implementing the partial automation of concept recognition in source code. Further, I propose a new model for both representation and solution of the partial explanation of code which integrates well with the interactive model of understanding sketched in Figure 2.2.

I motivate the picture of an expert understanding a software system in terms of an expert constructing a mental model in which his/her set of existing knowledge is mapped against the system in question. This view of program understanding is supported in earlier software engineering work.

> It has already been suggested that the cognitive process of developing a program involves building an internal semantic model of the problem and its solution. Building this model may involve utilizing existing knowledge and, except for trivial problems, is an iterative process. [Sommerville, 1982, p. 236]

I explore this conception further specifically in Section 2.2.1, and in a broader fashion throughout this work.

The *theory of program understanding* upon which most understanding efforts has been based is described by Rich [Rich, 1987] as:

> Plan instances can be recognized in existing programs to recover the programmer's abstract concepts and intentions.

I describe program understanding in two primary parts: *partial local explanation* which identifies instances of a particular program plan in code fragments, and *global or integrative understanding* in which the partial local explanations are combined to give a broader interpretation of the given fragment(s). Following other understanding efforts [Hartman, 1992b] I describe the understanding tasks as hierarchical - both in terms of the source program and the plan library. In practice, any plan library is going to be incapable of completely "explaining" a complex program. Consequently, partial explanations will have to suffice as provided by an expert-assistant understanding toolset. I therefore build understanding tools based on a weaker, but still useful theory:

> Plan instances can be recognized in existing programs to recover *some* of the programmer's abstract concepts and intentions.

2.2 Program Understanding Methodologies

I have defined program understanding loosely as the process of constructing consistent mappings between a source program and its many possible representations such as control-flow and data-flow graphs, and some set of existing knowledge which may be partially represented formally and partially contained in an expert's general knowledge.

It has been reported from programmer studies (see [Pennington, 1987a] and [Pennington, 1987b]) that code understanding is undertaken differently based on the context of the task. For instance, understanding by novice programmers or by programmers unfamiliar with the code or domain in question is performed initially bottom-up, searching for low-level structures or clichés in the code. Subsequent to this low-level analysis, large structures are generated based on examination of the control-flow and data-flow constraints among the identified low-level structures. In contrast, when an expert is examining familiar code, it has been demonstrated [Soloway and Ehrlich, 1984] that programmers approach code top-down, first looking for familiar high-level constructs and only then determining their relative functional relationships. Most previous approaches to assisting program understanding have either implicitly or explicitly recognized this need to support both top-down and bottom-up understanding. Few, however, have described how these might be integrated into a single model of understanding. In a real sense, both top-down and bottom-up understanding involve mapping source code to a set of knowledge about programs. In the bottom-up sense the mapped sets of related source statements constitute *microstructures* [Citrin *et al.*, 1996] or clichés, while in the top-down sense larger sets of related source functional components are recognized as corresponding to known *source systems*. In each case, the key to successful recognition is a correspondence between a set of related or constrained components and a typical or known set of components related in the same fashion. Throughout this work the view of program understanding as mapping between an original source code and known program plans (templates or clichés) will be assumed.

2.2.1 An illustrative example of program understanding

Consider the C program fragment outlined on the left hand side of Figure 2.3. This example contains declarations, initializations and an embedded print loop for *each* of three strings. As an illustration, strings are treated as a primitive data type by the programmer, with no shared functionality assumed for string output.

To understand this program, one might use as a basis a library of program plans, as shown in Figure 2.4, which represents a portion of previously compiled knowledge about program composition within a particular domain. Figure 2.1 shows a program

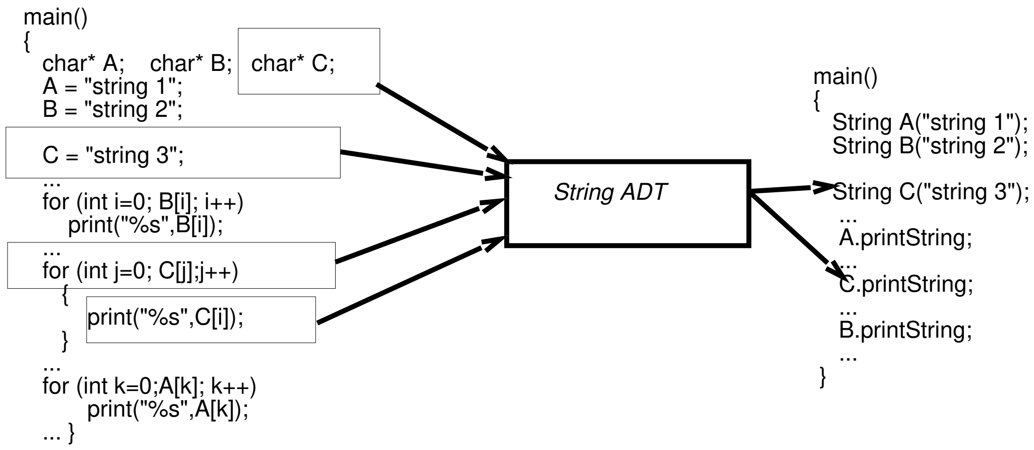


Figure 2.3: C source code mapped through a String ADT instance to C++ code

plan from the Abstract Data Type (ADT) or class **String** which is part of this library of plans. I consider that an ADT represents a "focal point" for a collection of plans or methods applicable to the data type represented. In particular, one may imagine that a particular ADT may have a subset of *necessary* operations for that type, and another subset of *possible* operations. Consequently, I view an "ADT" plan as merely a collection (necessary or and'ed) of related plans. Once a mapping is constructed between the source and compiled knowledge, one could replace the redundant functionality of the relevant portions of the source code with a single inclusion of the ADT from Table 2.1, as shown in the C++ code of Figure 2.3.

Given the source code on the left side of Figure 2.3, we would like to *understand* or *explain* some portions of the source program within the known context of the program plans. The C++ code shown at the right of Figure 2.3 is obtained with replacement of C source with references to **String** ADT functionality. This understanding process might be executed in two steps. First, one identifies all instances of a particular abstract program plan in a source code. I refer to this step as the *Template Matching* or *Mapping* problem. Second, one relates some set of identified plan blocks or components to conform

```
Class String {
 char localStr [MAXSIZE];

 String( char* inStr )
 {
   for (int j=0; inStr[j]; j++)
       localStr[j] = inStr[j]; }

 printString()
 {
  for (int j=0; localStr[j]; j++)
      printf("%s",localStr[j]); }   }
```

Table 2.1: Example abstract data type

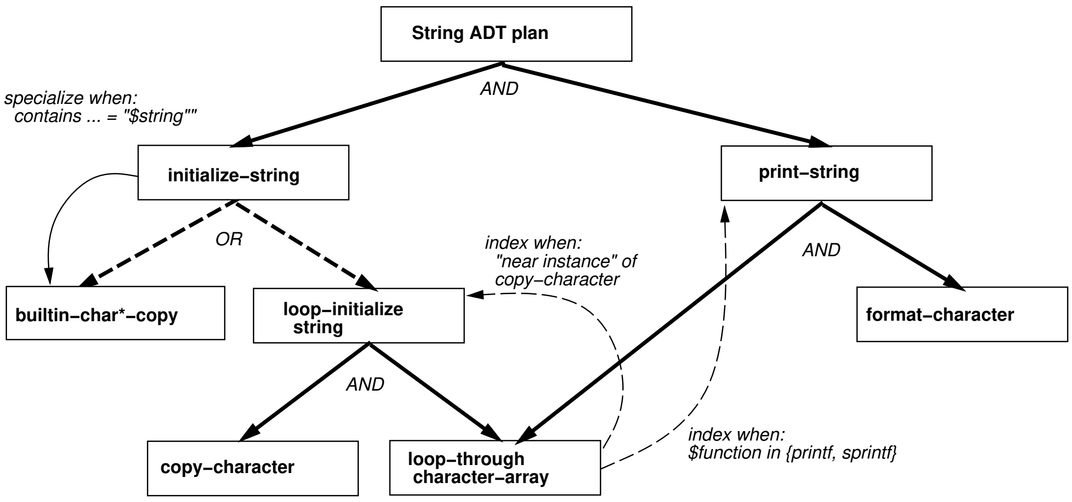


Figure 2.4: **String** ADT within a hierarchical program plan library

to the hierarchical structure in a given program-plan knowledge base. The latter process I refer to as the *Program Understanding* problem solution.

I identify two important benefits of locating local mappings between a programming plan library and an existing source code. First, the resulting replacement of code with ADT instances can result in substantial reduction in the amount of code required for a particular task. This size savings can reduce the amount of effort required for subsequent code understanding or maintenance by programmers. Second, the local mapping(s) between source and library plans can be used as building blocks in attempting to understand and translate a larger context of source code.

2.2.2 A Review of Past Program Understanding Work

Recently, researchers have adopted a direct approach to program understanding in which an explicit library of programming plan templates and concepts is constructed, and various top-down and bottom-up search strategies are utilized to implement the mapping process between source and templates. Over the past decade, researchers have proposed and implemented a wide variety of plan-based program understanding algorithms. These efforts have included [Quilici, 1994], [Kozaczynski and Ning, 1994], [Wills, 1992], [Rich and Waters, 1990], [Hartman, 1991b], [Johnson, 1986], and [Johnson and Soloway, 1985]. Other work including that of [Kontogiannis *et al.*, 1995] which extends code to code similarity measurement, has attempted probabilistic recognition of individual abstracted concepts with promising empirical results. It is currently open issue whether probabilistic or constraint-based approaches offer the most "exploitable" methodology for the construction of program understanding tools. It is interesting to note, however, that users of expert systems have frequently put a premium on systems which are able to explain their decisions on demand in simple easy-to-understand terminology. For this very reason, many expert system implementations have avoided probabilistic or statistical explanations. One advantage of a constraint-based approach in which problem constraints are propagated as part of the limited reasoning process is that a ready-explanation exists for the removal of any partial explanation in terms of a constraint violation.

While some of the plan-based research efforts including [Wills, 1992] and [Quilici and Chin, 1995] have reported promising empirical results in mapping plan libraries to reasonably sized program segments (up to 1,000 lines) of source code, none have been clearly demonstrated—either analytically or empirically—as scaling up for use in understanding real-world sized software systems. In addition, little work has been done comparing the relative performance of these approaches or analyzing in detail the similarities and differences among 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 because they tend to use collections of heuristic tricks to improve performance (indexing, specialized rule and constraint ordering, and so on). This lack of a common framework means it is difficult to systematically compare these different approaches to program understanding or to understand how their performance will be affected by variants in the plan library such as removals or additions of new plans, or the recognition of specific program plans in part of the existing source code.

What is needed is a framework for describing these heuristic approaches that supports empirical and analytical comparisons of their behavior. I propose, in Chapters 6 and 8, that viewing program understanding as a *constraint satisfaction* problem (CSP)[1] can provide such a framework. For this framework to be useful, existing understanding algorithms must be mapped into this constraint satisfaction framework despite their

[1]See Chapter 4 or [Kumar, 1992] for an overview of constraint satisfaction, or see [Tsang, 1993] for more detail.

differing representations and heuristic tricks. If this framework is sufficiently general to unify these approaches, then I can use it to compare their relative performance and better understand where these algorithms succeed and fail in attacking the program understanding problem. In addition, I 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 global [Kondrak and van Beek, 1995] and local search-based methods [Sosic and Gu, 1990, Minton *et al.*, 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.

In the following subsections, I very briefly describe several program understanding efforts that have motivated this work. In later sections of this thesis I elaborate on some aspects of these efforts where they directly impact my own work.

2.2.2.1 The Programmer's Apprentice

Rich and Waters [Rich and Waters, 1988, Rich and Waters, 1990] headed the Programmer's Apprentice project which focused on the development of a demonstration system (Knowledge-Based Editor in Emacs or `KBEmacs`) with the ability to assist a programmer in analyzing, creating, changing, specifying and verifying software systems. In addition, Rich and Waters [Rich and Waters, 1990, pp. 171-188] describe a cliché recognizer `Recognize` based in `KBEmacs`.

2.2.2.2 Wills' Graph Parsing Method : GRASPR

Wills [Wills, 1990, Wills, 1992], building upon the earlier work of Rich and Waters [Rich and Waters, 1990], outlined an approach to recognition in which stereotypical program or data structures known as *clichés* are represented as a type of graph grammar. A source program is translated into an intermediate representation as a flow graph. These flow graphs are parsed so as to identify all possible derivations of the flow graph based on the known *clichés*. These derivations each represent a possible *partial* interpretation of the source program or mapping to the library of clichés. Wills notes that although the parsing problem is NP-complete in general, experience suggests that attribute constraint checking significantly prunes the search space. Wills evaluates the effectiveness of such an approach empirically for two medium-size source code examples.

Wills' work differs from the approach described in this work in at least three important ways: (1) cliché and program representation, (2) library knowledge representation and exploitation during search, and (3) method of integrating cliché instances in the larger understanding problem. The purely graph-based parsing approach to explaining program source has several drawbacks which I believe are ameliorated using a constraint-based approach, the most important of which is that the representation of a graph is significantly distanced from the parsing (search) strategy, hiding somewhat the effect of structure on search performance. I construct my representation from the other viewpoint, that is, I adopt the terminology and structures of program understanding methodologies into a framework more closely coupled with search performance and search algorithm design. Kozaczynski and Ning [Kozaczynski and Ning, 1994] observe that during experiments users had difficult adapting to a strange syntax and semantic plan language, and the graph-based language may be equally awkward for software developers to utilize.

Rugaber, Stirewalt, Wills and others are part of an effort in reverse engineering being conducted at the Georgia Institute of Technology. Recent work [Rugaber *et al.*, 1995] describes one major research area in program understanding known as interleaving in which program plans intertwine.

2.2.2.3 Concept Recognizer

Kozaczynski and Ning [Kozaczynski and Ning, 1994] describe a method of automatically recognizing abstract concepts in source code given a library of concepts and rules for how to recognize the higher-level concepts in lower-level language concepts, essentially controlling the concept search in a top-down fashion. This system is referred to as the Concept Recognizer.

In the Concept Recognizer, source code is processed to add control and data-flow constraint information to an intermediate representation based on abstract syntax trees (AST). Code is described as having several progressive representations, from the raw *ascii* representation to an syntactic AST, to a "semantic" AST with control and data-flow annotations, eventually to the "conceptual" annotated AST with additional annotation of semantic or abstract concepts in the AST. The knowledge concepts which form the conceptual level of source code exist in a hierarchical *is-a* knowledge library. The intent is not to totally explain a source code with respect to a library, but rather recognize that a great deal of unrecognizable code will exist, and rather recognize "islands" of concepts in the source code. It is implicitly recognized that abstract concepts are not tightly related by syntactic structure, rather the existence of "semantic" information such as control and data-flow relationships is more closely related to the abstract concept level.

An abstract concept consists of sub-components and constraints among them (paralleling control and data-flow factors), and an unspecified matching process is undertaken to recognize these concepts. Since concepts may be built upon other concepts, a need is recognized to somehow order the recognition process so as to recognize lower-level concepts first. This process, however, is largely unspecified in the published work.

Concept Recognizer uses program plans as rules for recognition of concepts. That is, the structure of program plans identifies the portions of a concept to be recognized.

Thus the recognition of a plan element suggests the possible existence of the plan to which the element belongs. In essence, program plans are represented as AST entries which are then compared against a particular program plan AST searching for matches. Real-source examples are briefly reported for COBOL source code instances of approximately 15,000 lines of code. In recognition of approximately 60 abstracted concepts, results are reported to be on the order of one minute to recognize instances of a single concept per 100 lines of code. The ultimate conclusion of this work was that a severe bottleneck existed in the pattern-matching portion of the problem.

While the complete Concept Recognizer algorithm is unfortunately not published, it is observed in [Kozaczynski and Ning, 1994] that the following methods are of use in improving recognition performance for a single concept:

- Early constraint evaluation; In particular, constraints describe the relationship of sub-concepts, and are formulated as *concept recognition rules* [Hartman, 1991b] or *plans*. A portion of these recognition rules includes logical expressions (constraints) which may be thought of as including information about at least control-flow, data-flow, data dependencies and binding constraints.

- Reordering of sub-concept instances; In particular, a heuristic approach to ordering sub-concepts during attempts to identify concept instances is based upon static meta-knowledge about "relevance" between variables and constraints, and domain-independent problem-instance information such as the size of "instance" sets. This size information used to reduce the search space is precisely the kind of information which domain-independent constraint satisfaction algorithms described in Chapter 4 effectively exploit.

- Grouping sub-concept instances based on variable bindings; This may be thought of as another mostly domain-independent search strategy which selects sub-sets of variables to instantiate according to their particular inter-relationships. Attempting to instantiate sets of well-constrained variables may result in an earlier pruning of the search space and more efficient search. Once again, such search issues are raised further in Chapter 4.

The actual CONCEPT RECOGNIZER code was passed on to another researcher, and I touch on this extension to CONCEPT RECOGNIZER next. I examine an approximation of CONCEPT RECOGNIZER's matching process in Section 5.3.1 where the observed performance enhancements are specifically modeled and discussed in some detail. It is important to note that Kozaczynski and Ning observe that "... a limitation with the current (1994) version of the concept recognizer is that the user cannot provide active guidance to the concept recognition process to reduce search space and resolve recognition conflicts.". The method presented in this thesis has been designed with the intent of accommodating exactly this deficiency of knowledge integration. Chapters 8 and 9 outline in detail an approach to understanding which supports an expert-determined amount of interactive search control.

2.2.2.4 DECODE : Quilici's Memory-based Method

One recent approach for program understanding by Quilici [Quilici, 1994], a derivative of earlier work by Kozaczynski and Ning [Kozaczynski and Ning, 1994], is based on a construction of an explicit library of programming plan templates, complete with an indexing ability, which can quickly associate a particular recognized source code fragment with program plan templates in the knowledge base. In this "code-driven" fashion, a combination of top-down and bottom-up search strategies is utilized to implement the matching process. With his DECODE system, Quilici demonstrated how simple C programs could be translated to C++ programs. This approach marks one recent cognitively motivated[2] attempt to extend program understanding using a hierarchical library of program plans.

Program plans (such as those composed through ADTs) are organized hierarchically in a library as shown in Figure 2.4. These source examples may be thought of as derivative from typical class libraries. Software source code in the form of an abstract syntax tree is mapped to the plan library through the use of indices, which are pointers from the source code to parts of the plan library. Index tests indicate when to *specialize* or to *infer* the existence of other plans according to a set of conditions. As an example of specialization, consider Figure 2.4 in which the program plan **initialize-string** is specialized to **builtin-char*-copy** when a direct string assignment is observed in the source code. An example of an inference test is also shown in Figure 2.4, where the existence of **loop-initialize-string** is inferred when an instance of **loop-through-character-array** is "near" a related instance of **copy-character** in the source code.

Given a source code and a program plan, Quilici describes an approach to understanding the source based on a search in the plan library. Search behaves *bottom-up* when existing index tests indicate possible higher-level explanation plans for a particular lower-level component in the library. Quilici observes that people only make bottom-up inferences in particular "well-known" circumstances, and consequently limits the number of upward explanations by inferring only those specified by explicit indexes. On the other hand, search behaves *top-down* when low-level components are indexed and subsequently matched based on some hypothesized high-level plans. Quilici's algorithm attempts to specialize any matched plan as much as possible according to predefined specialization tests, and directs search for low-level plans based on high-level hypothesized plans.

There are, however, a number of shortcomings. First, the lack of a general mathe-

[2] Quilici's work has included observation of the behaviour of student programmers in manipulating source examples.

matical model of the indexing and search process makes it unclear as to how one should coordinate the top-down and bottom-up search. Second, Quilici's algorithm depends on a number of heuristics, such as specializing a plan as much as possible. It is not clear how these heuristics integrate or how they scale-up when the problem size increases. Finally, Quilici makes a substantial effort in capturing actual programmer's methodologies as heuristic enhancements to search control, but presents no empirical results.

DECODE represents the current (1994) state-of-the-art in program understanding implementations, and is built upon earlier work regarded as foundational [Kozaczynski and Ning, 1994]. Consequently, I use DECODE as a typical example for comparative purposes. I describe Quilici's extension to Kozaczynski and Ning's CONCEPT RECOGNIZER in detail in Section 5.4.

2.2.2.5 UNPROG : Hartman

Hartman [Hartman, 1992b] provides an excellent overview of the range of historical approaches to program understanding based upon one or other of these theories. From Hartman it is possible to observe how *constraint* exploitation has played a pivotal role in almost all recognition strategies. In effect, the constraints are formed from the described structure in library program plans and are satisfied through identification of source code fragments which satisfy these control-flow and data-flow constraints.

Following other understanding efforts [Hartman, 1991a, Wills, 1992] I choose to represent source code in an abstract model which make control-flow and data-flow information explicit along with the syntactic structure provided by a parser. The recognition of a particular fragment of code as an instance of a program plan may be thought of as a very limited reasoning problem. Hartman [Hartman, 1992b] indicates that the heart of a program understander is the "comparison" algorithm among plans (represented by schemas and knowledge constraints) and code (represented by abstract syntax trees annotated with structural constraints). Wills [Wills, 1992] and Hartman [Hartman, 1991b, Hartman, 1991a] both observe that even partial recognition is an exponential problem in the worst case. This result, confirmed in [Woods and Yang, 1996a] (and discussed in Section 6.2.0.3), is in itself uninteresting in that it is entirely possible that many (or even all) practical instances are well-constrained enough to be readily solved. In particular, Wills' GRASPR and Hartman's UNPROG both exploit constraints in unique ways during matching search.

Hartman [Hartman, 1992b] identifies several research insights as potentials for advancing the state of understanding. These issues, which have been addressed in previous work [Woods and Yang, 1995b, Woods and Yang, 1996c, Woods and Quilici, 1996a, Woods and Quilici, 1996c, Quilici *et al.*, 1996, Woods, 1995], include: (1) new formalisms, (2) new program and plan decomposition methods, (3) plan representations and pre-processing, (4) empirical study, (5) work towards isolating the effects of program characteristics, and (6) alternative reasoning methods, including existing paradigms. In particular, the focus has been on producing empirical results which can isolate precisely the usefulness of value-added plan representations and alternative reasoning methods in the shared context of algorithms for solving constraint satisfaction problems (CSPs)[3].

2.2.3 Visualization of Software Structure

All of the aforementioned understanding approaches assume that partial automatic understanding of a piece of source code is necessarily based upon mapping existing knowledge about how program plans are constructed into a particular source code instance. It is a general assumption, from which I do not deviate in this work, that the more knowledge that can be derived from a source instance, the more accurate and effective the under-

[3] See [Kumar, 1992] for an excellent overview of constraint satisfaction, or [Tsang, 1993] for more detail.

standing task. Two of the most difficult issues (outside the actual understanding task) in dealing with large and complicated source are visualization and knowledge annotation and extraction.

Effective visualization of software is an important sub-problem of user-interaction for the understanding problem. Program understanding is a knowledge-intensive task immensely difficult for human experts. Computers have traditionally been of great use to human experts in such tasks in very specific ways. Computers can do highly repetitive tasks very quickly and without fatigue. Mathematical correlations can be derived where a human eye or expert would be confused hopelessly with irrelevant clutter. As an example, variable pattern recognizers such as that discussed in [Woods, 1993] can recognize "template instances" or specifically arranged collections of objects dispersed in a field of objects in seconds whereas a human expert is generally unable to perform the task at all. On the other hand, humans are far superior in their ability to bring to bear particular information to a given problem instance. Experience and comprehension abilities might be said to typically outweigh speed in complicated tasks. Happily, a synergy of human and machine offers even better performance than human alone. Traditionally, decision support systems are designed so as to make use of computer functionality in visualization, what-if analysis, complexity organization and mathematical calculation.

Program understanding systems are one form of decision support system. Consequently efforts at partially automating program understanding are (or should be) directed towards providing useful tools to aid the expert analyst in their task of understanding. The work in this thesis is directed towards suggesting ways in which the automatic partial understanding of programs through mappings to existing abstract program plan libraries can be improved. This function is a tool — one of many — that should be provided to an expert. In order for this functionality to be of any use to an expert it must fit within the model of some toolset. Typically these toolsets are built in conjunction with a visualization tool[4] that allows the expert to dynamically configure the focus on source code in particular, useful ways. Visualization efforts seek to find an effective manner in which to both hide and emphasize information, according to an expert's desire and ability to perceive.

In addition to observing raw source code, many possible views on code are possible. These include views showing control-flow and data-flow, views showing modularity of source code, real or implied, views of call-graphs, execution traces, and many other software features. In addition, tools exist to display dynamic algorithmic behaviour at execution. Visualization purports to support the rapid isolation of problems, reveal unanticipated behaviour, reveal areas for program improvement (i.e. profiling revealing execution bottlenecks), and generally support efforts to explain application behaviour. Industrial application and experience is growing rapidly — examples of industrial-strength tools include IBM's OVATION and PV [Kimeham *et al.*, 1994, De Pauw *et al.*, 1994].

Typical partially automated understanding tools exploit as much constraining information about a software source as possible. Consequently, other tools such as annotating parsers are exploited to build a rich structure of information on top of or in conjunction with the source statements themselves. This information can itself be presented to the experts for their own benefit as mentioned earlier. One such tool for generating annotated source code is described below. In addition, one representative visualization research effort that has received a good deal of attention lately is also described.

RIGI

As part of the effort at providing explicit structure for maintenance engineering, reverse engineering and re-engineering, Müller et al. [Müller *et al.*, 1993, Müller *et al.*, 1994,

[4] I make particular mention of such tools as REFINE [Markosian *et al.*, 1994b], and RIGI [Müller *et al.*, 1994] which is mentioned in Section 2.2.3 below.

Tilley *et al.*, 1993], are involved in the construction of RIGI, a system for analyzing software systems which includes visual representations of data and control-flow structures in a code for the identification of subsystems and hierarchies of structure in the code. RIGI is designed to analyze and summarize the structure of large software systems through the use of two complementary interfaces for browsing software: a multiple-windows view similar to hypertext, and a fisheye view [Storey and Müller, 1995] of nested graphs. Clustering techniques have been successfully used to to identify related clusters of program statements.

Any utilization of algorithms for program plan pattern recognition is only useful in the context of such a visualization tool. Comprehensive models of how expert analysts exploit source code are integrated into the very conception of such a tool, and consequently any effective decision support system can only exploit additional structured "what-if" suggestions through careful presentation, and an interactive paradigm which allows an expert to accept or reject any such hypothetical reasoning.

GENOA and GEN++

One basic view of software structure is the abstract syntax tree. Others include control-flow and data-flow graphs. Parsers and annotated parsers that are capable of providing such information are available as commercial products. One system providing this ability is REFINE [Burn, 1992]. Other researchers have created similar tools as part of ongoing research, and under various kinds of agreements make them available for academic use. One such tool is GENOA [Devanbu, 1992], a language-independent code analyzer. With GENOA, Devanbu and Eaves [Devanbu and Eaves, 1994] have constructed `Gen++`, a proprietary tool which generates tools for analysis of C++ code. Specifically, `Gen++` can generate tools which in turn generate annotated abstract syntax trees (ASTs) of C++ code showing control and data-flow information.

2.2.4 My Two-Phased View of Program Understanding: Local and Global

The PU approaches described briefly in this chapter provide "non-iterative" or all-at-once *partial* local explanations of a source code given a particular program plan library. The primary origin of information is program source annotated for control-flow, data-flow and other statically available information that strongly constrains explanations for any element of the code. The implicit assumption in this type of model is that a great deal of contextual information is available in advance of any plan recognition. Consequently, explanatory conclusions are reached through controlled reduction of sub-part explanations, by exploiting the fact that structural constraints must match knowledge constraints for consistent explanations.

In particular, all of these approaches conceive of a generic, abstracted local program plan or cliché instance in a piece of a source code as a partial **local** explanation. That is, an instance of a program plan in a piece of source code describes at least (spatially) that subset of the code taken up in forming the particular plan instance. The explanation is necessarily local to the code affected by the particular instance. Thus, the task of recognizing instances or all instances of one or a set of program plans gives a series of partial explanations. These explanations are necessarily partial in the sense that some code elements are not covered by a particular instance and so remain unexplained. The integration of these local explanations into a coherent (but still possibly partial) picture of what program plans are represented by the combination and inter-relationship of the local explanations may be thought of as **global** explanation. Throughout the rest of this work I will make careful use of the distinction between partial local explanations (as evidenced in program plan template matching), and global explanations (which try to draw conclusions about the inter-relations of local explanations).

While studying earlier program understanding approaches, I determined that the program understanding problem could be broken down into a number of choice points. Examples of these choices include: (1) choosing among candidate unexplained components, (2) choosing among multiple initial plan assignments for a component, (3) choosing among several plans whose existence is implied top-down, and (4) choosing a particular index or specialization test from a candidate set. The existence and interactions of these decisions are buried in Quilici's presentation, but are very important in addressing the efficiency of the search problem. In Chapter 4 I will introduce a mathematical model known as *constraint satisfaction* which constitutes a possible method for representing and exploiting these types of choice points during the search for local and global program explanations.

Chapter 3

Plan Recognition

In Artificial Intelligence research, the problem of program understanding has been approached indirectly from the perspective of plan recognition [Kautz and Allen, 1986, Carberry, 1988, Carberry, 1990a, van Beek *et al.*, 1993, Song and Cohen, 1991, Song, 1990]. In this work, existing human knowledge in a particular domain is represented in hierarchies (of varying types) of plans that describe relevant actions and goals. Program understanding research has taken similar representational approaches. For example, my own hierarchical program plan representational scheme described in Chapter 8 is closely related to that of [Ardissono and Cohen, 1996a, Ardissono and Cohen, 1996b] in which Kautz's hierarchical representation is extended to clarify how sub-components may be shared between specialized children of a given abstraction. Given a hierarchy of this type, and an observation of another agent's plan, a plan-recognizer would typically construct a mapping from input plan fragments to the leaf nodes of the knowledge-base and infer upwards toward a goal. To disambiguate among alternative goals, the mapping processes may employ knowledge about relations between parts of the plan. Many plan recognition examples rely heavily upon temporal relationships, while paying less explicit attention to other types of constraint information. Plan recognition also frequently makes use of cognitively-motivated heuristics such as evidenced through the way context or focus models are used to restrict alternative explanations. These plan recognition programs have been applied mostly to *toy domains* (such as the cooking domain), involving small knowledge bases and a small amount of search.

Plan recognition is the task of interpreting the actions of agents in the environment, in the context of the knowledge possessed about how action occurs in the world, and why. The recognition task involves constructing a mapping, possibly partial, between an existing repository of plan and domain knowledge and observations of some subset of the actions taken toward a goal. Program understanding can be viewed as a variant of plan recognition, where the task is to recognize the plans used to construct a program, and it therefore appears sensible to apply standard plan recognition algorithms to program understanding. This chapter demonstrates that this view is, however, too simplistic. I show (1) that program understanding differs significantly from generalized plan recognition along several key dimensions, (2) that these differences lead to inadequacies in applying typical plan recognition algorithms to program understanding, and (3) that the program understanding task has properties that make it particularly amenable to constraint satisfaction techniques discussed further in Chapter 4 which can lead to appropriate and effective solutions for the program understanding problem.

3.1 The Relation between Plan Recognition and Program Understanding

3.1.1 Software Engineering and Planning

Planning (see for example [Yang, 1996, Allen *et al.*, 1990]) is the process of generating sequences of actions in order to provide a method for agents to modify the state of the world in which they are situated. Typically in planning, plan *operators* are specified which describe the types of actions a particular agent or system is capable of performing. These operators are defined in terms of *preconditions* which must be met before a given operator (or action) may be applied or undertaken, and *postconditions*, the set of effects, either primary or incidental that result from the application or use of the chosen operator. The planning "world" or state is typically defined in terms of a set of predicates which model the "real" world in which the planning or executing agent(s) are situated. These predicates are exactly those which preconditions and postconditions may affect. Thus, given a particular *initial* state and a desired *goal* state, the job of a planner is to determine an appropriate sequence of actions which will essentially change the situation from the initial to the desired. A very wide range of approaches have been taken to this problem. In *classical* planning an entire sequence is traditionally determined in advance of execution based primarily on the assumption that the world will not change during deliberation or execution (other than as intended), and that actions have their desired effects. In *reactive* planning [Agre and Chapman, 1987], long-range goal-based plans are not generated, but rather, an agent undertakes very small "reactive" actions in accordance to environmental stimuli. In *anytime* problem solving or planning, a system plans only for an arbitrary amount of time, returning a "best-guess" solution at the end of this time period. Other approaches exist also and [Yang, 1996] provides a wide breadth of background for the planning process.

Just as planning is a process of action generation to meet goals, the construction of computer programs is also a process of sequencing actions so as to meet goals. For example, the use of *design patterns* [Gamma *et al.*, 1995, Gamma *et al.*, 1993] in software design mirrors case-based planners. Case-based planning [Hammond, 1990] is an area of planning in which previously constructed plans are placed in a library, and when a new situation arises requiring a new plan, an attempt is made to find a "closely related" old plan and modify it to fit the new situation. Similarly, in software or design reuse efforts, efforts are being made to compile libraries of typical design patterns and then apply them to appropriate situations with only minor changes.

These analogies to planning are currently being exploited in the software-development side of software, and rely on the similarity of *forward engineering* (see Section 2.1.2) of software and the process of generating a plan appropriate to a given situation. However, a similar parallel can be drawn between the process of *reverse engineering* (Section 2.1.2) or understanding software and *plan recognition*. If one assumes that the current state of some world viewed through a set of perceptions is the result of the execution of some agent's plan (constructed as already introduced), then it seems reasonable to observe that it may be possible to discern this plan based on the perceptions. This process may be loosely defined as plan recognition. It is obvious to observe that software is an artifact of intelligent planning and execution. In fact, it may be thought of as exactly the desired (or nearly so!) result of some agent's process of planning a design, planning a program to meet that design, and executing an implementation plan to create such a program. Further, the resulting program is in fact a very precise representation of some plan in the domain of implementation. A program accepts a set of inputs (or preconditions for its execution), and creates a set of outputs (or postconditions) that may be thought to change the world into which they are sent. If the domain in which the program or plan was intended to function is well known, it would be possible to describe the purpose or goals of the program or plan. In this chapter I discuss more closely the relation of plan recognition research and the task of understanding software.

3.1.2 Motivation and Introduction

Every day people make decisions about how to act and what to say based on the coupling of perceptions of actions and events undertaken by other actors or agents in the world and *interpretation or understanding* of these actions and events. In particular, we understand that people react to their environment in ways that they believe will help them satisfy their own goals. From this assumption, and our perceptions of the actions of others, we plan our own actions in the world. In a similar manner people interpret human-generated artifacts which they encounter in their everyday lives. For example, we all expect that reports, tools, books and phone messages are created by intelligent agents in our world who have fulfilled specific goals of their own in creating the artifacts which we perceive. Programmers routinely conceptualize software source code as created with particular goals in mind in order to further the process of deciphering exactly what is or was intended in the code. In general we all infer the goals of other agents in a similar manner as a matter of course each day.

Plan Recognition (PR) is the task of creating a contextual model of the intentions underlying the actions of agents. *Program Understanding* (PU) is the task of creating a contextual model of the intentions underlying actions encoded into program source code. From these simple descriptions of PU and PR, it may be tempting to view PU as simply an instance of PR, and further, recognize that methodologies presented for PR should readily apply to PU. This chapter will clarify the classes of problems that PR and PU methodologies intend to address, and describe the ways in which these classes both differ and resemble one another. As part of this explanation, I show that a straightforward interpretation of PU as a particular kind of PR is incapable of exploiting the particular temporal and causal structures embedded in source code. I point out that PU may be thought of as a *simplified* or more tightly-constrained version of PR that remains NP-hard (see Section 8.2). PU provides an interesting example problem on which to build methodologies which may be extended to effectively address instances of the more general PR problem.

I describe the close relationship between PR and PU in the following Sections. In Sections 3.1.3 and 3.1.4 we examine each of PR and PU in turn, and attempt to clarify the structural differences in these problems through use of examples. In Section 3.1.5 I describe an attempt to model an approach to PU in the spirit of one seminal, typical PR algorithm, and illustrate the inadequacy of this approach. In Section 3.1.8 I summarize the relationship between PR and PU before continuing.

3.1.3 The Plan Recognition Paradigm

Plan Recognition can be thought of as the task of determining the *best*[1] unified *context* which causally explains a set of perceived events as they are observed. A context is essentially a hierarchical set of plans and goals that accounts for the observed actions. This process generally assumes a specific body of knowledge which describes and limits the types and combinations of events that may be expected to occur. This knowledge body is frequently represented as a specialization and decomposition structure of events and actions.

Kautz and Allen [Kautz and Allen, 1986] formalized an approach to PR that has served as a primary building block for many subsequent PR methodologies, including [van Beek *et al.*, 1993]. In addition, others address similar issues of explanation selection

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

in separate representational and conceptual schemes [Carberry, 1988, Carberry, 1990a]. Kautz and Allen define the PR process as that by which "a set of observed or described actions is explained by constructing a plan that contains them". A model of PR is formed with the intent of both representing actual events or occurrences, and of proposing hypothetical explanations of actions. Explaining action through PR involves uncertainty, and therefore it is necessary to somehow recognize some *particular* plan that another agent is performing from a possibly large set of explanatory plans. The process of arbitrating this uncertain selection process is the primary focus of the work of Kautz and Allen, and of plan recognition systems in general. For the purposes of this study, I focus on the interpretation of Kautz and Allen primarily.

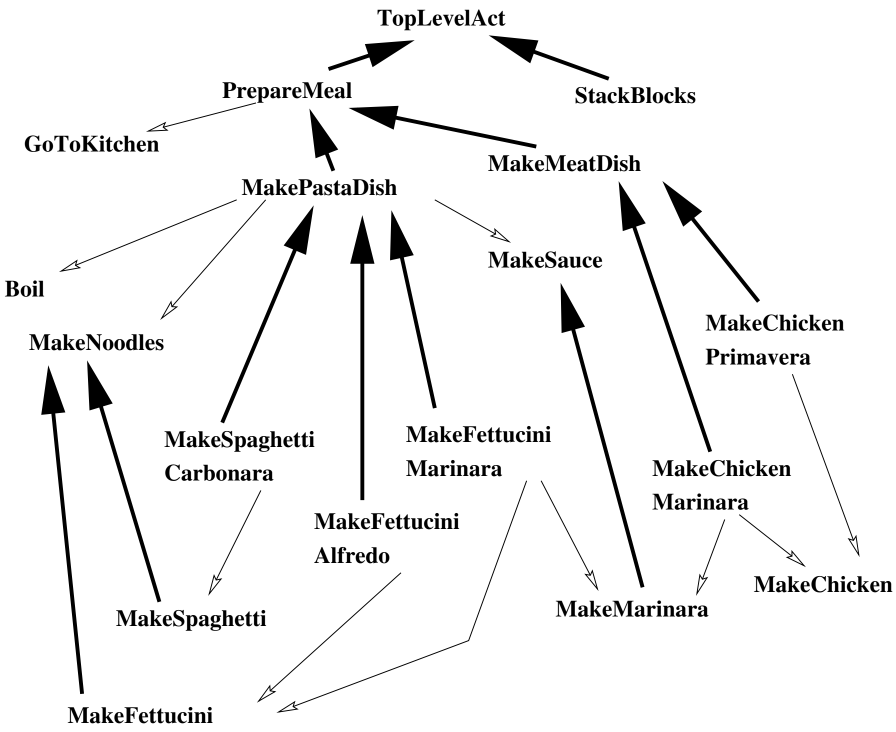


Figure 3.1: Action hierarchy for the cooking domain

Kautz and Allen's approach is based upon ordinary deductive inference. The *rules* for

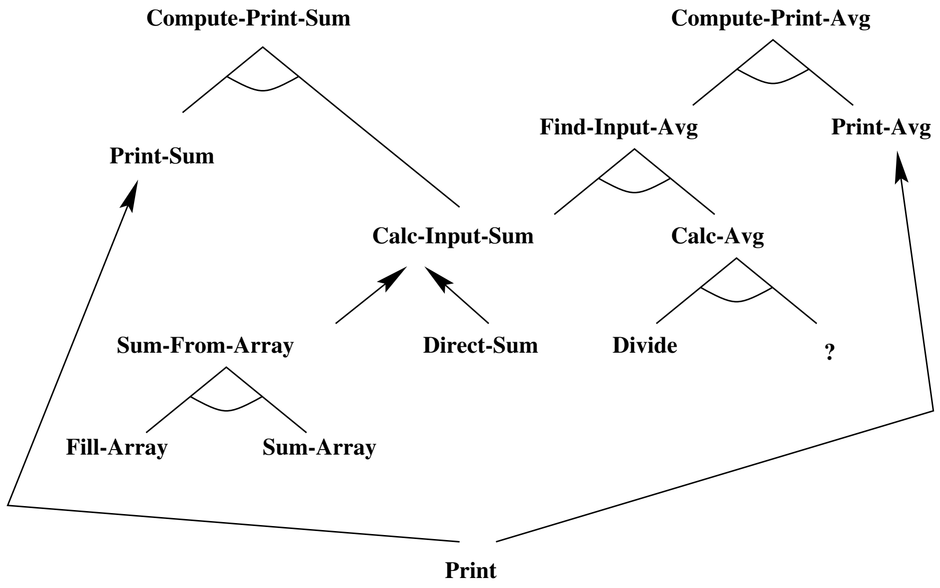


Figure 3.2: An Example Action Hierarchy

deduction are rooted in the exhaustive body of knowledge about actions in a particular domain encoded in the form of an **action hierarchy**. The hierarchy of Figure 3.1 taken directly from [Kautz, 1987] depicts specialization relations as dark arrows from specific to general actions. The thinner lines encode decomposition of actions into a set of sub-actions. Not encoded in this figure are additional domain constraints such as temporal relations among sub-actions, although this information is assumed to be available for the plan recognition process. For instance, in **MakePastaDish** it is assumed that the constraint that **Boil** precedes **MakeNoodles** is included.[2] For illustrative purposes, a similar hierarchy in the program understanding domain (note that composition is indicated by an arc and specialization with an arrow) is given in Figure 3.2.

The action hierarchy describes all ways in which any expected action may be performed or used as a step in a more complex action. The *trigger* for deduction is the

[2] Note that **MakeNoodles** refers to the process of transforming dry noodles to hot noodles ready for serving, not the process of actually making the individual noodles from pasta dough.

perception of an action. As an example, observe that the hierarchy encodes that **Boil** and **MakeNoodles** are sub-actions of **MakePastaDish**, and further that they are sub-actions of no other action. Perception of an instance of **Boil** then results in the deduction that the more abstract task being undertaken is **MakePastaDish**, and similarly, **PrepareMeal** and **TopLevelAct**. Actions are perceived one at a time, with a model of the agent's intention maintained incrementally following each perception. Although at any point in the process the determination of the perceived agent's plan may be ambiguous[3] (or rather, disjunctive), specific predictions about future activities can still be made. For instance, imagine the perception of an action which is identified as either **MakeSpaghetti** or **MakeFettucini**. Since both of these actions are instances of **MakeNoodles**, it can be deduced that the higher level task **MakeNoodles** is being undertaken. Now, since **MakePastaDish** has the additional sub-action **Boil**, we can expect to perceive **Boil** in the future (if it indeed has not yet been perceived).

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

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

[3]Other work in PR [van Beek *et al.*, 1993] addresses the issue of resolving ambiguity only *when necessary* through interactive dialogue focusing on explanatory plans that share particular faults — where a fault is an inability of a plan to perfectly match the observations.

only decompositions.

- *Full Sensibility*; All actions are purposeful, that is, any non top-level action occurs only as part of the decomposition of some top-level action.

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

Kautz and Allen explain the plan recognition process as follows. First, the plan hierarchy is processed into a set of axioms according to the hierarchy structure and the assumptions stated. Next, a specialized forward chaining reasoning process embodying a particular inference strategy over these axioms is undertaken. As each observation is received, the system chains up both the abstraction and decomposition hierarchies until a top-level action is reached. The intermediate steps may include many disjunctive statements, such as in the **MakeFettucini**, **MakeSpaghetti** example introduced in Section 3.1.3. The action hierarchy is used as a *control graph* which directs and limits this disjunctive reasoning. After more than one observation arrives, the system will have derived two or more top-level action instances (that is, it will have found a set of paths from each observed action, through the action hierarchy, to top-level actions). The system then applies a simplicity heuristic to unify the disjoint explanations. This heuristic is to prefer as few high-level actions as possible, or, in other words, to reduce the explanation to the set of actions and the minimal set of higher-level plans that "cover" all of them. When this heuristic is applied, the result is a set of restrictive assertions about the functions of each observed actions. If this causes an inconsistency, the system backtracks up the explanation path to where the simplicity heuristic incorrectly merged the explanation paths.

Kautz [Kautz, 1987] identifies two primary problems that must be dealt with in incremental recognition systems. The first of these is the *combinatorial problem* which arises when the minimum cardinality assumption is relaxed to include two or more primary actions. This relaxation allows the number of possible ways of grouping together the set of observations to grow exponentially[4]. The second problem identified is the *persistence problem*: once two observations are tied together or interpreted in a particular context (say as a result of the minimum cardinality assumption), entirely discarding this context simply on account of the arrival of a contradictory piece of information *seems unnatural* from a human reasoning viewpoint.

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

Refer once again to Figure 3.1 from page 54. A **Boil** action is perceived. **Boil** only occurs as part of **MakePastaDish**, and consequently **MakePastaDish** is adopted as the covering plan. Next, a **MakeMarinara** action is perceived. **MakeMarinara** can be covered by **MakeFettuciniMarinara**, and subsequently **MakePastaDish**, or by **MakeChickenMarinara**, and subsequently the additional high-level covering plan **MakeMeatDish**. What then is the plan being undertaken? According to minimum cover, **MakePastaDish** covers both **Boil** and **MakeMarinara**, and so **MakePastaDish** is accepted as the current plan; **MakeChickenMarinara** and **MakeMeatDish** may be denied. Now suppose the next perception is a **MakeChicken** action.

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

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

Kautz's algorithm

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

Kautz's algorithm [Kautz, 1987] as depicted in Table 3.1, has three main parts: (1)

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

Recognition summary

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

3.1.4 Program Understanding Recalled

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

Algorithm KAUTZRECOGNIZE(Obs_{set}, $Hier$);
Input: A set of observations $Obs_{event} \in Obs_{set}$, a plan hierarchy $Hier$ structure;
Output: A hypothesis $Hypo$ consisting of all explanation graphs $ExplGraph$ instantiating part of the hierarchy covering the set of observations Obs_{set};

SubRoutines

A. $ExplainObservation(Obs_{event}, Hier)$:returns an event graph $ObsEventGraph$ which explains Obs_{event}.

B. $MatchGraphs(ObsEventGraph, ExplGraph)$: returns a new $ExplGraph$ including $ObsEventGraph$ and TRUE; or FALSE.

Main Routine

1 $Hypo_{set} := NULL$;
2 **for each** Obs_{event} **in** Obs_{set} **do**
3   $ObsEventGraph := ExplainObservation(Obs_{event}, Hier)$;
4   **for each** $Hypo$ **in** $Hypo_{set}$ **do**

5     $Hypo_{set} := Hypo_{set} - Hypo$;
6     $Hypo_{set} := Hypo_{set} + Union(Hypo, ObsEventGraph)$;
7   **for each** $ExplGraph$ **in** $Hypo$ **do**
8       $(ExplGraph_{new}, MatchOkay) :=$
        $MatchGraphs(ObsEventGraph, ExplGraph)$;

9   **if** $MatchOkay$
      then
        $Hypo_{set} := Hypo_{set} + Union((Hypo - ExplGraph), ExplGraph_{new})$;
      endif
10   **endfor** (step 7)
11   **endfor** (step 4)
12 **endfor** (step 2)

13 **Return** $Hypo \in Hypo_{set}$ **with** minimum $cardinality(Hypo)$;

Table 3.1: The Kautz Non-dichronic Program Understanding algorithm

Recent work in PU (of which this thesis is also an example) [Woods and Yang, 1995b, Woods and Yang, 1995a, Woods and Yang, 1995c, Quilici, 1994, Quilici and Chin, 1994, Quilici, 1995b, Quilici and Chin, 1995, Kozaczynski and Ning, 1994, Rich and Waters, 1990, Wills, 1990, Wills, 1992, Rugaber *et al.*, 1995] has tended to describe approaches to PU based on the existence of a domain-dependent knowledge library which consists of programming plan templates and concepts. Source code is interpreted within the context of a specific body of knowledge that describes how programs in general, and domain-programs in particular, are known to be structured. Various top-down and bottom-up search strategies are utilized to construct partial mappings between the source code and knowledge. To some extent, these approaches are all aimed at improving the effectiveness of the mapping process through exploiting heuristic knowledge. One primary goal of this thesis is to bring together the range of program understanding strategies and heuristics into a single representational framework.

In Figure 1.2 on page 6, a subset of expert knowledge about a particular application domain is represented as a fragment of a hierarchical library of program templates.[5] The encoded structure, or *knowledge constraints*, includes temporal, control-flow, and data-flow relations among the components of plans. In addition, typical or expected structure can be represented in the hierarchy as preferences for certain common specializations or indices for frequently related plans. Figure 1.2 shows one possible mapping between a plan template from the library and a specific source fragment, in this case a single source statement. The existence of such a mapping essentially *explains* the presence of the low-level source statement at a higher level of abstraction, in this case as an instance of the plan template **copy-character** specified in the library.

[5] In PR, the existence of hierarchical plan libraries for a particular domain is assumed.

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

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

- *Closed Specialization*; The known ways of specializing a particular abstract plan are the only ways to consider, despite the fact that others may exist.

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

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

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

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

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

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

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

In Chapters 7 and 8 I present results which demonstrate that an effective approach to partial PU is possible with large source code examples. In particular, I utilize existing technology in the form of specialized algorithms for solving a class of problems which is shown to include program understanding. Additionally through this project I have represented the particular problem structures, constraints, and solution strategies in a unified framework. The model I have chosen for this generalized representation of PU is as a constraint satisfaction problem (CSP), as described in detail in Chapter 4.

3.1.5 Program Understanding as Plan Recognition

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

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

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

Incorrect Plan Recognition with Kautz

For a moment let us assume that we will try to apply the Kautz plan recognition representations and algorithm to a simple program understanding example. As a consequence of the *Open Perception* assumption and the simplicity heuristic used to deal with it, the Kautz and Allen approach can find an incorrect explanation, despite there being sufficient knowledge to eliminate it as a candidate. To illustrate, consider the following simple line of code representing a self-contained plan or function:

```
c := (a + b)/2;
```

Consider for this example the hierarchy of Figure 3.3. One may view this example as a pair of observed actions: **Sum-Pair**, **Divide-Pair**, and **Assign**. Ignore assignment and other structural constraints for this example. We wish to find plans for explaining this program fragment with response to the hierarchy of Figure 3.3. Consider that each portion of this complex program statement constitutes a block or program action that must be explained in the context of the program plan hierarchy. Note that in this example and throughout the remainder of this thesis, I adapt the hierarchical representational scheme for specialization and composition: specialization relations are indicated with upward arrows (from specific to general) and composition as a set of joined arcs from a parent to its children. This compositional scheme is framed clearly in Figure 8.13 on page 216.

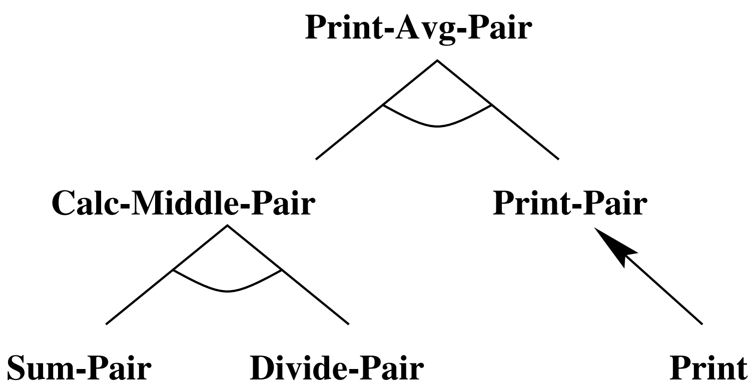


Figure 3.3: Another Example Action Hierarchy

Upon recognizing **Sum-Pair**, the explanation is **Sum-Pair** is-part-of **Calc-Middle-Pair** is-part-of **Print-Avg-Pair**. After recognizing **Divide-Pair**, the explanation is substantially the same, except that it now covers the **Divide-Pair** as well. However, if the **Print** never occurs, the problem is that this explanation cannot be the case. The explanation, in fact, should be that **Sum-Pair** and **Divide-Pair** are a part of **Calc-Middle-Pair**.

The problem is that the explanation **Print-Avg-Pair** is actually wrong, given that we know no more actions relevant to these plans will appear in the program. Although this explanation is minimal *in terms of top-level actions*, it allows for the assumption that future actions will be encountered. In program understanding, it is inappropriate for the covering set to cover more actions than have already been encountered. Consequently, an exact covering set that is not necessarily minimal would give the correct explanation.

Situations like the one in this example can occur frequently in program understanding because of *incomplete plan libraries*. It is unlikely that a plan library will contain all the plans necessary to understand a program [Chin and Quilici, 1996, Quilici, 1995a]. **Sum-Pair**, for example, has a myriad of uses, only some of which will be captured by any plan library. The result is that any algorithm we use must be capable of producing a forest of intermediate plans and not attempt to infer potentially incorrect high-level groupings.

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

It is reasonable to wonder whether Kautz's approach can easily be modified to address this problem. An initial modification might be to change it so that, after all events have been processed, it runs through all of the explanation chains it forms and eliminates any that covers an event that did not occur. This guarantees that the algorithm never terminates with an incorrect explanation. However, since this algorithm does not carry along every candidate explanation, it may lose the correct portion of the overall explanation (**Calc-Middle-Pair**, in the example). To address this, it is necessary to modify the algorithm to carry around all potential explanations and to use search to eliminate those that are missing events, a process which is likely to be inefficient.

Inefficient Plan Recognition

Another problem with applying the Kautz approach directly is the *Combinatorial Problem* that occurs because any given action can be a component of a multitude of plans that can themselves be actions within a multitude of plans, and so on. The result is that the number of possible explanations for a given set of observations can grow exponentially. To determine a minimal event cover of perceived actions from a plan hierarchy, it is necessary to generate potential covers and search to select the minimal one. This problem can be thought of as defining a search space of covers. Each action needs to be covered by some plan. Consider the analogical program understanding problem. Each perceived program statement needs to be covered by a program plan, in the order in which they appear in a source code. The Kautz method essentially imposes a single ordering on the domain values or program statements, resulting in a statically defined search order, hence a potentially very inefficient search tree. The effects such ordering are analytically evaluated in the search strategy discussion in Chapter 4, and empirically in the program understanding domain in Chapter 7.

This problem is especially relevant to program understanding since most programs involve thousands and thousands of actions. Kautz explicitly notes this problem, and suggests that in some domains the combinatorial problem may be largely mediated through constraints on event types; however, he asserts that in realistically sized problems additional principles will be required.

3.1.6 Comparing PR and PU

The simple aforementioned example highlights one important difference between PU and PR. Program understanding is a special form of plan recognition in which PU has more knowledge about the source domain structure available. However, the differences between general plan recognition and program understanding also imply very different approaches for tackling the problems.

In general plan recognition, it is fundamental to assume that the observations are incomplete. This makes it necessary to use algorithms that leave as much room as possible for incorporating potential new observations in the future. In plan recognition, given the fact that any observation set may be incomplete, it is therefore natural to judge all potential explanations on an equal basis — the minimality basis built on cardinality (of covering plans).

On the other hand, for program understanding, the observations are always complete — the entire source program is available to the programmer at once, and no guesswork is permissible. In this case, cardinality-based incremental algorithms are no longer the best choice. Instead, explanations must be found in such a way that

1. explains all observations (program source code), and
2. does not admit anything else.

That is, the recognition algorithm for PU must be as tight as possible.

It is also important to distinguish whether the knowledge base, or plan hierarchy, can be assumed complete or not. If the plan hierarchy is assumed to be incomplete, as in the case of general plan recognition, one must utilize non-monotonic algorithms to account for potential new observations. When the plan hierarchy is complete, however, the algorithm designed for recognition is monotonic; once a hypothesis is ruled out, it is never considered again.

3.1.7 Dimensions of Comparison

The inputs to a plan recognition algorithm and a program understanding algorithm, are the following:

1. observations,

2. plan hierarchy.

One may therefore discuss the similarity and difference of PR and PU approaches according in terms of the following points:

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

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

- The set of observations or perceived actions is either **strongly** or **weakly** causally constrained. For example, in PU there is no ambiguity regarding the causal relatedness of some actions (such as in the case of actions in a loop structure), while in PR for the cooking domain, action perceptions require additional assumptions to connect them causally. This difference is of great importance in controlling the combinatorial problem Kautz outlines: the *structural constraints* available through

Observation Set ↓ : Library Structure →	InComplete	Complete
InComplete	Extended PR	Kautz PR Algorithm
Complete	Partial PU	Special case PU

Table 3.2: PU versus PR Comparison of assumptions

preprocessing of source code are what support efficient solution of large PU problems. It is important to note that Kautz makes use of this type of source constraint, where possible, in the form of observable temporal relationships in the cooking domain.

Below, the PU and PR approaches are categorized based on their assumptions about hierarchy and observation set completeness. These results are highlighted in Table 3.2, and discussed as follows:

- **Library InComplete, Observation Set InComplete**

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

- **Library InComplete, Observation Set Complete**

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

- **Library Complete, Observation Set InComplete**

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

- **Library Complete, Observation Set Complete**

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

3.1.8 Comparative Summary of PR and PU

Planning and software *forward* engineering are highly related. Similarly related are plan recognition and software understanding as part of *reverse* or *re*-engineering. In particular, past PR and PU approaches exhibit a great deal of similarity:

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

However, the approaches differ in very significant ways:

- The Kautz PR strategy assumes a complete library and incomplete observation set, and consequently is difficult to apply to a more restricted PU domain in which an incomplete library and complete observation set are the norm.
- The differing assumption sets can result in over-committed solutions when the PR concept of observation set minimal covering is applied to PU. The assumption of an incomplete observation set is the basis for preferring few top-level plans rather than a number of apparently disjoint partial plans.
- PR has a less-restrictive constraint set upon which to limit the combinatorial problem of disjunctive explanation. While PU may exploit the wealth of structural constraints easily-extracted from the source before recognition, PR examples have

been typically largely limited to explicit temporal constraints[6]. Consequently, one may expect to solve larger PU problems more efficiently than comparably sized PR problems.

- PU can be thought of as a special, well constrained, case of PR which remains difficult (NP-hard). While we have seen why general PR approaches are inapplicable to typical PU problem instances, it should be emphasized that one important result of this study is the suggestion that the techniques used in PU be considered for the more general PR problem. In particular, certain PR problem instances could admit pre-processing of the observation set to identify particular causal relationships. These explicit relationships should be applied in conjunction with action representations so as to increase the number and type of constraints available in the problem solution.

In addition to significant similarities and differences in approaches to mapping a set of perceived artifacts to a knowledge body, plan recognition and program understanding are related in other ways as well — a large body of work in plan recognition is highly complimentary to program understanding efforts. In particular, that work which intersects user modeling and plan recognition can be utilized in program understanding. This work focuses on cooperative solution of the plan recognition problem between an "expert" advisor-type system and a particular user who is undertaking a particular, possibly unknown, goal. Examples of such work includes [Ardissono and Cohen, 1996a, Ardissono and Cohen, 1996b, van Beek *et al.*, 1993, Carberry, 1990b]. In such systems, plan recognition is undertaken by the advising system in an effort at determining exactly goals the user is pursuing, and adapting responses and information-provision strategies appropriately. As outlined in Chapter 2, any successful effort at providing automated

[6] Of course, related work in user modeling has attempted to restrict the range of explanation with "focus" heuristics which may be thought of as explicit constraints an explanation.

tools to assist the program understanding effort is predicated on the existence of good interaction strategies and effective user interface tools. This plan recognition and user modeling work is a necessary step in bridging the gaps between visual tools for recognition, representation of hierarchies, user interaction and understanding tools such as are described more precisely in the following chapters.

3.1.9 Looking Ahead: Adapting PR for PU

In some sense, program understanding has more knowledge available than is present in typical AI plan recognition domains. In particular, program understanders have the complete set of actions that are present in the program and many detailed data-flow and control-flow constraints among those actions. This allows program understanding to take a breadth-first approach to plan recognition, which avoids carrying along unconfirmed and possibly incorrect hypotheses. There exist other similar domains appropriate to plan recognition techniques which exploit all-at-once action knowledge. For example, text comprehension or examination of execution or action traces offer much the same advantages as program understanding in terms of complete action sets. However, it is difficult to envisage other plan recognition domains which have as rich a set of inter-action or structural information available about the connectivity of action.

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

In program plan recognition, one may immediately verify a portion of any hypothesized explanation chain, and gradually construct explanation chains from verified pieces.

In particular, given an action that is potentially part of a set of plans containing only actions (and not sub-plans), one can immediately verify whether that plan actually exists by locating the plan's other actions and verifying any constraints between them. That is, it is possible to use each action in the AST (abstract syntax tree) as an index to the set of potential plans that might contain it, and then check whether each of those plans are present. Thus, at the end of a pass through all actions, the plan recognizer has located verified-single plan explanations for each of the actions. Quilici [Quilici, 1994] adopts just such an approach which is discussed in detail in Chapter 5.

One simple, complete way to locate total, verified-explanation chains is to organize the plan library in layers, where the first layer consists of those plans that are solely events in a program's AST, the next layer consists of those plans that depend only on the events in the AST and plans in the first layer, and so on. After recognizing those plans in the initial layer, the plan recognizer runs through each of those plans and verifies whether the plans in the next layer that can contain them are actually present, creating a new set of verified recognized plans. This process is repeated until there are no newly recognized plans.

A question remains as to how to perform this verification process. That is, given that an action suggests a set of possible plans that might explain it, how can one verify which of these plans are actually present? Given the presence of many constraints among the actions in any plan, this suggests using a constraint satisfaction approach, which I discuss in detail in Section 4 and apply to recognition of local sub-plans for programs in Chapter 6 and global program plans in Chapter 8.

A Constraint Satisfaction Problem consists of three major components: A set of variables, a finite domain value set for each variable, and a set of constraints among the variables which restrict domain value assignments. A solution of a CSP is a set of domain value to variable assignments such that all inter-variable constraints are satisfied. These mechanisms include global [Kondrak and van Beek, 1995] and local search-based methods (see [Sosic and Gu, 1990], [Minton *et al.*, 1992], and [Yang and Fong, 1992]), constraint-propagation problem simplifications (see [Nadel, 1989], [Dechter, 1992], and [Prosser, 1993]), hierarchical exploitation of problem structure [Freuder and Wallace, 1992], as well as hybrid combinations of these approaches.

In using a CSP for the task of verifying whether a single plan is present in a particular program fragment (or situation), the variables correspond to the actions in the plan, the domain values are the source statements (or sub-plans) with the same type within the program, and the constraints are reflexive type constraints on each variable, along with inter-variable constraints such as data and control-flow. Variables here can have attributes such as (**print**, **for**) that may be seen as *constraints* on allowable assignment of program statements (values) to plan features (variables). Other *constraints* are on the sharing of information among variables, and on the order in which plan components or variables are expected to appear in source code. Example plans using this representation have been presented in [Woods and Quilici, 1996c] and [Woods and Yang, 1995b], and are given here in Chapter 6.

A solution to the CSP consists of the set of all assignments of plan features by source code statements, where each assignment must satisfy all constraints. The solution to a CSP provides a mapping that *explains* the matched source statements as parts of an instance of the abstract program plan or ADT. When one start solving a particular CSP for recognizing a particular plan, the variable which represents the action which triggered this plan's consideration is restricted to the single domain value corresponding to that action. Thus, in some sense, each CSP is starting off partially solved. For example, if a recognized sub-part of a program plan template for computing an average was a summation of a vector, then the template variable for "summation" would be instantiated to the recognized summation code, and other, currently unrecognized plan sub-parts such as "compute-set-size" would be initially uninstantiated.

Applying ordinary plan recognition to program understanding imposes an ordering of the program statements — essentially they are considered in temporal order, top to bottom. Consider the simple case of attempting to recognize a single program plan in the CSP framework using the Kautz imposed order. A search space results in which the components of the CSP have domain ranges which include all program statements. A "cover" of the components that satisfies the existing component constraints is a potential solution. The domain ranges are ordered temporally (early program statements first), thus resulting in the generation of potential solutions with "earlier" combinations first, "later combinations" second, and an eventual generation of all combinations. Kautz's insight that "additional principles" would be required to mediate the search can be at least partially satisfied for program understanding through the use of intelligent backtracking strategies during this process. In contrast, a constraint satisfaction algorithm relaxes the temporal ordering of domain ranges by dynamically re-arranging the domains (in the spirit of some types of forward checking algorithms), and reaping the benefits of improved search results through more effective constraint applications which reduce entire sub-parts of the search space.

Program understanding is often viewed as a task of understanding the plans inherent in a software code. I have demonstrated that there are serious problems with the naive notion of simply applying AI plan recognition algorithms, and that these problems in some sense justify the rejection of this AI algorithm by researchers in program understanding. However, as we shall see in subsequent chapters of this work, as a result partially of careful analysis of the problems that arise in applying at least one existing AI plan recognition algorithm to program understanding, it is possible to construct a variant of that algorithm that appears effective in efficiently recognizing certain classes of plans in real-world programs.

I investigate the constraint satisfaction paradigm in some depth next in Chapter 4.

Part II

Modeling Framework

Chapter 4

The Constraint Satisfaction Paradigm

4.1 Motivation and Background

I present here a framework in which to represent program understanding, and a corresponding approach for attempting to generate solutions from a base of situational information. One possible method of expression for this problem would be as a constraint satisfaction problem (CSP). CSPs have been studied extensively, and a variety of problem domains have been formulated in this framework. To list even a representative cross section of these papers would be difficult, however, some recent applications and evaluations include [Tolba *et al.*, 1991], [Yang and Fong, 1992], [Norvig, 1992], [Van Hentenryck *et al.*, 1992b], [Guan and Friedrich, 1992], [Nadel, 1989] and [Nadel, 1990]. Many authors have been noted for continued interest in broad discussions of the applicability of CSP and specific theoretical and practical contributions, however, the most notable are perhaps Mackworth, Dechter and Freuder (see for example [Mackworth, 1981], [Mackworth, 1987], [Mackworth, 1992], [Mackworth and Freuder, 1993], [Mackworth and Freuder, 1985], [Mackworth, 1977], [Dechter and Pearl, 1987], [Dechter and Pearl, 1989], [Dechter and Meiri, 1989], [Dechter and Dechter, 1987], [Dechter, 1990a], [Dechter, 1990b], [Dechter, 1992], [Hubbe and Freuder, 1992], [Freuder and Mackworth, 1992], [Freuder and Wallace, 1992], [Freuder, 1982], and [Freuder, 1991]).

In this section I intend to provide a brief description of constraint satisfaction problems, and of how they are typically solved with a combination of heuristic search and constraint propagation (CP) techniques. I discuss how the issues of constraint propagation and search affect a specific problem instance. Of particular interest in this discussion will be the performance of solution strategies, and possible ways in which interaction with the user of a constraint-based system might be engineered. In a later section I will formulate a spatial template recognition problem as an example of how domain heuristics such as spatial locality are exploited in solving CSPs.

Constraint satisfaction problems (CSPs) provide a simple and yet powerful framework for solving a large variety of AI problems. The technique has been industrially applied in a wide variety of domains [Van Hentenryck, 1989]. A successful application of this technique to knowledge-based planning is presented in [Yang, 1992].

A good introduction to CSP can be found in [Kumar, 1992], which presents a general overview of the formulation of many AI problems as CSP including those in machine vision, belief maintenance, scheduling, temporal reasoning, graph problems, floor plan design, genetic experiment planning, and the satisfiability problem. Kumar outlines several different approaches to solving these problems such as backtracking search, constraint propagation, and some hybrid combinations of the two. A more detailed discussion of the field and a complete description of relevant algorithms may be found in [Tsang, 1993].

4.2 A Simple Example

A constraint satisfaction problem (CSP) can be formulated abstractly in terms of three components:

1. a set of **variables**, $X_i, i = 1, 2 \ldots n$,
2. for each variable X_i a set of values $\{v_{i1}, v_{i2}, \ldots v_{ik}\}$. Each set is called a **domain** for the corresponding variable, denoted as $\texttt{domain}(X_i)$,
3. a collection of **constraints** that defines the permissible subsets of values assignable to particular variables.

The goal of a CSP is to find one (or all) assignment of values to the variables such that no constraints are violated. Each assignment, $\{x_i = v_{ij_i},\ i = 1, 2, \ldots, n\}$, is called a **solution** to the CSP.

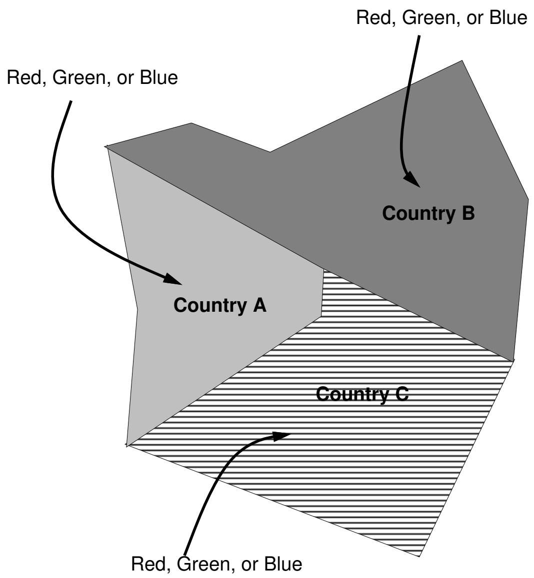


Figure 4.1: A Map Coloring Problem

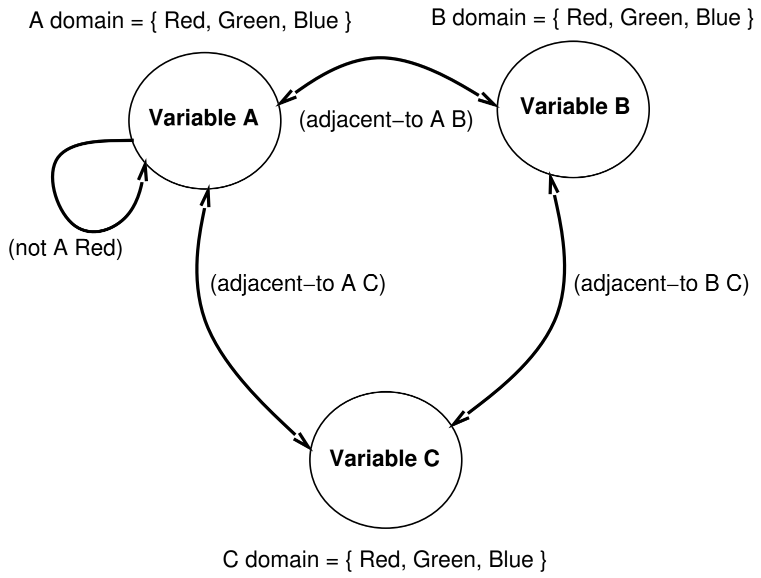


Figure 4.2: Map-Coloring CSP

As an example of a CSP, consider a map-coloring problem, where the variables are regions A, B, C that are to be colored (see Figure 4.1). In any final solution every region must be assigned a color such that no two adjacent regions share the same color. The CSP representation of this problem is given in Figure 4.2. A domain for a variable is the set of alternative colors that a region can be painted with. For example, a domain for A might be $\{\text{Green}, \text{Red}, \text{Blue}\}$. A constraint exists between every pair of adjacent variables which states that the pair cannot be assigned the same color. Between adjacent regions A and B, for example, there is a constraint $Color(A) \neq Color(B)$. An additional "node" or local constraint is given that region A must not be assigned the domain value or color Red. A solution to the problem is a set of colors, one for each region, that satisfies the constraints.

In a problem specified in this way, it should be noted that if a constraint relates only two variables then it is called a **binary constraint**. A CSP is binary if all constraints are binary. For any two variables X and Y, one says $X = u$ and $Y = v$ is **consistent** if all binary constraints between X and Y are satisfied by this assignment. The power of constraint-based representations lies partly in the flexibility of a constraint definition. In the binary case (a n-ary constraint scheme is convertible to a binary case), a constraint is defined as a decision function (returning true or false) between domain *values* from the source and target variables. Any computable function may be thus represented as a constraint.

In Chapter 6 I utilize the CSP representation scheme of components and constraints to correspond to program plan components (variables) and known data and control-flow constraints. The domain of each variable ranges over all program statements. In Chapter 8, I utilize the CSP scheme to represent physical program components (variables) and structural constraints among these components. The domain of each variable ranges over all program plans in a known library.

4.3 CSP Solution Approaches

A solution to a CSP is simply an assignment to each variable of a domain value from that variable's domain such that all constraints restricting assignments are satisfied. In some cases we may be interested in only a single satisfying assignment, while in other problem instances all possible satisfying assignments may be of interest. Much research has been done in creating algorithms for solving CSPs. These mechanisms include global [Kondrak and van Beek, 1995] and local search-based methods (see for example [Sosic and Gu, 1990], [Minton *et al.*, 1992], [Yang and Fong, 1992], [Gent and Walsh, 1993], [Gu, 1992], [Selman *et al.*, 1994], [Selman and Kautz, 1993], and [Minton *et al.*, 1990]), constraint-propagation problem simplifications (see for example [Nadel, 1989], [Dechter, 1992], and [Prosser, 1993]), hierarchical (or partial) exploitation of known problem structure (see for example [Freuder and Wallace, 1992], and

[Mackworth *et al.*, 1985]), as well as hybrid combinations of these approaches. Many real-world implementations merge these approaches in various ways.

It is important to note here that a particular problem instance such as the map colouring problem may be represented in many different ways, each a different constraint satisfaction problem with different numbers of variables, domains, and constraints. The particular representation chosen directly affects the structure of the problem search space and consequently the performance of various search methods applied to this space. In [Nadel, 1990], the relative costs and benefits of selecting different representations for the same problem are discussed. In addition the performance trade-offs are determined both experimentally and theoretically.

In terms of program understanding, CSP solution approaches represent methods of identifying both local program component explanations and global interpretations of source code. The methodologies described in this section are domain-independent. They may, however, be easily adapted to make use of domain-dependent knowledge and heuristics.

4.3.1 A naive solution: Generate-and-Test

The most straightforward method to solving a CSP is to generate possible solutions and to test if the constraint set is satisfied for a given instance. In the simple example of Figure 4.2, a simple search is undertaken through the space of all possible assignments of colours to variables, and each assignment found to be consistent with the set of constraints is a solution. Of course, such a method is quite inefficient and will not be adequate for more complex examples involving more variables, larger domain sizes, and more elaborate constraint networks. Specific search methods and heuristics that have been shown to reduce the overall search for certain types of problems will be discussed later in this paper.

This example exhibits constraints between pairs of variables indicating that certain pairings of assignments are not acceptable (*arc constraints*). One could have also added constraints on a particular individual variable or node, that restricted the set of allowable domain values which that particular variable could be assigned (*node constraints*).

In terms of partial local program understanding, generate-and-test would correspond to the very expensive process of permuting all program statements combinations for each program plan component and identifying which combinations were consistent with the expected control and data-flow constraints.

4.3.2 Local Consistency

An alternate approach to solving CSPs is known as constraint propagation and is an attempt to take advantage of the limiting characteristics of constraints among the variables in the CSP. Essentially, any solution of a CSP has the quality that domain assignments to the variables must be *consistent* with one another. Any inconsistency discovered with a domain assignment and a constraint signals that a particular assignment is invalid and may be discarded. An inconsistency discovered as a result of a constraint between two[1] variables indicates that that a particular pairwise assignment is invalid and that pair of assignments may be discarded.

Local consistency methods follow the theme of *preprocessing*. That is, before a more costly method is used, a consistency-based method could be applied to simplify a CSP and remove any obviously incompatible values. Often these methods yield tremendous headway toward eventually solving the problem by reducing the total search space involved with a relatively small, often predictable amount of pre-search effort.

Let X and Y be two variables. If a domain value A of X is **inconsistent** with all

[1] Note that for simplicity I refer to the simple case in which only binary constraints are allowed. In general, constraints may involve many variables.

values of Y, then A cannot be part of a final solution to the CSP. This is because in any final solution S, any assignment to X must satisfy all constraints in the CSP. Since $X = A$ violates at least one constraint in all possible solutions, A can be removed from the domain of X without affecting any solution.

If for a pair of variables (X, Y), for every value of X there is a corresponding **consistent** value of Y, then one says (X, Y) is arc-consistent. By the above argument, enforcing arc-consistency by removing values from variable domains does not affect the final solution. A CSP in which every pair of variables has been made arc-consistent is said to exhibit *arc-consistency*.

Local consistency can be used in local program understanding to reduce the explanation set of locally constrained program plan components. For instance, the knowledge obtained through the identification of one program statement as corresponding to a given plan component could be propagated by following constraints from this component to other related components and appropriately eliminating explanations for those components which fail the constraint in question.

4.3.2.1 Simple or Node Consistency

If for each variable V_k we were to enforce any node constraints pertaining to its domain D_k by removing elements d_i from D_k where d_i violates a node constraint of V_k, then the resulting constraint graph would exhibit *node consistency*. If one or more of the variables were to have a resulting domain which is empty D_k, then the CSP has no possible solution. Algorithms for making a CSP node consistent are straightforward in that each potential domain value d_i for a particular variable V_j must be checked for all node constraints nc applicable to V_j. The worst case complexity of this process can be determined simply by taking the factor of the number of variables n, the largest number of domain values A of any of the n variables, and the largest number of node constraints applicable to any of the n variables. Of course, the actual performance will be considerably better than this since each variable v will have fewer than the maximum number of constraints on average, and fewer than the maximum domain size also. In addition, once a domain value for a particular variable fails one node constraint there is no need to check that value against subsequent node constraints for that variable. This fact suggests a simple heuristic in the ordering of node constraint checking, where those constraints most strict, or most likely to fail are checked first for a particular variable and domain value.

It is known that if a solution can be found via search for the original CSP problem, then one can be found for the problem formulated in a node consistent fashion, CSP_{nodeC}. The CSP_{nodeC} is nothing more than a version of the original CSP with many possible dead ends and their subsequent derivation paths removed from the search space in advance. One would expect the effect on search performance of making a CSP graph node consistent to be quite noticeable as the search space branching factor implied by the set of domain values has been reduced with each removal.

In global program understanding terminology, input and output typing information available about program functional blocks and program plan templates forms the range of node constraints. For example, only those library program plans which agree in terms of input and output typing are possible explanations of a given functional source code block.

4.3.2.2 Arc Consistency

If for each *pair* of variables V_i and V_j one were to enforce all constraints c_{i_j} from V_i to V_j, and remove each domain element d_j of D_j where d_j is not consistent for c_{i_j} with any assignment of d_i from D_i to V_i, then the resulting constraint graph would exhibit *arc consistency.* Once again completeness is retained and any solution reachable in the original CSP can be found in the arc consistent CSP, CSP_{arcC}.

Several algorithms have been published that can convert a general CSP graph into an arc consistent CSP graph. Mackworth's algorithm [Mackworth, 1977] AC-3 is certainly the most well known. AC-3 and other variations of arc consistency algorithms are discussed and presented in a common framework in [Nadel, 1989]. An optimal algorithm, AC-4, for arc-consistency is presented in [Mohr and Henderson, 1986] where the worst case complexity of AC-4 is shown to be $O(ed^2)$ where e is the number of arcs (constraints) in a CSP graph, and d is the size of the largest variable domain in the problem. This algorithm is generalized further into AC-5 in [Van Hentenryck *et al.*, 1992a], where AC-5 can be instantiated to produce an $O(ed)$ algorithm for a number of classes of constraints.

See Appendix A.1 for a discussion of a generalized version of arc consistency, and Appendix A.2 for a discussion of the utility of constraint propagation, and Appendix A.3 for a discussion of algorithms for arc consistency which propagate only a subset of all arc constraints.

4.3.3 Combining Generation and Constraint Propagation

I have so far considered two approaches to solving constraint satisfaction problems. The first, *generate-and-test*, is inherently simplistic, and the size of the search space is clearly going to be enormous in any complex domain. The second, *constraint propagation* can be either an incomplete solution as in the case of those algorithms which insure a degree of consistency short of total n-consistency for a problem of size n, or extremely inefficient, as in the case of total n-consistency algorithms. Several options present themselves for immediate improvement of this situation. First, generate-and-test can be replaced by a complete and more efficient heuristically directed depth first search approach. Second, one can take advantage of the simplification properties of arc consistency algorithms in several different ways. An obvious approach is to use some degree of arc consistency propagation as a preprocessing phase, simplifying the original CSP and thus limiting much of the search required to solve the problem. This preprocessing behaviour is discussed in [Dechter and Meiri, 1989]. I have already mentioned this option, and I will discuss it further through this paper. Still another option is to consider a slightly different *hybridization* of search and constraint propagation which has been suggested and has been the focus of experimental analysis by several authors including [Nadel, 1989].

Consider the nodes of a search tree as each representing a new CSP which is simpler in that one variable from the original CSP has already been instantiated. At each level of the tree, another variable is instantiated as in normal search, but each node is considered independently now, and a consistency algorithm may be first applied to that node before attempting a solution or refinement from that level. Essentially this process interleaves search and constraint propagation so as to limit the branching options further at each search node and the particular CSP problem that node represents. Several search algorithms and variations of search algorithms behave as if there were a particular degree of constraint processing being performed at each node, even if this fact is not stated explicitly. Nadel [Nadel, 1989] attempts to standardize this behaviour in order to determine how much constraint processing is of benefit during search, and consequently developed the notion of *fractional* arc-consistency algorithms which make a problem graph only partly arc-consistent for some particular subset of constraints and domain values.

One could possibly locate a solution (if the variable domains become all size 1 and the CSP is arc-consistent) or prune the node (if a variable domain becomes empty), or generate successor nodes in the search space (for some selected variable one generate all possible assignments to that variable). Each new successor CSP will be progressively "smaller" than its parent since the variable chosen at the succession point will never be selected again, and there is now have one less variable in the CSP.

Experiments reported in [Haralick and Elliott, 1980] and [Nadel, 1989] have indicated that best performance is obtained across a variety of domains when constraint propagation is only applied in a limited form. Results with hybrid algorithms are discussed in the context of partial constraint propagation in [Nadel, 1989], and many other papers give experimental results for particular search strategies that exhibit partial constraint propagation without explicitly stating their behaviour as such. In the next section I will discuss various search approaches often utilized and presented in conjunction with constraint satisfaction problems.

In global program understanding terms, combinations of generation and constraint propagation correspond to hypothesizing particular candidate explanations of given functional program blocks at one time, and making a limited degree of inference based on the relative structural constraint information based on these hypotheses. The amount of inference is modeled by the amount of constraint propagation, while the number of hypotheses is modeled by the number of generations.

4.3.4 Backtrack-based Algorithms

CSPs can be solved using search alone, using consistency propagation alone, with search and using consistency propagation as a preprocessing phase, or by using some hybrid approach interleaving constraint propagation and search.

Arc-consistency algorithms only work on pairs of variables, and as such can only handle binary constraints and cannot always guarantee a final solution to a CSP. A more thorough method for solving a CSP is backtracking, where a depth-first search is performed on a search tree formed by the variables in the CSP. A thorough examination of these techniques can be found in [Nadel, 1989] and [Kumar, 1992]. During a backtracking search, each variable instantiation might be interpreted as extending the partial solution one step further.

A backtracking algorithm instantiates the variables one at a time in a depth-first man-

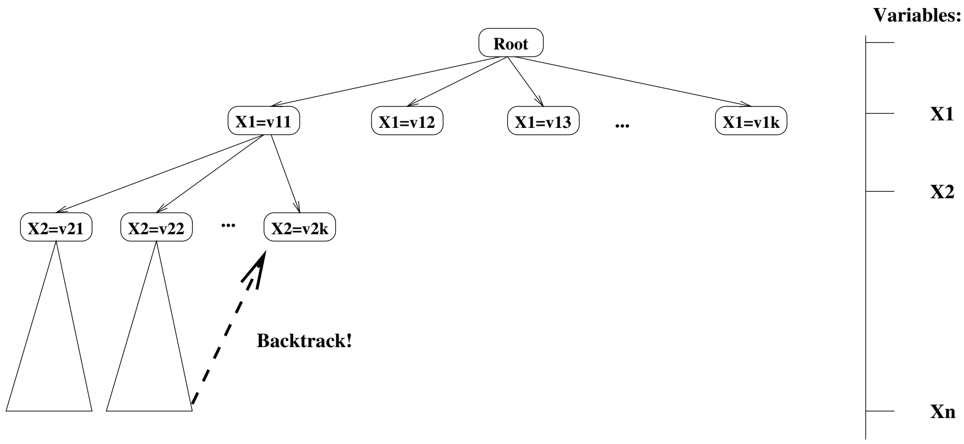


Figure 4.3: A search tree for a backtrack-based algorithm

ner. The algorithm "backtracks" (or re-instantiates a previously instantiated variable, typically the most recent) when the constraints accumulated so far signal inconsistency. In Figure 4.3[2] I show this process. First, variables are ordered in a certain sequence. Different orders of variables might entail different search efficiency, and heuristics for good ordering of variables are called **variable-ordering heuristics**. Similarly, for each variable, the values are tried out one at a time, and the heuristics for a good ordering of values are called **value-ordering heuristics**.

Using the CSP representation, one could also consider a more systematic study of different search algorithms. In Table 4.1 I present a general interpretation of a backtracking algorithm for solving a CSP. In this algorithm, there are a number of hooks where one could place different search heuristics. They correspond to heuristics for ordering variables and constraints, as well as heuristics for deciding the amount of constraint propagation.

There are several choice points which both individually and in combination affect the resulting search performance. These choice points are explained as follows:

[2]Thanks to Qiang Yang for the use of this figure.

Generic CSP Search
V: variables in a CSP, $Dom(X)$: the domain values of X.

1. **[Initialization] for** each variable $X_i \in V$, find the set of domain values for X_i;
2. **[Initial Constraint Propagation]** Reduce $Dom(X)$ by constraint propagation.
3. Solution = NULL
4. **[Variable Selection]** Select and remove a variable X from V
5. **[Value Selection]** Select and remove a value of X from $Dom(X)$.
   The value must be consistent with all assignments in Solution.
6. **[In-search Propagation]** Apply a subset of constraints to V.
7. **[Backtrack Point Selection]** Backtrack if any $Dom(X)$ in V becomes empty.
8. **[Solution Evaluation]** If V is empty, exit with Solution (if all-solution, continue); else, goto Step 4.

Table 4.1: Generic CSP Search Algorithm

1. *Initialization* and *Initial Constraint Propagation* are the determination of variables and domain values before the search starts. It can be viewed as a special type of localized constraint propagation algorithm, but one that is directed according to pre-defined domain knowledge. The determination of the set V and of $Dom(X)$ controls how much work is done in advance. This reduction could also be performed as an in-search propagation at Step 6 of the Generic CSP algorithm.

2. *Constraint Propagation* is the reduction of domains locally or globally within the CSP problem graph via AC algorithms introduced earlier.

3. *Variable Selection* is the determination of which component variable should be chosen next for instantiation during search. The decision may be based on domain independent measures, such as the size of a variable's domain; on information specific to the domain and problem instance, or based on some combination of these types of information.

4. *Domain Value Selection* is the determination of a particular variable domain value

to assign to the current variable. Typically this selection should be made so as to most effectively limit the remaining variable ranges, that is, to be the most context limiting.

5. *In-search Propagation* is the reduction (as for Step 2) of the remaining uninstantiated variable domains according to some constraint propagation algorithm. Problem characteristics such as variable domains that exceed some average or absolute bounds are potential signals that constraint propagation may be useful before continuing search. In [Nadel, 1989] the advantages of exploiting various algorithms for achieving a limited degree of partial consistency amongst variable sets are examined.

6. *BackTrack point selection* is the determination, after it has become evident that no possible solution exists along a particular variable-instantiation path, of which instantiation to retract. Intelligent backtracking approaches such as BackJumping and BackMarking[3] attempt to determine the origin of the conflict that caused the failure, and to BackTrack as far up the search tree as possible to avoid a repeated failure of the same condition.

7. *Solution Evaluation* determines whether or not a particular solution is satisfactory. In a cooperative interactive approach to program understanding, it is at this point that an expert might interact and evaluate a particular partial solution for adequacy. Similarly, if there exist particular measures of adequacy (such as soft or *preferential* constraints that may have been relaxed during search), such measures may be applied here.

In the generic search algorithm a set of choice points are presented in the new context of CSP solving. Variations that I shall examine empirically later in this dissertation

[3]These and other intelligent backtracking algorithms are described in detail by Nadel in [Nadel, 1989].

include applying AC-3 as Step 1 combined with BackTracking, and an intelligent search algorithm known as Forward Checking [Haralick and Elliott, 1980], which performs a limited amount of in-search propagation at Step 6. In addition, the intelligent search algorithm dynamically rearranges the order of variables during search according to the size of the variable domains, selecting the variable with the smallest domain value set first.

The order in which constraints are applied can dramatically affect search performance. Constraint ordering or selection would occur at Step 6. In particular, it is advantageous to apply constraints that are inexpensive computationally and that (potentially) prune a large number of domain values. In a particular domain it may be possible to determine or estimate such relative benefits either from past empirical results or through analysis of the domain structure itself. For instance, in Section 4.3.6.1 it is seen that the property that spatially defined template components features tend to be found *spatially* near each other can be exploited through heuristics that limit the range of search for related components. The effectiveness of such abstraction-inspired heuristics has been reported in [Holte *et al.*, 1995] and [Woods, 1993].

There are, in addition, several other ways to improve search efficiency. One method is to exploit particular hierarchical structure of the domain values using a *hierarchical constraint satisfaction algorithm* [Mackworth *et al.*, 1985]. In such an approach, a value hierarchy represents sets of values at varying levels of abstraction. A set of low-level hierarchical values mapped to a particular variable may be grouped according to their functional similarity and form a single higher-level component. This abstracted component may now be treated as a domain value with regard to constraint application. Should a failure or success be detected using this abstracted value, the children of the abstract value may "inherit" the success or failure. In this way, many values which share a functional relationship can be processed with a single constraint application. I discuss an extended novel approach along these lines in Chapter 8.

In the global program understanding problem, backtracking may be thought of as simply generating successive candidate explanations of functional program blocks which are consistent with previous explanations. If a block (variable) has no consistent explanations, one of the previous explanations must have been wrong and should be retracted.

4.3.5 Hybrids of Backtracking and Propagation

For the purposes of search in CSPs, if one makes the assumption that each successive search layer results in one more variable instantiated, then each node towards a leaf represents a simpler CSP left to solve. So if there are n variables in some CSP, then the search space will be exactly n levels deep. Since all total solutions are necessarily leaf nodes, any *complete* search strategy (such as directed depth first search) identify all possible solution instances. The algorithms often cited in constraint propagation search literature include *Simple Backtracking*, *Forward Checking*, and *Partial Lookahead*, *Full Lookahead*, *Really Full Lookahead* [Haralick and Elliott, 1980]. Kumar [Kumar, 1992] describes Nadel's work [Nadel, 1989] which shows an empirical comparison of the performance of these algorithms in a unified framework constructed by Nadel. In this comparison, these algorithms are described primarily in terms of the degrees of arc consistency which is performed at the nodes of the search tree, essentially what level of hybridization they each represent. The definitions are shown below with a depiction of each algorithm as a combination of pure tree search and algorithms which guarantee at each backtrack node varying fractions of arc consistency as described earlier.

- Generate-and-Test (GT)
- Simple Backtracking (BT = GT + AC 1/5)
- Forward Checking (FC = GT + AC 1/4)

- Partial Lookahead (PL = FC + AC 1/3)
- Full Lookahead (FL = FC + AC 1/2)
- Really Full Lookahead (RFL = FC + AC)

In global program understanding terms, the structural and knowledge constraints are utilized in varying degrees of inference. For example, Forward Checking involves checking of approximately 1/4 of all possible arc constraints between selections of component hypotheses.

I will look at each strategy in more detail in subsequent subsections. I am interested in how each method combines various consistency checking with search, and in later experiments in determining which may be appropriate and effective in a specific domain.

BT: Simple Backtracking

BT incorporates some consistency checking implicitly since whenever a new variable is selected for instantiation, any values which are inconsistent with previous instantiations indicates a successor that may be pruned. This differs from Generate and Test in that GT would only determine consistency or inconsistency at the leaf nodes. All possible combinations would be generated, and then tested.

FC: Backtracking with Forward Checking

FC utilizes consistency checking more than BT by filtering all yet uninstantiated variables for consistency with previously instantiated variables values at successor creation time. In this way, one avoids generating some successor nodes that will fail against previous instantiations.

FCDR: Backtracking with Forward Checking and Dynamic Rearrangement

The Dynamic Rearrangement heuristic is basically an adjustment to any Forward Checking algorithm so that after a particular node is filtered removing any domain values for the remaining variables that are inconsistent with previous instantiated variable assignments, the next variable selected for instantiation is the one with the smallest domain size. This is essentially an additional attempt to limit the branching factor of the tree by selecting the next variable to be instantiated more intelligently.

RFL: Really Full Lookahead

Really Full Lookahead, Full Lookahead, and Partial Lookahead are essentially augmented versions of FC in that they perform additional arc consistency checks even between uninstantiated variables. RFL essentially limits all remaining domains by insuring full arc consistency before selecting the next variable for instantiation using the Dynamic Rearrangement or some other method.

FL: Full Lookahead

Full Lookahead differs from RFL in that it utilizes the partial consistency algorithm "AC 1/2" described by Nadel in [Nadel, 1989]. Essentially, this algorithm is the same as Haralick's "Look_Future" described in [Haralick and Elliott, 1980].

PL: Partial Lookahead

Partial Lookahead, essentially Haralick's "Partial_Look_Future", is described by Nadel in terms achieving a degree of consistency suggested by his algorithm "AC 1/3".

4.3.5.1 Intelligent Backtracking

In previous results including those presented in [Nadel, 1989] and [Haralick and Elliott, 1980], it has been observed that during the constraint application in consistency checking, some constraints are repeatedly checked between pairs of variables despite no possibility of a different result in each subsequent case. Also, portions of the search space that have been discarded in earlier search as a result of arc inconsistency are revisited in later search. This behaviour is referred to in the literature as *thrashing*. Modifications have been suggested by several authors to try to avoid this redundant constraint checking and search. Two methods known as BackJumping and BackMarking are discussed in Appendix A.4. Other more complex and expensive methods are described in [Kumar, 1992], some of which minimize redundant checking even further, but at a much greater computational, space and complexity cost.

4.3.5.2 Heuristic extensions to the search process

In any domain it is possible to exploit domain-specific knowledge during search. For example, in the global understanding domain, if one knew that a particular program plan only occurred in conjunction with another program plan, this is a very strong local constraint. This constraint might be used such that whenever the "key" plan was suggested as an explanation, the related plan must occur as the explanation of one of the components related to the component explained by the "key" plan. If the pair-plan did not exist, it would be fruitless to explore the "key" plan as an explanation further. In this way specific knowledge can be used to reduce search. Each of the following paragraphs describes one such exploitation of domain knowledge during search for coherent global explanations.

Preprocessing with advance consistency checking

It has been seen in earlier sections how CSPs can be solved using various search approaches alone, using consistency algorithms without search, or using a hybrid combination of search and some degree of partial consistency checking during search. A more simplistic approach is possible in which some degree of consistency checking is performed prior to search only, and then some search strategy is performed on the simplified version of the CSP resulting.

Preprocessing with variable ordering

It is known that the order in which variables are selected for instantiation directly determines the shape and structure of the search tree for a particular CSP. Dynamic rearrangement is one such heuristic improvement to Forward Checking search which attempts to control the shape of the search tree by selecting the variable with the smallest domain for next instantiation.

It is also known that if given a particular CSP, one can either attempt to solve it or other simplified versions obtained through advance consistency checking by some search method. Since the search space is determined by the order in which the variables are selected for instantiation, it is conceivable that some heuristic for ordering these exists in a particular domain that may be of benefit. Certainly one option is to simply order the variables from smallest domain size to largest which is similar to the approach taken by Dynamic Rearrangement, except that this would be more accurately labeled Static Advance Rearrangement.

Constraint application ordering

As a consequence of backtracking search, each domain value selected as a variable assignment at some branching point in the search space must be checked to see if it is consistent with each variable assignment made at previously visited search levels. Any constraints between this variable and already instantiated variables must be checked. As with node constraints, a single such arc failure is enough to signal a backtracking point, and consequently it would be advantageous to attempt evaluation of these constraints in an order which would maximize the chances of finding a failure earlier. In other words, satisfying the "tightest" constraints first would seem to be an appropriate heuristic for avoiding unnecessary constraint checks.

Heuristic Application

I have identified several interesting areas for development of appropriate heuristics during search for this domain. These areas include:

1. Propagating only certain types of constraints after variable instantiation. Specifically, in a spatially-oriented domain, one might consider propagating spatial constraints that limit the "field of vision" of remaining variables at little cost. Some form of spatial index might be used to quickly filter variable domain sets, or possibly this filtering could be done via some kind of call to a data parallel machine, passing sets of variables and requesting the same elimination process applied to all sets simultaneously.[4] This type of domain-exploitative strategy has a potentially tremendous search improvement potential through the elimination of many search dead-ends.

2. Dynamic selection of variables to instantiate can dramatically reduce the branching factor of search. Whether or not to undertake this action would depend on understanding better the amount of work that results from more or less constraint checking. This is an open area of research.

[4] This parallelized conception was realized in collaboration with Dr. Guy Vezina of the Defence Research Establishment Valcartier.

3. During search, even in depth-first search, the question of which domain value to attempt to instantiate next (or the order), is an issue of heuristics. Abstraction may be viewed as an attempt to impose a partial order on variable selection during search (see for example, [Woods, 1991]), however, the question of instance selection is not addressed. Conceivably it might be beneficial to choose instances with close "logical" proximity to some other value already assigned. At any rate, some ordering might be imposed on domain instances according to space or other method.

4. As discussed in Appendix A.4.2.1 and A.4.2.2, it may be possible to propagate certain constraint reductions of a problem through the entire search space in limited instances. Determination of what heuristics might identify when this effort is justified is an open area of research.

5. The order in which constraints are applied during constraint propagation or verification of variable instance consistency during search is a matter of heuristic. Selection of expensive constraints which seldom yield a negative response can result in much wasted computation. In some fashion one would want to order the application of these such that if a negative response was anticipated, the most "cost-effective" and "restrictive" constraints would be applied first, perhaps according to some ratio measuring their chance of returning a negative value to cost of application.

4.3.6 Local Search

Local-search methods exemplified by [Minton, 1990] and GSAT [Selman and Kautz, 1993] represent a kind of greedy approach to CSP solution currently enjoying popularity. While I do not examine local search solutions in detail in this dissertation, the later use of CSP as a modeling tool has been structured so that local-search might be eventually adopted.

Selection of an appropriate search strategy can be highly problem dependent. For instance, if one requires all solutions, then some form of complete search will be required, likely with addition of heuristics to shorten the search time. If any single solution will satisfy the requirements in a particular case, then some type of quick but incomplete search may be what is required in order to more efficiently find the single answer. Possibly some combination of these two search strategies will be appropriate in cases where a single solution may be inadequate but several solutions may suffice, or where a particular solution must meet some outside (possibly manual) criteria to determine it's appropriateness. If possible, an interactive search utilizing an algorithm for *ALL* solutions may be appropriate if it can be stopped after a "satisfactory" solution has been detected. In such cases where a complete strategy is used, the order of arrival at various solutions can be accommodated directly into the search strategy in the form of control heuristics for the selection of variables to instantiate, the order of domain values to attempt to assign, and for the control of expanding the search strategy deeper or wider in the quest for a "good enough" solution. A later section of this paper will be dedicated to the discussion of this type of interaction and with the presentation of a particular overall heuristic to add to complete search that facilitates this ongoing interaction in search of appropriate or satisfactory solutions.

One single solution strategy that has shown great promise in various domains is referred to as "Local Search". Local Search works by performing a random assignment of domain values to each of the variables in the CSP, and then by attempting to repair the assignment according to some heuristic function. One common repair heuristic is to minimize the conflicts that arise with a particular assignment using a hill climbing approach. Local Search has been shown in [Minton *et al.*, 1992, Sosic and Gu, 1990] to be orders of magnitude faster than previous methods at finding single solutions for difficult problems involving large variable and domain sets with many constraints. Recent ongoing work in the university course scheduling domain has also been discussed with similar results

[Yang and Fong, 1992].

4.3.6.1 Locality Heuristics in Spatial Problems

In Appendix A.5 I discuss several advantages and methods of decomposing constraint satisfaction problems, and also ways in which one may control or guide the recomposition of these separate solutions with user interaction or/and by heuristics. A primary observation I make is that, given a certain problem decomposition, it may be possible to exploit local problem partial-solutions in domain-specific ways.

One example of such a domain-specific heuristic might be the observation that partial solutions can be directly exploited in the search for more complete solutions. In particular, locality heuristics may be employed to restrict the range of solution completion. For instance, consider the example of the spatial template problem (STR) [Woods, 1993] in which a template is defined in terms of a set of spatially situated objects of varying types. These objects have positions based solely in relationship to spatial distances and orientations with respect to one another. Consider each object (or template slot) as a variable which potentially corresponds to one of many object instances in a particular field of view. Any set of assignments of instances to template components such that the set of spatial constraints is satisfied is a solution.

Spatial locality is an example of an important "simplification" or restriction of the general CSP nature of STR in my formulation. This heuristic is applicable to CSP problems that are grounded in spatial coordinates. Essentially, one may take advantage of the fact that our spatial templates may be "pre-compiled" such that it is possible to encode rough boundary information for an entire solution based upon only partial variable assignment information. For instance, if a template has no two slots farther apart than some measure max, then during search if an assumption is made about the location of some template slot t, then it may be concluded that no subsequent template slot can be assigned a situation element farther than max from the slot t assignment. Given that one may easily determine the distance between situation objects (perhaps through a pre-built index or similar method), it seems easy to dramatically reduce overall search complexity by pruning large numbers of situation elements as candidates for slot assignment "locally".

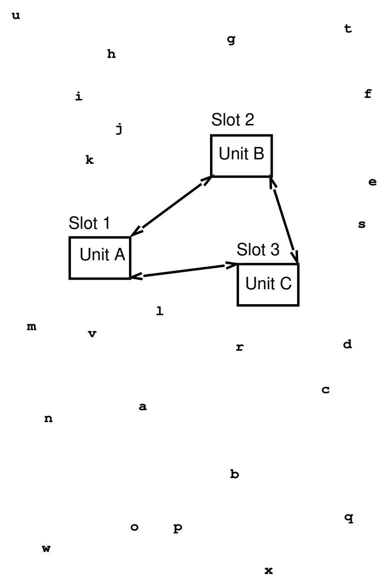


Figure 4.4: SCH Example with level 1 solution

Once assignments have been made to some variable (or set of variables), a boundary is created limiting the scope of this particular solution. Now, when subsequent variables and constraints are added making the problem more specific, the scope of the search is restricted according to the outlined area for each problem. Thus the earlier partial solution has not only been used as a basis for limiting later constraint checking, but also in quickly eliminating some domain values for each of the new-level variable additions. In the example, Figure 4.4 shows a partial solution where some template slots 1, 2 and 3 are matched to situation elements Units A, B and C respectively. Before search continues, the spatial locality heuristic derived from A, B and C assignment positions is applied, limiting the area of search for the remaining slots as shown in the shaded area of Figure 4.5.

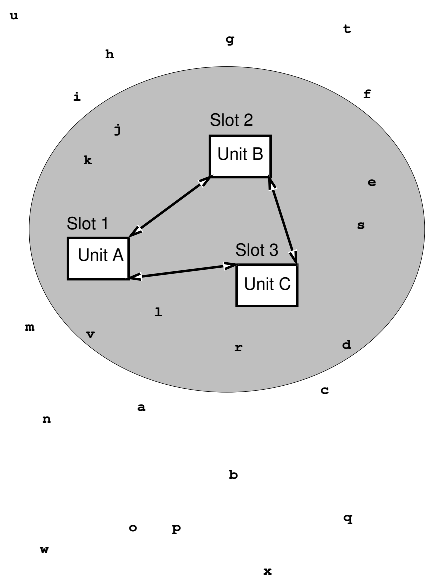


Figure 4.5: SCH Example limiting range of level 2 instances

Consequently the only candidate instances in the selected area for the next level slots are *j, k, v, l, r, d s and e* , while the others outside this area are excluded.

Locality is an important part of restricting the combinatorics of problems of this nature, and need not be limited to purely spatial problems. In Chapter 8 I examine how to exploit partial local solutions as part of the task of reducing global search in program understanding.

In this chapter I have introduced the paradigm of constraint satisfaction as a framework for the representation and solution of problems which can be represented in terms of a set of components and constraints. These components are related with each other in just exactly those ways specified by the set of constraints. A solution to such a problem is a mapping between variables and domain values which may co-designate with those variables such that no constraints are violated. In subsequent chapters I demonstrate how this model may be seen as an obvious cognitive analog of the program understanding problem, and further, how representing understanding as a CSP provides for both a basis for heuristic and empirical comparison of previous program understanding approaches and a new set of ready-made algorithms for program understanding.

Chapter 5

Understanding as Constraint Satisfaction

5.1 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, 1991b, Johnson, 1986]. Although I show in Sections 6.2 and 8.2 that both partial local program understanding and global program understanding are NP-hard, some of these research efforts have presented promising empirical results in mapping plan libraries to reasonably sized (up to 1,000 lines) source code [Wills, 1992, Wills, 1990, Chin and Quilici, 1996]. None, however, have been clearly demonstrated—either analytically or empirically—as scaling up for use in understanding real-world sized software systems. In addition, little work has been done in comparing the relative performance of these approaches or analyzing in detail the performance similarities and differences among these algorithms.[1] Wills [Wills, 1992] has, however, provided an extensive descriptive comparison of PU strategies before 1991.

In part, this situation has resulted because the algorithms tend to be based upon different representational frameworks, such as flowgraphs [Wills, 1992, Wills, 1990], 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]. As a consequence, it is difficult to systematically compare these different approaches or to understand how their performance or effectiveness at understanding programs will be affected by variants in the plan library, such as adding large numbers of new plans, or programs being understood, or 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 the previous chapter I outlined a problem representation and solution technique, the *constraint satisfaction* problem (CSP) technique. In Chapters 6 and 8 I demonstrate 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 one can take

[1] Much of the work in this chapter has appeared previously in work co-authored with Dr Alex Quilici [Woods and Quilici, 1996c]. This collaboration stemmed from a need to accurately represent the work undertaken by Dr Quilici in his memory-based approach to program understanding [Quilici, 1994]. Dr Quilici shares my desire to unify program understanding methodologies as part of the effort to provide heuristic efficiency and scaling as a part of automated understanding tools for software reverse engineers.

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, there is the potential to achieve improved scalability of these approaches by augmenting them with the mechanisms developed for efficient heuristic solving of different classes of constraint satisfaction problems introduced in Chapter 4, including various domain-independent search guiding heuristics in conjunction with partial constraint-propagation during search.

In this chapter I describe how a constraint satisfaction framework can model program understanding, and demonstrate how one well-known heuristic program understanding algorithm can be placed within that framework. In addition, I discuss how this model improves understanding of the behavior and performance of understanding, and how this viewpoint facilitates comparing a specific algorithm's performance with other program understanding approaches. In particular:

- Section 5.2 outlines the two phases of program understanding introduced in Section 2.2.4 in terms of constraint satisfaction representations.
- Section 5.3 is a more detailed look at a representation for program plans and program understanding algorithms. In particular, program understanding is viewed as a composition of local explanations.
- Section 5.4 describes an existing extension to DECODE's 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 5.5 presents a detailed example of my implementation of the extended algorithm in the constraint-satisfaction based framework.
- Section 5.6 provides a comparison in performance between the constraint satisfac-

tion approach and the original approach.

- Section 5.7 summarizes my future research path and the conclusions which have been from this current work.

5.2 CSPs for Two Phases of Program Understanding

Section 2.2.4 introduced the idea of program understanding being represented in two primary phases: (partial) local explanation and (partial) global explanation. Local explanations consist of identified program plan instances in source code, and global explanations are consistent interpretations of how sets of local explanations may be fit together in a particular domain. In this section I briefly discuss how each of these phases can be mapped into a constraint satisfaction framework. In particular I label the local explanation phase as **MAP-CSP**, and the global phase as **PU-CSP**. It is important to note that there are a multiplicity of ways in which local explanations may be combined to give partial global explanations. For example, plan recognition research as discussed in Chapter 3 suggests one set of methods to forming global views from local observations. In addition, the (partially) automated approaches to program understanding through mapping to a knowledge library each take different approaches to this unification. The PU-CSP methodology which is introduced in Section 5.2 and expanded in detail in Chapter 8 forms one such model of integration which accommodates a variable amount of user-guidance during search, and which is capable of propagating information about partial local explanations to limit the range of potential global explanations. There are many possible ways in which source code may be decomposed, and partial explanations might be integrated. Several decomposition and integrative possibilities are introduced as examples below.

PU-CSP

- Decomposition according to spatial locality in which source code is considered in functional blocks typically connected spatially.

- Integration in the PU-CSP framework: integrate MAP-CSP solutions as partial explanations of source code blocks which are directly derived from the source code based primarily on provided functional/procedural separations. Constraint application based upon the knowledge constraints of a hierarchical plan library reduces the combinatorics of a coherent global explanation.

Ordered-PU

- Decomposition and recomposition based solely on the existence of program plan instances.
- Integration follows a methodology in which the hierarchical plan library is structured with the "lowest-level" plans on the bottom of the hierarchy. First one matches all plans with the "lowest" components in the hierarchy and identifies instance parts as belonging to new, abstracted instances of the MAP-CSP program plan. Progressively, one matches all library plans lowest first, matching higher level plans only after all lower level plans are recognized. Constraints are applied at each stage based on structural connection.

Scattered-PU

- Decomposition based loosely on spatial conceptions. These are extended, however, through connectivity such as data and control-flow to enlarge the search range for related local explanations.
- An integration strategy is based on the observation that Ordered-PU is highly inefficient in the sense that very many MAP-CSP applications are required to match all plan instances. Further, the existence of a complete plan library is highly unlikely in general, and Ordered-PU would require such a library to guarantee recognition of higher level plans. Based on identified local explanations, one may select other local explanations that potentially combine with identified instances and search for these "locally" based on possible segments of code which are capable of meeting the connectedness constraints between

the plans in the knowledge library.

Each of these might be regarded as a simple heuristic strategy for program understanding, and each can be modeled as a constraint problem merely as variations on PU-CSP. While PU-CSP represents only one possible integrative model, it must be kept in mind that the entire purpose of modeling with CSP and utilizing CSP solution strategies is to provide a conceptual framework in which other methodologies can be compared. The PU-CSP formulation provides for an interpretive structure in which to view the effect of mapping hierarchically structured source programs into hierarchical program plan libraries. Other approaches at integrating local explanations all exploit the relative constrainedness and connectedness of these locally explained source portions. PU-CSP merely formalizes this conception and suggests that extended CSP algorithms can be adapted to represent search with various constraint propagation techniques. In past approaches frequent mention is made of reducing the space of explanation through the use of constraints, however, CSP provides a way of formalizing this notion.

Global Explanation as PU-CSP

The broader program understanding problem can be represented as a constraint satisfaction problem, called PU-CSP. This problem is represented in the following way.

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 conjunction. The possible explanations correspond to a set of plans[2] in a hierar-

[2] While in this description it is imagined that a single domain value or explanation candidate corresponds to precisely one simple program plan, it is quite reasonable to point out that in fact each program plan is a member of the hierarchical program plan library and consequently has a much different structure than a simple discrete domain value. The conception of domain values is extended to include hierarchical

chically 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). Interesting related work in identifying procedural or functional blocks which possess a particular given input/output typing may be found in the domain of *signature matching* [Zaremski and Wing, 1995b, Zaremski and Wing, 1995a, Zaremski and Wing, 1993]. In Appendix B I discuss related work in which architectural components and constraints are dealt with much as program components and constraints are dealt with here.

Within the CSP framework, constraints are classified 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 plans 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 among 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

domain values later in Chapter 8, however, for descriptive purposes the simpler explanation is retained here.

set of structural relationships present in the source code.

PU-CSP seeks 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 ability to recognize familiar functionality by using these partial explanations when explaining large blocks or chunks of code [von Mayhrhauser and Vans, 1995]. One 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 which is now introduced.

5.2.1 Partial Local Explanation as 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 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 among them, and which are represented as arc constraints in the MAP-CSP. A solution to the MAP-CSP problem is therefore any assignment of domain values (AST elements) to template variables (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 is identified in the source code, and thus explains that part of the source code being mapped. Given a plan library, repeated naive application of MAP-CSP can be used to recognize all instances of plans whose components correspond solely to abstract-syntax tree elements. I expand upon this view of program understanding later in this chapter.

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 among 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 three different program plans in the hierarchy: C1, C2 or C3. The knowledge constraints present in the library for program plans A1 and B1 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.

5.3 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 hierarchically 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.

5.3.1 Concept Recognizer Program Understanding

One way to represent program plans, originally used in the Concept Recognizer [Kozaczynski and Ning, 1994, Kozaczynski *et al.*, 1992] and introduced in Section 2.2.2.3, is as a combination of attributes, components, and constraints.

A concept or plan is defined as a set of common implementation patterns, where each code pattern is a collection of *components* or particular language items or sub-plans that must be recognized to have a potential instance of the plan. Attributes are parameters to the plan components which become instantiated when a plan instance is recognized. *Constraints* are simply inter-component relationships that must be found to be true in order that a particular component set may be said to be a plan instance. Figure 5.1 contains an example of a simple representation of a plan. The language utilized in this figure is derived directly from [Quilici, 1994], and is an intermediate template representation which encodes a small amount of canonicalization of the templates through reference to the data dependencies, control-flow, and variable instantiations determined to be common to instances of the particular template being specified.

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 sub-plans. In particular the components are a `DECL-ARRAY` to declare the character array, a `ZERO` sub-plan to initialize the index variable to zero,

```
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 5.1: An example code pattern

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 among its components, such as a data dependency between the test of the index variable and its initialization. Only if all these constraints hold is there an identified 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 in which the recognized plan may be, in turn, a component. Thus, the Concept Recognizer describes a methodology in which all local explanations are incrementally built upon in order to eventually arrive at all instances of all plans in the program library.

5.3.2 An Initial CSP Framework

How can one place the CONCEPT RECOGNIZER's program understanding approach in a CSP framework? First the MAP-CSP definition must be recalled. As was stated in Chapter 4, 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 arc constraints are satisfied. Also recall that a key aspect of the larger program understanding problem (shared in all approaches studied), introduced in Sections 2.2.4 and elaborated in 5.2, is the subproblem of finding all instances of a particular plan within a given program's internal representation. The CSP version of this problem is known as MAP-CSP.

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 three 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 among them) as inter-variable constraints between plan variables. For instance, in the example plan of Figure 5.1, 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 constrained to logically precede any instance of the variable corresponding to the `LOOP` component.

Figure 5.2 details the variables and constraints of the resulting MAP-CSP for the example plan.

It is important to note that this transformation to a CSP is representational only. Many domain-independent methods such as those described in Chapter 4 exist to solve a given CSP. In order, however, to model a particular domain-specific solution strategy as a constraint satisfaction search algorithm it is necessary to map the specific heuristic as some subset of a CSP solution approach. 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 has been identified.

A single MAP-CSP application corresponds to searching for all the instances of a given plan. While PU-CSP was described as one methodology towards integrating these partial local explanations into a global explanation, there are other possibilities. Consider the following example. It would be possible to make use of only MAP-CSP within a single control strategy. One can find all instances of all plans present in the source by repeated applications of MAP-CSP. In particular, one 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

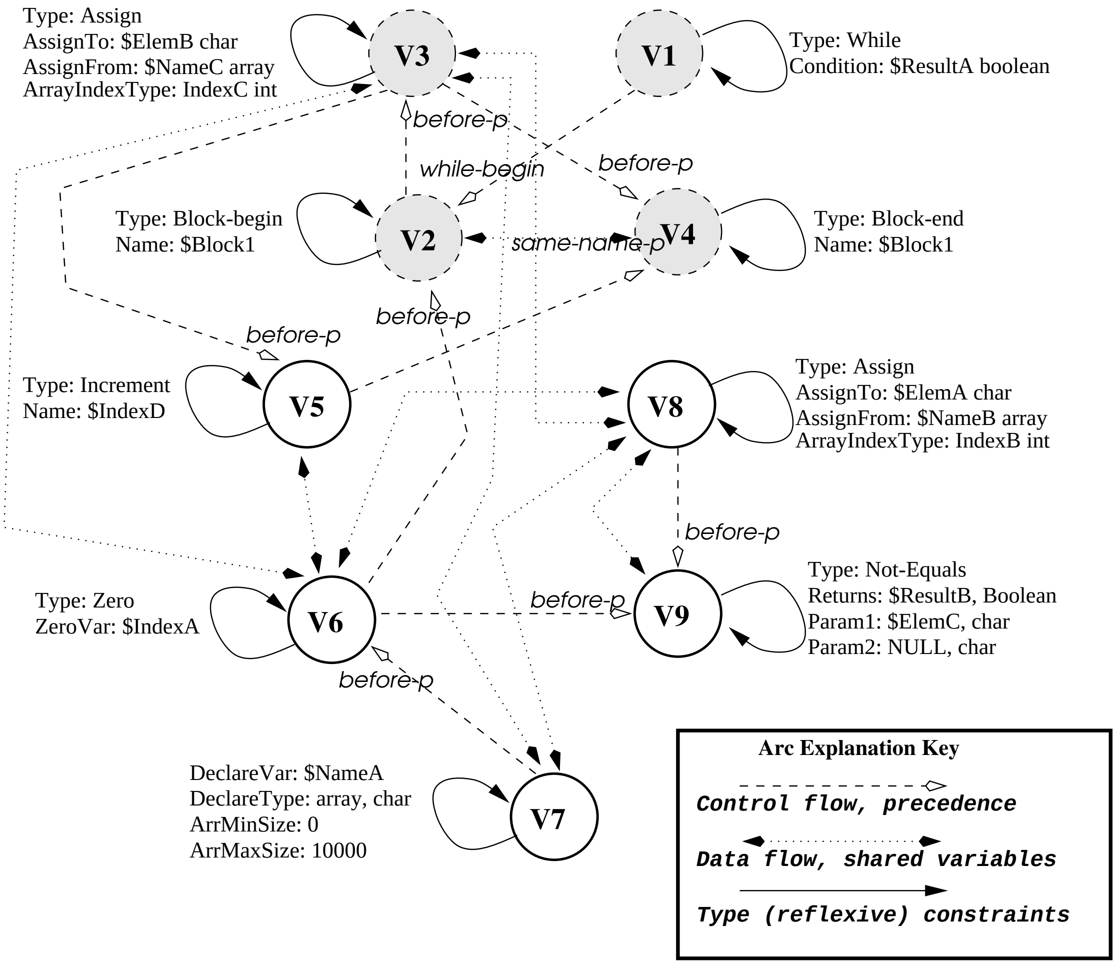


Figure 5.2: MAP-CSP representation of `TRAVERSE-STRING` plan (index shaded)

only on abstract syntax tree items. At the next level are plans, like `TRAVERSE-STRING`, that depend on these sub-plans. One then runs 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. One may 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 one deals with hierarchical plan structure through a layered plan library and applications of MAP-CSP a layer at a time.

The problem with such 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. It is not immediately obvious how the solution of one MAP-CSP can be exploited to reduce the effort made by other MAP-CSPs.[3]

In contrast, if the PU-CSP approach is considered 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 one considers 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 does not correspond to any plan in the existing plan library [Quilici and Chin, 1995]. As a result, the program understanding task corresponds to efficiently partially reverse-engineering the code. In the repeated application of MAP-CSPs, it is difficult to imagine how the programmer can help the process.

[3]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.

However, in the PU-CSP approach, both the algorithm and the programmer can exploit local partial solutions to restrict other, possibly higher-level solutions. Larger code components such as procedures or functions form nicely coupled code chunks with clearly defined constraint relations among them in the form of calling and type relationships. The identification of plans that interact with one of these function blocks can potentially reduce the combinations of explaining a set of these function blocks.

Finally, earlier work with spatial templates [Woods, 1993] has demonstrated that sets of complex constraints, such as those involved in MAP-CSP's plan templates, are very difficult for experts to quickly identify in noisy situations, such as are provided by confusing or cluttered source code. Iterative large-scale understanding of complex spatial situations was greatly assisted by local identification of difficult-to-see spatial relationships. For example, in Figure 5.3 300 spatial objects of four types (plus, diamond, square and cross) are presented in an apparently random manner. A "WarpCross" configuration consists of two plus objects book-ending a diamond object with all three in a nearly straight line, and with the diamond and a square book-ending a cross, also all in a nearly straight line, where the two straight lines cross at between 45 and 90 degrees. If one were told that such an instance occurs in the set of spatial objects (within a set of well-defined tolerances) it is nearly impossible to manually sort through the noise and recognize the instance. However, the solution shown in Figure 5.4 can be found easily using constraint processing techniques similar to MAP-CSP. 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. Applying this idea to program understanding suggests doing as many of these micro-observations (MAP-CSPs) as is computationally affordable, and then attempting to couple instances 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.

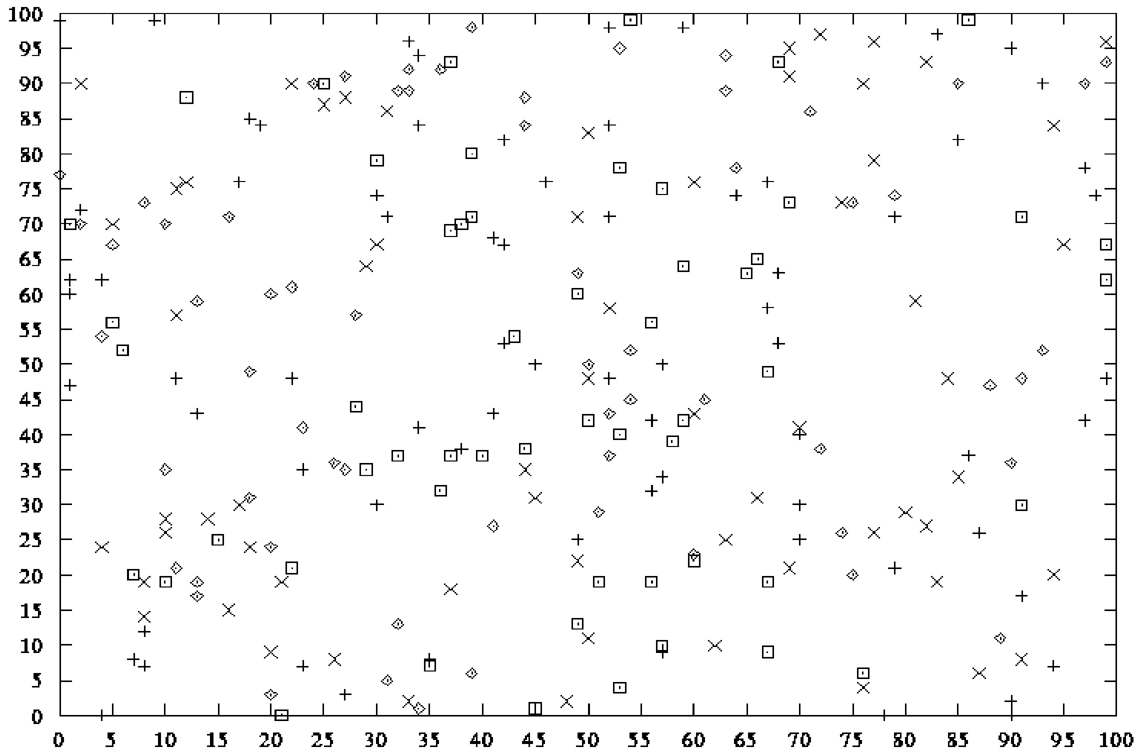


Figure 5.3: Spatial situation with 300 objects of four types.

An 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 structured constraints that exist between high-level plans and source code. However, this is exactly what PU-CSP is meant to do.

In general, there are two primary concerns when trying to model a particular program understanding methodology in a constraint-based framework: representation and control.

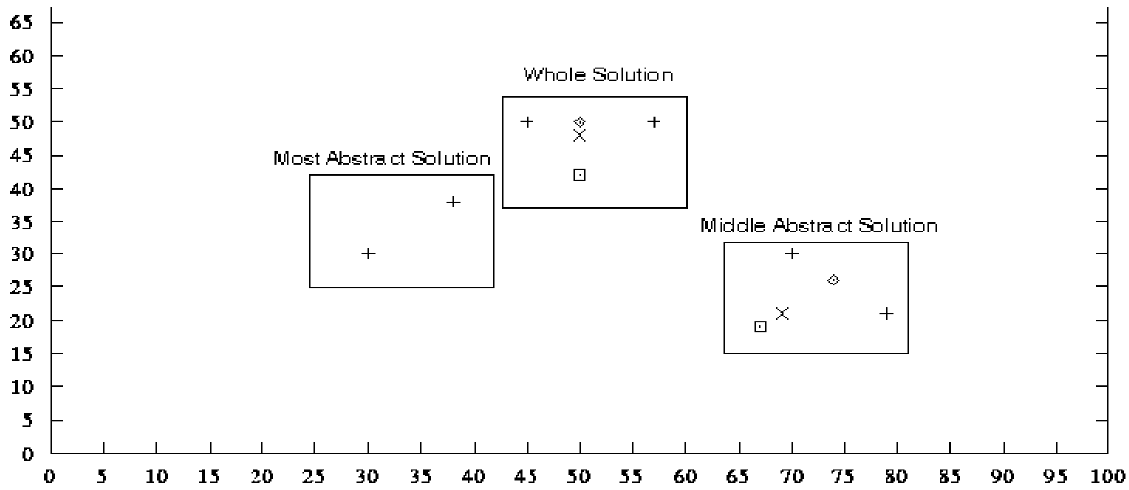


Figure 5.4: One complete "WarpCross" template instance and two partials.

It must be ensured that the CSP representation is general enough to capture the complexities and nuances of the original while not abstracting away important details, and also that the original control strategy can be interpreted in terms of a particular control strategy for solving CSPs.

In the case described, the CSP representation captures exactly the original component and constraint representation of plans. The CSP control strategy is analogous 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 left unspecified in the original Concept Recognizer description.

5.4 Heuristic Program Understanding as a CSP

I 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 with plan library size [Kozaczynski and Ning, 1994].

As introduced in Section 2.2.2.4, later work, in a system called DECODE [Chin and Quilici, 1996], 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. In the following section these heuristic tricks are described and their encoding in a CSP-based framework is presented.

5.4.1 Decode's Heuristic Approach to Program Understanding

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

5.4.1.1 Representation

Figure 5.5 contains several examples of DECODE's extended plan representation.

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, or matched against program pieces and recognized plans. The index combines a

```
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 5.5: An example code pattern

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 `DECL-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 unneeded 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* (inheritances) 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.

5.4.1.2 Control

Figure 5.6[4] is 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 sub-plan that is recognized after the index triggers consideration of `TRAVERSE-STRING`. To avoid this problem, DECODE's algorithm

[4]Thank you to Alex Quilici for the use of this representation of his algorithm.

assumes that the plan library has been organized in layers, just as in the 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 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.

5.4.2 DECODE's Approach to Program Understanding as a CSP

DECODE's approach to program understanding may be captured with several extensions to the CSP-based framework. The additional parts of DECODE's plan representation which must be mapped 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.

One 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. There are therefore 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. The CSP which includes the index behaviour of DECODE is referred to as the *Memory-CSP*.

This separation into two phases is important, as illustrated by the following example. A template is defined with five 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 *independent* 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 three 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. One may view this as a hierarchical view of the CSP in which the "key" portion is "more important" and thus matched first.[5] 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

[5]See [Freuder and Wallace, 1992, Yang and Fong, 1992] for a detailed discussion of representing CSPs as partially ordered sets of variables and constraints.

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.

Implication is handled 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 one processes a plan library layer, MAP-CSPs are also done 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 one runs through each layer of MAP-CSPs, indexing is relied upon to guarantee that the MAP-CSPs in a given layer fail quickly if the indexed component hasn't been recognized from the previous layer.

5.5 An Example of MAP-CSP In Action

I 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 (see also [Zaremski and Wing, 1995b]), 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 5.7 is an example showing how the portion of the plan of Figure 5.5 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 (labeled `ACCESS` in Figure 5.5) as a variable `v3` of a particular type of assignment, `Assign`, for assigning a value to a character array. The complex operation `LOOP` in Figure 5.5 is mapped as a combination of a variable `v1` of type `While`, a variable `v2` of type `Begin`, and a variable `v3` of type `End`. The program plan index constraint that the `Assign` exist inside the control environment of the `While` is represented 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. I discuss a particular approach used for comparison purposes in the next section.

5.6 Some Comparative Experiments

I have run several experiments to compare the performance of DECODE's heuristic program understanding algorithm against various generic 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. The overall approach was to implement a general constraint satisfaction-based framework and then place DECODE's algorithm within that framework as described.

5.6.1 Experimental Description

My prime interest is the performance comparison of different approaches to program understanding in terms of the size of the programs being understood. In particular, my focus is on comparing the amount of effort expended (and consequently, time) in recognizing *all* instances of a single plan template as the source program is increased in size. To keep the focus on scale issues alone, our desire was to have programs of varying sizes available where those programs have the same relative distribution of different program entities (the same percentage of loops, etc...) regardless of size. The test programs used as sources are automatically generated in the following way. An instance (or instances, depending on the experiment) of the program plan template is generated, and program statements are added randomly according to a pre-determined distribution of program statements. This distribution is derived directly from a cross-sectional study of student C programs undertaken by Quilici and described in [Quilici, 1994].

Figure 5.8 shows the internal representation of the 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, I approximate control and data-flow constraints using locality and containment constraints. In addition, I require a `same-name-p` constraint to capture the notion that a variable appearing in multiple places represents the same underlying entity, a notion that is implicit in DECODE's representation for plans. The component set elements consist of a general-component label, a general-component type, and component-identify constraint information (i.e. component q1-c has type "While", signifying that the template requires a loop component whose access is controlled by the boolean value "ResultA").

Given this experimental framework, I generated programs of varying sizes at intervals of 50 lines of code, from 50 to several thousand added lines, with 10 programs generated and tested at each size interval. Based on these 10 data points at each size level, we generate a 95% confidence interval for the number of constraint checks occurring during the search. I show both summary charts which compare the mean of the 10 experiments among many algorithms, or individual experiments which indicate the size of the 95% confidence interval. The working assumption (shared throughout the study of CSP performance) is that the number of constraint checks performed are a reasonable measure of relative work performed. While certain methodologies require slightly differing amounts of computational overhead during search, I have verified that the CPU-second graphs of these same experimental results yields comparable graphs. For comparison purposes, examples requiring approximately 2,500 constraint checks utilize roughly six CPU seconds. Since CPU usage is highly variable across implementations and platforms, constraint checks offer a more system-independent reference point.

5.6.2 Methodologies Tested

The primary goal has been is to model DECODE's heuristic approach (Memory-CSP) as a particular type of constraint-satisfaction problem and compare its performance to various techniques for solving constraint satisfaction problems. In particular, I 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, I also generated naive solutions using simple backtracking for as many problem instances as practicable.

5.6.2.1 MAP-CSP

I 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) which was introduced in Section 4.3.5. 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 the 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.

I 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.

5.6.2.2 Memory-CSP

As was shown earlier, DECODE's approach to program understanding can be modeled by a variant of MAP-CSP with two phases: the index phase and the resolution phase. My 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 5.9 shows the index for our earlier example.

Since Memory-CSP has two distinct phases, it is necessary to select search heuristics and parameters for each of the phases. I 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.

5.6.3 DECODE and CONCEPT RECOGNIZER Experimental Results and Discussion

There were 5 specific algorithms tested in this series of experimental runs. Each of these runs makes use of a single program template instantiation, and the algorithms are run until all instances are found - in fact, no other instances satisfying the constraints exist. Figure 5.10 shows the results of these tests. In particular, this figure shows a performance for FCDR in source examples up to 1,000 lines of code which appears linear. In fact, the Memory-CSP approach appears relatively stable over this size of problem as well. In later experiments detailed in Chapter 7 I extend these preliminary results over larger samples. The distribution of program statements (derived from the Quilici study) used is shown in Table 5.1. As well, when a variable was to be generated, it was generated with the following type distribution: array type (1/7), simple int (2/7), char (2/7), real (1/7), and boolean (1/7). If an array was generated, it was instantiated according to this type distribution: int (2/6), char (2/6), real (1/6) and boolean (1/6). In this experiment, I allowed that certain of the program examples would have no complete instance. Partial instances, however, exist. In particular, for problems of size 50-550, 1 instance is inserted for each of the 10 cases. For problems of 600 and greater, the number of generated programs out of 10 *without* an instance are as follows (by size index): 600, 650 (2), 700, 750, 800 (3), 850, 900 (5), 950, 1000 (8).

Statement Type	Frequency	Percentage
While	1/22	4.5
Zero	1/22	4.5
For	1/22	4.5
Block	2/22	9.0
Increment	2/22	9.0
Not-Equals	2/22	9.0
Print	2/22	9.0
Assign	3/22	13.5
Decl	4/22	18.0
Check	4/22	18.0

Table 5.1: Program statement type distribution

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. All graphed instances completed inside this time restriction.

5.6.4 Summary of Results and Analysis

The bottom-line of these studies is that *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 is, by its very nature, static—the assumption is that an expert can look at a 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 problem instance being examined.

However, there are some artifacts of my particular experiments that need to be explored further before the results are validated. One is that the experiments used locality to approximate data and control-flow constraints, and the 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 the artificial programs. I discuss future work and improvements of these experiments in Section 10.3.

5.7 Conclusions

This chapter has demonstrated how it is possible 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, one can adapt previously developed constraint propagation and search algorithms to improve their efficiency and lead to program understanding algorithms that scale.

The performance results presented here 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 propagation methods.

The study of program understanding algorithms in terms of this constraint satisfaction-based approach appears promising as a unifying framework for describing and comparing program understanding algorithms. My hope is that others 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 well enough to be demonstratably applicable in a reverse engineering toolset.

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

```
(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 5.7: MAP-CSP representation of code patterns

```
'( "quilici-t1"

   ( `` Component Set ``

     (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)))   )

   ( ``Constraint Set''

     (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 5.8: CSP-based internal representation for plans

```
'( "quilici-t1-index"

   ( ''Component Set''

     (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)))   )

   ( ''Constraint Set''

     (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 5.9: The representation for a plan index

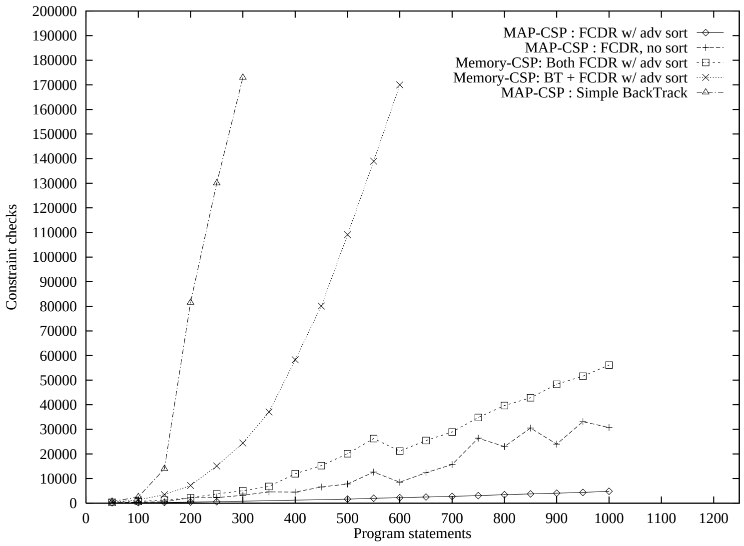


Figure 5.10: The median results for each of 5 algorithms

Part III

Partial Local Explanation

Chapter 6

Partial Local Explanations (MAP-CSP)

In Section 5.2 I introduced the conception of global program understanding (in CSP terms, the PU-CSP), which resolves integration of partial local explanations of source code blocks. The identification of these partial local explanations I represent as the MAP-CSP problem described in Section 5.2.1. I have described MAP-CSP as integral to the more ambitious understanding task. In Section 5.3.2 I have described specific instances of MAP-CSP in the context of modeling earlier program understanding approaches. In this chapter I examine the complexity of the MAP-CSP analytically. Keeping in mind that the MAP-CSP is a sub-problem of the larger understanding problem I have described, the question is — what is the relative cost of applying this algorithm in general?

In general, reverse engineers are interested in very large computer programs, programs with hundreds of thousands or even millions of lines. Just as experts limit their focus to various functionally or spatially restricted sections of code at one time, the work in this thesis is directed towards describing a model which is capable of dealing with both the size and complexity of such code through careful use or interleaving of narrow and broad focus on the source at hand. Thus, while it may be overwhelming to consider a million or even a hundred thousand lines of code looking for semi-global, distributed, or even partial local explanations, it may be possible to deal empirically with foci of tens of thousands of lines initially. Subsequently, one may exploit the partial local knowledge gained "bottom-up" to reduce the potential explanation space of those code segments on a global scale and thus defeat the size barrier through careful narrowing and widening of focus.

The empirical challenge is consequently to show applicability in ever larger narrow focuses. In particular, the ability to scan large segments of code at once increases the chance that a plan instance spread across several functional blocks may be recognized. While it is likely that a simple algorithm for constructing template to instance mappings is an intractable problem, it is nonetheless conceivable that the task of locating partial local explanations could be achieved through the discovery of a polynomial algorithm of some kind. The convenient bounding that such an algorithm would provide could be very useful in a larger understanding context. For example, it would be possible to quickly estimate how much time a particular query would require in advance. Unfortunately not all problems have polynomial solutions. In this chapter, I show that the program template matching approach to partial local explanation construction in fact does not have such a solution. The consequence of this result is that one is well-justified in pursuing heuristic approximations or local partial solutions in ways that may not be polynomial in some worst case. Of more concern then will be discussions about applicability and effectiveness in particular problem instances.

6.1 Program Template Recognition Model

The MAP-CSP or program template matching problem can be stated as follows: given a plan template with a number of elements and constraints among the elements, find all instances of the template (or idiom) in a source code. For example, consider finding all instances of an abstract data type (ADT) in a C program. It is considered that an ADT is a collection of closely related program plans centered about a particular data structure or set of structures. Figure 6.1 is a **String** ADT plan template modeled after those present in a plan library described in the literature [Quilici, 1994]. The ADT is described in terms of five features describing various key components of a string class. In addition, there are constraints among the different parts as well, such as the one that requires one component to go before another.

I model this problem as a CSP as follows. For the given plan template (or ADT), each feature is a variable in the MAP-CSP. The *domain range* consists of all possible source program statements. Variables here can have attributes such as (**print**, **for**) that may be seen as *constraints* on allowable assignment of program statements (values) to template features (variables). Other *constraints* are on the sharing of information among variables, and on the order in which template features or variable are expected to appear in source code.

A solution to the MAP-CSP consists of the set of all assignments of plan template features by source code statements, where each assignment must satisfy all constraints. As an example, consider the ADT of Table 2.1. When represented as a plan template as in Figure 6.1, the variables of the MAP-CSP are: $X_i, i = 1, \ldots, 5$. Initially the domain for each variable ranges through all source statements in Figure 2.3. The constraints are as shown in the figure. The solution to this problem corresponds to the three alternative consistent assignments to the variables, one for each character string A, B and C, respec-

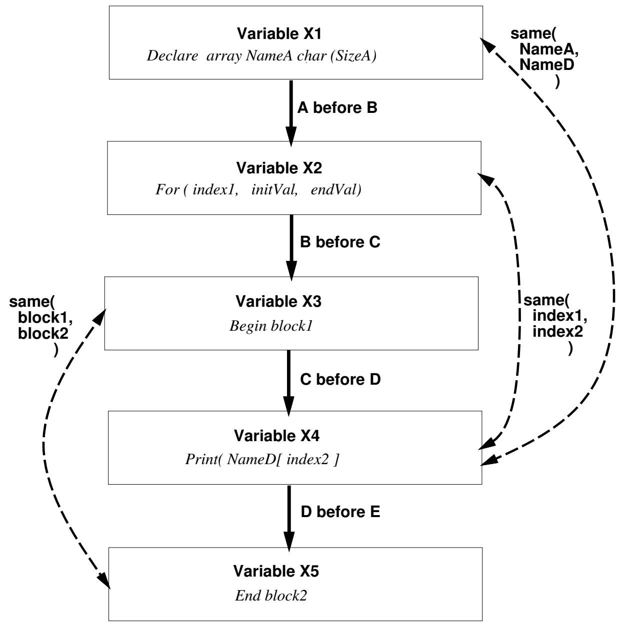


Figure 6.1: The **String** ADT in MAP-CSP

tively. Thus, the solution to a MAP-CSP provides a mapping that *explains* the matched source statements as parts of an instance of the abstract program plan or ADT.

6.2 Complexity Issues

6.2.0.1 Program Template Matching is NP-hard

As stated earlier, many recognition approaches attempt to recognize typical program plans or clichés, and then integrate these instances into a coherent or consistent global understanding. In Section 8.2 I show that the simple program understanding problem is, in fact, NP-hard. Now, what about the seemingly much simpler potential sub-problem of finding instances of a given pattern in a program source code? In this section, I establish that even this "simpler" problem is NP-hard.

6.2.0.2 The Program Template Matching Problem

The **Simple Program Template Matching Problem (SMAP)** problem is depicted in Figure 6.2. Consider that we are given the following:

- There exists a collection B of program blocks $B_i, i = 1, 2, \ldots, m$. These blocks can be viewed in terms of a corresponding graph $P = (B, D)$ where D is the set of edges of the graph $D_k, k = 1, 2, \ldots, n$, such that an edge $D_{i,j}$ exists between node B_i and B_j if and only if a data-flow exists between the program blocks.

- We are also given a program template plan $T = (N, C)$ where each member N_i of N participates in data-flow relationships with some other nodes in N. These relationships are specified by C, the set of edges between nodes N, such that $C_{k,l} \in C$ exists between nodes N_k and N_l if and only if a data-flow exists between the two.

Given the above structure, the SMAP problem is to determine if a mapping exists from the template nodes N to a subset of program blocks in B. The mapping is a function between relationships among template nodes and data-flows among program blocks. As an example, in Figure 6.2, the following correspondence gives rise to a matching instance of the program template:

$$\begin{aligned} N_1 &\Longleftrightarrow B_2 \\ N_2 &\Longleftrightarrow B_4 \\ N_3 &\Longleftrightarrow B_1 \\ N_4 &\Longleftrightarrow B_3 \end{aligned}$$

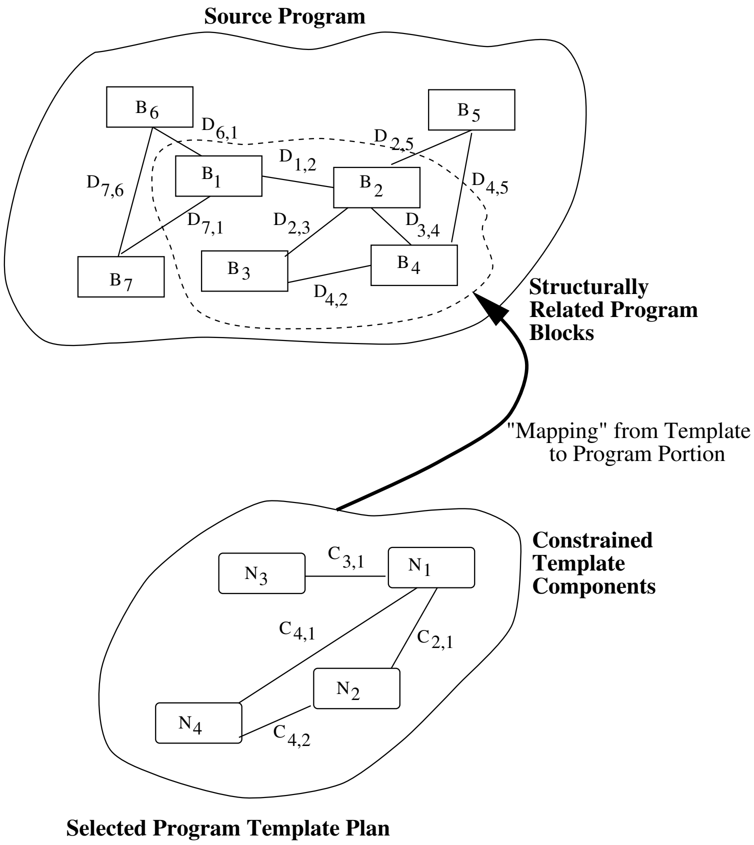


Figure 6.2: Program Template Matching

6.2.0.3 Program Template Matching is NP-hard

One can prove that SMAP is NP-hard by a reduction from the NP-hard problem **Subgraph Isomorphism**, which is known to be NP-hard [Garey and Johnson, 1979, p. 202]. The Subgraph Isomorphism problem may be stated as follows:

> Given a graph $G = (V_1, E_1)$ and a graph $H = (V_2, E_2)$, Does G contain a subgraph isomorphic to H, i.e., a subset $V \subseteq V_1$ and a subset $E \subseteq E_1$ such that $|V| = |V_2|$, $|E| = |E_2|$, and there exists a one-to-one function $f : V_2 \rightarrow V$ satisfying $\{u, v\} \in E_2$ if and only if $\{f(u), f(v)\} \in E$?

The transformation to an SMAP problem can be done as follows. Every vertex of V_1 in G is a program block, and every edge of E_1 in G signifies a data-flow between blocks. Each vertex of V_2 in H is a program template and each edge of E_2 in H is a data-flow between templates. A mapping between a template T and a subset of program blocks in a program P exists if and only if H is a an isomorphic subgraph of G.

6.2.1 MAP-CSP and Search

So partial local explanation as embodied by MAP-CSP is NP-hard. It is now tempting to ask "but does it matter ?". The answer of course is that if it is possible to integrate MAP-CSP as a part of PU-CSP in such a way that the combinatorics that lead to the abstract NP-hard result can be controlled in practice, then no, it does not matter.

In the worst case we are faced with a situation in which, for a fixed template of N components, and M situation elements (source code statements) that possibly match a given component, that there are M^N possible combinations of components and elements. In practice of course it is not necessary to check all of these! Consequently, one may say that for a fixed template size, the solution to this problem is no worse than polynomial of degree equal to the number of components in the given template. In general, however, for a template of arbitrary size, the problem is exponential. The good news is that as M, the size of the source fragment increases, the problem does not become polynomially worse.

The expression of this problem as a CSP serves the goal of demonstrating this heuristic adequateness in the simple way that many relatively large CSPs can in fact be solved. Local search strategies such as GSAT [Selman and Kautz, 1993] have been shown effective for very large problems of particular structure. MAP-CSP offers only a set of partial local explanations of source. It is conceivable then that a different, less powerful local strategy that identified fewer local explanations, but with less effort, might still be of use. Since the PU-CSP which is discussed more fully in Chapter 8 is designed entirely to exploit such local knowledge it is quite conceivable that these few local solutions can be exploited to great advantage in practice. This question of local search applicability is to be the focus of future investigation.

A complete search using the MAP-CSP representation has one definite advantage over a local search. Complete search allows one to make negative claims about the existence of a solution to a given CSP. For example, one can state categorically that no solution exists such that the full set of constraints could have been satisfied. Of course, it is possible that a solution existed with just one constraint failing. At any rate, this negative information can be used effectively in constraining a set of options, some of which may have required a positive solution. We discuss the use of such information directly in Chapter 8.

Chapter 7

MAP-CSP Experimental Results

In Section 5.6.2 of Chapter 5 we briefly described and outlined some experimental results obtained though implementing MAP-CSP and modeling several search heuristics in this paradigm. In particular, in Figure 5.10, we detailed a summary of several search strategies for MAP-CSP including the Memory-CSP approach suggested by Quilici's indexing in DECODE, as well as several domain-independent strategies such as forward checking with dynamic re-arrangement.

In this chapter a wider range of experimental results are described more completely, providing more detail and offering further comparison and analysis. Over the past decade, researchers in program understanding have formulated a wide variety of program understanding algorithms. Unfortunately, however, there have been few, if any, published studies on the scalability of program understanding algorithms. This makes it difficult to understand the limitations of these algorithms and to determine whether the field of program understanding is making progress.

7.1 Source Data and Program Plans

Before examining experimental results from MAP-CSP for recognizing particular instances of program plans, I shall first describe precisely the nature of the generated source programs examined, and also the program plan templates considered.

While this ongoing research effort is directed towards the eventual demonstration of feasibility of both PU-CSP and MAP-CSP techniques in the domain of large commercial source libraries and software sources, that is not the focus of the experiments outlined in this chapter. Rather, I am interested in determining whether, with a minimum of *structural* constraint information, it is possible to utilize the CSP algorithm for MAP-CSP template recognition in reasonably size source programs. For example, if the combinatorics of recognition stopped at 100 lines of code it would be necessary to re-consider this approach. However, if it is possible to scale to code on the order of a thousand lines, it is conceivable that MAP-CSP may be seen as a model or prototype of an integral sub-component of a future understanding toolset. In the future, such a model can be extended to take advantage of further structural source constraint information.

7.1.1 Program Plan Templates

In Section 5.6.1 the **quilici-t1** program plan template is used for purposes of comparison. This template is derived directly from the `TRAVERSE-STRING` program plan utilized by Quilici [Quilici, 1994], and which are described in Section 5.4.1.1 of Chapter 5. The MAP-CSP template version is composed of nine primary components and 20 constraints. The index version of this template (**quilici-t1-index**) is extrapolated from Quilici's indexing strategy (described in Chapter 5), is shown in Figure 5.9. This index-plan is representative of a smaller program plan (in that it is a connected plan in its own right). The index plan contains four primary components and five constraints among them. An extended version of the quilici-t1 template (**quilici-t1-large**) has been created in order to evaluate the empirical results of searching for an instance of a more complex program plan. This extension represents a `TRAVERSE-PRINT` program plan from the Quilici domain, and is shown in the MAP-CSP representation in Figure 7.1. The `TRAVERSE-PRINT` program plan consists of 14 primary components and 29 constraints.

My experiments are concerned with all-instance template identification in large generated source instances. One primary concern is how the relative size of a program plan template (in terms of components and constraints) will affect the empirical performance of MAP-CSP. Another is how the relative size of a given source affects recognition performance.

7.1.2 Generated Examples

I have described three particular program plan templates that we shall be utilizing the MAP-CSP algorithm to search for in generated source code examples. Before describing the process of source generation that we adopt, an obvious question is: "what does an instance of these program plans look like?". In answer to this question, Figures 7.2, 7.3 and 7.4 present sample program fragment instances of the **quilici-t1-index**, **quilici-t1** and **quilici-t1-large** plans.

As I introduced in Sections 5.6.1 and 5.6.3, source examples are generated according to particular statistical distributions of program statements. In particular, the experiments of Chapter 5 were based upon a distribution shown in Table 5.1 provided by Quilici following a study of student programmers and their programs. In addition to this distribution, one must also be interested in the effect on MAP-CSP of notably different distributions of program statements. In particular, while I refer to the Quilici distribution as "Standard", I also make use of "Equal"(see Table 7.1), and "Skewed"(see Table 7.2). The "Equal" distribution generates variables of the varying types with equal probability, and similarly with specific array types. The "Skewed" distribution generates variables with the same type distribution as the "Standard" distribution: array type (1/7), simple int (2/7), char (2/7), real (1/7), and boolean (1/7). If an array was generated, it was instantiated according to this type distribution: int (2/6), char (2/6), real (1/6) and boolean (1/6).

Statement Type	Frequency	Percentage
While	1/10	0.1
Zero	1/10	0.1
For	1/10	0.1
Block	1/10	0.1
Increment	1/10	0.1
Not-Equals	1/10	0.1
Print	1/10	0.1
Assign	1/10	0.1
Decl	1/10	0.1
Check	1/10	0.1

Table 7.1: **Equal** program statement type distribution

Statement Type	Frequency	Percentage
While	1/75	0.013
Zero	1/75	0.013
For	1/75	0.013
Block	2/75	0.026
Increment	2/75	0.026
Not-Equals	2/75	0.026
Print	2/75	0.026
Assign	30/75	0.4
Decl	30/75	0.4
Check	4/75	0.0520

Table 7.2: **Skewed** program statement type distribution

7.1.3 Problem Instances

As discussed in Section 5.6.1, experiments with a given search strategy are performed based on the results of 10 MAP-CSP problem instances at each source code sample size. These 10 problem instances are generated according to the particular distribution in use. If a particular distribution is not discussed in a given experiment, the distribution used is assumed to be the Quilici "Standard". Problem instances are created as follows: a particular program plan instance is generated from the template in question, including an assignment of line numbers for the instance according to the separation specified in the template. Source code statements are now generated according to the given distribution until a source program of appropriate size is generated. The statements are given line numbers randomly within the range from zero to the maximum line number specified in the template instance plus one hundred lines. Certain statement types (such as **Loop** with a corresponding **Begin** and **End**) require more than a single line in their generation. If a conflict occurs in which a new generated line number is already in use, a simple stepping algorithm selects the next available line number. If this algorithm hits the end of the allowed line range, the range is extended by one hundred additional lines.

As an example, utilizing the "Standard" distribution, a generated "program" containing one instance of the **quilici-t1** program plan together with 10 generated source statements is given in Figure 7.5. The template-related components may be identified in this (and subsequent) figures through the statement labels prefixed with "ADT". The initial template instance has nine related component lines, and the remaining 12 added lines arise as a result of the insertion of a for-loop statement with three associated lines. Experiments of a particular size are generated at intervals of 50 source code lines typically (although not in all cases). In such a case, the 10 examples at (say) size 250 would be graphed according to the average size of the 10 examples keeping in mind that each example has a slight variation depending on how many multiple-line insertions are made.

7.1.4 Experimental Results

In this section I present a range of experiments which are intended to show the feasibility of the MAP-CSP representation and related solution algorithms in relatively large (several thousand lines) problem instances.

The experimental results depicted here are based upon models of constraint satisfaction described earlier in this thesis. The solution algorithms referenced in the following figures include combinations of the following algorithms:

1. Standard BackTracking (BT, see Section 4.3.5).

2. Arc consistency propagation (AC-3, see Section 4.3.2.2).

3. Forward Checking with Dynamic Rearrangement (FCDR, see Section 4.3.5).

This list is not intended to be a complete set of solution strategies to constraint satisfaction strategies. Rather, these approaches represent a range of strategies that together are capable of capturing an initial subset of the heuristic strategies undertaken by previous program understanding researchers. Chapter 4 provides a more comprehensive overview of these strategies.

7.1.4.1 Detailed Individual Results

Single template instances

The following examples contain a single template instance in each generated source example. All of these examples are generated using the "Standard" Quilici distribution and make reference to identifying instances of the **quilici-t1** program plan template. The results are graphed showing a 95% confidence interval over the 10 sampled sources. All of these experimental instances were generated such that the inserted template was not destroyed, that is, the template was identified successfully in each case. In addition, for these examples no false solutions were identified. At the end of these *complete* searches, one may conclude that no other instance possibly exists that satisfies the template constraint set.

1. *Simple Backtracking* with no advance variable order, Figure 7.6. The experiment was terminated for source examples exceeding 400 lines. In fact, several individual cases failed to complete a total search of the given example in less than 20 cpu minutes, an arbitrary boundary. In particular, at 250 there was 1 failure, at 300 (1), at 350 (2) and at 400 (3).

2. *Simple Backtracking* with advance variable ordering, Figure 7.7. This experiment shows a rapidly increasing number of constraint checks as source example size increases. In particular, 25,000 constraints are checked with an example size of approximately 725 source lines. Simple backtracking displays an extremely large variance between examples at the same source size. For instance, at 200 source insertions, one of the 10 cases required more than 500,000 constraint checks to solve, while another at the same size required only 3,400. For purposes of comparison between constraint checks with CPU time, Figure 7.8 details this same experiment with regard to CPU seconds consumed on a shared SPARCserver 1,000 running Allegro Common Lisp.

3. *Quilici's Memory-CSP* with FCDR at each level of the problem, including advance variable ordering at each stage, Figure 7.12. This experiment shows 25,000 constraint checks taken for approximately 1,000 source lines. An interesting, unexpected and currently unexplained result for this experiment is that one of the 10 source examples proves increasingly difficult to solve for source examples of size up

to and including 550 source insertions, and then becomes easier at approximately 600 insertions, once again increasing in difficulty from 600 upwards. This test case is notably more difficult (i.e. by a factor of 3 to 5) than others at levels of 400 to 550 source insertions, and consequently affects the confidence interval of these instances significantly. After 600 insertions, this example problem returns to a difficulty similar to its fellows. This behaviour is not noticed in other strategies for this example.

4. *AC-3 with Forward Checking and Dynamic Rearrangement* with advance variable ordering, Figure 7.11. For this experiment, the 25,000 constraint check limit is surpassed at approximately 1,200 source lines.

5. *Forward Checking and Dynamic Rearrangement* with advance variable ordering, Figure 7.9. This standard CSP solution strategy has typically performed well on a wide range of problems. The FCDR strategy essentially propagates constraints between each search assignment to a depth of one "look-ahead" variable. This experiment results in the checking of only 5274 constraints at 1,500 source lines. A similar example for FCDR shown in Figure 7.20 shows a result of 25,000 constraint checks for an example source size of more than 3,000 source lines. For purposes of comparison between constraint checks with CPU time, Figure 7.10 details this same experiment with regard to CPU seconds consumed on a shared SPARCserver 1000 running Allegro Common Lisp.

6. Figure 7.13 graphs the median of the previous examples in a unified chart to show the relative performances. Note one line is extended for FCDR with advance variable ordering (2 solutions) so as to demonstrate the relative rate of increase for that heuristic. Figure 7.20 indicates how this particular heuristic scales in much larger examples.

7. Figures 7.14 and 7.15 demonstrate the relative utility of estimating effort through constraint checks as opposed to CPU time. In particular, I chart time on the X axis and constraint checks on the Y axis. Notice that initial overhead matters only as a constant factor. The results are taken from the experiments utilizing BT and FCDR (advance ordering) with the Standard template and Standard code distribution.

```
'( "quilici-t1-large"

   ( ``Component Set''

    (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-a2 Decl      (NameA2 (char)))
    (q1-a3 Decl      (NameA3 (char)))
    (q1-a4 Decl      (NameA4 (char)))
    (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)))
    (q1-p1 Print     (NameP1  (char)))
    (q1-p2 Print     (Newline (char)))  )

   ( ``Constraint Set''

    (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-g))
    (before-p    (q1-b q1-c))
    (before-p    (q1-d q1-f))
    (before-p    (q1-d q1-i))
    (before-p    (q1-a2 q1-g))
    (before-p    (q1-a3 q1-f))
    (before-p    (q1-g  q1-p1))
    (before-p    (q1-p1 q1-e))
    (before-p    (q1-i q1-e))
    (before-p    (q1-f q1-h))
    (before-p    (q1-h q1-e))
    (before-p    (q1-a3 q1-p1))
    (before-p    (q1-a4 q1-p2))

    (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))

    (same-name-p (q1-a2 q1-g) (NameA2 ElemB))
    (same-name-p (q1-a3 q1-f) (NameA3 ElemA))
    (same-name-p (q1-p1 q1-g) (NameP1 ElemB))
    (same-name-p (q1-a4 q1-p2) (NameA4 Newline))          ))
```

Figure 7.1: Extended program plan **quilici-large**.

```
Stmt Id  Line  Statement

(ADTQ1-C  300 (WHILE RESULT))
(ADTQ1-D  310 (BEGIN BLOCK1))
(ADTQ1-G  400 (ASSIGN A IDX ELEMB))
(ADTQ1-E  600 (END BLOCK1))
```

Figure 7.2: Instance of **quilici-t1-index** plan.

```
Stmt Id  Line  Statement

(ADTQ1-A  100 (DECL ARRAY A CHAR 99))
(ADTQ1-B  200 (ZERO IDX))
(ADTQ1-C  300 (WHILE RESULT))
(ADTQ1-D  310 (BEGIN BLOCK1))
(ADTQ1-G  400 (ASSIGN A IDX ELEMB))
(ADTQ1-I  500 (INCREMENT IDX))
(ADTQ1-E  600 (END BLOCK1))
(ADTQ1-F  700 (ASSIGN A IDX ELEM))
(ADTQ1-H  800 (NOT-EQUALS ELEM NULL RESULT))
```

Figure 7.3: Instance of **quilici-t1** plan.

```
Stmt Id  Line  Statement

(ADTQ1-A  100  (DECL ARRAY A CHAR 99))
(ADTQ1-A2 200  (DECL CHAR ELEMB))
(ADTQ1-A3 300  (DECL CHAR ELEM))
(ADTQ1-A4 350  (DECL CHAR NEWLINE))
(ADTQ1-B  400  (ZERO IDX))
(ADTQ1-C  500  (WHILE RESULT))
(ADTQ1-D  510  (BEGIN BLOCK1))
(ADTQ1-G  600  (ASSIGN A IDX ELEMB))
(ADTQ1-P1 700  (PRINT ELEMB))
(ADTQ1-I  800  (INCREMENT IDX))
(ADTQ1-F  900  (ASSIGN A IDX ELEM))
(ADTQ1-H  1000 (NOT-EQUALS ELEM NULL RESULT))
(ADTQ1-E  1100 (END BLOCK1))
(ADTQ1-P2 1200 (PRINT NEWLINE))
```

Figure 7.4: Instance of **quilici-t1-large** plan.

```
Stmt Id         Line     Statement

(|sit-gen5|     (0 88)   (ASSIGN FIRSTINT FIRSTINT))
(ADTQ1-A        (1 100)  (DECL ARRAY A CHAR 99))
(|sit-gen15|    (2 144)  (NOT-EQUALS FIRSTINT |var-name13| FIRSTBOOLEAN))
(ADTQ1-B        (3 200)  (ZERO IDX))
(|sit-gen1|     (4 289)  (CHECK FIRSTINT FIRSTCHAR))
(|sit-gen14|    (5 297)  (ASSIGN |var-name7| |var-name7|))
(ADTQ1-C        (6 300)  (WHILE RESULT))
(ADTQ1-D        (7 310)  (BEGIN BLOCK1))
(|sit-gen3|     (8 362)  (ASSIGN FIRSTARRAYI FIRSTINT FIRSTINT))
(ADTQ1-G        (9 400)  (ASSIGN A IDX ELEMB))
(ADTQ1-I        (10 500) (INCREMENT IDX))
(|sit-gen6|     (11 574) (DECL INT |var-name7|))
(|sit-gen2|     (12 584) (ASSIGN FIRSTARRAYB FIRSTINT FIRSTBOOLEAN))
(ADTQ1-E        (13 600) (END BLOCK1))
(|sit-gen0|     (14 632) (CHECK FIRSTCHAR FIRSTREAL))
(|sit-gen8|     (15 682) (FOR |var-name13| 14 7))
(|begin-sid11| (16 683) (BEGIN |block10|))
(|end-sid12|    (17 685) (END |block10|))
(|other-sid9|   (18 686) (ASSIGN FIRSTARRAYI |var-name13| |var-name13|))
(ADTQ1-F        (19 700) (ASSIGN A IDX ELEM))
(|sit-gen4|     (20 765) (CHECK FIRSTINT FIRSTINT))
(ADTQ1-H        (21 800) (NOT-EQUALS ELEM NULL RESULT))
```

Figure 7.5: Instance of **quilici-t1** plan with 10 inserted statements.

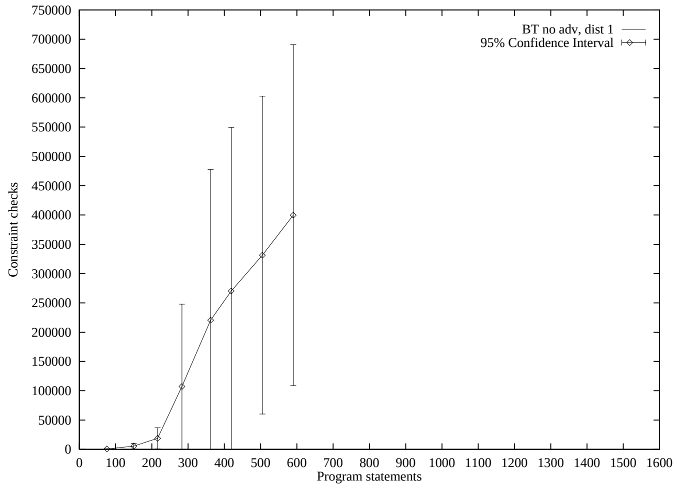


Figure 7.6: Standard BackTrack (95% conf. interval)

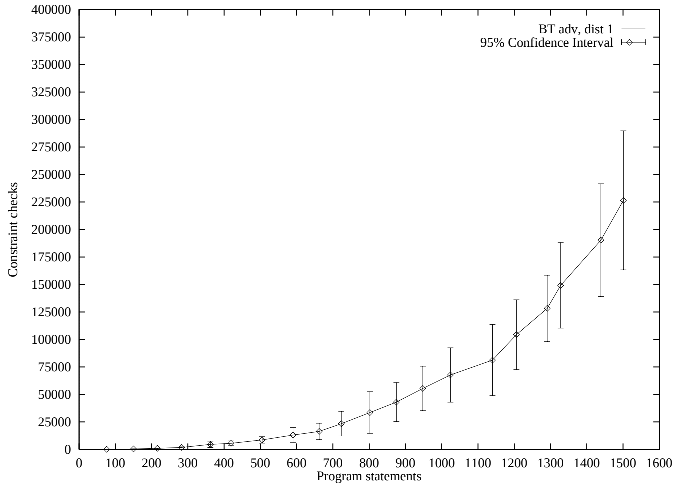


Figure 7.7: BackTrack, variable order (95% conf. interval)

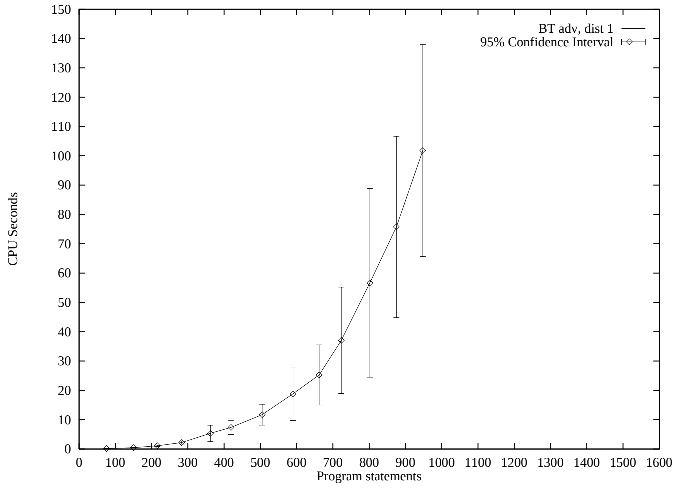


Figure 7.8: BackTrack CPU-time, variable order (95% conf. interval)

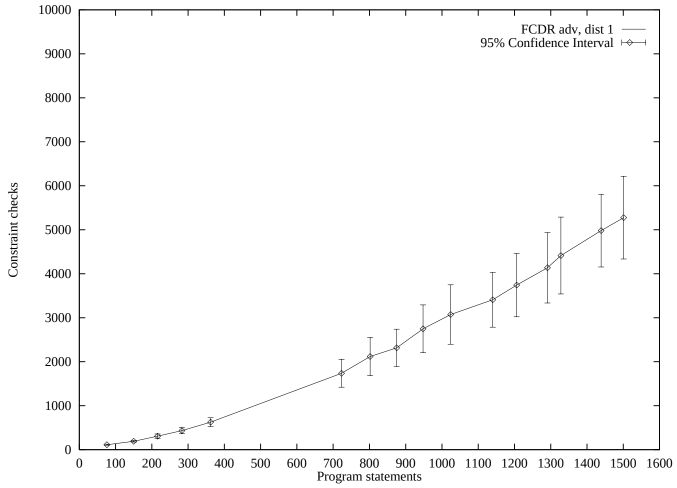


Figure 7.9: Forward Checking, DR (95% conf. interval)

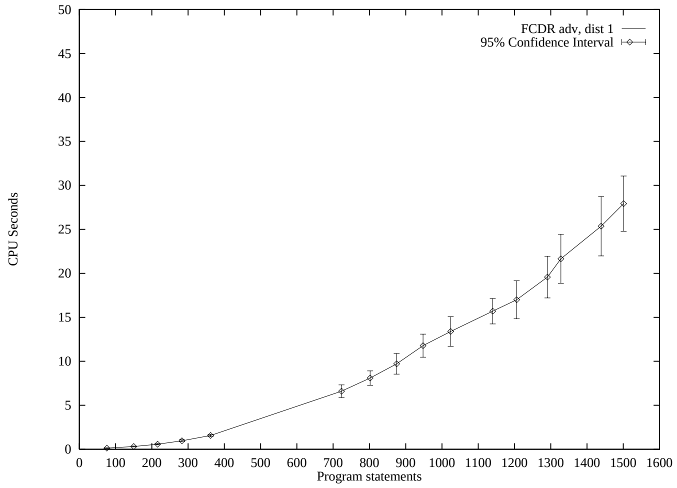


Figure 7.10: Forward Checking, DR CPU-time (95% conf. interval)

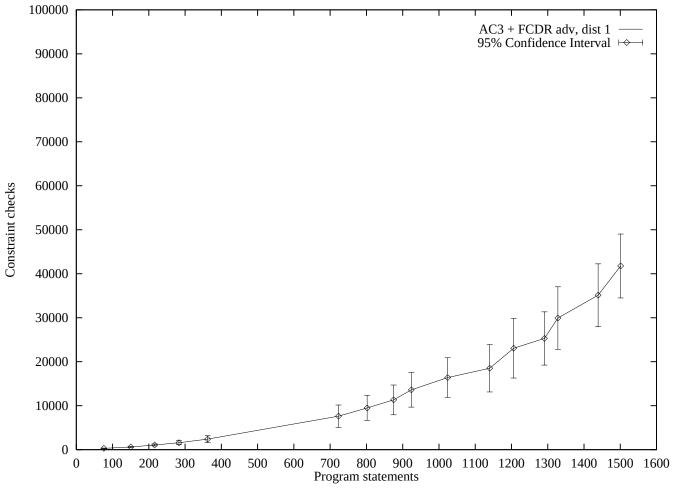


Figure 7.11: AC-3 with FCDR (95% conf. interval)

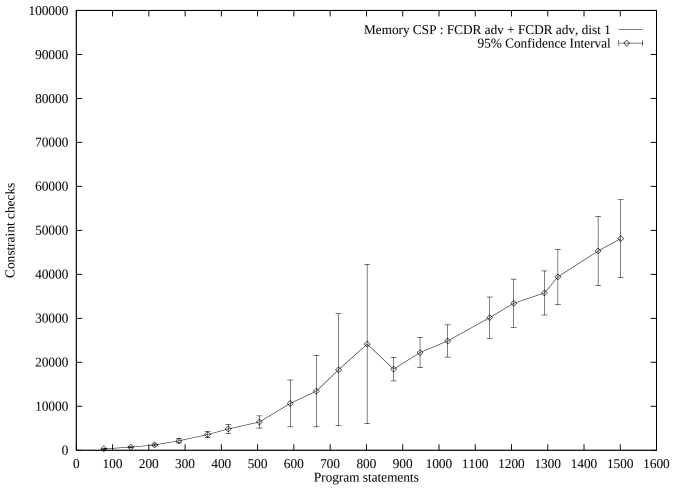


Figure 7.12: Memory-CSP with FCDR (95% conf. interval)

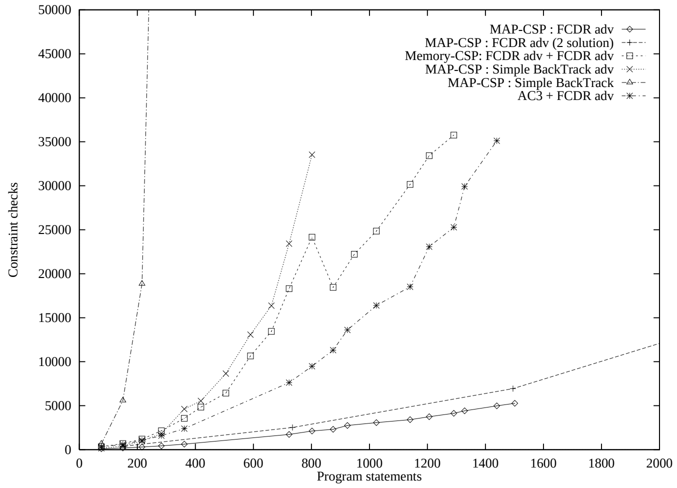


Figure 7.13: A range of strategies (medians graphed)

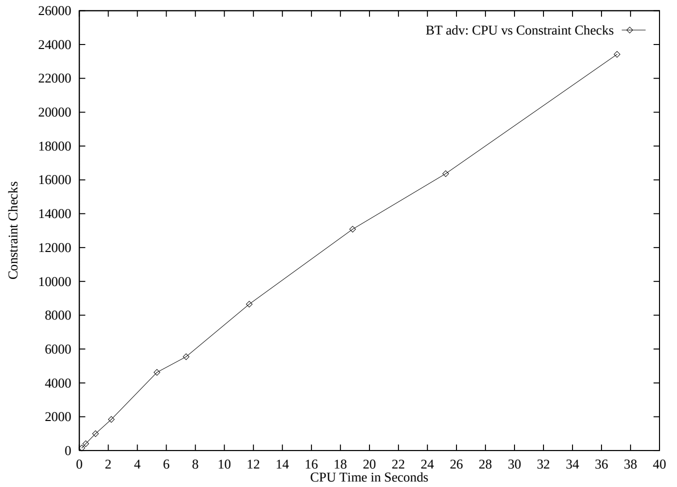


Figure 7.14: BT adv Constraints vs Time, Standard distribution

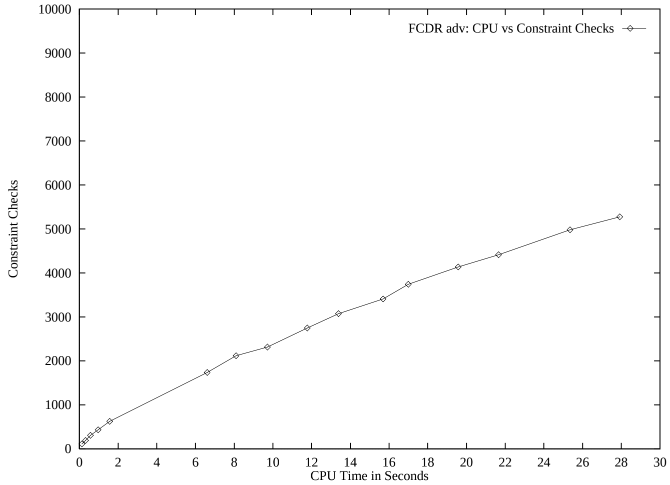


Figure 7.15: FCDR adv Constraints vs Time, Standard distribution

Multiple (2) template instances

The examples shown in the previous paragraphs for single template identification were conducted only up to a maximum of about 1,500 source code lines. In the following experiments I extend these results for the FCDR heuristic with advance variable ordering over a range of distributions and templates, allowing for multiple (2) intended template instances in each example, and performing MAP-CSP in 3 distributions at each source code size.

1. Figure 7.16 graphs a 95% confidence interval using FCDR with random initial variable ordering for Standard *quilici-t1* template, Standard distribution, while Figure 7.17 charts the same for Equal distribution, and Figure 7.18 for Skewed. Figure 7.19 charts the median result for all three distributions. I do not chart the confidence interval for the combined tests to avoid clutter — as can be seen, there exists an extremely large range of values for each problem size in this experiment as a result of the random variable ordering. For example, using the Equal distribution, different problems of size 1,500 took as many as 115,000 constraint checks and as few as 12,500.

2. Figure 7.20 graphs a 95% confidence interval using FCDR with advance variable ordering for Standard *quilici-t1* template, Standard distribution, while Figure 7.21 charts the same for Equal distribution, and Figure 7.22 for Skewed. Figure 7.23 charts the median result and confidence interval for all three distributions.

3. Figure 7.24 graphs a 95% confidence interval using FCDR with advance variable ordering for the Index *quilici-t1-index* template, Standard distribution, while Figure 7.25 charts the same for Equal distribution, and Figure 7.26 for Skewed. Figure 7.27 charts the median result and confidence interval for all three distributions.

4. Figure 7.28 graphs a 95% confidence interval using FCDR with advance variable ordering for the extended *quilici-t1-large* template, Standard distribution, while Figure 7.29 charts the same for Equal distribution, and Figure 7.30 for Skewed. Figure 7.31 charts the median result and confidence interval for all three distributions.

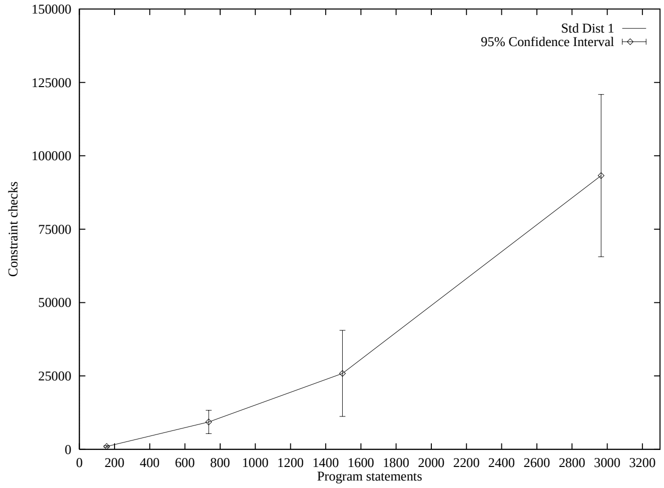


Figure 7.16: FCDR (Random), Standard Template, Standard code distribution

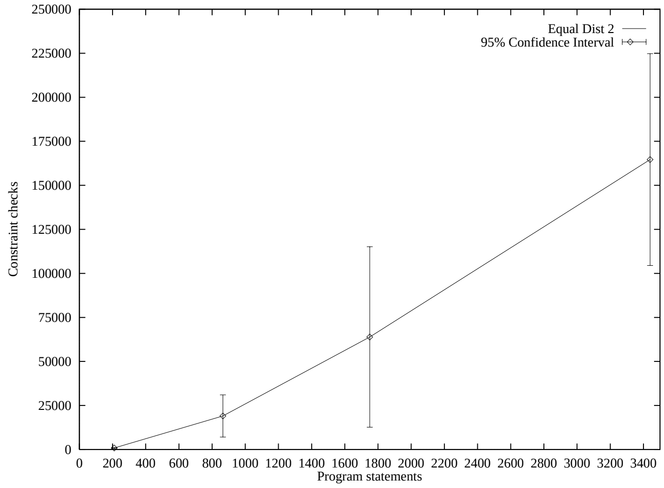


Figure 7.17: FCDR (Random), Standard Template, Equal code distribution

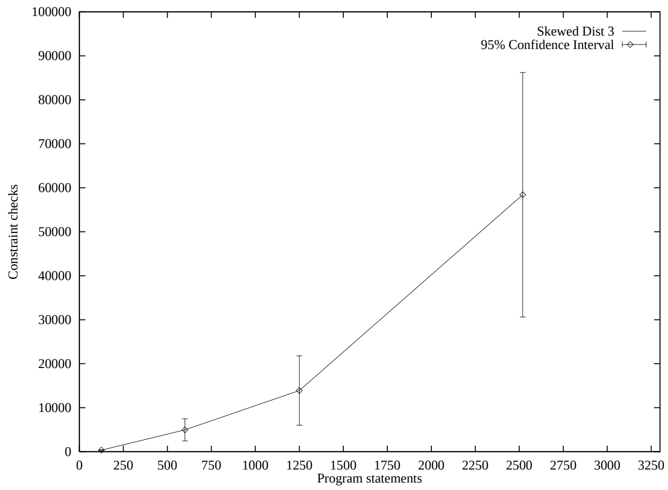


Figure 7.18: FCDR (Random), Standard Template, Skewed code distribution

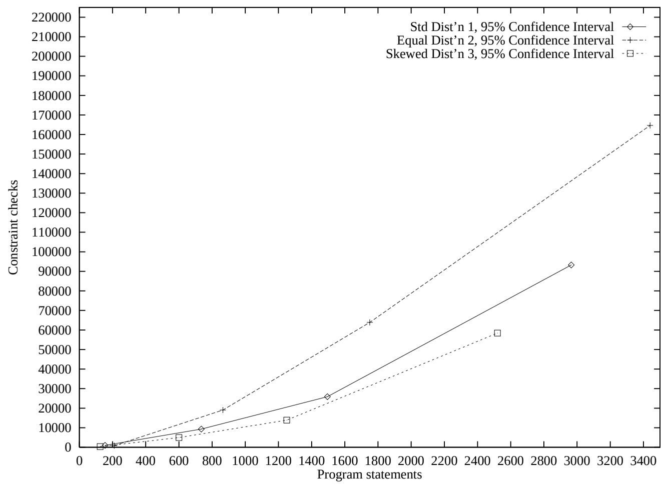


Figure 7.19: FCDR (Random), Standard Template, three distributions

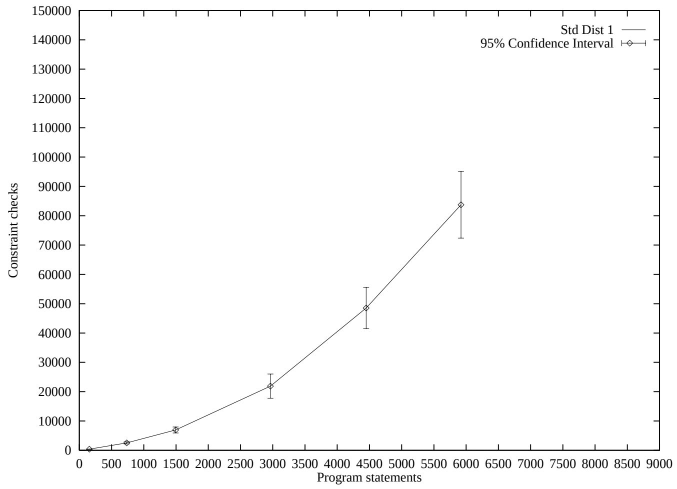


Figure 7.20: FCDR Standard Template, Standard code distribution

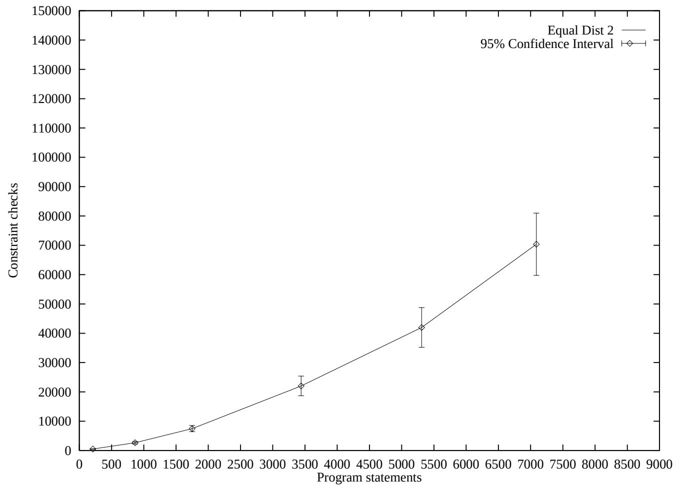


Figure 7.21: FCDR Standard Template, Equal code distribution

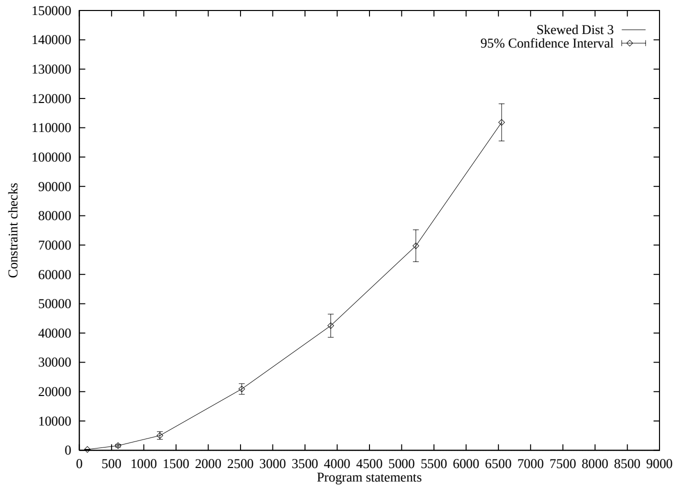


Figure 7.22: FCDR Standard Template, Skewed code distribution

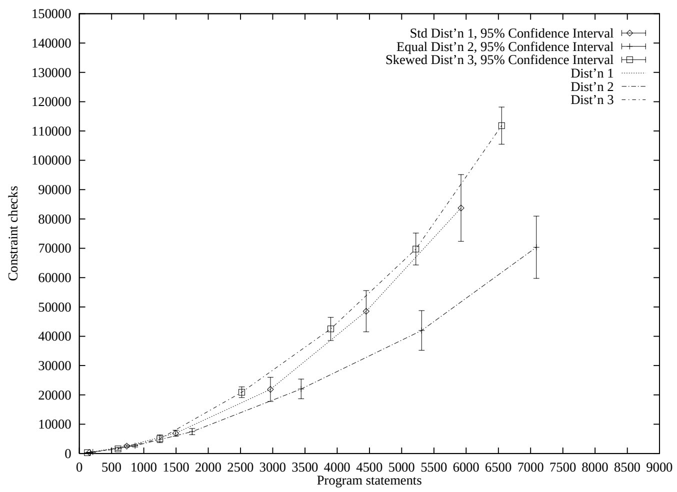


Figure 7.23: FCDR, Standard Template, three distributions

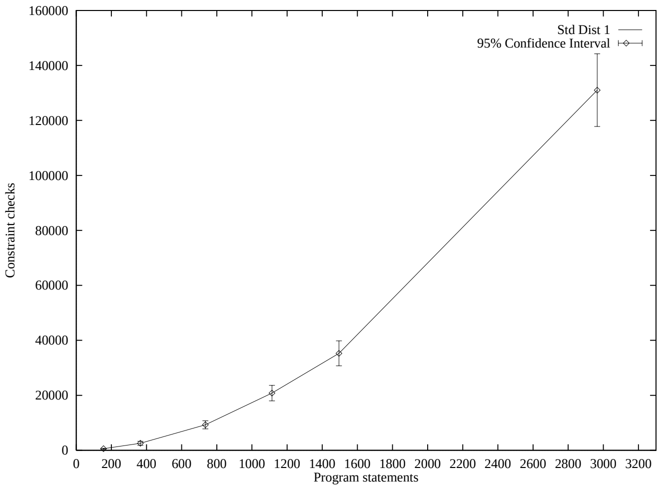


Figure 7.24: Index Template, Standard code distribution

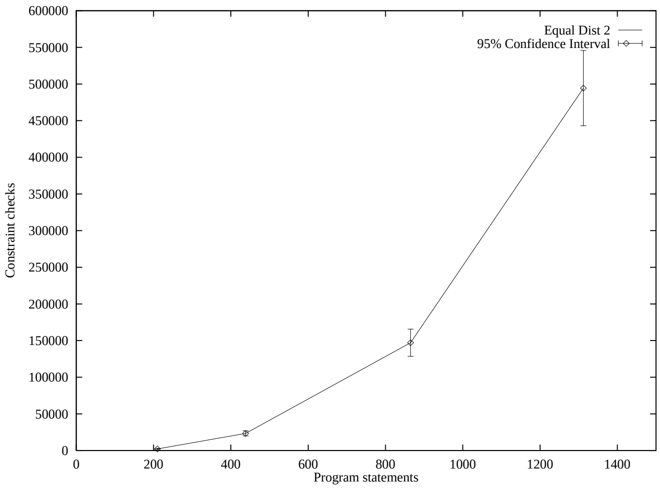


Figure 7.25: Index Template, Equal code distribution

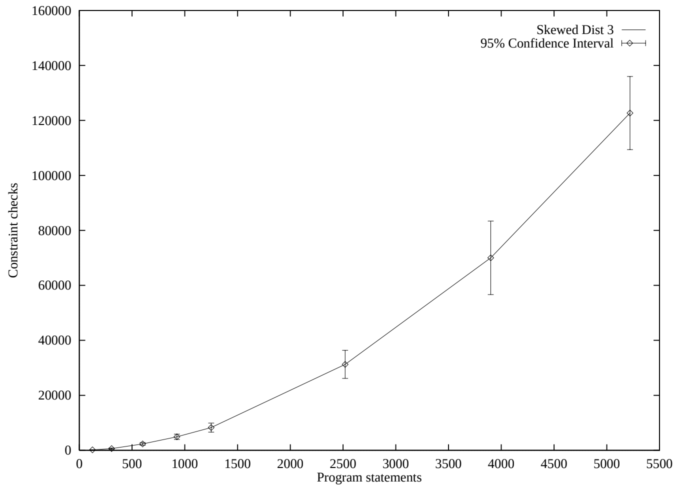


Figure 7.26: Index Template, Skewed code distribution

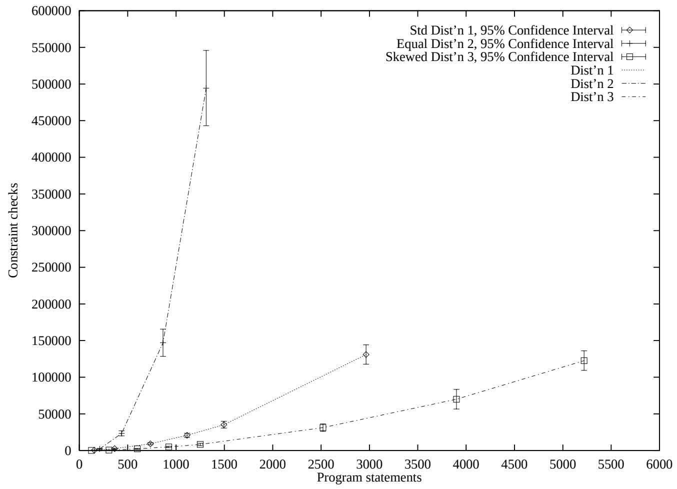


Figure 7.27: FCDR, Index Template, three distributions

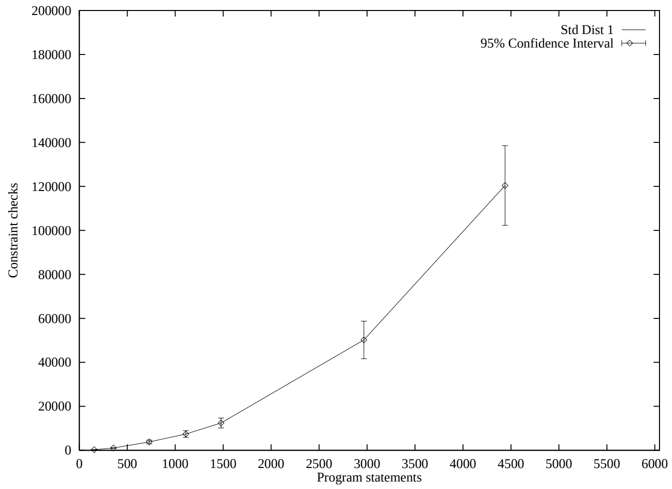


Figure 7.28: Large Template, Standard code distribution

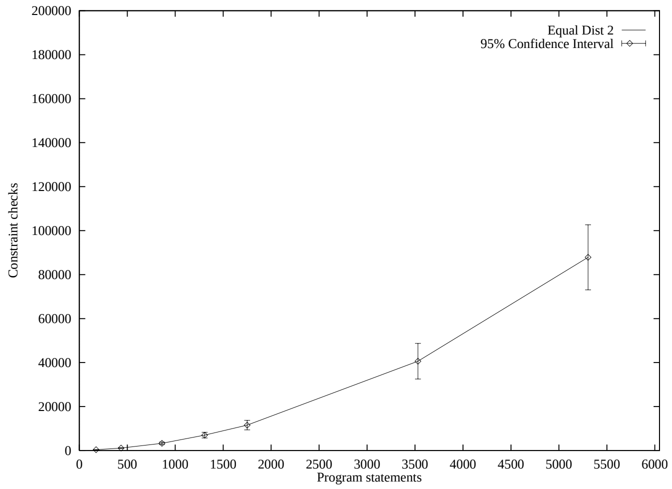


Figure 7.29: Large Template, Equal code distribution

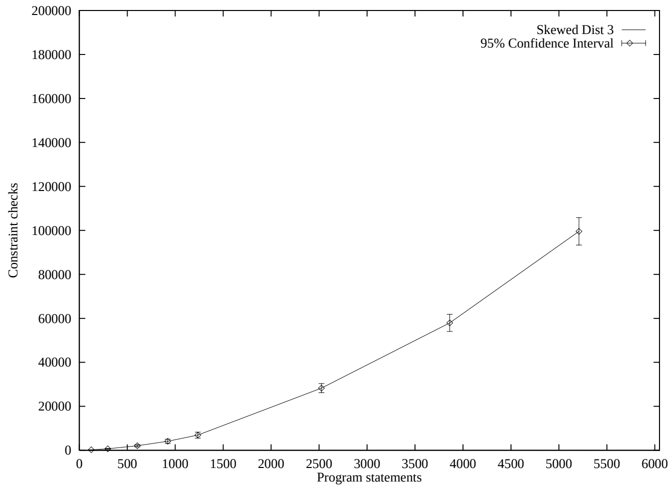


Figure 7.30: Large Template, Skewed code distribution

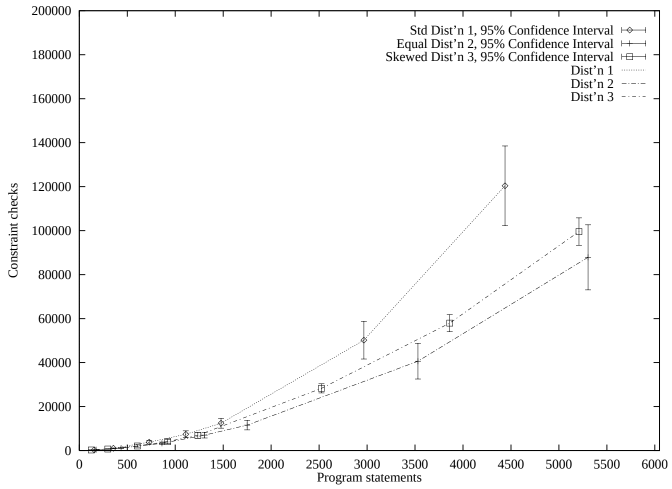


Figure 7.31: FCDR, Large Template, three distributions

7.1.4.2 Comparative Results

Figure 7.32 shows the same (median) results charted in Figure 7.13 for a variety of search strategies, with the extension that the FCDR with advance variable ordering (median) tests have been extended to almost 6,000 lines of code.

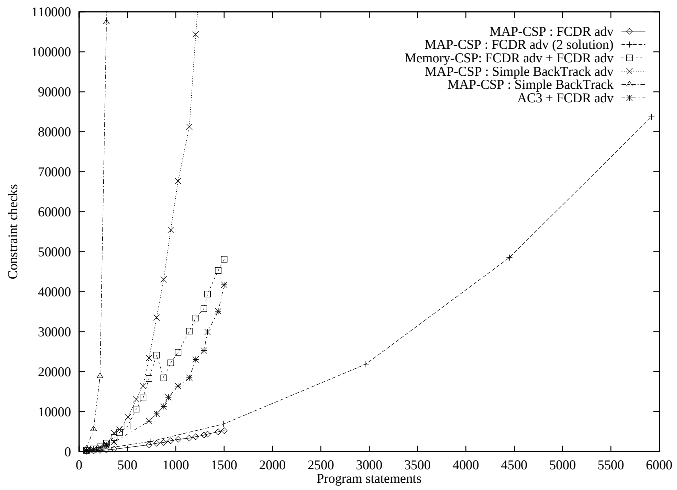


Figure 7.32: Extended results: strategy range

I wish to demonstrate that the MAP-CSP representation and algorithm is capable of providing all-instance results in moderately sized program blocks. An efficient MAP-CSP algorithm could make the execution of the larger PU-CSP algorithm more feasible. In addition, the MAP-CSP algorithm for template matching could potentially be stand-alone as a tool for assisting in the identification of source portions that may be replaced with existing source library objects.

Several observations can be made from these test results:

- Standard Backtracking exhibited very unstable performance in examples of the same size. As hoped, more intelligent strategies behaved in a more stable manner. Forward Checking was considerably more stable, and the applications using AC-3 in advance of search exhibited very small variance across test cases of similar size.

Stability is an important factor in any application that may be used as part of an online or interactive tool. In addition, Standard Backtracking was unable to complete in less than 600 CPU seconds for source instances exceeding 500 program statements.

- For source examples of up to approximately 600 lines of code, the FCDR strategies located all instances of the example program plan template in less than approximately 5 seconds of CPU time. In examples of up to 1,500 lines of code, all instances were identified in approximately 35 seconds. In such prototyped (Lisp) near-real-time circumstances it would appear that a tool could be fashioned that could be called up to run as a background process supporting an expert working with some reasonably sized source code.

In terms of evaluating the scalability of the CSP-Based approach, the results may be summarized as follows. The first set of tests involved expanding the size of the source examples considered to include various sizes up to approximately 6,000 lines of code. For each of these sizes, I used three different distributions of program events: a "Standard" distribution, an "Equal" distribution, and a "randomly Skewed" distribution. The standard distribution shown in Table 5.1 (relative to the example plans) is arranged this way:

1	While Zero For Array-Access
2	Block Arithmetic-Operators Logical-Operators Print
3	Assign
4	Declare Test

That is, declarations and tests appear four times as frequently as loops and array accesses, assignments appear three times as frequently, and so on. In an equal (Table 7.1) distribution, each construct appears approximately the same number of times. And in a random distribution (Table 7.2), I randomly chose particular event types to appear much more frequently in the program than others.

Figure 7.23 shows the results of running this set of experiments on a set of plan instances of the template from Figure 5.8. This template contains 9 components and 20 constraints. One way to summarize this graph is by the ratio of constraints evaluated per statement. At 1,250 statements, this ratio is approximately 4–1 for all of the approaches. At 3,000 statements, the ratio varies between 10–1 for the random distribution to approximately 6–1 for the equal distribution. At 5,000 statements it is 13–1 for the standard distribution and 8–1 for the equal distribution. At 6,000 statements this ratio is 14–1 for the standard distribution.

These numbers imply several things:

- The distribution of types of components within the program has a big effect on the performance of the CSP algorithm.
- The apparent linearity seen in the original experiments [Woods and Quilici, 1996c] details the results of Figure 7.16; however, experiments were only conducted up until approximately 1,000 insertions (resulting in approximately 1,500 lines of code) appears to be the flatter portion of a polynomial curve[1]

It is clear that the algorithm performs significantly better when all components appear equally within the program as opposed to either a randomly skewed distribution or a fixed distribution. A possible explanation for this behavior is that in the Standard and randomly Skewed distribution, entities that appear frequently as components of the plan tend to appear frequently in the program. This suggests to us that an extension

[1]Recall that the size of an unconstrained search space for this problem is bounded by M^N where N is the number of components in a template, and M is the size of the source. Consequently, one would expect that the relation of increase in work with source size is polynomial.

of these results might be useful in future work. In particular, one might generate a distribution similar to real-world C programs performing a variety of tasks *in different real domains*, generate testing programs based on those distributions, and then examine the performance of the algorithm across a variety of distributions. It may well turn out that the CSP approach will perform noticeably better on some types of programs than others.

It is also clear the ratio of constraints evaluated to program statements is growing rapidly for the "standard distribution". This is problematic, but there are several reasons to remain optimistic. One is that the "simulated locality constraints" are much looser than the *structural constraints* such as control-flow and data-flow constraints found in real programs. I therefore predict that this curve will flatten significantly when experiments are undertaken in future work using programs that preserve the structural properties of real-world programs. This is because tighter constraints should reduce the size of the domain value sets, leading to a speedier solution to the CSP and fewer constraints to evaluate. It is an open question, however, just how much the curve will flatten and how much that will slow the rate of growth of the number of constraints evaluated.

The other reason to expect even better scalability using CSP is that, even if real-world structural constraints do not prevent the currently modeled plan recognition algorithm from going exponential, it appears from the curves that instances of individual plans can be recognized in programs in the 5,000 statement range in a reasonable amount of time. For instance, the 60,000 or so constraints needed to evaluate takes less than 25 seconds or so on a shared SPARCserver 1000 workstation. While 5,000 statements is small compared to the sizes of real-world programs, it is within an order of magnitude of the size of modules in real-world programs or modules that have been created from software systems using semi-automatic techniques [Newcomb and Markosian, 1993].

A second set of experiments has also been run which have been designed to begin exploring the scaling properties of the CSP approach as the program plans or templates get larger in size. To do this, plan were explored that were both smaller (four components and give constraints) and larger (14 components and 29 constraints) than the original plan.

Figure 7.31 shows the results of running this set of experiments for the larger plan over the three distributions. This graph can be summarized using the same ratio of constraints evaluated per statement. At 3,000 statements, the ratio varies between 20–1 for the standard distribution to 10–1 for the equal distribution. At 5,000 statements, the ratio is 30–1 for the standard distribution and 12–1 for the equal distribution. My conclusion here is that with smaller programs, at least, doubling the size of plan leads to approximately double the number of constraints that must be evaluated. This practical result is encouraging for problems of this size, particularly in light of the analytical results of Section 6.2.1 which demonstrates that the complexity of a template matching problem for a template of N components with M source statements is bounded by a polynomial M^N.

7.1.5 Implications for Program Understanding Research

This chapter has explored some scalability issues for a constraint-based approach to partial explanation using plan recognition. This, however, is only one part of the overall program understanding problem. Program understanding is often viewed as a three-step process:

Parsing: Turning the program into an annotated AST using standard parsing and flow analysis techniques.

Canonicalization: Simplifying this internal representation to minimize the number of different plans that must be in the library. A simple example is transforming all relational expressions so that they involve only the greater than operator and not the less than.

Plan recognition: Recognizing instances of each plan in a library of program plans.

This chapter has focused on the scalability of one part of the plan recognition process: determining whether a given program plan is present based on the existence of the constrained plan components in the internal representation of the program. There are other aspects to this problem, that have not yet been addressed, including how to decide which plans to try to locate within a given program, and in which order to try those plans. Chapter 8 investigates integration issues in detail, and discusses the use of abstraction as a mode of canonicalization.

My experiments have several important implications. One is that it may well be necessary to have a modularization step that precedes the plan recognition process, where this step breaks the program into pieces of whatever size the program understanding algorithm can comfortably handle before the combinatorics become problematic. In fact, this is precisely the point of the PU-CSP stage for integrating these partial solution stages. Some work on semi-automatic modularization of COBOL programs has already been done that has demonstrated that large COBOL programs can be broken into modules of 25,000 or so statements [Newcomb and Markosian, 1993]. This is only a factor of five larger that the point that the CSP approach can comfortably handle, which makes it appear worthwhile to determine whether those techniques can be extended to break programs down into even smaller modules.

In addition, even if one successfully recognizes plans at the module level, there also needs to be a mechanism for combining this modular understanding that needs to follow the plan recognition process. It is an open question as to how one might accomplish the task of coming up with an understanding for a program as a whole, especially if the library is incomplete and there exists only partial understanding of what a module does.

Finally, the success in using CSPs in the local or MAP-CSP understanding process suggests that perhaps they can be applied to other related tasks, such as selecting plans to recognize, or as part of the canonicalization process. However, it is an open and interesting research question how to do so.

7.1.6 Conclusions

This chapter (and also some results presented in Chapter 5) presented empirical results on the scalability of program understanding algorithms. I first compared the performance of the CSP approach to a CSP-based version of several existing program understanding algorithms. The initial results suggested that the CSP approach is significantly more efficient than these other approaches. I then took an initial look at scaling properties of the CSP approach to large programs. Experimental results appear to back up the observation from Section 6.2.1 that the CSP approach takes a polynomially increasing amount of time to recognize plans as the programs grow in size. In particular, the rate of increase of this curve has been found to be bearable in the 5,000-statement or less range for the CSP algorithms tested.

These results may be thought of as some initial, overdue, data points in a progress report on the state of the art of program understanding. In particular, the specific amount of work done by the CSP recognition algorithm can be reduced, perhaps significantly, by moving to real control-flow and data-flow constraints, an experiment I am intending to set up in a future extension to this work. This may well mean that significantly larger programs can be successfully understood. In addition, the relative amount of work done by the algorithm may increase rapidly as one moves toward exploring larger and larger plans, rather than slowly as it had with the first few plans. This may mean that there is an practical upper bound on the size of individual plans that can be efficiently recognized in a program of a particular size. If the initial empirical results hold up, they suggest that automatically modularizing large programs and combining modular understanding are several important areas of future research. The PU-CSP model addresses integration of these modularized partial explanations.

My hope is that this work will spur others working in the area of program plan recognition to do one of two things: either map their understanding algorithms into the CSP framework so that others may easily compare their performance with the CSP approach, or to provide data on the performance of their program understanding algorithms as programs grow in size. This step is crucial to move beyond the understanding of "toy" programs and into the world of being a useful aid in the re-engineering of real legacy software systems.

Part IV

Global Explanation

Chapter 8

Managing Global Explanations (PU-CSP)

The entire program understanding problem is now viewed as a constraint satisfaction problem. In this model, a long program code is first divided into blocks, where each block is a set of closely related source code. The program understanding problem is to identify the top-level function of each of these program blocks, so that not only the inter-relationships among the blocks are explained, but also the constraints specified by a program library on the program plans are respected. A key problem, then, is to assign one plan component to each block, subject to a set of constraints. This problem is called the **program-understanding CSP**, or PU-CSP.

The number of program plan components that one could assign to each block could be enormous. To be practical, it is crucial to first reduce the number of explanations for each block as much as possible. This process could be helped by a related constraint satisfaction problem, one that was explained in detail in Chapter 6: the problem of finding all instances of a given program plan or pattern in the entire source code, or MAP-CSP.

In the following subsections I explain in detail how to accommodate hierarchically valued domain plans and partial local explanations in a larger understanding model.

The solution of PU-CSP with a complete search strategy (or one based on consistent constraint propagation) completes with a *set* of all consistent mappings between program blocks and program plans. Plan recognition approaches such as that described in Chapter 3 return a similar, possibly ordered set. Each mapping in this set may be thought of as a consistent explanation of the program blocks. Constraint propagation approaches (such as arc consistency) may leave exact mappings partially unspecified, in which case a search algorithm over the reduced problem could extract the possible solutions.

8.1 Overall Understanding Model

PU-CSP is formed in the following way. Suppose that an initial decomposition or slicing of the source code is given. Each block of source code corresponds to a *variable* in the PU-CSP. The *Variable domains* correspond to all possible explanations of an individual source code block. As an example, consider the source code program statements of Figure 8.1 (from Chapter 2) as the blocks. Take each block as a PU-CSP variable which ranges over all possible program plans of corresponding statement type, such as “declaration”, “assignment”, “print”, etc, in the plan library of Figure 8.2 (from Chapter 2).

8.2 PU-CSP Complexity Issues

The survey of the approaches to program understanding outlined in Section 5 has resulted in the PU-CSP model. Simply put, one is given a source program to understand in terms of a library of program plan templates. From these, one is to compose a solution in the form of a mapping from portions of the source code to part of the plan library. In this section, I prove that this problem is intractable.

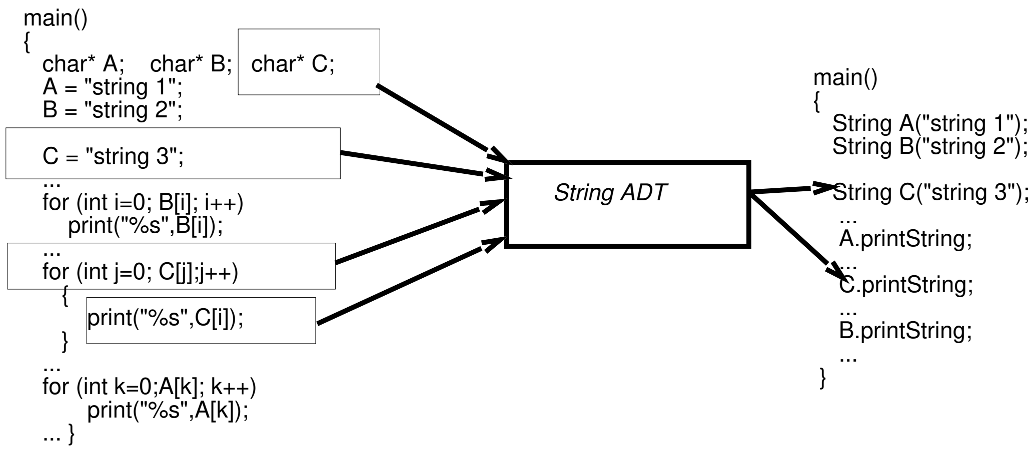


Figure 8.1: C source code mapped through a String ADT instance to C++ code

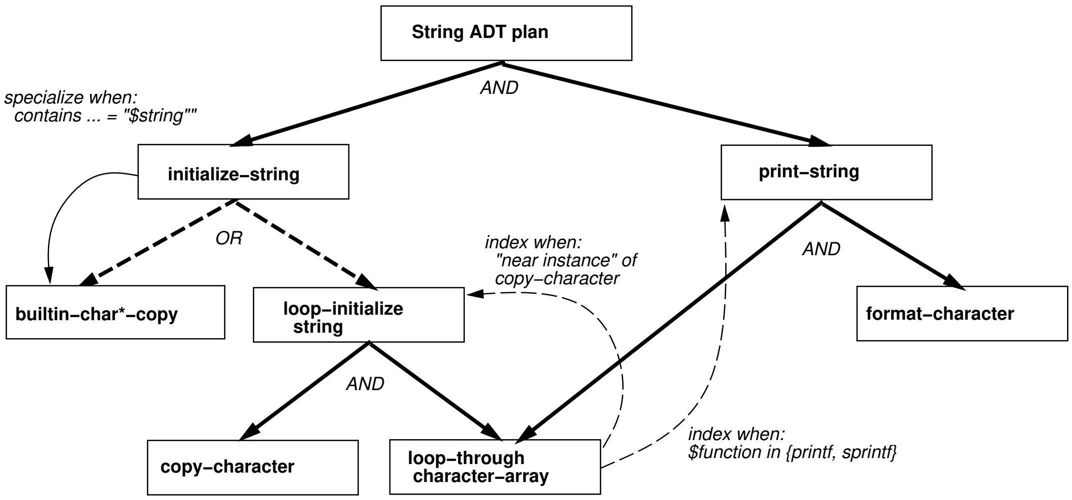


Figure 8.2: **String** ADT within a hierarchical program plan library

8.2.1 Simple Program Understanding Problem

8.2.1.1 The Modeling Process

The SPU program-understanding constraint-satisfaction problem (referred to as PU-CSP when formed as a constraint satisfaction problem), is formed in the following way. Suppose that an initial decomposition[1] of the source code is given. Each **block** of source code corresponds to a *variable* in the PU-CSP. The *variable domains* correspond to all possible explanations (mappings to the knowledge or library) of an individual block of the source code. The constraints between the variables can be specified via both the structural relationships in the source program, and subsequently, knowledge relationships in the program plan library. A *solution* is a mapping between the source code blocks and the library such that the constraints are satisfied.

The modeling process may be illustrated in more detail. A Program Understanding CSP (PU-CSP) is formulated via four distinct steps shown in Figure 8.7. First, the source code is pre-processed[2] creating an intermediate representation which precisely captures many interrelationships among the elements of the abstract syntax tree implicit from a parsing of the source. This representation includes data-flow and control-flow among functional blocks. Second, the source code is partitioned into spatially localized, cohesive code blocks[3] which exhibit several inter-block functional relationships. Third, a

[1]This decomposition is such as would be created from a simple reading of the code in which function or procedure blocks are identified as the portions of the code to understand. These decompositions are structurally related as might be discerned from the annotated abstract syntax tree construction presented in [Devanbu and Eaves, 1994].

[2]Using, for example, a parsing tool such as GEN++ or REFINE.

[3]These code blocks may be of varying size and complexity. The actual determination of appropriate blocking characteristics will be investigated empirically in later work. It is important to note that since the library of knowledge is arranged hierarchically it will often be the case that smaller blocks will tend to correspond to lower-level program plans and vice-versa. Consequently, the problem itself may be thought of as the need to generate a sequence of multi-layered (hierarchical) mappings. It has been suggested by Alex Quilici in a personal communication that these mappings would best be generated bottom-up from small code fragments and plans to larger. This is, however, not the only possible approach.

skeleton CSP is formulated consisting of one variable for each identified source block, and constraints among these variables are derived from the intermediate representation level artifacts. The combination of "typed" input and output flows for each particular block are adopted as *reflexive* constraints on the corresponding variable, effectively limiting the range of program plans that might explain that block. An example of block typing might be where an instance of a plan component to update a given database field may be known to required the passing of this field both into and out-of a function, and additionally this field may have to be both readable and writable - the typing information thus restricts which functions are capable of being instances of the given plan. Finally, each CSP variable is matched (or indexed by type information) against the templates in the program plan library. Potentially matching plan templates are then composed as the domain ranges of each variable.

In a PU-CSP, the constraints among variables are of two types:

- *Structural* constraints are determined from the source code. They include such things as scope or called/calling relations, precedence relations, or shared information relations among component blocks.
- *Knowledge* constraints are independent of the source code. These constraints reside in the AND/OR hierarchical program plan library, restricting program plan inter-relationships. The AND connections indicate a parent-child component relationship, while the OR connections indicate specialization/generalization relations. Each of ANDs and ORs can serve to indicate important details of the parent-child relationship, such as the *role*[4] of a child as part of the higher level parent, or what details specialize a child from a parent. Specializing an abstract plan in one of several ways. Assigning one program plan as an explanation of a particular PU-CSP

[4] For instance, a compositional role might describe what data-flows a child provides in its function as part of a parent. In a more abstract instance, this role might be a service rather than a low-level data-flow.

variable thus constrains consistent assignments of other component variables. This detail effectively describes the allowable range of known program plan structure.

A solution to the PU-CSP is an assignment to each variable by one program plan component in the plan library, such that no structural constraint from the source code, or knowledge constraint from the plan library is violated.

The representation of program understanding as PU-CSP provides a convenient framework for the interpretation of earlier program understanding heuristics as particular constraint manipulations. For example, the Quilici-style indexing outlined earlier in which an index instance in a source code signals the need to attempt to match a particular program plan from the library can be thought of as a specific constraint ordering during CSP search. Quilici-style specialization preferences can be viewed as a heuristic for ordering the application of hierarchical knowledge constraints, essentially reducing the range of domain variables in a hierarchical CSP. Similarly, Quilici and others refer to inferences or *implications* which indicate the likely existence of plans based on the identification of other plans. Such behaviour can be interpreted as a special kind of dynamic variable-ordering heuristic in which successful instantiation of a particular variable suggests the need to attempt to instantiate a related variable next.

My strategy will be to simplify the problem. Consider the following **Simple Program Understanding (SPU)** problem, depicted in Figure 8.3. We are given the following:

- The source code consists of a collection B of program blocks $B_i, i = 1, 2, \ldots, m$. These blocks can be viewed in terms of a corresponding graph $P = (B, D)$ where D is the set of edges of the graph $D_k, k = 1, 2, \ldots, n$, such that an edge $D_{i,j}$ exists between nodes B_i and B_j if and only if a data-flow exists between the program blocks.

- We are also given a library of program plan templates represented as a graph $L =$

(T, C) where T, the set of templates $T_o, o = 1, 2, \ldots, t$ in L are related to one another through data-flow relationships. These relations are specified by a set C of edges, such that an edge $C_{k,l}$ exists between templates T_k and T_l if and only if a data-flow possibly exists between them.

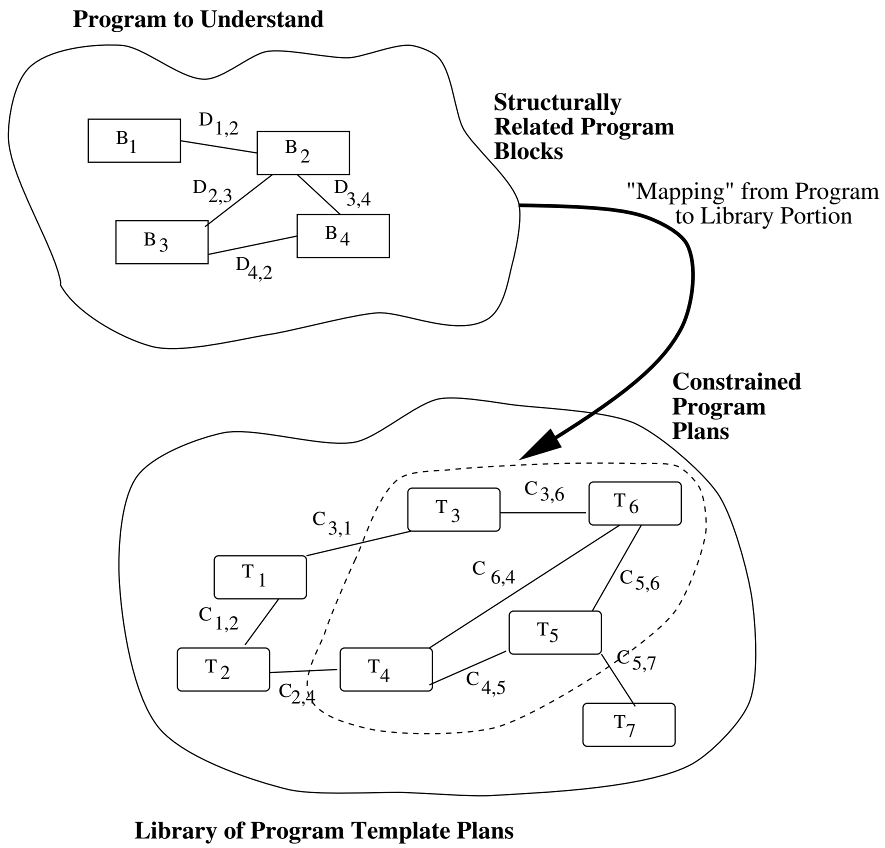


Figure 8.3: Simple Program Understanding

Given the above structure, the SPU problem is to determine if a correspondence exists from program blocks to a subset of templates. The correspondence is of the form of a mapping between templates and program blocks, and between their data-flow relationships. As an example, in Figure 8.3, the following correspondence gives rise to an understanding of the example program:

$$
\begin{aligned}
B_1 &\iff T_3 \\
B_2 &\iff T_6 \\
B_3 &\iff T_4 \\
B_4 &\iff T_5
\end{aligned}
$$

It may be contended that the SPU problem is representative of many program understanding tasks. In Kozaczynski and Ning's approach in the `Concept Recognizer` system, program plans which consist of components and constraints abstracted away from a particular implementation language or method are utilized. Quilici extends these plans with the provision of indices (memory) that control the selection of candidate plans more selectively than in `Concept Recognizer`. The library of interrelated program plan templates in each approach are essentially the same as outlined in SPU, with the exception that a hierarchical structure may be imposed on the library. In Wills' approach, program components are modeled as graph grammars and are used to parse an intermediate flow graph representation of a source program. A component's make-up is constrained by its grammar, and these components are composed in a library (of constraints). The SPU program understanding model abstracts these differing representations of components and constraints into a unifying constraint-based library format. Understanding approaches uniformly assume that source programs have been pre-processed into a intermediate representation (annotated abstract syntax trees, annotated flow graphs) which makes explicit use of data-flow and control-flow information. This visually implicit and frequently difficult-to-see information can be readily obtained with existing polynomial algorithms, and the exploitation of this structure is precisely the kind of lever that can assist in the understanding process. In SPU, this information is represented as a simple program graph of related program blocks.

8.2.2 NP-hardness Proof

The formal definition for the SPU problem is similar to that given in Section 6.2.0.2 for the SMAP problem. The SPU problem could be stated more formally as follows. Given a library of program plans $L = (T, C)$ and a source program $P = (B, D)$, does there exist at least one subgraph of the library $L^S = (T^s, C^s)$ where the templates in the subgraph $T^s \subseteq T$, and the constraints among the templates $C^s \subseteq C$, are matched to the source program by a mapping function X, defined as follows:

- X maps every program block B_i to a member of T^s; and
- X maps every program data-flow edge $D_{i,k}$ to a corresponding member $C_{u,v}$ of C^s, where $u = X(B_i)$ and $v = X(B_k)$.

It is possible to prove the claim that SPU is NP-hard by a reduction from the **Subgraph Isomorphism** problem, described in Section 6.2.0.3 on page 152.

The transformation to an SPU problem can be done as follows. Every vertex of V_1 in G is a program template, and every edge of E_1 in G signifies a data-flow between two templates. Each vertex of V_2 in H is a program block and each edge of E_2 in H is a data-flow between blocks. A mapping between a program P and a subset of a library of related templates T exists if and only if H is a an isomorphic subgraph of G. Furthermore, this transformation can clearly be done in polynomial time.

8.2.3 Applicability of Local and Global Strategies

The PU-CSP-based solution strategy outlined in this chapter and expressed most generally in the overall interactive strategy of Table 8.1 on page 230 includes the MAP-CSP sub-problem as an integral tool in the process of reducing the explanatory range of PU-CSP program blocks. MAP-CSP, as has been described, is based on *complete* solution strategies that find all instances of a particular program plan. As has been mentioned elsewhere, it is possible to utilize local search strategies for MAP-CSP in place of global ones. The advantage of such a selection is that it may be possible to find single local solutions very quickly. Similarly, the complete strategy of MAP-CSP might be adjusted to quit after finding a single instance. In this way, a significant incremental refinement may be made to the PU-CSP without paying the entire cost of a complete search. The downside of such a selection is that negative information such as the confirmation of the *absence* of a program plan cannot be utilized in reducing the range of the PU-CSP explanations. An interesting area for future work is to explore the relative interactive utility of both local/incomplete and global/complete strategies.

8.2.4 Applying Local Constraint Propagation

In general, the more constrained a particular CSP, the easier it is to solve, provided it is known in advance which parts of the problem are the most highly constrained. It is my contention that program understanding can be thought of as a well-constrained problem in many useful instances. Software is ideally well-structured and compartmented by design, and a rich system of structural constraints between functional blocks can be extracted through known methods. Program plan libraries such as commercial or shared object libraries contain a similar structure of knowledge constraints which can be annotated with design information much more readily than is the case with any particular piece of software since it has been intended for wider distribution and use.

The large number of knowledge and structural constraints in a particular problem instance combine to limit the number of consistent explanations or mappings for collections of related program blocks. In particular, the application of even one such structural constraint among blocks could reduce the domain size of a program block significantly. This reduction can in turn be cascaded through adjoining block relations into successive reductions of other domains. This process is known in CSP related algorithms as local constraint propagation. An algorithm which enforces that all domains be consistent with their immediate neighbours is known as an *arc consistency*(AC) algorithm[5]. Many variations and extensions to the original AC algorithm, AC-3 [Mackworth, 1977] have appeared in the literature, some of which are mentioned in [Van Hentenryck *et al.*, 1992a]. These algorithms and many variations have been extensively applied and tested with a wide range of problems.

Consider a pair of variables (X, Y) and a relation $R(X, Y)$. The *arc* R is said to be *consistent*, if for every domain value of X there is at least one consistent domain value of Y. If this condition is not satisfied, an AO-Revise routine can be applied to the pair to remove any value of X that does not have a corresponding consistent value of Y. If, for every pair of related variables in a problem, all are consistent, the problem has been made *arc-consistent*.

8.2.4.1 A Simple PU-CSP Example using Local Constraint Propagation

In this section I demonstrate the applicability of local constraint propagation with an example. The repeated application of local constraints to reduce variables domains admits a solution with no search in this example. For purposes of illustrative simplicity, this example is constructed using PU-CSP variable domain values that are "flat" only. No hierarchical domain values are utilized. Examples utilizing hierarchical values are given in Section 8.3.2.

A piece of input source code is shown on the left of Figure 8.4. The code is parsed and

[5] Other algorithms enforce different degrees of consistency, from only partial arc-consistency over a subset of all arcs, to consistency along paths of arcs of varying lengths.

program blocks extracted along with data-flow information as shown on the right side of the figure. This figure has been significantly simplified for the purposes of this example.

```
main()

struct2 isrt( struct1 *this, struct2 * inL )

  if ( mbr(this,inL) )
  then return inL
 else return app(inL,this)

end isrt

int mbr(struct1 *ck, struct2 *cks)
if ( (cks == nil)  or (cks.first ==nil)) then return 0
struct1 *try = cks.first
while( not(try.id == ck.id) and not(try.next == nil) ) do
    try = try.next
end while

if  (try.id == ck.id)
then return 1
else  return 0

end mbr

struct2 app(struct2 * intL, struct1 *putE)

if(intL.first == nil)
then  intL.first = putE
          putE.next = nil
else  struct1 *temp = intL.first
        intL.first = putE
        putE.next = temp
return intL
end app
```

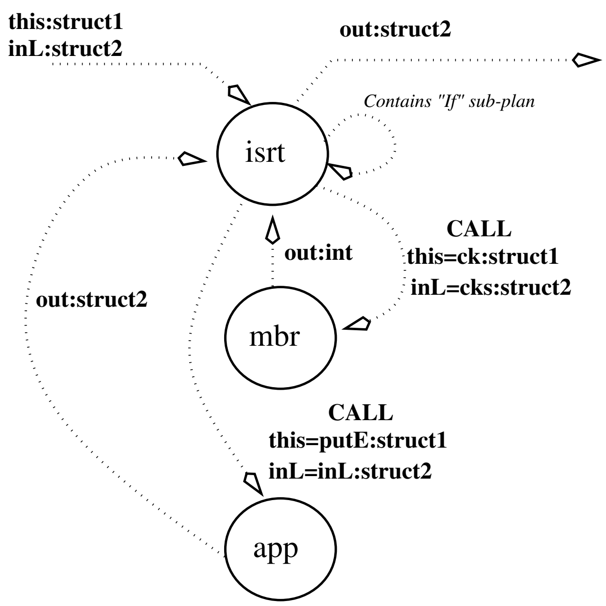


Figure 8.4: One “Blocking” of a Source Fragment

I wish to utilize the PU-CSP[6] framework to *understand* the source code in terms of the hierarchical program template library fragment given in Figure 8.5. This library fragment has been assembled by observing the structure of commercial C++, SMALTALK, and persistent-object libraries. The source blocks are mapped to variables, and initial domain ranges are assigned according to block input and output types as shown in Figure 8.6. Variable `isrt`, for example, potentially maps to several library plans based solely on input and output typing. A further (reflexive) constraint application based on observation of key components (`If`) in the structure of `isrt` could significantly reduce this set. In this

[6]PU-CSP corresponds to the SPU model.

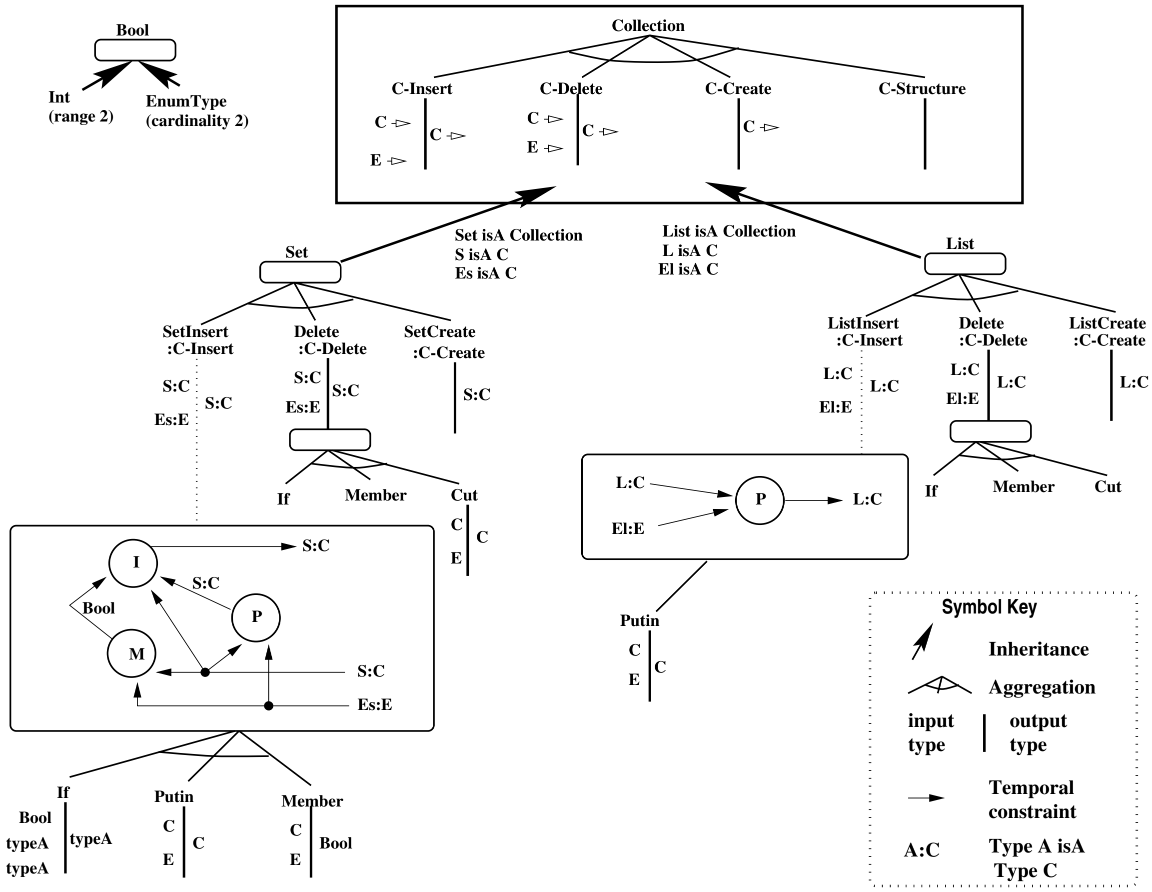


Figure 8.5: Library Fragment

example, only plans $\mathbf{Insert}_{Set}$, $\mathbf{Delete}_{Set}$, and $\mathbf{Delete}_{List}$ satisfy this constraint. Similarly, `app` maps to $\mathbf{Member}_{Set}$ and $\mathbf{Member}_{List}$; `mbr` to $\mathbf{Putin}_{Set}$, $\mathbf{Putin}_{List}$, $\mathbf{Cut}_{Set}$, $\mathbf{Cut}_{List}$, $\mathbf{Insert}_{Set}$, $\mathbf{Insert}_{List}$, $\mathbf{Delete}_{Set}$, and $\mathbf{Delete}_{List}$.

The domain range of variable `isrt` may be *revised* with respect to `mbr`. In this case no values may be removed since $\mathbf{Insert}_{Set}$ is consistent with value $\mathbf{Member}_{Set}$, $\mathbf{Delete}_{Set}$ is consistent with value $\mathbf{Member}_{Set}$, and $\mathbf{Delete}_{List}$ is consistent with value $\mathbf{Member}_{List}$. Revising `app` with respect to `isrt` yields consistent mappings for values $\mathbf{Putin}_{Set}$, $\mathbf{Cut}_{Set}$, and $\mathbf{Cut}_{List}$.

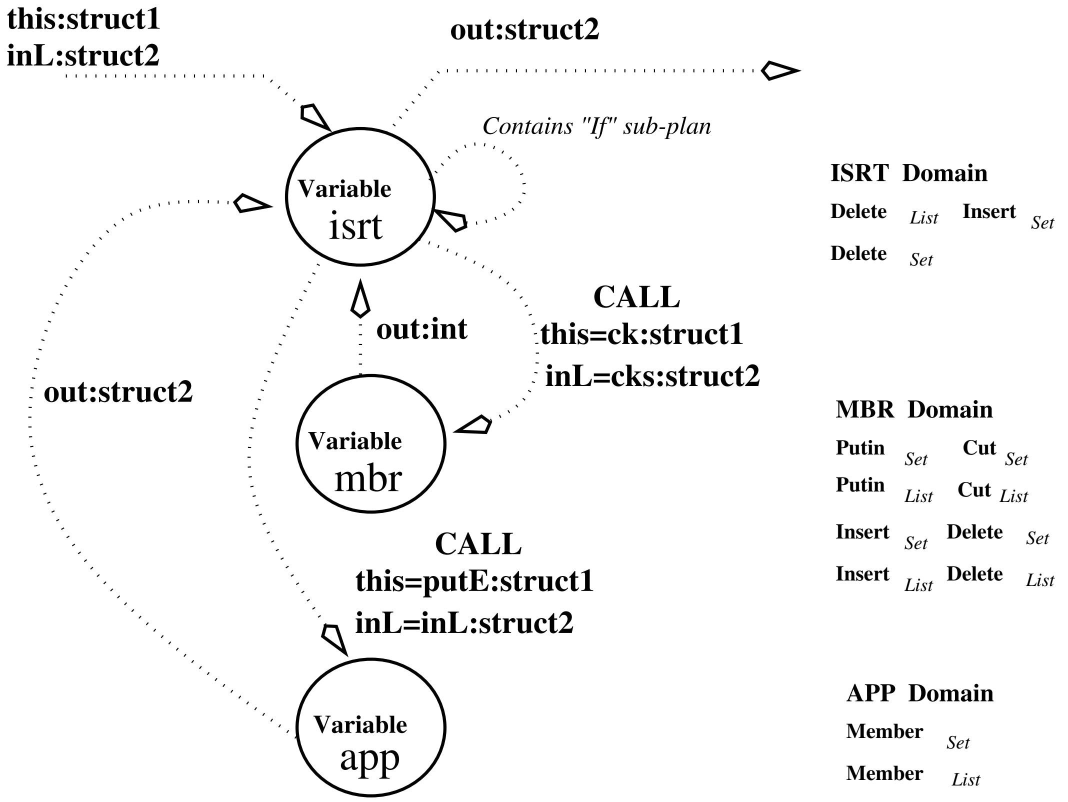


Figure 8.6: Initial PU-CSP

After this revision, no further reductions can be made, one is left with three combined alternate explanations that are consistent with the structure outlined. These three are: (1) a set insertion plan, (2) a set deletion plan, or (3) a list deletion plan.

This ambiguity can be easily resolved if one more closely expanded the structure of `app`, showing that the structure is an insertion rather than a deletion plan on the basis of the lack of an iteration which a deletion would require. A reduction in the range of `app` would result, leaving only the value $\mathbf{Putin}_{Set}$ in the range. A revision of `isrt` with respect to `app` would now result in a singleton value $\mathbf{Insert}_{Set}$ remaining, and subsequently `mbr` could be reduced to only $\mathbf{Member}_{Set}$.

This example problem is completed with the successful construction of a single mapping to the given program plan library. The source consisting of three slices is an instance of an $\mathbf{Insert}_{Set}$ program plan with two primary sub-plans $\mathbf{Member}_{Set}$, and $\mathbf{Putin}_{Set}$. $\mathbf{Insert}_{Set}$ occurs in the library fragment only as a part of the **Set** abstract data type plan group, and further as part of the **Collection** abstract data type plan. No other interpretations are possible given the knowledge constraints and structural constraints of this example.

8.3 The Modeling Process

A Program Understanding CSP (PU-CSP) is formulated via four distinct steps shown in Figure 8.7, and which may be explained as follows.

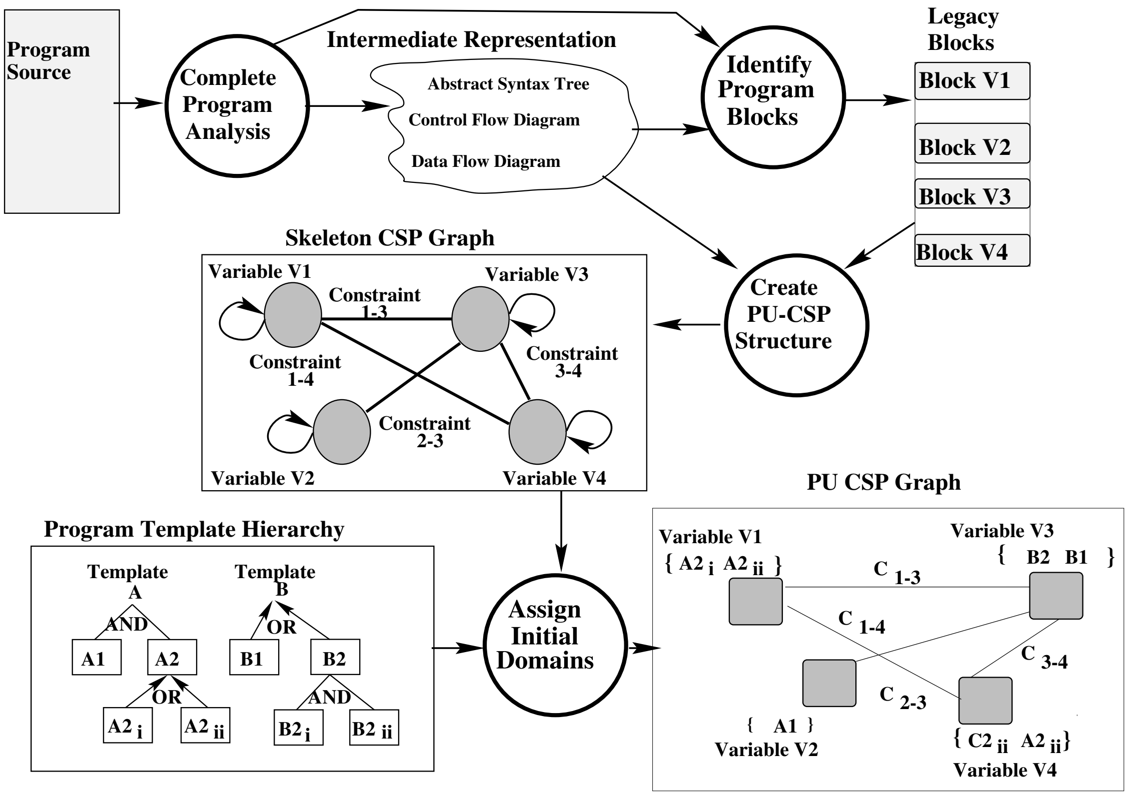


Figure 8.7: PUCSP Formulation; CSP Graph exploded in Figure 8.8

Step 1. First, the source is pre-processed creating a set of artifacts that describe some precise interrelationships in the source. These relationships include data-flow relationships between functional blocks, control-flow among the functional blocks, and the creation of an abstract syntax tree in an intermediate abstract language via parsing of the source.

Step 2. Second, the source code is partitioned according to existing program decomposition methodologies (or actual program procedural structure, if applicable) into spatially localized blocks of code which are known to exhibit functional relationships among one another, and cohesive properties within one's boundaries.

Step 3. Third, a skeleton CSP is formulated consisting of one variable for each identified source block, and constraints between these variables are derived from the intermediate representation level artifacts. Each variable is 'typed' via the addition of reflexive constraints on the variable which describe properties of the block such as *kinds* of input or output.

Step 4. Finally, each CSP variable is compared against the templates in the program plan library. Any templates which share input and output characteristics with that of the variable are added to the variable's domain.

Figure 8.8 shows an example formulated PU-CSP in which the domains of each variable are shown as instances identified in the program template hierarchy. During discussion of the PU-CSP two distinct types of constraints will be used. The first type of constraint is *structural constraints*, examples of which are depicted in Figure 8.8 as inter-variable constraints, or exactly those constraints derived from the intermediate source representation. Structural constraints describe how program components are structurally related. The second constraint type is *knowledge constraints*, examples of which are depicted in Figure 8.8 as compositional and specialization constraints in the program template hierarchy. These constraints describe how program plans or templates may fit together to form larger (more abstract) plans in this domain. These may be summarized in the following way.

- *Structural* constraints are determined from the source code. They include such things as scope or called/calling relations, precedence relations, or shared informa-

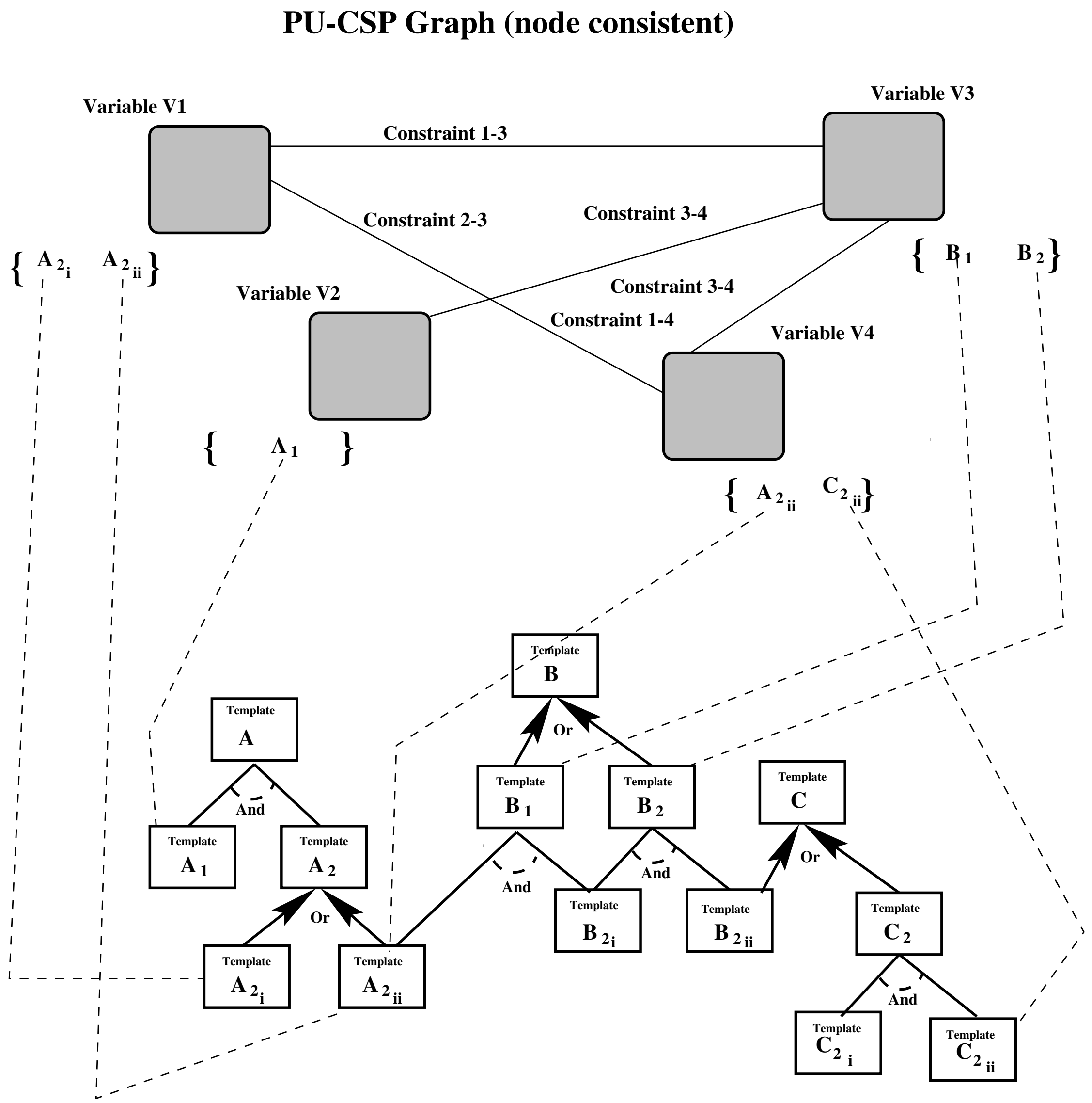


Figure 8.8: PUCSP Graph

tion relations between component blocks. For instance, in the source in Figure 8.1, the **print** statements appear within the scope of **for** statements, **declarations** precede their initial **assignment**, and print statements act upon array positions indexed by corresponding **for** statement indexes.

- *Knowledge* constraints are independent of the source code. They are program plans restricted in their relationship by the AND/OR structure given in the plan library. AND constraints are for composing program plans into higher level plans, and OR's are for specializing an abstract plan in one of several ways. Assigning one program plan as an explanation of a particular PU-CSP variable constrains consistent assignments of other component variables.

  As an example of a knowledge constraint mandated from the library structure, if a variable corresponding to program component **A = "string 1"** in Figure 8.1 were instantiated to program plan **builtin-char*-copy** as shown in Figure 8.2, then it is consistent to assign the last **for-loop** variable an explanation of **print-string**, where the strings are the same.

A solution to the PU-CSP is an assignment to each variable by one program plan component in the plan library, such that no *structural constraint* from the source code, or *knowledge constraint* from the plan library is violated.

PU-CSP program understanding provides a convenient framework for interpreting Quilici's *index* tests [Quilici, 1994] in which candidate explanations are suggested based on the identification of "key" plan components. One may view such heuristically-guided constraint applications as one part of a CSP search control strategy as described in Chapter 4. It is possible to similarly interpret Quilici's *specialization* tests which mandate a preference for maximally-specific explanations. These tests may be regarded as specific instances of knowledge constraints that are used to systematically reduce the range of domain variables in a hierarchical CSP. Quilici's *inference* tests identify "related" program plan templates according to earlier component instantiation. In CSP terminology, inference tests may be seen as a special kind of variable-ordering heuristic in which these "related" program components are explained "next" in a general search strategy.

The program template hierarchy is composed of hierarchically related plan templates[7]. A template plan may be broken down into several sub-plans, in which case this is recorded as an **And** relationship between the sub-plans and the parent plan. Further, any required structure between the sub-plans such as necessary ordering, data-flows between the sub-plans or control-flow between the sub-plans is recorded with the **And** relationship. Similarly, a template plan may be a specialization of another plan (or one of many such specializations), and in this case the constraints that constitute the specialization such as restriction of variable type or a particular restriction of data or control-flow is recorded with the **Or** relationship. Figure 8.9 shows a simple **And** example in which **Template A** is composed of two sub-plans A_1 and A_2 where A_1 provides the data-flow r which A_2 requires. In addition, a simple **Or** example is given in which **Template A** may be specialized by either of the plans B_1 (which exports n in addition to the primary exports of B) or B_2 (which exports p).

8.3.1 General Hierarchical Constraint Satisfaction Model

In Section 4.3.2 of Chapter 4 I introduced the notion of consistency in a constraint graph. In particular, I discussed the notion of arc-consistency. Arc-consistent graphs have the property that all domain values for the source variable in a relation with a target variable have at least one consistent mapping with a target variable domain value. This property can be thought as a non-search reduction in the overall combinatorics of solving

[7] For a formalization of hierarchical planning knowledge, see [Yang, 1990].

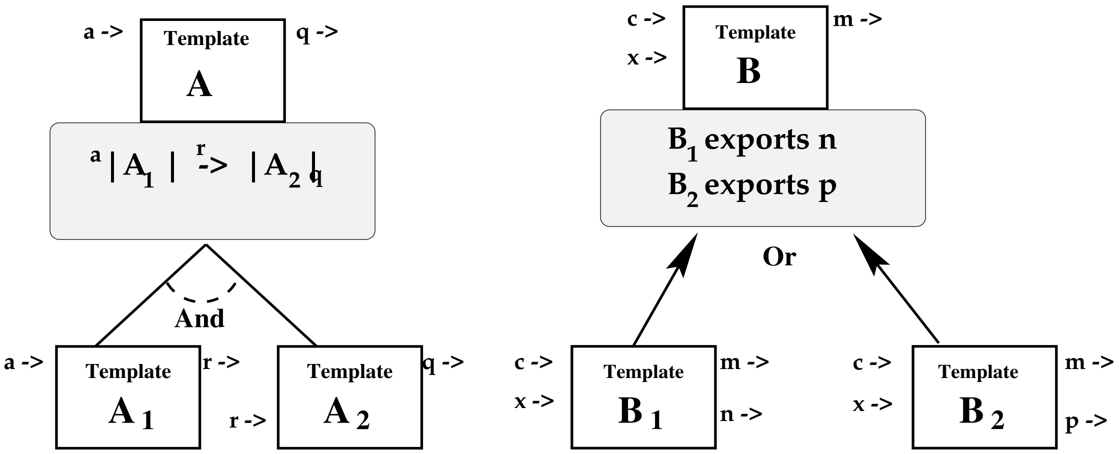


Figure 8.9: Library knowledge constraints

a particular problem. Well constrained problems can result in a large reduction of the size of variable domains in such a fashion. Traditional algorithms for arc-consistency are concerned with simple discrete domain ranges. In other work [Mackworth *et al.*, 1985] domain values are considered in which a value may in fact represent a set of domain values. The algorithm described in this earlier work is known as HAC or Hierarchical Arc-Consistency algorithm. In fact, the domain values may be considered to be a limited form of hierarchy consisting of is-a relationships. More specifically, any domain value with children can be categorized specialized by exactly one of its child values.

Hierarchies are formed in such a way that a failure of a given relation at a high level signals that all children will subsequently fail to satisfy this same relation. There is, therefore, no need to re-apply this constraint to those values. Problem domains supporting such a decomposition allow for significant reduction of the number of constraint checks for particular problem instances. Mackworth states:

> Indeed, for many real problems the domain elements often cluster into sets with common properties and relations. Those sets, in turn, group to form higher level sets. This clustering or categorization into "natural kinds" can be

> represented as a specialization/generalization (is-a) hierarchy. [Mackworth *et al.*, 1985, p. 119]

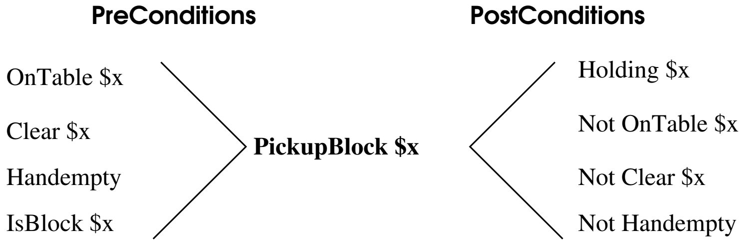
Observe that the program understanding domain fits this suggested need for hierarchically structured domain values. In particular, domain values will need to represent possible program-plan explanations of given source code components. In most recent program understanding and plan recognition research efforts, plan libraries are represented as accumulations of program plans structured with **is-part-of** relationships (or conversely *has-parts*), and **is-a** relationships. Consequently, any constraint-based algorithm supporting domain values that can be structured in these ways must also be capable of dealing with the additional complexity of reasoning with these relationships.

Making a constraint graph "more consistent" is a limited reasoning process which systematically limits the possible solution set of the problem. Considering domain values as hierarchies rather than simple discrete values implies that one must be capable of not only keeping or deleting a particular hierarchy, but also of simplifying a given hierarchy.

The remainder of this section is structured as follows. First, I briefly discuss the issues of hierarchical representation and justify the particular assumptions made for representation. Second, I describe a novel algorithm, with several variations, designed specifically to induce arc-consistency in a constraint graph composed of domain values which may be structured through *is-part-of* (And) and *is-a* (Or) relations. In contrast to the (or-only) consistency property HAC imposes, I refer to this expanded consistency as And/Or-consistency or AO-consistency. In Section 8.3.2, I present a pair of program understanding examples of the operation of hierarchical PU-CSP, and in Section 9.1 I provide an extended generic hierarchical example from which I describe the operation of hierarchical CSP.

8.3.1.1 Hierarchical Domain Representation

Earlier works have frequently (and often loosely) discussed *abstraction hierarchies* in what Wilkins [Wilkins, 1988] defines as two different senses. I outline these two conceptions as *hierarchical approximation* and *hierarchical decomposition* in the following way.



Criticality Values

OnTable $x	1
Clear $x	1
Handempty	2
IsBlock $x	0
Holding $x	n/a

Figure 8.10: A single action with criticality hierarchy

Early planning systems [Sacerdoti, 1974] used *abstraction* in the sense that hierarchies of actions were created based upon precondition and postcondition elimination. The conditions of an action were assigned a numeric value known as a "criticality". At the highest levels of abstraction, an action was viewed only in terms of those conditions with a criticality at least as great as the current abstraction level (starting at 0 with the most abstract level, and increasing to a level corresponding to the maximum criticality value assigned). An action condition is then considered more "abstract" (or perhaps more integral to the action) if it had a small criticality value and more specific (or perhaps more of a detail to the action) if it had a large criticality value. I detail one such Blocks-World action (**PickupBlock**) and its associated criticality values in Figure 8.10, and the resul-

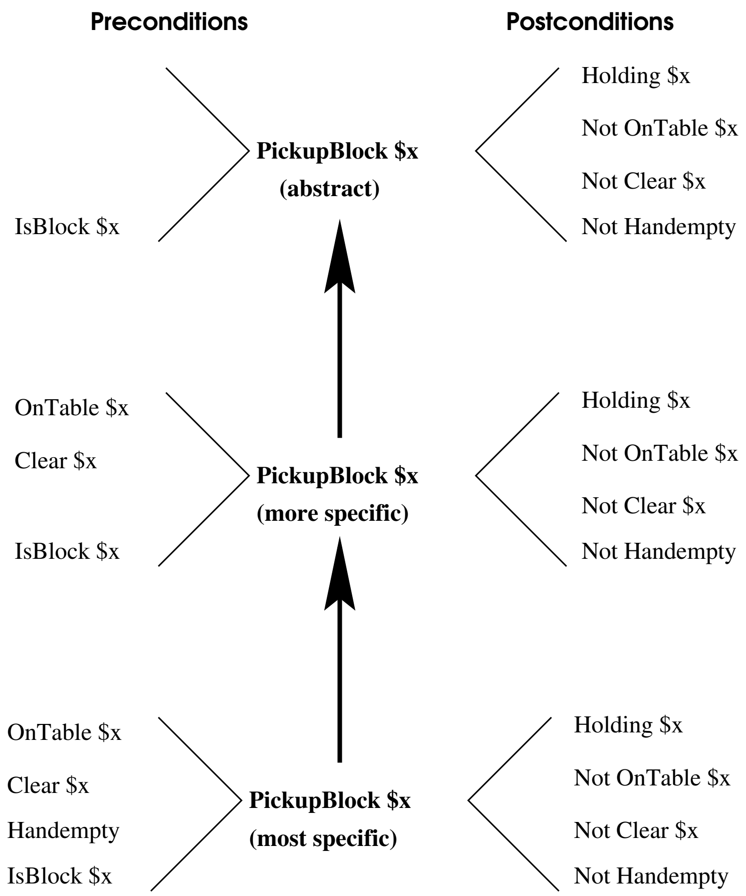


Figure 8.11: Criticality-based action hierarchy for **PickupBlock**

tant action hierarchy in Figure 8.11. The abstraction of an action by such *precondition elimination* [Sacerdoti, 1974, Tenenberg, 1988] may be viewed more accurately as action approximation, and I refer to this procedure and resulting hierarchy as **hierarchical approximation**.

In plan recognition (see [Kautz and Allen, 1986], [Kautz, 1987], [Carberry, 1990b], and [Ardissono and Cohen, 1996a]), it is common to represent abstraction in terms of a view in which a high level action (or plan) might be decomposed into a set of lower-level actions that together form a plan for the completion of the high-level action. In [Ardissono and Cohen, 1996a, Ardissono and Cohen, 1996b] the abstraction hierarchy is extended to better represent the shared portions of specializations which constitute an "abstract" action. In Figure 8.12 I detail a decomposition hierarchy for the same high-level plan **PickupBlock**. The specification of an action by replacing it entirely with a plan consisting of partially ordered sub-actions may fit more closely with the common "intuition" of what abstraction means. I refer to this structuring of actions as **hierarchical decomposition**.

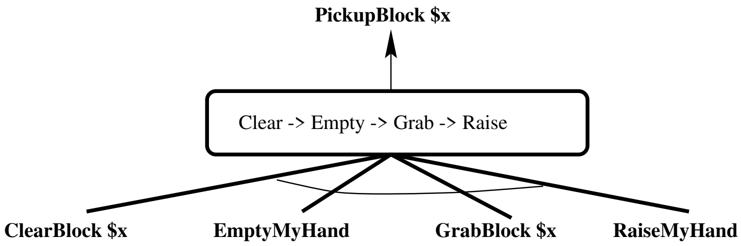


Figure 8.12: A simple decomposition hierarchy

In HAC described earlier, a hierarchical view of domains was encountered in which the objects of the domain were structured such that each high-level object description was either *ground* (at the lowest level of abstraction possible), or it would be specializable into two (or, in general, two or more) objects at the next lower level. This structuring is similar to the hierarchical approximation approach in so far as the child objects specialize the parent objects through possession of properties that extend the child over the parent, and consequently differentiate the child from each of its siblings. In this way, a parent object may be considered to be representative of a class that includes each of the child objects as members. In Section 4.3.2 it was seen that a domain object found to be inconsistent with all of the objects in another related domain can be removed without loss of any potential solutions. HAC combined these two conceptions of representation and consistency to give a new algorithm for propagating consistency in problems where the domains are arranged in a limited hierarchical manner.

While the HAC algorithm effectively exploits a structure of hierarchical approximation possessing a particular distribution of objects and sub-objects, this hierarchical consistency algorithm does not incorporate the notion of abstraction interpreted as hierarchical decomposition very well. For example, one wonders what it would mean in such an algorithm for a particular domain object to be defined such that it may be decomposed into an ordered, constrained, set of sub-objects that may be viewed as a plan.

Extending the HAC representation

The hierarchical approximation approach defines a hierarchy in which a particular object is *specializable* into exactly one of a set of possibilities. I label this type of parent-children specialization relationship as an "OR" or **specialization** relation. In contrast, the hierarchical decomposition approach defines a hierarchy in which a particular object is *specializable* into a plan composed of many children arranged in a particular way. Each of the children play a particular *role* in the specialization. I label this parent-children specialization as an "AND" or **decomposition** relation. Combining these two approaches, one may arrive at a model of hierarchical specialization that contains branch points of both "AND" and "OR" types. Figure 8.13 on page 216 shows these relations. First, PLAN A specializes into one of PLAN A_1 or PLAN A_2. Next, PLAN B decomposes into the composite of PLAN B1 and PLAN B2. Finally, a given plan such as PLAN C may have parents through either a specialization relation, or a decomposition relation. In the event of specialization, a particular instance of PLAN C is assumed to be capable of only specializing more than one single parent plan. Similarly, in the event of decomposition, a single instance of PLAN C may be a sub-part of more than one parent plan.

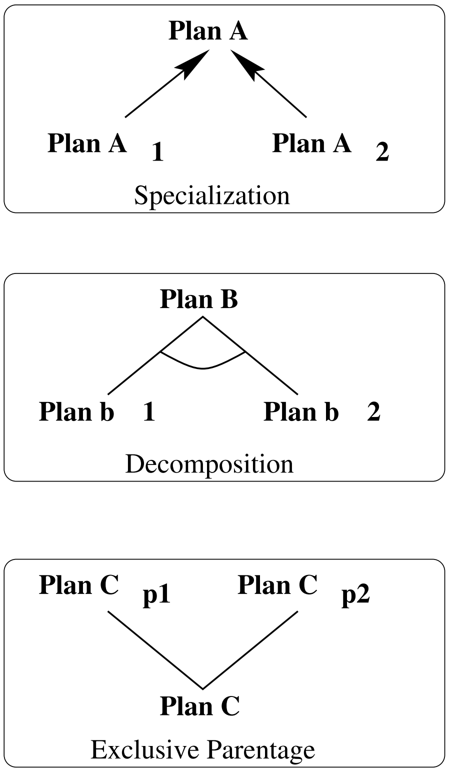


Figure 8.13: Specialization and decomposition represented

8.3.1.2 And/Or Arc-Consistency Algorithms

In a CSP with simple, discrete domain values, arc-consistency algorithms check a given constraint for each source domain value with the range of target values. In the absence of even one constraint success (or *justification*) for that source domain value, it may be deleted. A side-effect of this type of algorithm is that it is possible to generate a set of justification links which specifically records the pairs of source and target domain values consistent with respect to a particular constraint.

With hierarchically structured domains, the problem is essentially unchanged. The primary difference is that the application of a particular constraint between a given source and target domain value has a result that is considerably more complicated than simple success or failure. For example, if one considers a source value S with two specialization possibilities S_1 and S_2, and a target value T with two specialization possibilities T_1 and T_2, and a constraint that ranges over S, T, and their specializations, then constraint values of "true", "false", or "depends" are possible for each constraint evaluation between hierarchical domain values at corresponding levels. While "true" or "false" have obvious meanings, "depends" is unusual. It is possible in such a hierarchical structure to define a constraint valuation at a particular level in terms of its success or failure at subsequent sub and super levels. Specialization links are like "or" relations - if something is a T, then it is certainly either a T_1 or a T_2. The consequence of this relationship is evidenced through the simple example. For S and T a given constraint C might yield a "local" value of "depends". The actual answer for $C(S, T)$ might have to be determined by a subsequent evaluation of C for the children of S and T. If one considers the logical entailment of the hierarchy, $C(S, T)$ can be said to be "true" if and only if one of $C(S_1, T_1)$, $C(S_1, T_2)$, $C(S_2, T_1)$ or $C(S_2, T_2)$ is "true".

Typical arc-consistency algorithms are constructed in the following way. First, a basic subroutine *revise* is defined which revises a given source domain with respect to a given target domain by removing any source domain value which is incompatible with *any* target domain value. Next, the *revise* subroutine is used repeatedly until there is no source domain remaining which can be reduced. If no remaining domain may be reduced with respect to any target domain, the problem graph may be said to be arc consistent.

The extension of an arc-consistency algorithm to hierarchical domains follows the same basic approach. The major parts of such an algorithm may be summarized as follows.

1. A *revise* algorithm is composed based on the premise that two hierarchical values are consistent only when their corresponding hierarchies are consistent to a given

hierarchical depth. In the case of **and**-type hierarchical components, both must succeed, and in the case of **or**-type components, at least one must succeed. The revise algorithm is built through repetitive application of one primary sub-part, *apply*.

2. An *apply* algorithm applies a given constraint between a single pair of hierarchical source and target values. A success at the basic or root level is only achieved if the constraint holds at the basic level, and both *up* against the source and target parent sub-trees, and *down* against the source and target child sub-trees.

While a source value in a discrete domain is either retained or deleted during a revision, this is not the only possible result in a hierarchical algorithm. It is possible that some portion of the hierarchy of a particular value may be removed, while the root value is itself retained. For instance, if one *or*-child of a root value was inconsistent with the appropriate sub-trees of any target value, then that child could be pruned. This process is broken up into two stages.

1. The first stage is *marking*, performed during a source-target revision. When a hierarchical child or part-part of a source value is determined to be inconsistent with a given target hierarchy, that source value is "marked".

2. The second stage is *simplification*, performed after a given source value has been checked against the entire range of target values. Any source value marked a number of times equal to the number of possible target values may be deleted.

Some arc-consistency algorithms attempt to "remember" the constraint application successes encountered during revisions. For instance, it is possible to build a *justification graph* in which the successes of a constraint application between source and a variety of target values are recorded. It may be said that those target values "justify" the continued existence of the source domain value. If one of those target values is removed through a later revision, but others remain, then the source value remains justified. However, if all target "justifying" values are removed, then the source is no longer justified and may itself be deleted. In this way, it is possible to "chain" deletions, removing domain values that are known to not participate in any possible solution, without subsequent constraint application. I refer to this chaining process as *DeleteSourcePropagation*.

Since we are dealing with hierarchical values, the *DeleteSourcePropagation* problem is complicated somewhat. For instance, if a source domain value S_1 were justified by a target domain value T_1, then one knows that a constraint C between S_1 and T_1 holds to a particular hierarchical depth. That is, the hierarchies of S_1 and T_1 have been determined to be hierarchically consistent with respect to C. However, what if the hierarchy of the T_1 domain value is pruned? In this case, the justification relationship needs to be re-verified in case the hierarchy changed affects the C evaluation. A similar chaining process may ensue in which some domain values are hierarchically simplified, and other domain values are simply deleted. I refer to this chaining process as *KeepSourcePropagation*. *DeleteSourcePropagation* and *KeepSourcePropagation* are described in Sections 9.2.2.2 and 9.2.2.3 of Chapter 9.

One other distinction is of prime importance in understanding hierarchical arc consistency algorithms. Some discrete arc-consistency algorithms aggressively check all target values for a given source domain revision, while others simply step through the target values until a justifying target value is found. It is possible to formulate hierarchical algorithms in this same fashion. I thus specialize the revision algorithms as either *aggressive* or *stepped* (see Tables 9.13 and 9.14). Subsequent versions of of *Delete-SourcePropagation* and *KeepSourcePropagation* are similarly implied. The algorithms *(Delete/Keep)SourcePropagation(Aggr/Step)* are described in detail in Appendix C in Tables C.1, C.2, C.3 and C.4.

8.3.2 Hierarchical CSP and Program Understanding : An Example

In Chapter 9 I describe the hierarchical arc-consistency algorithms introduced in this general overview, and provide specific examples of the operation of these algorithms. In this section I demonstrate the utility of such an algorithm in the program understanding process. We have seen how constraint satisfaction algorithms may be applied to partial local program explanation. In order to apply constraint satisfaction to the problem of integrating these local explanations into a global view of a source program one must be able to match program component/variables to a hierarchically structured program library.

In this chapter I have described in some detail the operation of a hierarchical constraint propagation method in terms of generic examples and explicit algorithmic descriptions and explanation. In particular, Sections 8.3 and 8.3.1 outlined the overall PU-CSP model and the hierarchical domain structure and Section 8.3.1.2 presented a detailed generic example of solving hierarchically structured constraint satisfaction problems. In this generic example, it was demonstrated how it was possible to limit both the range of domain values for a particular variable in the spirit of previous arc-consistency algorithms, and also reduce the hierarchy of particular domain values themselves. This reduced hierarchical structure provides a "clearer" or more restricted domain value, and consequently a more constrained solution. Applying this conception directly to the program understanding problem results in a "clearer" or more precise explanation of a series of program components each represented by a CSP variable. In this section I extend the illustration of the effectiveness of this hierarchical approach with a pair of simple examples rooted in the program plan domain.

8.3.2.1 Downward Hierarchical Revision

Consider the small hierarchical plan library fragment shown in Figure 8.14. This selection of plans have been chosen from the image processing domain. **Calc-Area-Rectangle (C-A-R)** has two sub-parts, **Find-Height-Width (F-H-W)** which returns the two size parameters from a given rectangle object, and **Multiply (Mult)** which returns a product of height and width as an area measurement, and **Input-Object (I-Obj)** which identifies an object of a particular type in a given scene. **Input-Object** has two specific instantiations, **Input-Rectangle (I-Rect)**, and **Input-Circle (I-Circ)**.

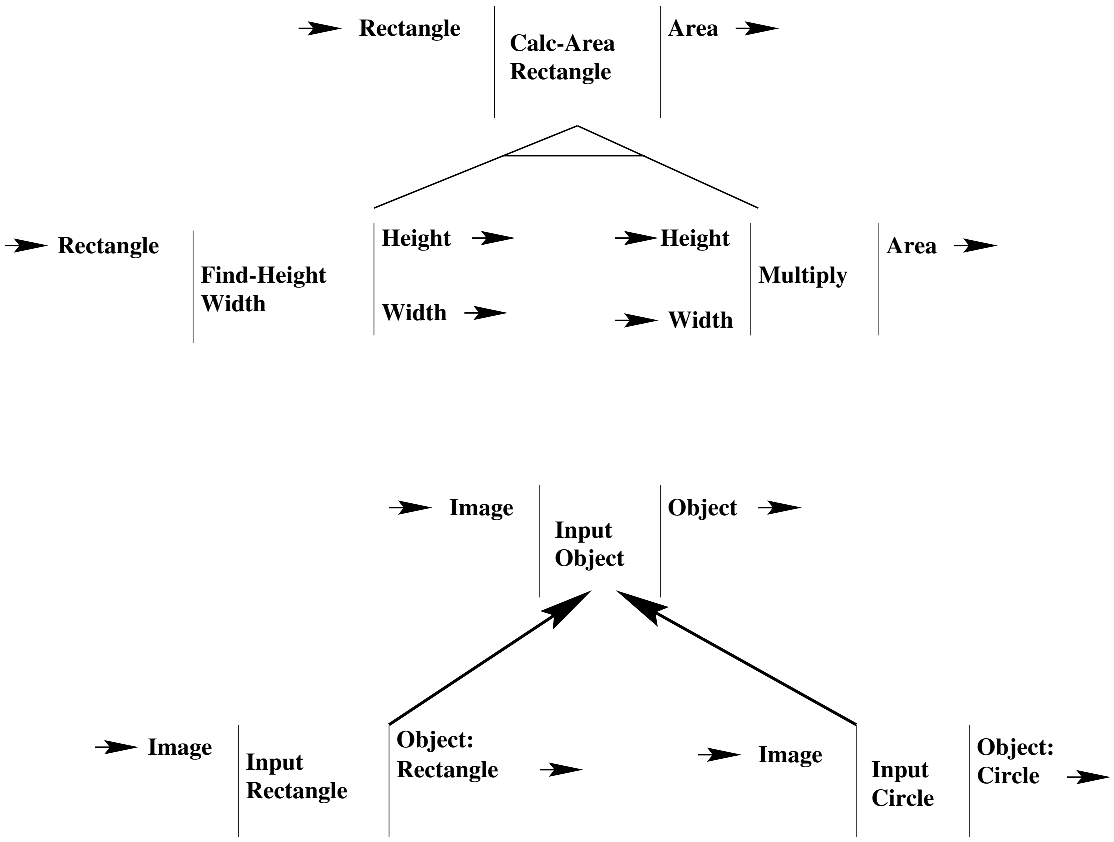


Figure 8.14: Image Processing Plan Library Fragment

Suppose that there exist two blocks of code *Block 1* and *Block 2*. Block 1 has an input typing of a super-type of Rectangle and Circle, *Shape*, and an output typing of *Area*. Block 2 has an input typing of **Image** and an output typing of **Object**. In addition, a structural data-flow constraint is identified such that the **Object** input to Block 1 necessarily originates from Block 2. One PU-CSP formulation is shown in Figure 8.15 in which Variable 1 corresponds to Block 1 and Variable 2 to Block 2. According to an initial variable assignment based on input and output typing, the Variable 1 domain includes only **C-A-R**, and Variable 2 only **I-Obj**. We wish to apply the revision algorithm introduced in Section 8.3.1.2 (for detail, see Section 9.2.2) in order to revise the domain of Variable 2 with respect to Variable 1. We follow the execution of such an algorithm as follows.

Step 1. First, a revision is undertaken at depth 0, verifying the compatibility of target **C-A-R** and source **I-Obj** with respect to the structural constraint between Block 1 and Block 2. This constraint holds "locally" since **I-Obj** is capable of supplying an **Object** to **C-A-R**.

Step 2. Next, the revision is undertaken at level 1 **down**. Any use of **Object** at the next level needs to be verified for compatibility between the two potential plans. The revision between **C-A-R** sub-plan **F-H-W** and **I-Obj** specialization **I-Rect** succeeds in that **I-Rect** is capable of supplying **Object:Rectangle** for **F-H-W**. **C-A-R** sub-plan **Mult** does not make use of any specialized plan portions from **C-A-R** and consequently succeeds by default. Thus both necessary sub-parts of domain value **C-A-R** succeed in the downward application against **I-Obj** left specialization **I-Rect**.

Step 3. The second downward application at depth 1 is between target **C-A-R**'s children and source specialization **I-Circ**. **I-Circ**'s specialization of **Object** is **Object:Circle** is not compatible with **C-A-R** sub-part **F-H-W** Rectangle input, and consequently, domain sub-value **I-Circ** fails and is marked for potential deletion.

Step 4. The overall constraint application between **I-Obj** and **C-A-R** succeeds as a result of the compatibility between **I-Rect** and the sub-parts of **C-A-R**. Since there are no further domain values of Variable 1 to check value **I-Obj** against, the *Simplify* sub-algorithm results in the removal of specialization **I-Circ**, and the explanation hierarchy of Block 2 is reduced to a single specialization of **I-Obj**, **I-Rect**.

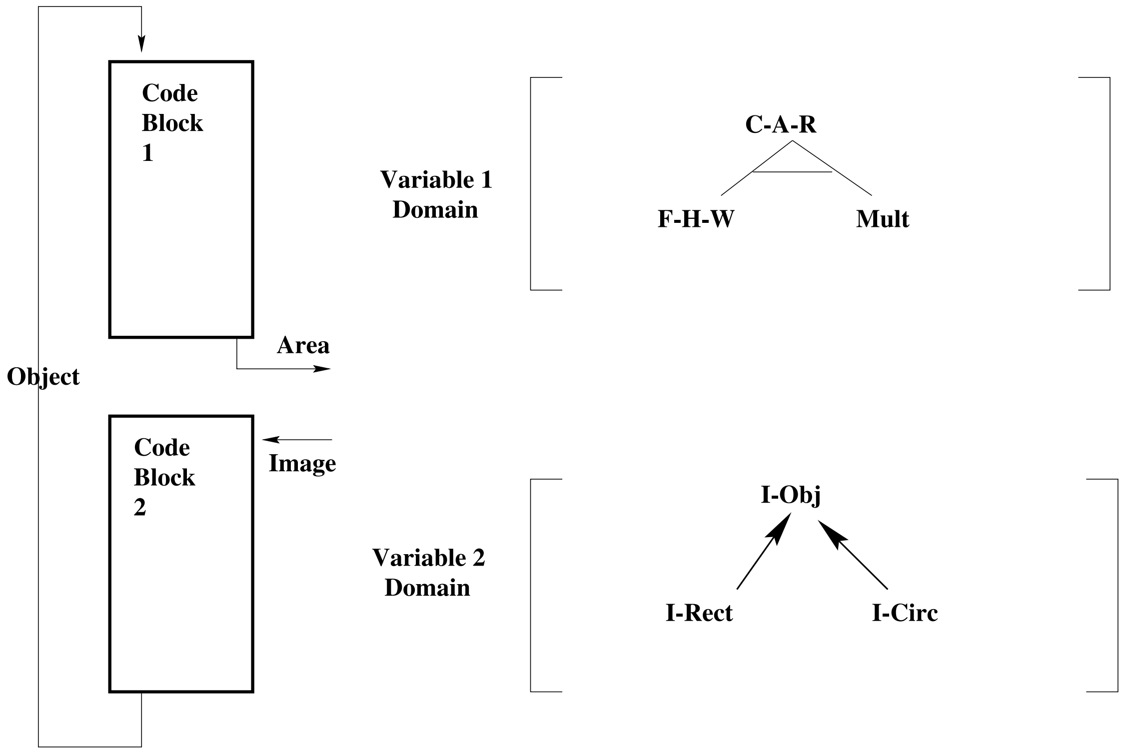


Figure 8.15: Example 1 PU-CSP formulation

8.3.2.2 Upward Hierarchical Revision

Consider the small hierarchical plan library fragment shown in Figure 8.16. This selection of plans have been chosen from an extension of the previous example to include plans to exploit the area calculations. **Calc-Area-Rectangle (C-A-R)** has two possible "uses" - as a sub-part of **Calc-Area-Composite-Object (C-A-C-O)**, or as a sub-part of **Approx-Area-Circle (A-A-C)**. A new plan **Calc-Area-Cost (C-A-C)** has two possible "uses" - as a sub-part of the larger **Calc-Cost (C-C)** plan, or as a part of **Calc-Internal-Cost (C-I-C)**.

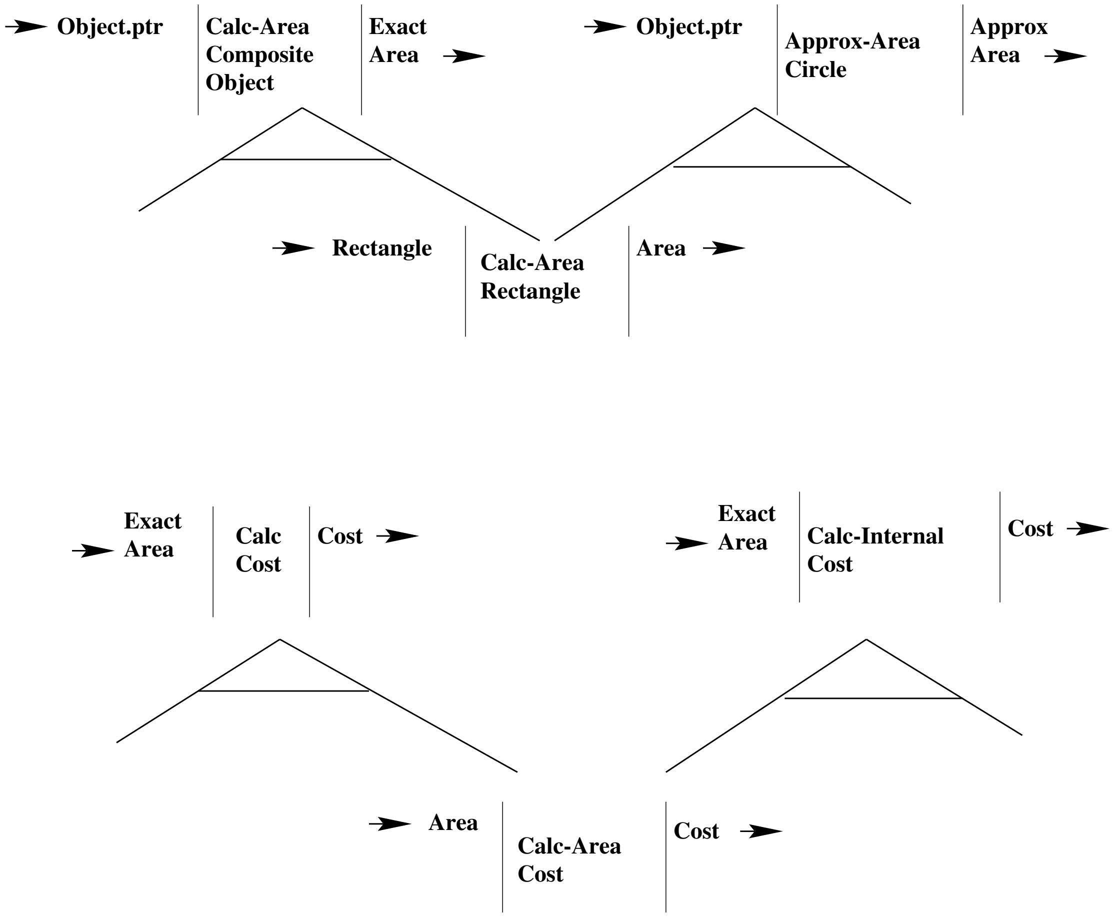


Figure 8.16: Example 2 Area Management Plan Library Fragment

Suppose once again that there exist two blocks of code *Block 1* and *Block 2*. According to the input and output typing of these blocks, new variable domains are created such that Variable 1's domain contains only **C-A-R** and Variable 2's only **C-A-C**. A structural constraint is identified such that Block 1 provides an **Exact-Area** data structure to Block 2. One PU-CSP formulation is shown in Figure 8.17 in which Variable 1 corresponds to Block 1 and Variable 2 to Block 2. Once again we wish to follow the revision of a variable domain. This time we wish to hierarchically revise Variable 1 with respect to Variable 2.

Step 1. At level 0, **C-A-R** successfully provides **Area** for use by **C-A-C** and so the constraint holds "locally".

Step 2. Ascending to apply the constraint checking algorithm to the parents of the source and target, it may be observed that source value **C-A-C-O** succeeds with respect to both target values **C-C** and **C-I-C**.

Step 3. On the other hand, the source value **A-A-C** fails in that **Approx-Area** cannot be used as **Exact-Area**.

Step 4. The identification of at least one source parent with the target results in an overall successful application for **C-A-R**. **A-A-C** is, however, marked and deleted by the *simplification* algorithm previously outlined.

8.3.3 One Unified Algorithm for Program Understanding

To this point I have introduced at a high-level the range of algorithms required to deal with a hierarchically structured domain values. Chapter 9 provides a level of detail for those interested in pursuing the hierarchical CSP algorithms in depth. In this section, I bring all of these algorithms together in an integrated model of program understanding based on the two-phases of MAP-CSP and PU-CSP. The algorithms introduced in Section 8.3.1 extended the discrete CSP model to accommodate the traditional representation of program plans within the context of hierarchically structured program plan

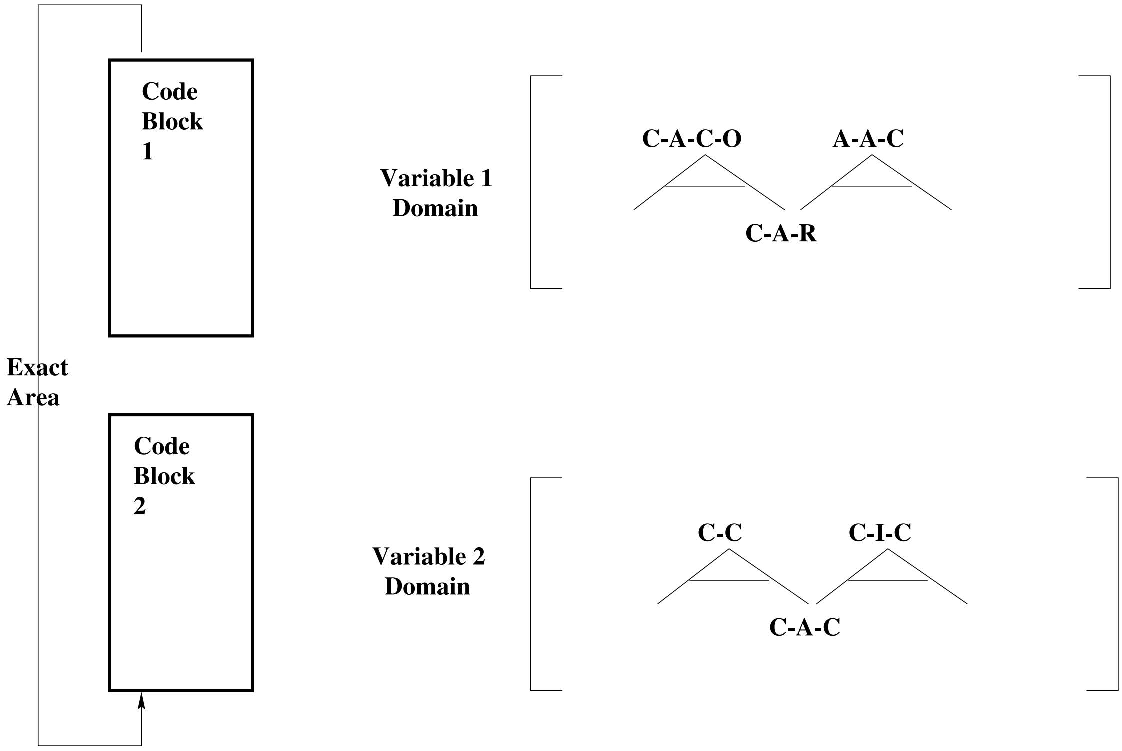


Figure 8.17: Example 2 PU-CSP formulation

libraries. In Section 8.3.2, range of examples were identified which clearly map particular program plan hierarchies within this domain valued representation.

Almost all of the pieces of the program understanding puzzle have now been presented. Program plans are represented hierarchically as part of a larger program plan library. Source code is represented as well constrained syntax trees. Partial local explanations of source are identified through the use of MAP-CSP. In a global sense, hierarchical program plan libraries themselves may be mapped against source program blocks, reducing the possible global explanations through hierarchical arc-consistency (and search) algorithms. What is missing, however, is an integrated (interactive) model of understanding that supports the use of all of these techniques. In this section I outline one such model that accommodates all of these methodologies in a single model. In particular, this model shows how the partial local explanations of MAP-CSP can be used to reduce the overall explanation space of the more global PU-CSP explanation problem. The expert (or user) may direct and control the explanation process interactively as well as select plans to assist in local explanations.

8.3.3.1 Algorithm Understand Explanation

The algorithm *Understand* outlined in Table 8.1 shows the major steps required for integrating local and global strategies. Recall that the PU-CSP variable/block domains D are initialized according to input and output typing information as described in the introduction to PU-CSP setup in Section 5.2. This initial explanation range could next be reduced in Step 2 through application of any of the hierarchical arc-consistency algorithms. In Step 3, a heuristic selection (perhaps by the expert, or according to some estimate of greatest utility) is made of a template in the knowledge library. In Step 4 MAP-CSP is utilized to identify all instances in either the entire source, or perhaps in some subset of the whole source range. In Step 5 the recognized instances are utilized to reduce the range of explanations in the PU-CSP variables. One example algorithm for *MergeRevise* is outlined in Table 8.2. The algorithm revises the variable domains of D according to information obtained through an identified partial local explanation, T_i. The algorithms accepts as input, the T_i template instance including I_{set} the instance set of D variables "hit" in the instance, and produces as output the revised domain values for the variables in D.

This example demonstrates how a local plan instance can be used to reduce the range of explanation - in this case by removing PU-CSP domain values which do not accommodate the plan instance as a sub-component. Step 8 from Table 8.1 reflects the existence of a similar revision phase against the variable domains of the PU-CSP, this time with the new knowledge about the non-existence of a T plan instance in particular variable ranges. Similar algorithms to $MergeRevise$ may be constructed for Step 8. For example, there may be some cases in which the negative MAP-CSP presence of a plan indicate it's actual non-presence (such as perhaps when the connectivity of such a hypothesized plan has a very strong connection to other identified plan portions) and some negative revision can be propagated through the set of partial explanations. On the other hand, in other situations it may be the case that there is less confidence in the canonical ability of a given plan and such a negative conclusion would be an example of over-commitment. In Step 9 some heuristic or expert decision determines whether the iterative process of propagation, local explanation, integration of local instances and merging is to continue. For example, an expert may wish to direct attention elsewhere in the program source, or perhaps has learned enough from this particular analysis. The key to the usefulness of this strategy is that the expert is in control of the process - further research is required in order to better identify which kinds of completion conditions experts would likely employ.

It is important to note that this algorithm is merely an outline of how I have envisaged the integration of local and global explanation strategies. The primary focus of this algorithm has been to incorporate MAP-CSP as a tool which the expert may choose to use or to ignore. Expert generated revisions of the PU-CSP space (or in fact, or identification of MAP instances) can be easily accommodated between Steps 3 and 5, with selective use made of Step 5 and 2 in order to propagate the effect of new knowledge acquisition to the model.

In particular, it is important to note the capacity of the $Understand$ algorithm to exploit expert-provided knowledge during the interactive process. In Table 8.1, user interaction can be provided in the following steps.

Step 1. Domain initialization provides the first, very rough set of possible explanations for each program block independent of the others. The expert may discard any individuals or subset of individuals before search continues. In particular, it may be possible to discard large sets of the values based on the determination of a very-high level goal or domain for the code on the part of the expert.

Step 2. Hierarchical arc-consistency propagation eliminates values or portions of values which have no consistent assignment. Once again it is possible that the expert may wish to prune from some of the explanation sets.

Step 3. During the global explanation process the expert may wish to search for a particular program plan as a "clue" to reducing the number of possible local or global explanations. The selection of a template to search for, and the "focus" within the code in which to search may be computed algorithmically according to some search-based heuristic (such as maximal partitioning of a particular domain, for instance) or directed explicitly by an expert.

Step 4. The identification of partial local explanations in the code can be undertaken by MAP-CSP or in fact directly assigned by the expert. This expert assignment can then be later propagated via PU-CSP. In addition, the use of constraint *relaxation* techniques may be warranted in order to widen the range of possible plan instance identification, either automatically or perhaps through expert-identified constraints to relax.

Steps 5 through 8. The propagation of positive or negative template instance information can be guided by an expert choosing which instances to propagate. For instance, an expert may determine that a given negative result is too uncertain to allow this information to propagate and eliminate other possibilities.

Step 9. The determination of a completion condition is an excellent example of a decision best left to a domain expert. However, it may be the case that an expert would like the automated process to run for a period of time and completion signaled by a detectable condition such as a suitable reduction of possible explanations of some subset of the total problem. These conditions might be identified dynamically by an expert and saved for later use for example.

Algorithm $Understand(L, S, B, D)$;
Input: L library of hierarchical program knowledge, S attributed source rep'n, B set of procedural source blocks, D the variables and domain values *explaining* the procedural blocks.
Output: Incrementally improved explanation of procedural blocks in the context of a given knowledge library.

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 $D := MergeRevise(T_i, L, S, B, D)$;
7 **End Loop**

8 Optionally, MergeRevise negative information about T matches

9 **if** (**not** $Done$)
10 **then GoTo** 2
11 **else Exit**;
12

Table 8.1: The overall understanding algorithm

8.3.3.2 Algorithm MergeRevise Explanation

The $MergeRevise$ algorithm is a reflection of the specific integration process, and demonstrates the usefulness of a unified program plan library. Plan instances used for MAP-CSP matching are members of the hierarchical plan library, and consequently identifying instances of these plans allows us to reason about PU-CSP explanations based on the structure of the library.

Specifically, $MergeRevise$ (detailed in Table 8.2) follows the following procedure.

Step 1. Identify the variable blocks affected by the identified template T instance, T_i - call these the I_{set}.

Step 2. If T_i is completely within a single variable block, then any explanation of this block must contain the partial explanation T_i.

Step 3. 3A. If T_i is completely contained by two variable blocks (V1 and V2), then if it is *possible* to generate a decomposition of T into sub-parts T_1 *and* T_2 where T_1 corresponds to the portion of T in V1, and T_2 to the portion in V2 then do so - if necessary update the hierarchy to reflect this new fact. Note that if this decomposition is *not possible*, for whatever reason, that no conclusions may be drawn.

3B. With T_i decomposed into T_1 in V1 and T_2 in V2, for all explanations $d1$ of V1 where there is at least one child with T_1 as a necessary sub-part, if there exists an explanation $d2$ of V2 such that T_2 is a necessary sub-part, then prune any or-subtrees of $d1$ where T_1 is not necessarily required. More simply, get rid of possible explanations (or parts of explanations) of V1 which do not have a T_1 as a necessary sub-part, and explanations (or parts of explanations) of V2 which do not have T_2 as a necessary sub-part.

Step 4. In the case where T_i is contained in N variable blocks, split the template T into N sub-parts, and remove explanations (or parts of explanations) according to the logic of Step 3.

As a simple example of the potential utility of the integrated MAP-CSP and PU-CSP approaches look ahead for a moment and consider the example PU-CSP detailed in Figure 9.2 on page 238. Consider that a MAP-CSP matching instance identified that there was an instance of a U11 plan within the scope of focus assigned the variable $V2$. Of the domain values in $V2$, only value B3 contains U11 as a sub-part (within T6). Consequently, step 6 of the overall understanding algorithm would reduce the range of variable $V2$ to contain only B3 in the example. Early elimination of B1 and B2 would short-circuit constraints associated with removing B1 and B2 in the traditional propagation manner. In larger contexts, multiple instances of program plans of the same template or even failures of such matching attempts could potentially reduce many variable domains in a similar fashion. Subsequent propagation of constraints over the reduced domain ranges (as per the aggressive or stepped revision algorithm) can be seen as a limited form of "triggered" reasoning, that is, the enforcement of entailed knowledge based on one small piece of acquired knowledge.

Algorithm $MergeRevise(T_i, L, S, B, D)$;
Input: Instance T_i of template T, Library L, Source S, Blocks B, and D variables and domains
Output: Revised variable domains D according to T_i position in S

$NumVarsAffected := sizeof(I_{set})$;
$Case(NumVarsAffected = 1)$
  $V := GetAffectedVar(T_i, D)$;
  if V totally covered by T_i
  then ForAll d **in** domain of V
      Remove d **iff** $d \neq T$;
  else ForAll d **in** domain of V
      Remove Or-subtrees of d where T not necessary;

$Case(NumVarsAffected = 2)$
  $V1, V2 := GetAffectedVars(T_i, D)$
  if $(CanDecompose(T_i) \rightarrow T.1 + T.2)$
  then
    if not((T.1 in V1) **and** (T.2 in V2))
    then ... add new decomposition of T to L
      $L := addNewDecomposition(T, T.1, T.2)$;

    ForAll d_1 in domain of $V1$
      if Exists $d2$ in domain of $V2$ s.t.
        T.1 necc **is-subpart-of** $d1$ **and**
        T.2 necc **is-subpart-of** $d2$
      then Remove Or-subtrees of $d1$ where T.1 not necc req'd;
      else Remove $d1$;

    ForAll d_2 in domain of $V2$
      if Exists $d1$ in domain of $V1$ s.t.
        T.1 necc **is-subpart-of** $d1$ **and**
        T.2 necc **is-subpart-of** $d2$
      then Remove Or-subtrees of $d2$ where T.2 not necc req'd;
      else Remove $d2$;

Note: General Case for N variables also possible.

Table 8.2: Merging partial local explanations to global view

Chapter 9

Hierarchical CSP: A Detailed Solution

One unique feature of the global, integrative, understanding problem or PU-CSP is that it involves a mapping between a variable corresponding to a block of program code and domain values which are members of a *hierarchical* library of plans. In particular, any potential *explanation* of a program block is going to be a hierarchical rather than "simple" value as is the case with discrete constraint satisfaction problems. For example, the explanation of a particular block might be a mapping to a "sorting" plan from the library where "sorting" might later be specialized into one of "bubble" or "quick". In this chapter we detail the new algorithms required to deal with this hierarchical domain structure, and provide a running generic example to illustrate the function of the algorithms.

Throughout this chapter one must recall the terminology of constraint satisfaction introduced in Chapter 4 as well as accommodate some new conceptions. The relevant terminology required for this chapter may be reviewed and introduced as follows.

- *CSP variables* or *variables* refer to the program blocks which one wishes to explain.

In particular, each block corresponds to a variable in a CSP.

- *Domain values* refer to the values which are candidates for assignment to a CSP variable. In the understanding domain, domain values are possible explanations for a program block.

- *Hierarchy* or *plan library* refers to the hierarchical structure or program plans (clichés, or templates). A particular domain value in the PU-CSP is actually a hierarchical value selected with structure as indicated by its presence in the program plan library.

In addition, in Chapter 8 I have introduced several concepts and terms relevant to hierarchical CSP that should be recalled.

- Hierarchical *simplification* refers to the process of removing a portion of the hierarchy of a domain value based on the discovery that that portion is inconsistent with any target domain hierarchy for a given constraint.

- *Justification Links* refer to the links created during constraint application which indicate that a source domain value is "justified" in its continued existence in the domain of the source variable as a result of a satisfied constraint check with a particular target value. A justification link may have multiple targets if it has been determined that the source value is consistent with multiple target values.

- *DeleteSourcePropagation* refers to the algorithm in which the deletion of the last target value involved in a particular justification link for a source value is removed and causes a subsequent deletion of the source value. Similarly, *KeepSourcePropagation* refers to the algorithm called to re-establish justification links in the event that a target domain value has been hierarchically simplified.

- *Aggressive* revision refers to the process of removing inconsistent domain values from a source variable based on checking against *all possible* target variable domain values. In contrast, *stepped* revision refers to the same process with the exception that only target values are checked only until one value is found consistent with a particular source value. A CSP aggressively revised for all source domains may be thought to have a complete set of *justification links* which depict every pair of arc-consistent source and target domain values. A CSP revised in a stepped fashion has only a subset of all possible justification links; however, any undeleted source domain value has at least one such link.

9.1 A Generic Hierarchical Example

In order to better understand the hierarchical domain structure, and for purposes of a running generic example, Figure 9.1 outlines a constraint satisfaction problem with three variables, $V0$, $V1$, and $V2$. $V0$ has two hierarchical domain values $E1$ and $E2$, $V1$ has three domain values $A1$, $A2$ and $A3$, and $V2$ has three values $B1$, $B2$, and $B3$. A directed constraint function $Constraint$ exists among the variables as shown by the constraint arcs. The hierarchical structure of the domain values in each of the three variables is shown in Figure 9.2. The constraint function is defined over the hierarchical values according to the directed constraint function by constraining $V0$ with respect to $V1$ (Table 9.1), $V1$ with respect to $V2$ (Table 9.3), and $V2$ with respect to $V0$ (Table 9.2).

A successful constraint application between two variables results in the construction of a *justification link* from a source domain value to a target domain value indicating that the directed constraint between these two values is necessarily satisfied (to a particular assumed hierarchical depth). It is possible (as is the case with aggressive constraint checking) that more than one source domain value justifies a particular target domain value. Figure 9.3 exemplifies this justification structure as it happens to exist for the given example with respect to variable $V0$ and $V1$ domain values. Figure 9.4 shows a complete justification structure as it would exist if each constraint was applied against all applicable domain values to a hierarchical depth of 2.

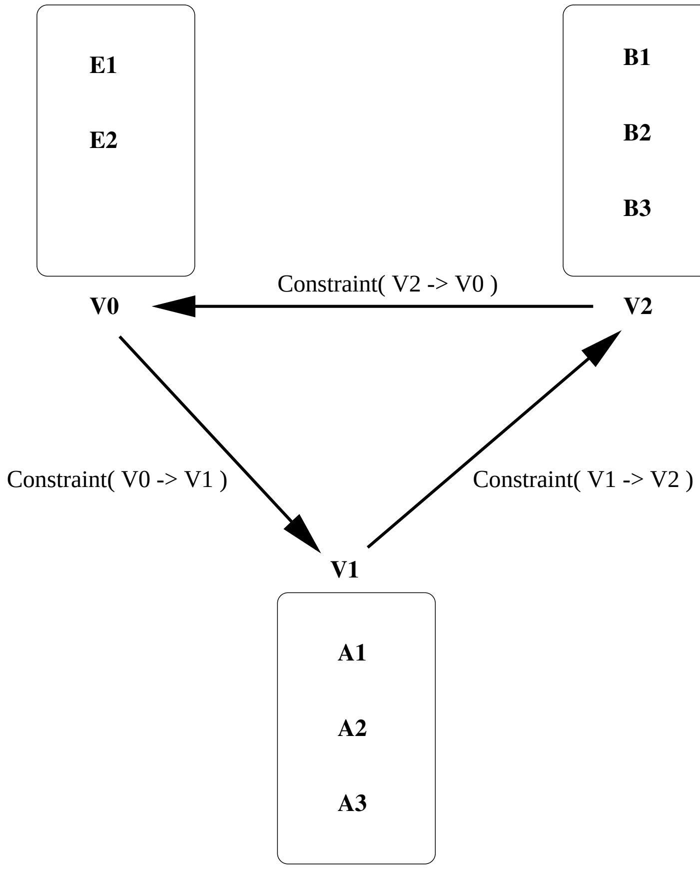


Figure 9.1: An example (flattened) CSP structure

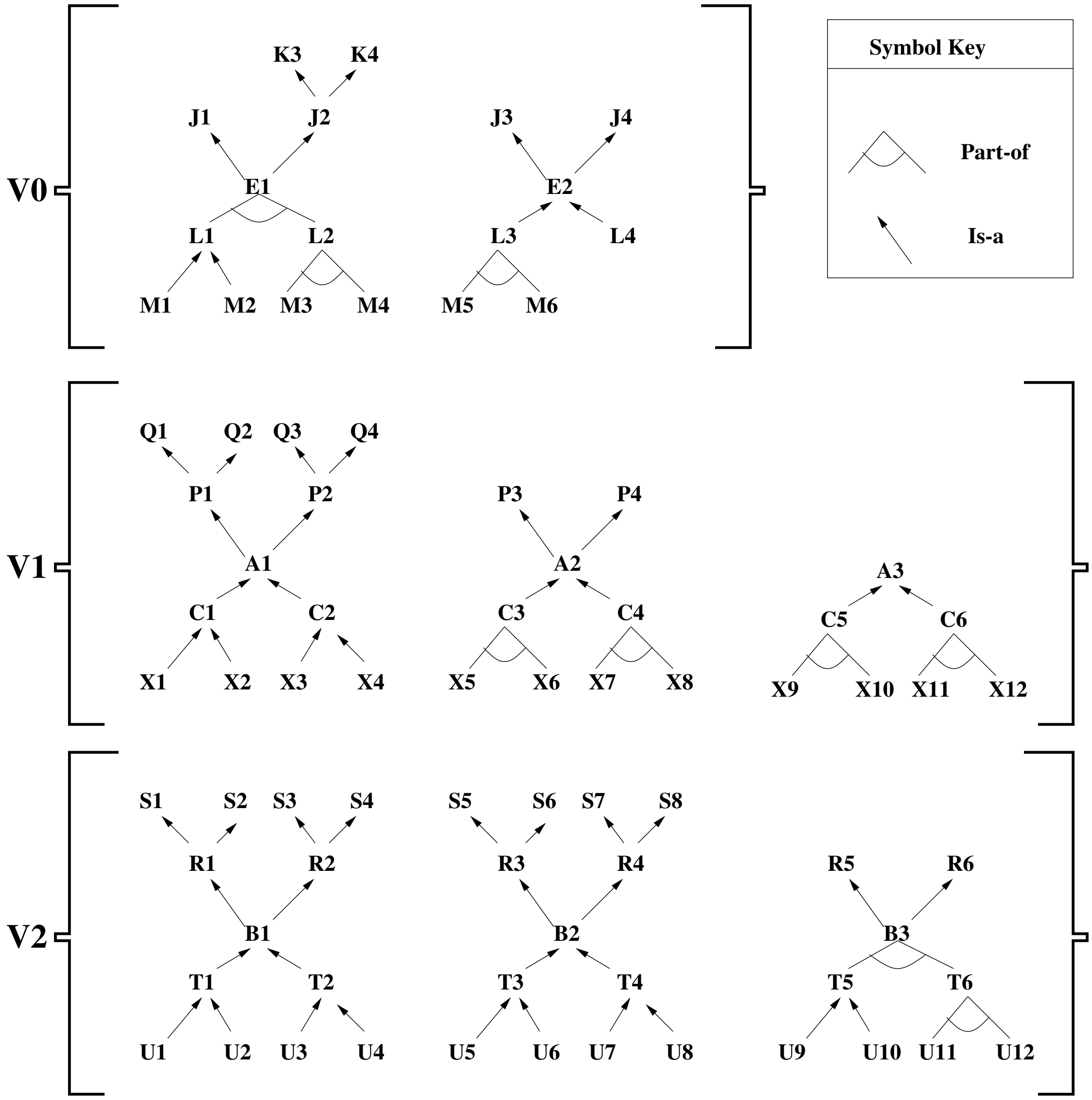


Figure 9.2: An example hierarchical domain value structure

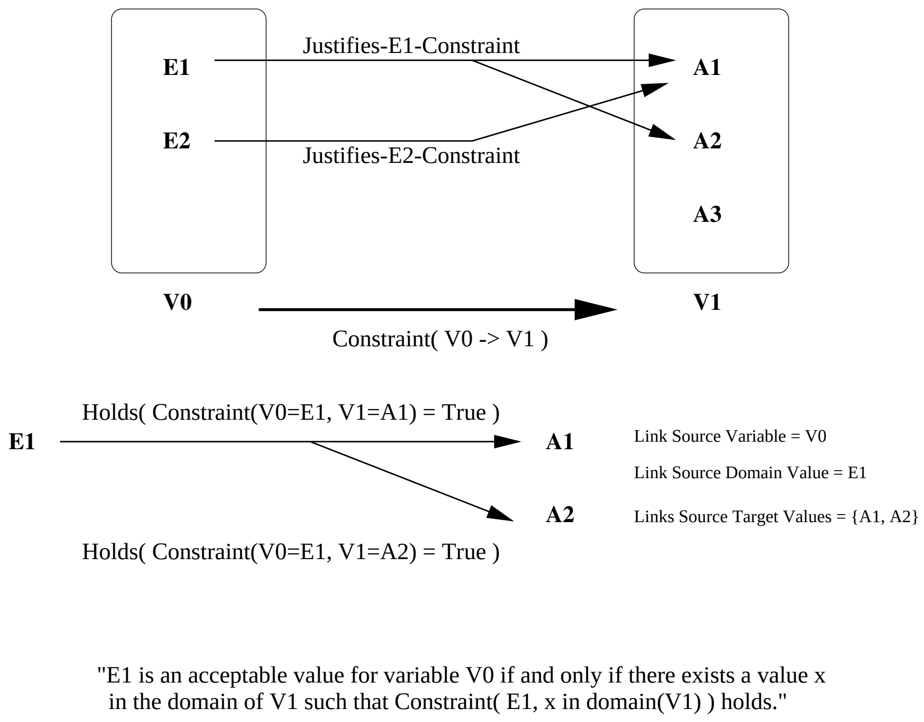


Figure 9.3: Close-up of CSP justification linkage

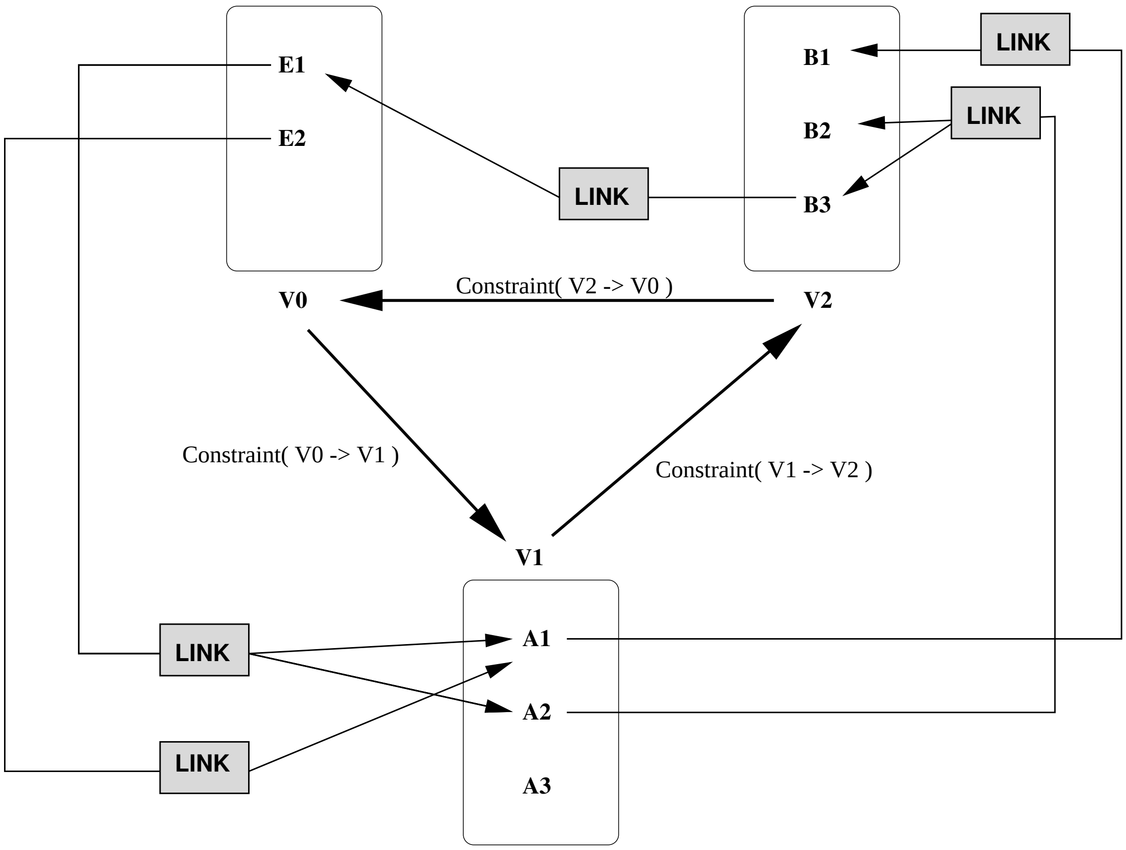


Figure 9.4: Example complete justification linkage

V0 ↓ $V1 \rightarrow$	A1	A2	A3
E1	T	P	F
E2	P	P	P

V0 ↓ $V1 \rightarrow$	P1	P2	P3	P4
J1	T	F	F	F
J2	F	F	T	T
J3	T	F	F	F
J4	F	F	F	F

V0 ↓ $V1 \rightarrow$	C1	C2	C3	C4	C5	C6
L1	T	F	T	F	F	T
L2	T	F	T	F	T	T
L3	T	F	F	F	T	T
L4	F	F	F	F	T	F

V0 ↓ $V1 \rightarrow$	X1	X2	X3	X4	X5	X6	X7	X8	X9	X10	X11	X12
M1	T	F	-	-	T	T	-	-	-	-	F	T
M2	F	F	-	-	F	F	-	-	-	-	T	F
M3	T	F	-	-	T	P	-	-	F	F	F	T
M4	T	F	-	-	P	T	-	-	F	F	T	F
M5	T	F	-	-	-	-	-	-	F	F	T	F
M6	F	F	-	-	-	-	-	-	F	F	F	T

Table 9.1: Example hierarchic constraint f'n between **V0** and **V1**

Given a hierarchy of arbitrary depth, hierarchical constraint checking can yield varying results depending on the depth to which a constraint is checked. Consequently, a new parameter is required for a constraint application - the depth. The example above describes an "or" application downwards in a hierarchy. In addition, it is necessary to check downwards for composition or "and" relationships, and "upwards" against parents. Once one has an algorithm for determining the value of a source/target domain value constraint application, it is necessary to elaborate this into a revision algorithm that will

V2 ↓ $V0 \rightarrow$	E1	E2
B1	F	F
B2	F	F
B3	T	F

V2 ↓ $V0 \rightarrow$	J1	J2
R5	F	F
R6	F	T

V2 ↓ $V0 \rightarrow$	L1	L2
T5	T	T
T6	p	P

V2 ↓ $V0 \rightarrow$	M1	M2	M3	M4
U9	T	F	T	F
U10	F	F	F	F
U11	T	F	T	P
U12	T	F	P	T

Table 9.2: Example hierarchic constraint f'n between **V2** and **V0**

allow us to maximally restrict the range of the source domain both in terms of reduction of the individual domain value hierarchies and the domain values themselves. For instance, if the particular specialization linkage of S_1 of S fails in application against all possible specialization linkages in the potential target domain, the S_1 branch can be eliminated entirely. If all such branches are similarly eliminated for S, S itself may be removed.

The remainder of this chapter is structured as follows. I first describe the hierarchical application algorithm which applies a constraint between two hierarchical domain values. This algorithm determines whether the constraint possibly holds, necessarily holds, or fails. As well, the application algorithm needs to mark portions of the hierarchy which necessarily fail the constraint (as introduced in Section 8.3.1.2). These marked portions can serve to later prune the hierarchy in the revision stage. For example, if a particular sub-tree of a domain value necessarily fails a constraint with respect to all values possibly

V1 ↓ $V2 \rightarrow$	B1	B2	B3
A1	P	P	F
A2	P	P	T
A3	F	F	P

V1 ↓ $V2 \rightarrow$	R1	R2	R3	R4	R5	R6
P1	F	P	P	F	-	-
P2	F	P	F	P	-	-
P3	F	F	F	F	F	T
P4	F	P	T	F	F	F

V1 ↓ $V2 \rightarrow$	T1	T2	T3	T4	T5	T6
C1	P	F	F	P	-	-
C2	F	F	F	F	-	-
C3	P	F	F	P	T	T
C4	F	F	F	P	F	F
C5	-	-	-	-	P	P
C6	-	-	-	-	P	P

V1 ↓ $V2 \rightarrow$	S3	S4	S5	S6	S7	S8
Q1	T	T	F	F	-	-
Q2	T	F	F	F	-	-
Q3	F	F	-	-	F	F
Q4	F	F	-	-	F	F

V1 ↓ $V2 \rightarrow$	U1	U2	U7	U8	U9	U10	U11	U12
X1	F	T	T	T	-	-	-	-
X2	F	F	F	F	-	-	-	-
X5	F	P	T	T	-	-	-	-
X6	T	F	F	F	-	-	-	-
X7	-	-	P	F	-	-	-	-
X8	-	-	T	F	-	-	-	-
X9	-	-	-	-	T	T	T	P
X10	-	-	-	-	F	F	P	T
X11	-	-	-	-	T	F	P	P
X12	-	-	-	-	P	F	P	T

Table 9.3: Example hierarchic constraint f'n between **V1** and **V2**

satisfying the constraint, then this sub-tree may be pruned. I next outline a hierarchical revision algorithm which reduces and simplifies the source domain accordingly. I present two instantiations of this revision algorithm: **Aggressive**, in which all possible reductions are made as soon as possible, and **Stepped** in which only those reductions necessary to justify the continued existence of a particular source domain value are made. Each of these revision algorithms implies a different amount of reasoning about compatible domain values, and in particular, a different time at which such reasoning should be undertaken. Next, I outline how to utilize the revise algorithms in an arc-consistency algorithm. Once again it is possible to structure this algorithm in two versions: **Traditional** (AO-HAC), in which arc-consistency is enforced through careful re-application of constraints, and **Simplified** (AO-HAC-NEW), in which an approximation of arc-consistency with less constraint application effort is accepted. Finally, I compose the two revision algorithms and two consistency strategies into four possible methodologies for enforcing or increasing hierarchical arc-consistency.

9.2 My Hierarchical Arc-consistency Algorithm

9.2.1 Algorithm Apply

9.2.1.1 Informal Description

The algorithm ApplyR outlined in Table 9.7, insures AO-consistency for a particular source and target pair of hierarchical domain values, a corresponding constraint, and a hierarchical depth. The source domain value may have some or all of its hierarchical structure pruned to exclude those portions that are not possibly consistent with the given target.

Inputs ↓→	$TRUE$	$FALSE$	$POSS$
$TRUE$	T	T	T
$FALSE$	T	F	P
$POSS$	T	P	P

Table 9.4: OR_3 logical operator

Inputs ↓→	$TRUE$	$FALSE$	$POSS$
$TRUE$	T	F	P
$FALSE$	F	F	F
$POSS$	P	F	P

Table 9.5: AND_3 logical operator

Input →	$TRUE$	$FALSE$	$POSS$
	F	T	F

Table 9.6: NOT_3 logical operator

In the ApplyR algorithm, a constraint γ is applied against a hierarchical source and target domain value pair. In order for the constraint to necessarily hold (i.e. return $TRUE$), the constraint must hold at all levels of the hierarchy up to and including $MaxD$ levels up and down in the hierarchy. This algorithm applies the constraint at the current level, and calls both the ApplyUp and ApplyDown algorithms which recursively apply the constraint against appropriate portions of the domain value hierarchies. Note that the results are "and'ed" together in a ternary logic (see Table 9.5) that yields $TRUE$ only when the operands are all true, yields $POSS$ only when the operands are all $POSS$ or $TRUE$, and yields $FALSE$ in all other cases. We now look at the upward and downward aspects of the apply algorithm in turn.

Algorithm ApplyUp Description

Upward hierarchical revision is an attempt to determine which (if any) parents of a domain value, of all possible parents, are consistent with that domain value, a given constraints, and a particular target domain value hierarchy. For example, imagine that (in the automated banking domain) that the act of *depositing money to machine* can be part of either a plan to *deposit to account* or one to *pay a bill*. Similarly, the act of *filling out a slip* can be part of either a plan to *pay a bill* or *deposit US dollars*. This circumstance can be represented as a simple hierarchy in which *deposit to machine* is child to both *deposit to account* and *pay bill*, and *fill out slip* is child to both *pay bill* and *deposit US*. Consider that two actions have been observed in watching someone at a bank machine. The only candidate value for the first or "source" action is *deposit to machine*. The only candidate value for the second or "target" action is *fill out slip*. If we consider that both these actions are known to be "constrained" to be part of the same transaction, then **upward revision** is the identification of which parents of the "source" domain value are compatible with which parents of the "target" domain value. In this

Algorithm ApplyR(Node:X_i, Node:Y_j, Relation:γ, int $MaxD$, Mode $CurMode$, int $CurDepth$);
Input: Two nodes X_i and Y_j, a relation γ between the corresponding variables, an integer $MaxD$ giving the maximum depth to penetrate the hierarchy, $CurMode$ signifying ascent or descent in the hierarchy, and an integer *depth*, the current penetration.
Output: $TRUE$ if application of $\gamma(X_i, Y_j)$ holds to a hierarchical distance $MaxD$ from X_i, $POSS$ if it may, $FALSE$ otherwise.

SubRoutines

A. ApplyDown(Node:X_i, Node:Y_j, Relation:γ, int $MaxD$, int $depth$) : returns one of $TRUE$, $FALSE$, $POSS$.
B. ApplyUp(Node:X_i, Node:Y_j, Relation:γ, int $MaxD$, int $depth$) : returns one of $TRUE$, $FALSE$, $POSS$.
C. $AND_3(LogicVal : x, LogicVal : y)$: returns a pessimistic AND of $TRUE$, $FALSE$, $POSS$.

Main Routine

```
R_local := γ(X_i, Y_j);
if R_local = FALSE
  then  if IsRoot(X_i) then Mark children of X_i;
    return FALSE
  else
    switch(CurMode)
      case = "initial"
        return AND3(ApplyDown(X_i, Y_j, γ, depth), ApplyUp(X_i, Y_j, γ, depth))
      case = "down"
        return ApplyDown(X_i, Y_j, γ, depth)
      case = "up"
        return ApplyUp(X_i, Y_j, γ, depth)
```

Table 9.7: The ApplyR algorithm

case, only *pay bill* admits a consistent evaluation of the constraint for each domain value, and consequently the *deposit to account* parent of the "source" value *deposit to machine* may be eliminated. Note that in this case the constraint is in fact bi-directional, and consequently we can consider the target also as the source, and thus parent *deposit US* of *fill out slip* can also be eliminated.

In the ApplyUp algorithm outlined in Table 9.8, the source and target domain hierarchy is checked for an "up" link. Each of the source or target "up" links can be either absent, AND, or OR. Figure 9.5 details the four primary source and target hierarchical combinations. If either of the "up" links are top-leaf nodes, the constraint cannot be checked at the next up-level (i.e. the next level up in the hierarchy) and consequently an indeterminate result $POSS$ is returned. This $POSS$ result is returned based on the assumption that the lack of definition of additional higher-level hierarchical values does not necessarily negate their existence. In each of the remaining possible source and target combinations, any pair of source and target parents that satisfy the constraint indicate a success at the next highest level (see the pessimistic "or" function defined in Table 9.4).

In addition, if the source's left parent fails a constraint application against both of the targets' parents, then this arc is marked by the $Mark$ algorithm for potential deletion. Specifically, such an arc is deleted for a particular source and parent if this parent fails against all other target parents at this level.

Algorithm ApplyDown Description

In Section 9.2.1.1, an example was given in the automated banking domain. We continue that example in explaining downward hierarchical revision. The plan for *depositing money in a machine*(of currency Cur) consists of at least two component parts, *taking money out of a wallet*(of currency Cur) and *putting money in an envelope*(of currency Cur). The plan for *filling out a slip* can be specialized as either *filling out a US deposit slip*(where

Algorithm ApplyUp(Node:X_i, Node:Y_j, Relation:γ,int $MaxD$, int $depth$);
Input: Two nodes X_i and Y_j, a relation γ between the corresponding variables, an integer $MaxD$ giving the maximum depth to penetrate the hierarchy, and an integer $depth$ giving the current penetration.
Output: TRUE if application of $\gamma(X_i, Y_j)$ holds to a hierarchical height $MaxD$ from X_i, FALSE if γ is known to fail, POSS otherwise.

SubRoutines

A. OR_3 : returns a pessimistic OR of $TRUE$, $FALSE$, $POSS$.
B. $FAIL_3$: true if ternary logic value is $FALSE$.
C. $MARK(X_i, X_k)$: increment failed arc count between X_i and X_k.
D. $TopLeaf(X_i)$: true if X_i is a leaf node at the top of hierarchy.
E. $Left(Right)ParentFail(X_i)$: true if the given parent has been deleted

Main Routine

$NewD := depth + 1$;

/ if we have exceeded the given hierarchical depth, go no further */*
if $(NewD > MaxD)$ **then set return** $TRUE$;

/ if either source or target is a top-leaf, result is indeterminate */*
if ($TopLeaf(X_i)$ **or** $TopLeaf(Y_i)$) **then return** $POSS$;

/ identify the undeleted parents of the current node */*
$X_{LP} := X_i.LeftParent$; $Y_{LP} := Y_j.LeftParent$;
$X_{RP} := X_i.RightParent$; $Y_{RP} := Y_j.RightParent$;
$Y_{LPfail} := Y_j.LeftParentFail$; $Y_{RPfail} := Y_j.RightParentFail$;

/ check the left source and left target parent subtrees */*
R_{LL} := **if** ((**null** Y_{LP}) **and** Y_{LPfail})
  then $FALSE$
  else ApplyR($X_{LP}, Y_{LP}, \gamma, Up, MaxD, NewD$);

... **continued** ...

Table 9.8: The ApplyUp algorithm, part 1 of 2

Algorithm APPLYUP(Node:X_i, Node:Y_j, Relation:γ,int $MaxD$, int $depth$);
Continued ...

Main Routine

/ check the left source and right target parent subtrees */*
R_{LR} := **if** ((**null** Y_{RP}) **and** Y_{RPfail})
  then $FALSE$
  else APPLYR($X_{LP}, Y_{RP}, \gamma, MaxD, NewD$);

/ check the right source and left target parent subtrees */*
R_{RL} := **if** ((**null** Y_{LP}) **and** Y_{LPfail})
  then $FALSE$
  else APPLYR($X_{RP}, Y_{LP}, \gamma, MaxD, NewD$);

/ check the right source and right target parent subtrees */*
R_{RR} := **if** ((**null** Y_{RP}) **and** Y_{RPfail})
  then $FALSE$
  else APPLYR($X_{RP}, Y_{RP}, \gamma, MaxD, NewD$);

/ Mark source subtrees that fail against both target parents */*
$R_L := OR_3(R_{LL}, R_{LR})$; **if** $FAIL_3(R_L)$ **then** $MARK(X_i, X_{LP})$);
$R_R := OR_3(R_{RL}, R_{RR})$; **if** $FAIL_3(R_R)$ **then** $MARK(X_i, X_{RP})$);

/ Return true if any source, target subtree pair succeeds */*
set return $OR_3(R_L, R_R)$;

Table 9.9: The APPLYUP algorithm, part 2 of 2

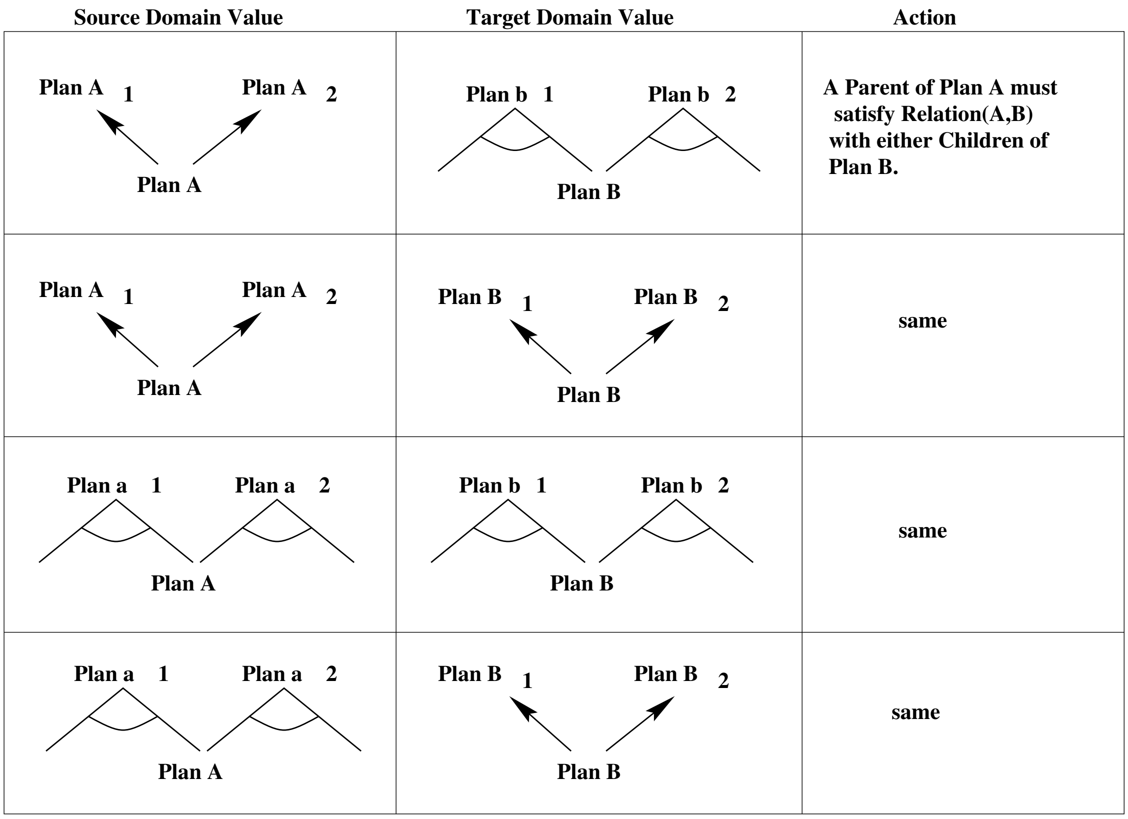


Figure 9.5: Upward cases for source, target structure

Cur = "US") or *filling out a bill slip*. Consider that two actions are observed and the only candidate for the first or "source" action is *deposit money in a machine*, and the only candidate for the first or "target" action is *fill out slip*. If we consider that both these actions are know to be "constrained" in that currencies involved must be the same if the plans are related, then **downward revision** is the identification of which children of the "source" domain value are compatible with which parents of the "target" domain value. In this case, neither of the children of *deposit money in a machine* are consistent with *fill out US deposit slip* (since the currencies do not match), however, both are consistent with *fill out bill slip*. Since once again the constraint is bi-directional, we can consider the target as source, and eliminate any children of *fill out slip* which are inconsistent with children of *deposit to machine*. In this case, *fill out bill slip* is consistent with both children, however, *fill out US deposit slip* is inconsistent due to the different currency involved. Thus, the child *fill out US deposit slip* can be eliminated as a candidate and we can thus conclude that the *fill out slip* action is actually specialized as *fill out bill slip*.

In the ApplyDown algorithm shown in Table 9.10, 9.11, and 9.12, the source and target domain hierarchy is checked for a "down" link. Each of the source or target "down" links can be either absent, AND, or OR. Figure 9.6 details the four primary source and target hierarchical combinations[1].

If either of the "down" links are bottom-leaf nodes, the constraint cannot be checked at the next down-level (i.e. the next level down in the hierarchy) and consequently an indeterminate result $POSS$ is returned. This $POSS$ result is returned based on the assumption that the lack of definition of additional lower-level hierarchical values does not necessarily negate their existence. In each of the remaining four possible source and target combinations a successful application is determined as follows. In the first case, the target and source both possess "or" child links. The constraint application succeeds at the next down-level if any pair of the source and target down links succeed. Failure of the child link for a source value against both targets indicates a potential link for deletion. In the second case, the target and the source both possess "and" child links. The constraint application succeeds if all possible combinations of the source and target children downwards in the hierarchy (down-links) succeed with the constraint application. If the application fails, the source's child links are marked for possible deletion. In the third case, the source node has an "or" child link and the target node has an "and" child link. In this case, if either of the source links succeed against both of the target links

[1] Note that in the cases shown in Figures 9.6 and 9.5 it is quite possible that a "diamond" relation exists in which some Plan B has two parents B1 and B2 which in turn share a single parent, BB. An example of such a relationship might be where B is a sub-part of both B1 and B2, and where B1 and B2 are different specializations of BB.

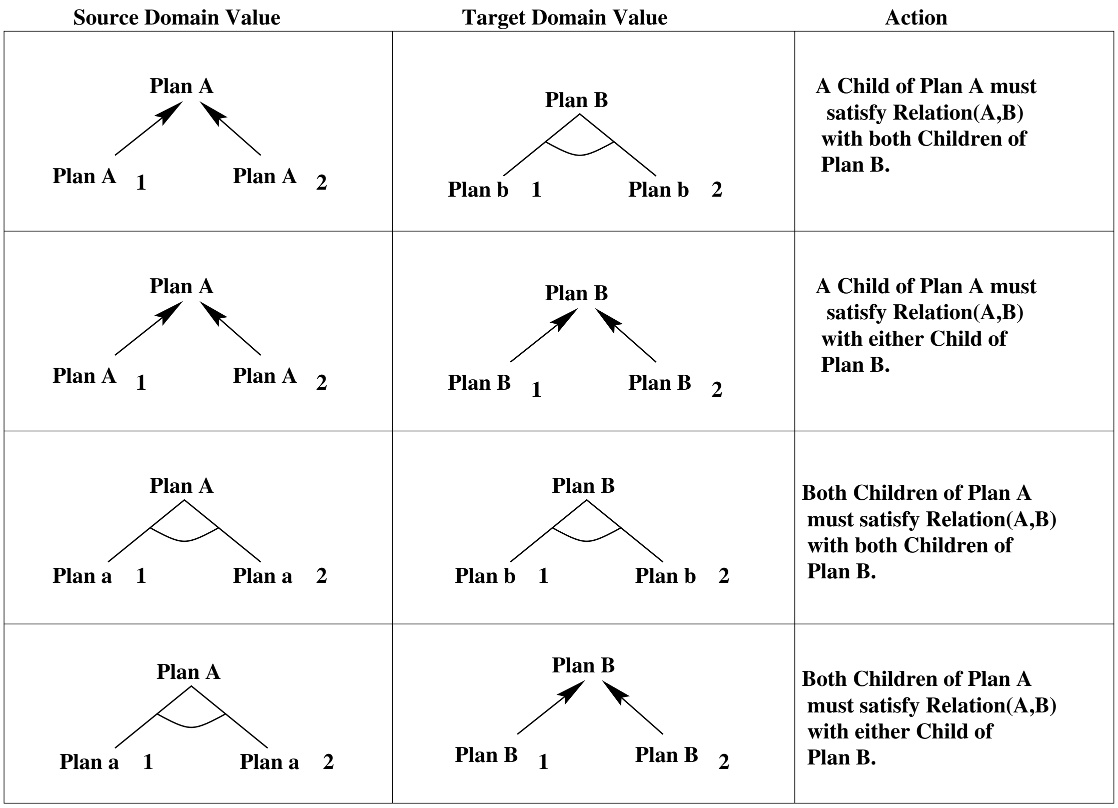


Figure 9.6: Downward cases for source, target structure

then the constraint application succeeds at the next down-level. If either of the source down-links fails against both target child links then it is marked for potential deletion. In this fourth and final case, the source node possesses an "and" link and the target node possess an "or" link. Here, a successful application at the next down-level occurs if *both* source's child links succeed in conjunction with the same target link (or links). If both source links fail, they are marked for deletion.

The ternary logical operators are defined in Tables 9.4, 9.5, and 9.6. These operators have been defined pessimistically in the sense that and *and* is true only in the sense that all values involved are necessarily true. Similarly, an *or* is true only if at least one value is necessarily true.

Algorithm ApplyDown(Node:X_i, Node:Y_j, Relation:γ, int $MaxD$, int $depth$);
Input: Two nodes X_i and Y_j, a relation γ between the corresponding variables, an integer $MaxD$ giving the maximum depth to penetrate the hierarchy, and an integer $depth$ giving the current penetration.
Output: $TRUE$ if application of $\gamma(X_i, Y_j)$ holds to a hierarchical depth $MaxD$ from X_i, FALSE if γ is known to fail, $POSS$ otherwise.

SubRoutines

A. AND_3 : returns a pessimistic AND of $TRUE$, $FALSE$, $POSS$.
B. $BottomLeaf(X_i)$: true if X_i is a leaf node at the bottom of hierarchy.
C. $Left(Right)ChildFail(X_i)$: true if the given child has been deleted

Main Routine

$NewD := depth + 1$;

/ if we have exceeded the given hierarchical depth, go no further */*
if $(NewD > MaxD)$ **then return** $TRUE$;

/ if either source or target is a bottom-leaf, result is indeterminate */*
if $(BottomLeaf(X_i)$ **or** $BottomLeaf(Y_j))$
  then set return $POSS$;

/ identify the undeleted children of the current node */*
$X_{LC} := X_i.LeftChild$; $Y_{LC} := Y_j.LeftChild$;
$X_{RC} := X_i.RightChild$; $Y_{RC} := Y_j.RightChild$;
$X_{LCfail} := X_i.LeftChildFail$; $Y_{RCfail} := Y_j.RightChildFail$;
$X_{type} := X_i.ChildType$; $Y_{type} := Y_j.ChildType$;

/ check the left source and left target child subtrees */*
$R_{LL} :=$ **if** ((**null** Y_{LC}) **and** Y_{LCfail})
  then $FALSE$
  else ApplyR$(X_{LC}, Y_{LC}, \gamma, down, MaxD, NewD)$;

... **continued** ...

Table 9.10: The ApplyDown algorithm, part 1 of 3

Algorithm ApplyDown(Node:X_i, Node:Y_j, Relation:γ, int $MaxD$, int *depth*);

Continued ...

Main Routine

/* *check the left source and right target child subtrees* */
9 R_{LR} := **if** ((**null** Y_{RC}) **and** Y_{RCfail})
  then $FALSE$
  else ApplyR($X_{LC}, Y_{RC}, \gamma, down, MaxD, NewD$);

/* *check the right source and left target child subtrees* */
10 R_{RL} := **if** ((**null** Y_{LC}) **and** Y_{LCfail})
  then $FALSE$
  else ApplyR($X_{RC}, Y_{LC}, \gamma, down, MaxD, NewD$;

/* *check the right source and right target child subtrees* */
11 R_{RR} := **if** ((**null** Y_{RC}) **and** Y_{RCfail})
  then $FALSE$
  else ApplyR($X_{RC}, Y_{RC}, \gamma, down, MaxD, NewD$);

/* *Combine pairwise results based on a hierarchical case* */
12 **switch**(XY_{type})

13   **case** = "OR + OR"
14     $R_L := OR_3(R_{LL}, R_{LR})$;
15     $R_R := OR_3(R_{RL}, R_{RR})$;

    /* *Mark source parent inconsistent with both targets* */
16   **if** $FAIL_3(R_L)$ **then** $MARK(X_i, X_{LC})$;
17   **if** $FAIL_3(R_R)$ **then** $MARK(X_i, X_{RC})$;
18   **return** $OR_3(R_L, R_R)$;

... **continued** ...

Table 9.11: The ApplyDown algorithm, part 2 of 3

Algorithm ApplyDown(Node:X_i, Node:Y_j, Relation:γ, int $MaxD$, int $depth$);
Continued ...

Main Routine

    case = "AND + AND"
      $R_{Down} := AND_3(R_{LL}, R_{LR}, R_{RL}, R_{RR})$;

      /* *Mark source parents inconsistent with either target* */
      if $FAIL_3(R_{Down})$
        then $MARK(X_i, X_{LC})$; $MARK(X_i, X_{RC})$;
      return R_{Down};

    case = "OR + AND"
      $R_L := AND_3(R_{LL}, R_{LR})$;
      $R_R := AND_3(R_{RL}, R_{RR})$;

      /* *Mark source parent inconsistent with either target* */
      if $FAIL_3(R_L)$ **then** $MARK(X_i, X_{LC})$;
      if $FAIL_3(R_R)$ **then** $MARK(X_i, X_{RC})$;
      return $OR_3(R_L, R_R)$;

    case = "AND + OR"
      $R_L := AND_3(R_{LL}, R_{RL})$;
      $R_R := AND_3(R_{LR}, R_{RR})$;

      /* *Mark source parents inconsistent with either target* */
      if $(FAIL_3(R_L)$ **and** $FAIL_3(R_R))$
        then
          $MARK(X_i, X_{LC})$; $MARK(X_i, X_{RC})$;
          return $FALSE$;
        else
          return $TRUE$;
  end switch;

Table 9.12: The ApplyDown algorithm, part 3 of 3

9.2.2 Algorithm Revise

9.2.2.1 Informal Description

In the REVISE algorithms overviewed in Tables 9.13 and 9.14, a particular source variable domain is reduced by removing values incompatible with any target domain value with respect to a particular constraint γ. The *aggressive* revise algorithm checks all target domain values for a given source and constraint, establishing justification links which indicate the results of each constraint application. In this way, any domain value (or part of a domain value's hierarchy) which is inconsistent with any target domain value may be deleted. This deletion is known as revision. Since justification links are generated for the entire problem graph in the aggressive approach, when a domain value is deleted, it is possible to see if this value *justified* the existence of another value in a different variable. If this is the case, and if no other justifications exist for that other variable value, then it is possible to propagate deletion (as introduced in Section 8.3.1.2 with *DeleteSourcePropagation*), deleting these values which are no longer justified.

In contrast, the *stepped* revise algorithm checks target domain values only as far as necessary to determine that a particular source value is *justified* in its continued existence as a value for the source variable. In this way, justification links are generated incrementally as needed, and domain values or hierarchies can be reduced only when all target values have been encountered. In addition, the removal of a justifying target domain value results in the need to try to re-justify the connected source value, with back-deletion possible only after checking all possible justifiers. The aggressive approach works on the assumption that the effort of aggressively justifying source value will pay off through effective back-deletion through the justification links. The stepped approach assumes that complete justification is over ambitious, and a more effective approach would be to check constraints only when forced to in order to justify the continued existence of a particular value.

In either aggressive or stepped revision, domain values are deleted only when they necessarily fail a given constraint. The way in which one or other of the revision algorithms are used as part of an arc consistency algorithm will determine the relative effectiveness of a given approach. Each algorithm is examined in detail in the following paragraphs.

9.2.2.2 Aggressive Revision Description

The algorithm AO-ReviseAggr outlined in Table 9.13 achieves AO-consistency for a source variable domain given a particular target variable and domain, a corresponding directed constraint, and a hierarchical depth.

Each domain in the source domain represents a set of domain values according to the domain value's particular hierarchical structure. Each source domain value is revised in turn. The constraint in question is checked against each target domain value. For each source and target value pair which satisfies the constraint, a *Link* is created indicating that the source value "depends" (at least partly) on the existence of the target domain value for its continued existence as a source domain candidate. After a particular source domain value has been checked against all target domain values, the source member may be deleted if no satisfying target values are found. If satisfying target values exist, then source target may be pruned insofar as some portions of its hierarchy may not have satisfied the constraint with any target domain value. It is important to note that it is not strictly necessary to check all target values for a particular source domain value in order to justifying keeping the source value. A single target domain value satisfying the given constraint with the source value is sufficient. However, each source domain value represents a hierarchy of other sub-domain value and pruning can consequently take place within the hierarchical structure corresponding to a particular source domain value even if that value itself is retained.

The primary subroutines shared in both aggressive and stepped revision routines may be summarized as follows:

1. **AddLink**, which generates a justification link between a source domain value x_i and its justifier, y_j with respect to the constraint γ. In the aggressive version of revise, this link can be one-to-one or one-to-many in the case of many target justifiers of x_i. As stated, the lack of a valid γ justifier is cause for deletion of y_j.

2. **DeleteSrcPropagateAggr**, which is invoked on the determination that the source value x_i has no γ justification. The deletion causes a back propagation in which it is necessary to delete any other values depending solely on x_i for their justification. Table C.1 in Appendix C.1 outlines this algorithm.

3. **Simplify**, invoked on determination that a value x_i is to be retained. Since x_i has been checked against all N target values with respect to the constraint γ, it is possible that some subpart of the x_i hierarchy is inconsistent with any target y_j value and may be removed. In particular, any arc marked with N failures may be removed. Tables 9.15, 9.16 and 9.17 outline this algorithm.

4. **KeepSrcPropagateAggr**, is called on the determination that x_i is to be retained, and further that the source value hierarchy of x_i has been simplified. This simplification may affect the use of x_i as a justifier in some previously identified justification, and so any such justification link needs to be re-evaluated. Table C.2 in Appendix C.2 outlines this algorithm.

5. **ResetHierarchyVisit**, is called to reset the hierarchy of a given source value so that all are unmarked and ready for the next constraint application against a particular target.

Algorithm REVISEAGGR(Var:$Source$, Var:$Target$, Relation:γ, int $MaxD$);
Input: Two variables $Source$ and $Target$, a relation γ between the variables, and an integer bounding the depth of revision;
Output: List of variables with reduced domains after AO-revision of $Source$ w.r.t. $Target$, and γ;

```
Iterate over all domain values x_i of source variable

  X_i ok := FALSE;          /* source value unjustified */

  Iterate over all domain values y_j of target variable

    x_i y_j ok := ApplyR(x_i, y_j, γ, initial, MaxD, 0);

    if (not FAIL_3(x_i y_j ok))
    then
      AddLink(x_i, Source, y_j, Target, γ);
      X_i ok := TRUE;

  End Iteration over y_j;

    if (not X_i ok)    /* source value still unjustified */
    then
      Add to UpdateVars, DeleteSrcPropagateAggr(x_i, Source, MaxD)
    else
      if Simplify(x_i, Target, γ)    /* try revise source hierarchy */
      then
        Add to UpdateVars, KeepSrcPropagateAggr(x_i, Source, MaxD);

    /* reset source domain hierarchy members as unmarked */
    ResetHierarchyVisit(x_i, γ);

End Iteration over x_i;
Return UpdateVars;
```

Table 9.13: The Aggressive REVISE algorithm

9.2.2.3 Stepped Revision Description

The algorithm AO-ReviseStep outlined in Table 9.14 achieves AO-consistency for a particular pair of variables, a corresponding directed constraint, and a hierarchical depth.

Step differs from Aggressive solely in that the target domain is traversed during revision only so far as required to justify the continued existence of a particular source variable value involved. Justification linkages are generated as with the aggressive version, with the exception that the justification structure for a particular source value is incomplete until such time as each and every target value relevant to a particular constraint has been checked. Consequently, any back-propagation needs to be undertaken only after such a restriction is ensured. As a result, a deletion of a source value x_i can trigger a series of re-justification efforts for other values and constraints.

While the subroutines listed in the previous section for the aggressive revision algorithm are still utilized in the stepped version, the **DeleteSrcPropagate** and **KeepSrcPropagate** algorithms need to be adapted to account for the new behaviour of dealing with modified or deleted source values. **DeleteSrcPropagateStep** is outlined in Table C.3 of Appendix C.3 and **KeepSrcPropagateStep** in Table C.4 of Appendix C.4.

Algorithm ReviseStep(Var:*Source*, Var:*Target*, Relation:γ, int *MaxD*);

```
Iterate over all domain values x_i of Source variable
   X_i ok := FALSE;          /* source unjustified */
   Iterate over all domain values y_j of Target variable
   if y_j has no siblings in target variable
   then                      /* last target value to check */
      ResetHierarchyVisit(x_i, γ);
      x_i y_j ok := ApplyR(x_i, y_j, γ, initial, MaxD, 0);
      if (not FAIL_3(x_i y_j ok))
      then                   /* last target justifies source */
         AddLink(x_i, Source, y_j, Target, γ);
         X_i ok := TRUE;
         if Simplify(x_i, Target, γ)
         then                /* revised source value hierarchy */
            Add to UpdateVars, KeepSrcPropagateStep(x_i, Source, MaxD);
      else X_i ok := FALSE;

   else more yj siblings exist
      x_i y_j ok := ApplyR(x_i, y_j, γ, initial, MaxD, 0);
      if (not FAIL_3(x_i y_j ok))
      then                   /* a justifying target is found */
         AddLink(x_i, Source, y_j, Target, γ);
         X_i ok := TRUE;
      else X_i ok := FALSE;  /* target justifier not found yet */
   End Iteration over y_j;
   if (not X_i ok)
   then                      /* source found unjustified */
      Add to UpdateVars, DeleteSrcPropagateStep(x_i, Source, MaxD)
   else                      /* source found justified */
      ResetHierarchyVisit(x_i, γ);
End Iteration over x_i;
Return UpdateVars;
```

Table 9.14: The Stepped Revise algorithm

Algorithm $Simplify$(Val:x_i, int:N γ);
I/O Comment: $Simplify$ updates the hierarchy of x_i **in place**.

SubRoutines

A. $SimplifyUp$: true if one of the parent subtrees is updated, null otherwise
B. $SimplifyDown$: true if one of the parent subtrees is updated, null otherwise
C. $(Left/Right)ChildExists(x_i)$: returns true only if x_i has a left/right child

Main Routine

$HierUp := SimplifyUp(x_i, N, \gamma)$; */* try to reduce the parent subtrees */*

if $LeftChildExists(x_i)$ */* try to reduce if left child exists */*
then
  $HierDnLeft := SimplifyDown(LeftChild(x_i), N, \gamma, \text{“}lc''\text{”})$;

if $RightChildExists(x_i)$ */* try to reduce if right child exists */*
then
  $HierDnRight := SimplifyDown(RightChild(x_i), N, \gamma, \text{“}rc''\text{”})$;

/ if any subtrees updates, return true */*
Return **or**($HierUp$, $HierDnLeft$, $HierDnRight$);

Table 9.15: The $Simplify$ hierarchical reduction algorithm

Algorithm $SimplifyUp$(Val:x_i, int:N γ);
I/O Comment: $SimplifyUp$ updates the parent subtrees of x_i **in place**.

SubRoutines

A. $DeleteUpSubtreeRootedAt(x)$: delete subtree incl x, upwards from x.

Main Routine

```
LP := LeftParent(x_i); RP := RightParent(x_i);
LPMk := LeftParentMark(x_i); RPMk := RightParentMark(x_i);
Update := FALSE;

if LP                     /* left parent exists ? */
then
  if Equals(LPMk, N)      /* parent marked n times ? */
  then                    /* yes, delete upwards */
    DeleteUpSubtreeRootedAt(LP); Update := TRUE;
  else                    /* no, but check subtrees */
    Update := SimplifyUpLP, N, γ);

if RP                     /* right parent exists ? */
then
  if Equals(RPMk, N)      /* parent marked n times ? */
  then                    /* yes, delete upwards */
    DeleteUpSubtreeRootedAt(RP); Update := TRUE;
  else                    /* no, but check subtrees */
    Update := SimplifyUpRP, N, γ);

/* any deletions indicates return true */
Return Update;
```

Table 9.16: The $SimplifyUp$ reduction algorithm

Algorithm $SimplifyDown$(Val:x_i, int:N γ);
I/O Comment: $SimplifyDown$ updates the child subtrees of x_i **in place**.

SubRoutines

A. $DeleteDownSubtreeRootedAt(x)$: delete subtree incl x, downwards from x.

Main Routine

$LC := LeftChild(x_i)$; $RC := RightChild(x_i)$;
$LCMk := LeftChildMark(x_i)$; $RCMk := RightChildMark(x_i)$;
$Update := FALSE$;

if LC /* *left child exists ?* */
then
  if $Equals(LCMk, N)$ /* *child marked n times ?* */
  then /* *yes, delete downwards* */
    $DeleteDownSubtreeRootedAt(LC)$; $Update := TRUE$;
  else /* *no, but check subtrees* */
    $Update := SimplifyDownLC, N, \gamma)$;

if RC /* *right child exists ?* */
then
  if $Equals(RCMk, N)$ /* *child marked n times ?* */
  then /* *yes, delete downwards* */
    $DeleteDownSubtreeRootedAt(RC)$; $Update := TRUE$;
  else /* *no, but check subtrees* */
    $Update := SimplifyDownRC, N, \gamma)$;

/* *any deletions indicates return true* */
Return $Update$;

Table 9.17: The $SimplifyDown$ reduction algorithm

9.2.3 Hierarchical Arc-consistency

The necessary parts of an arc-consistency algorithm have now been presented. With a revision algorithm for a single source variable which ensures consistency for source domain values with respect to other targets and constraints, one can assemble an algorithm that enforces consistency over a set of source variables. Simply iterating over all target variables and constraints enforcing consistency against applicable source variables is, however, not adequate. Since revision is done between a source domain set and a target domain set, a change to a target domain set may require that the source be revised again since a justifying value from the target may have been removed. Since the revision algorithms have been written such that explicit records are maintained of justification linkage, it is possible to exploit such linkage in the process of enforcing consistency.

I describe two separate arc-consistency algorithms here. The first, AO-HAC is a typical implementation which is modeled loosely after the AC-3 algorithm described in Section 4.3.2.2. First, iterate over all the target variables in the problem enforcing consistency with respect to all relevant sources. If a variable domain is modified, that variable is returned to the list of targets which must be re-revised. In this way a series of revision attempts which do not return additional variables to re-revise indicates that the problem is arc-consistent.

It is possible to envisage this consistency algorithm functioning with either the aggressive or stepped revise algorithm. It is an open research issue under what circumstances a particular combination would be most effective. The second arc-consistency algorithm, AO-HAC-NEW, is a somewhat simplified version of AO-HAC . AO-HAC-NEW simply iterates through each variable and considers each as a target one at a time. Once considered, a variable is never re-considered. It appears that this algorithm would only achieve partial arc-consistency since no re-revisions are performed. Recall that the aggressive revision algorithm generates a complete justification network for all retained domain values. This explicit linkage provides a framework on which to back propagate deletions. Such deletions in effect handle any re-revision through re-application of potentially destroyed justifications. The careful handling of this linkage structure is represented in the $DSPA$, $KSPA$, $DSPS$ and $KSPS$ algorithms shown in Tables C.1, C.2, C.3 and C.4 in Appendix C. The space occupied against linkage storage and the relative effort to compute all or partial linkage is thus used to trade-off against the time of re-revising variables. Making use of the stepped revision algorithm in AO-HAC-NEW will, however, achieve only partial arc-consistency as a result of its incomplete linkage structure coupled with a failure to return to all modified target variables for re-revision.

Altogether there are four possible arc-consistency algorithms depending on the base algorithm and the selected revision algorithm:

1. AO-HAC with aggressive revision, or AO-HAC-AGGR.

2. AO-HAC with stepped revision, or AO-HAC-STEP.

3. AO-HAC-NEW with aggressive revision, or AO-HAC-NEW-AGGR.

4. AO-HAC-NEW with stepped revision, or AO-HAC-NEW-STEP.

I describe the AO-HAC base algorithm briefly in Table 9.18, and the AO-HAC-NEW algorithm in Table 9.19.

9.2.3.1 Generic Hierarchical Examples

In this section I describe the behaviour of the hierarchic arc consistency algorithms through the use of the example I outlined in the CSP structure depicted in Figure 9.2 and the inter-variable constraint defined in Tables 9.1, 9.2, and 9.3. For simplicity and

Algorithm AO-HAC (Graph:Λ, int $MaxD$);
Input: A graph Λ structure containing variables V_i with domains $D(V_i)$, and relations among variables $\gamma(V_i, V_j)$, and an integer $MaxD$ giving the maximum depth to penetrate the domain hierarchy, ;
Output: An AO arc-consistent graph to depth $MaxD$;

SubRoutines

A. $GetSrcVars(Graph : \Lambda, Var : V_{target})$: returns list of variables V_{source} where some relation $\gamma(V_{source}, V_{target})$ exists.
B. $GetRelation(Graph : \Lambda, Var : V_1, Var : V_2)$: returns relation from Λ, $\gamma(V_1, V_2)$.
C. $GetVariables(Graph : \Lambda)$: returns variable list from Λ.
D. $GetDeleteMember(List : L)$: returns element of L, removing from L.
E. $Union(List : L1, List : L2)$: returns $\bigcup(L1, L2)$.
F. $Empty(List : L)$: returns $TRUE$ if L empty, nil else.

Main Routine

```
1    TargetList := GetVariables(Λ);

2    while not(Empty(TargetList)) do
3      ThisTarget := GetDeleteMember(TargetList);
4      SrcVarList := GetSrcVars(Λ, ThisTarget);

5      iterate over SrcVarList with SrcVar
6        γ := GetRelation(Λ, SrcVar, ThisTarget);
7        ReviseVars := Revise(SrcVar, ThisTarget, γ, MaxD);

8        if not Empty(ReviseVars)
         then
           TargetList := Union(ReviseVars, TargetList);

9      next SrcVar; (step 5)
10   end while; (step 2)
```

Table 9.18: The AO-HAC arc-consistency algorithm

Algorithm AO-HAC (Graph:Λ, int $MaxD$);
Input: A graph Λ structure containing variables V_i with domains $D(V_i)$, and relations among variables $\gamma(V_i, V_j)$, and an integer $MaxD$ giving the maximum depth to penetrate the domain hierarchy, ;
Output: An AO (possibly) arc-consistent graph to depth $MaxD$;

Main Routine

1 $TargetList := GetVariables(\Lambda)$;

2 **while not**$(Empty(TargetList))$ **do**
3 $ThisTarget := GetDeleteMember(TargetList)$;
4 $SrcVarList := GetSrcVars(\Lambda, ThisTarget)$;

5 **iterate over** $SrcVarList$ **with** $SrcVar$
6 $\gamma := GetRelation(\Lambda, SrcVar, ThisTarget)$;
7 $Revise(SrcVar, ThisTarget, \gamma, MaxD)$;

9 **next** $SrcVar$; (step 5)
10 **end while**; (step 2)

Table 9.19: The AO-HAC-NEW arc-consistency algorithm

the purposes of this example I assume aggressive constraint application in which a given source domain value is checked against all domain values in a constrained target variable.

Applying a constraint between domain values

In the problem structure of Figure 9.1 observe that a directed constraint exists between $V1$ and $V2$. This constraint (call it C) is defined for these variables in Table 9.3. In order to revise the domain of $V1$ to be consistent with respect to this constraint and the domain of $V2$ it will be necessary to check the individual constraint values between individual values, as outlined in the ApplyR algorithm of Table 9.7. Consider the constraint application between $V1$ domain value A1 and $V2$ value B1. This pairing is chosen for simplicity since both values consist entirely of *is-a* branchings. In this case the matching pairs much conform to an "or" logic, with any pair from source or target children or parents sufficient to produce a successful constraint application as shown in Figure 9.6.

$C(A1, B1)$ is evaluated as follows. Locally, $C(A1, B1)$ evaluates to P or possibly indicating that the constraint itself does not necessarily fail hierarchically. Since A1 is not a root domain value it is necessary to traverse the hierarchy of A1 and B1 shown in Figure 9.2 to further answer the query. Taking the And_3 of the result of applying both upwards and downwards in the relative hierarchy portions gives the following evaluation[2].

- **Upward** (Table 9.8) - A1 parent P1 possibly succeeds against B1 parent R2. Similarly, A1 parent P2 possibly succeeds against B1 parent R2. It is necessary to continue upward another level in each successful case. P1 and R2 may be resolved by checking the possible pairings of P1 parents Q1, Q2 against R2 parents S3,S4. All combinations succeed except for Q2 and S4, thus the $C(P1, R2)$ constraint has been determined to succeed. P2 and R2 may be resolved by checking the possible

[2] Notice we proceed exactly two levels up and downward in the hierarchical comparison. This corresponds to a particular depth for the hierarchic constraint application.

pairings of P2 parents Q3, Q4 against R2 parents S3,S4. All combinations fail, and consequently $C(P2, R2)$ has been determined to necessarily fail. Since A1 parent P2 necessarily fails against both B1 parents R1 and R2. Consequently, P2 can never succeed against any associated value of B1 and should be marked indicating this discovery. Should P2 similarly fail against all B values in $V2$, P2 is inconsistent and may be safely deleted. Thus, the upward portion of the $C(A1, B1)$ application succeeds in the P1 branch.

- **Downward** (Table 9.10)- A1 child C1 fails against all B1 children except T1 which produces a possible success. A1 child C2 fails against both B1 children T1 and T2, provoking us to mark C2 as incompatible with all children of B1. Should C2 fail similarly against other B values in $V2$, it may be deleted. Continuing downward, C1's child X1 succeeds with T1 child U2, other possibilities necessarily failing. This success implies a success at the C1, T1 level and subsequently a success in the downward hierarchic direction for $C(A1, B1)$.

Since both the upward and downward directions succeed, it one may conclude a successful value for the evaluations of $C(A1, B1)$. A justification *Link* is generated between A1 such that B2 justifies the existence of A1.

Revising a single value against a domain

In the example so far, P2 and C2 were marked as incompatible with B1's hierarchy. In fact, in the example (should one continue similarly) P2 and C2 are hierarchically incompatible with the $V2$ values B2 and V3 also. For example, based on the revise algorithm of Table 9.13, applying the constraint C between A1 and each of B1 B2 and B3 would result in success between A1 and B1 only. Subsequent application of the *Simplify* algorithm (Table 9.15) to the A1 hierarchy would remove the subtrees up and down rooted at P2 and C2 respectively.

Revising an entire domain

Following the aggressive revision strategy, applying each value of $V1$ (A1, A2, A3) against each value of $V2$ (B1, B2, B3) would result in a justification set in which A1 is justified by B1 (as above), A1 is justified by B2 and B3 (the justification *Link* has a single source, with two targets), and A3 remains unjustified and may be deleted. In particular, the failure of A3 for any $V2$ value is notices after the iteration stage, and the deletion is subsequently handled with the call to $DeleteSrcPropagateAggr$ in line 12 of the algorithm on Table 9.13.

Link **propagation of consistency**

A *link* from a source domain value to a target domain value (or values) indicates that the source domain value has been found to be *consistent* (for all constraints between the source and target variables) with the linked target values. We define a link in terms of uni-directional constraints in order to simplify the algorithms for dealing with links; that is, a bi-directional constraint can be represented by two uni-directional links. While an implementational scheme would be wise to exploit bi-directional constraints through a shared link, in the case of our prototype we assume uni-directional links only.

Figure 9.7 shows a linkage structure which represents an aggressive revision of the $V1$ domain with respect to $V2$. Figure 9.8 on page 274 extends this representation with the additional linkage after revising variable $V0$ with respect to $V1$.

Consider that an attempt now is taken to revise variable $V2$ with respect to $V0$. The first value in the domain of $V2$ is B1. B1 fails in any application of C against values E1 or E2 of $V0$. Consequently, B1 may be deleted according to $DeleteSrcPropagateAggr$. This deletion results in the removal of the only justification for the continued existence of A1 in the domain of $V2$. A1 has only a single justifier (the aggressive implementation of revise assures this), and so may itself be removed by *DeleteSrcPropagateAggr*. In turn, A1 is one of two justifiers for E1 in the domain of $V0$, and this justification must be removed. E1 remains justified by the existence of A2. As well, A1 is the lone justifier for E2 and so E2 is deleted recursively. Continuing in the revision of $V2$, B2 also fails in any application of C against the lone remaining $V0$ value E1 and is subsequently deleted. The deletion of B2 results in the adjustment of the justification *Link* for A2, leaving only B3 justifying A2. B3 succeeds in the application of C against E1 and a justification *Link* is constructed for B3. This situation now remaining is shown in Figure 9.9. In this situation each domain has a single value remaining, and each value is justified. This situation is in fact a solution.

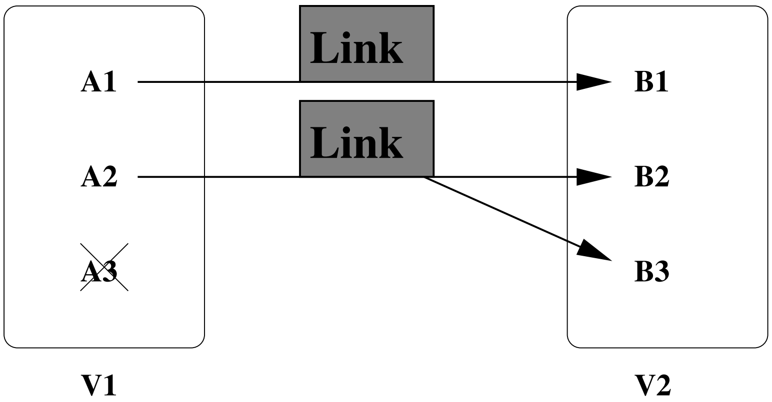


Figure 9.7: Justification of $V1$ domain values w.r.t. $V2$

Example hierarchical arc-consistency

In fact, during these constraint applications which have been described only in terms of the hierarchically central domain values for each variable, certain portions of the hierarchies have been pruned themselves. During the aggressive linking and revision, the pruning of a hierarchy value implies that any justification link based on the pruned value must

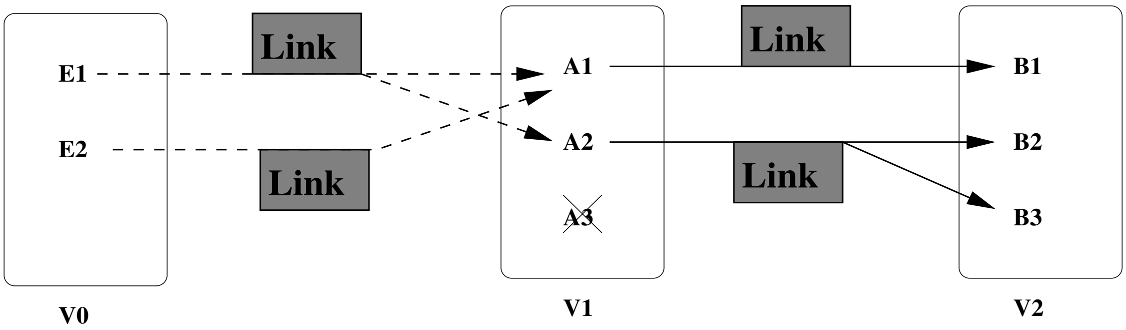


Figure 9.8: Justification $V1$ w.r.t $V2$ and $V0$ w.r.t. $V1$

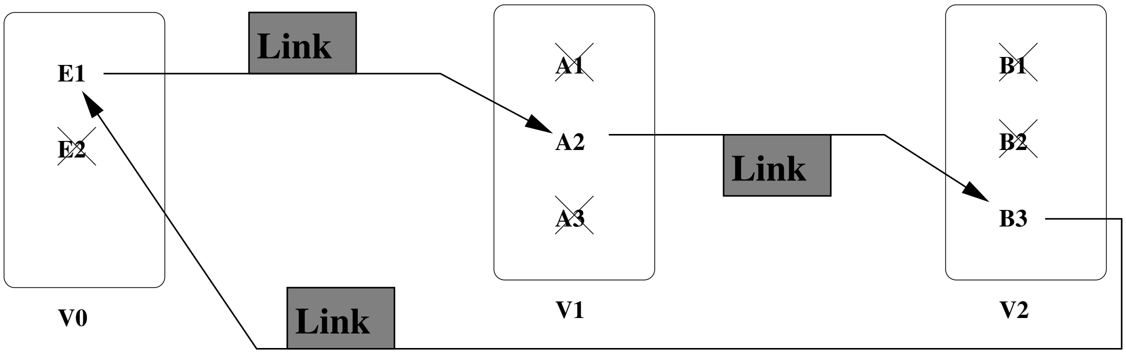


Figure 9.9: Final example justification structure

be re-evaluated. For example, If A1 was pruned as described in the discussion of single value revision, any *Link* which targeted A1 would be checked through the application of the *KeepSrcPropagateAggr* algorithm described on Table C.2.

The final hierarchical result of applying the revisions in the order described in the example above (i.e. revise $V1$ w.r.t. $V2$, revise $V0$ w.r.t. $V1$ and finally, revise $V2$ w.r.t. $V0$) is shown in Figure 9.10. Not only is each domain reduced to a single value, but in fact, these values are reduced with respect to their hierarchical structure, thus reducing the acceptable range of structure of each of the domain variables. For example, whereas E1 previously had a subcomponent L1 which specialized as either M1 or M2 in the original problem, M2 was found inconsistent, and now L1 may only specialize as M1.

Similarly, E1 generalized as one of J1 or J2 previously; however, it is only generalizable as J2 in this instance. In a similar manner, both A1 and B3 are reduced.

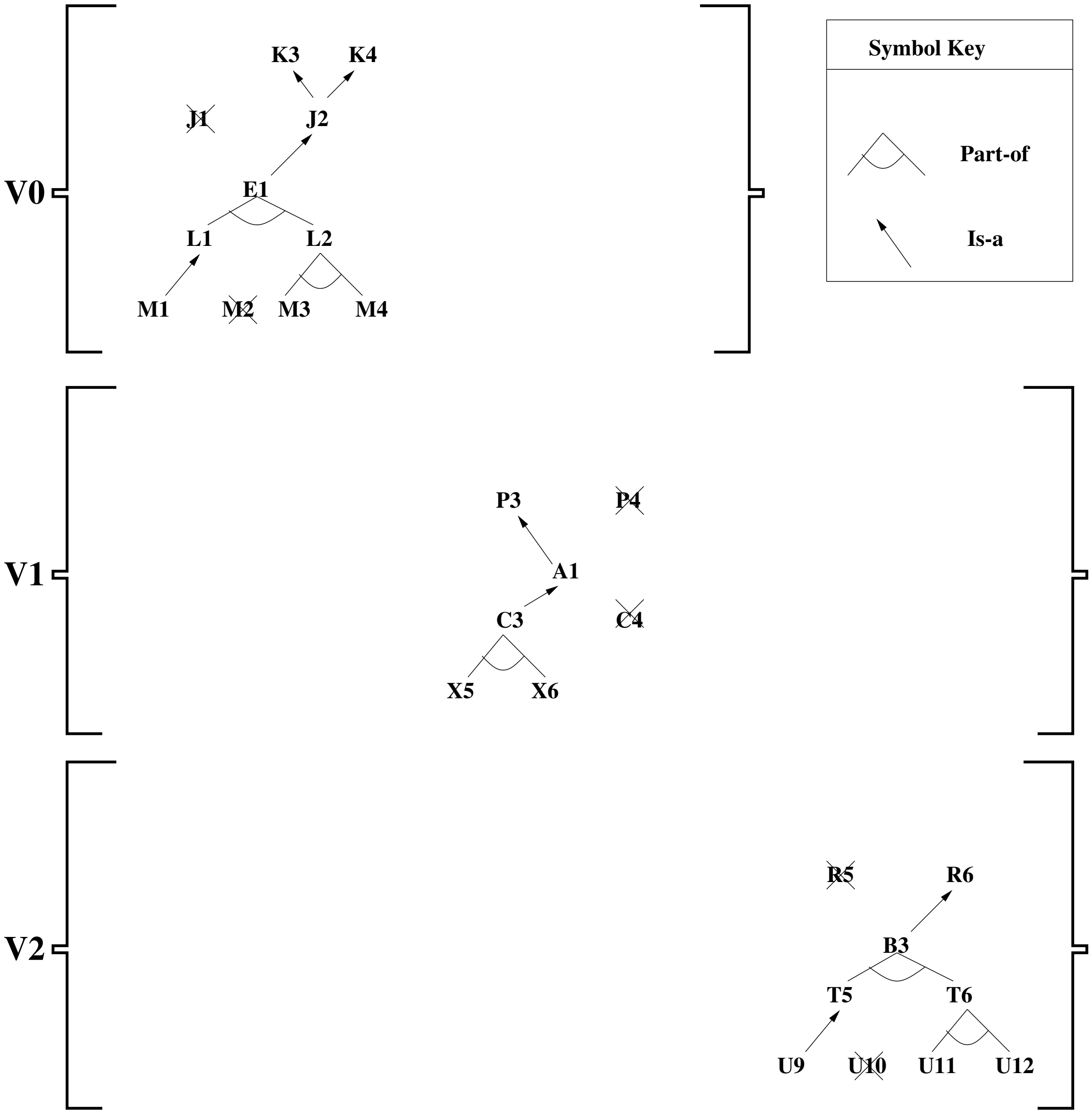


Figure 9.10: Final example hierarchic structure

9.3 Conclusion

9.3.1 Variations of Hierarchical CSP

As I have described, there are four separate arc-consistency algorithms available to us given the component algorithms described. In Table 9.20 I show the number of constraints checked in total for each of several initial variable orderings for each of the algorithms. The only incomplete results appear for stepped revision in the case in which the simplified AO-HAC-NEW algorithm is used. In all other cases the example results appear as described. The variation in constraint application value (or work) taken in each algorithm stems from early versus late eliminate of particular domain values which changes the number of times various constraints need to be checked and which also eliminates the need to ever check certain constraints in given situations. For instance, the removal of domain values B1 and B2 from the domain of $V2$ early during the algorithm(s) could save any need to check those values as possible justifiers of the E values in the $V0$ domain.

Algorithm	Order	Constraint Checks	Solution
AO-HAC-AGGR	$V0, V1, V2$	198	ok
	$V2, V0, V1$	226	ok
	$V1, V2, V0$	307	ok
AO-HAC-STEP	$V0, V1, V2$	211	ok
	$V2, V0, V1$	215	ok
	$V1, V2, V0$	218	ok
AO-HAC-NEW-AGGR	$V0, V1, V2$	173	ok
	$V2, V0, V1$	201	ok
	$V1, V2, V0$	272	ok
AO-HAC-NEW-STEP	$V0, V1, V2$	133	incomplete
	$V2, V0, V1$	173	incomplete
	$V1, V2, V0$	183	ok

Table 9.20: Hierarchical arc-consistency algorithm results

Table 9.20 has several implications. First, all of the strategies which we have indicated would provide correct, complete answers to the constructed example problem performed as expected. Second, the partial strategy AO-HAC-NEW-STEP obtained a nearly-correct solution with less work. The variations in work performed among the complete strategies opens an interesting issue — that of identifying under what circumstances each of the particular hierarchical algorithms is most appropriate or effective. We identify further investigation of the utility of the hierarchical arc-consistency algorithms as an interesting area for future research.

9.3.2 Novelty

The hierarchical CSP algorithm introduced in Chapter 8 and detailed in this chapter is novel in that it accommodates hierarchies of both **and** and **or** branch points in the definition of domain values in CSP. In particular, both decomposition and is-a relationships *downwards* and alternate-use or *or* relationships *upwards* are supported. I have shown how this extension has direct applicability to program understanding. However, this algorithm may well have other applications in hierarchical domains such as those of plan recognition which were introduced briefly in Chapter 3.

9.3.3 Correctness

I do not provide a formal proof of these hierarchical algorithms. All algorithms presented rely on the very few admissible cases evolving from "upward" and "downward" hierarchy checking. In particular, all possible combinations of these cases have been enumerated, with a behaviour specified for each particular resulting combination. After implementation of the algorithms, a complete set of test data was generated which covered all possible cases, and the results were verified as corresponding to the specified behaviour. From the viewpoint of correspondence of code and expected/desired algorithmic behaviour, the algorithms are correct. From the viewpoint of semantic correctness of the algorithms, it must be noted that the algorithms are designed to only accommodate the upward and downward cases enumerated, and consequently, situations requiring more flexibility are not covered at this time.

Part V

Conclusions

Chapter 10

Conclusions

The work presented in this thesis has several important contributions. The primary area of focus, program understanding, can be thought of as a fusion of software reverse engineering and artificial intelligence. Software engineering issues of software structure and mechanism are of primary importance in understanding since source code and program plan libraries form the basis of the input for most understanding methodologies. Artificial intelligence has long provided a venue for the study of computational knowledge representation schemes and the algorithmic means of manipulating these representations to extract new knowledge. In this research, issues in each of program understanding, software engineering, and artificial intelligence have been addressed. In this chapter the contributions of this work in each of these areas are summarized. At the end of this chapter possible future directions of this research are outlined.

10.1 Program Understanding

Program understanding was described as a primary sub-task of software reverse engineering and re-engineering in Chapter 2. The problems of dealing with large bodies of source code are areas of critical research, and promise to remain of interest for the foreseeable future. The timeliness of this work is evidenced in the ongoing quest for a way in which to dramatically increase the productivity of software development, or as Fred Brooks [Brooks, 1995] puts it, the quest for the "silver bullet". In addition, as the year 2000 looms closer, the need to address the plethora of embedded date calculation problems, which assume an ever increasing last-two-digit date sequence in COBOL and other source code, has never been greater. Visualization tools which provide a multiplicity of integrated (and possibly abstracted) views have been proposed and created which provide a basis from which it is hoped that such large scale software re-engineering projects can be launched. However, the availability of software tools (possibly embedded in such visualization toolsets) which can assist the expert programmer in the task of understanding software has been limited to relatively simple pattern matching programs to date. More ambitious projects performing abstract matching of program plans have been limited to toy sized problems as a result of, at least, an inability to deal with the combinatorics of even locally explaining large source fragments in terms of existing software libraries. An integrative model which demonstrates both how reasonably sized source code segments may be subjected to local explanatory techniques and how such partial local explanations might be merged into coherent global partial explanations has not been proposed to date.

In the area of **program understanding** it is possible to categorize the contributions of this dissertation in three ways: (1) a novel construction of a unifying, interactive model of partial global understanding is presented, (2) this model is used as the basis for the standardization of algorithms for search and control strategies for understanding, and (3) implementation and experimentation with the new model is presented showing improved scalability for the primary understanding sub-problem of partial local explanation through program plan matching.

Unifying Model of Understanding

The representational formalism of constraint satisfaction is used to construct a model of program understanding. This model enforces explicit specification of constraint information in inter-component relationships in program plan templates, and among program plan templates in a hierarchical library. Program plan templates from a hierarchically structured plan library are composed as sets of components related by *knowledge* constraints. Analysis of source code results in the production of a set of analogous *structural* constraints amongst code components. The process of understanding is expressed as the construction of (possibly partial) mappings between source and library components such that relevant knowledge and structural constraints correspond. These mappings may be made at varying degrees of decomposition and abstraction as a result of the hierarchical nature of both plan libraries and source code.

The novel algorithmic matching methodology I have presented has two primary algorithmic components which have been identified from analysis of previous program understanding methodologies. The first component finds all instances of a given program plan template in a source code. These instances may be thought of as partial local explanations of the source code. The second component constructs one or more explanations of a larger source code segment consistent with the program plan library and the current set of partial local explanations.

Previous understanding representations and approaches can be modeled within the constraints of this framework. These earlier approaches have tended to focus primarily on issues of representation and eschewed specific discussions of the role which problem structure and constrainedness plays in overall search effectiveness. In contrast, constraint satisfaction search and consistency-propagation paradigms focus almost exclusively on the relative constrainedness of certain problem features, and allow us to examine the relative effectiveness of various heuristic approaches and to determine how to best represent and exploit features of these heuristics in a single model. The goal of this work has been to provide the program understanding community a shared common reference point for the discussion of empirical effectiveness.

In this thesis, recent understanding efforts are illustrated within the context of the CSP model with the direct assistance of the authors of one of these efforts (Quilici), and the collaboration of another (Ning). The representation of these heuristic understanding strategies as variants of constraint satisfaction algorithms and heuristics has allowed us to compare these varying strategies directly both analytically and empirically. Further, in Section 5.6.2 and 7.1.4 it is demonstrated that portions of these strategies can be seen to be out-performed by the application of generic constraint-satisfaction strategies. The proposition of this model of program understanding has been widely reported (see [Woods and Yang, 1995b], [Woods and Yang, 1995a], [Woods and Yang, 1996c], [Woods and Yang, 1996a], [Woods and Quilici, 1996c], and [Woods and Quilici, 1996a]), and has met with growing acceptance and enthusiasm in the program understanding and search communities.

Modeling understanding as a CSP has had the direct benefit of allowing for the straight-forward creation of a skeletal model of program understanding. This simplified model expresses a minimal conception of what is generally referred to in the literature as program understanding. On the basis of this skeletal but concise model, a proof has been constructed that the problem of matching program plan templates to source code is NP-hard. In addition, the complementary problem of explaining source code components with respect to a given program plan library is also shown to be NP-hard. While these results are not particularly surprising, this effort marks the first formal work to demonstrate NP-hardness for this problem. The subsequent confidence in the lack of existence of a polynomial algorithm for even the sub-problem of partial local explanation is a strong motivating factor for investigating CSP solution strategies which have been created specifically for exponential but richly constrained problems.

Search and Control Standardization

Recent program understanding efforts have reported their empirical results either sparsely or not at all. The small amount of empirical reporting has itself been vague about important implementational details. For example, in both plan recognition and program understanding the expression of these tasks as one kind of search or other has been accompanied by allusions to the relatively "obvious" search-reducing benefit of propagating unspecified constraints during the recognition process. What has been lacking is a well-understood and precise model in which this propagation could be studied. My work builds upon these earlier models by making explicit this important propagation. Figure 10.1 details the major works from which my CSP model is derivative.

Quilici extended the CONCEPT RECOGNIZER of Kozaczynski and Ning through the addition of "indexing", a strategy in which a subset of the definitional constraints of templates are defined in advance and are used as a simpler "restriction" heuristic to identify possible matches. These indexed matches are later refined to check a more confident match condition. The conception behind such a heuristic is that student programmers were observed exploiting these simpler, more obvious "markers" of interest matching a particular subset of possibilities, and then refining these partial matches to either complete matches or failures.

It is possible to model such a strategy as a constraint satisfaction search heuristic. A template is partitioned, according to a simple application of the partial constraint satisfaction (PCSP) approach of Freuder and Wallace [Freuder and Wallace, 1992] into a subset of all template constraints and components capturing the "marker" aspects of the template separate from the remaining constraints and components. Identification of all

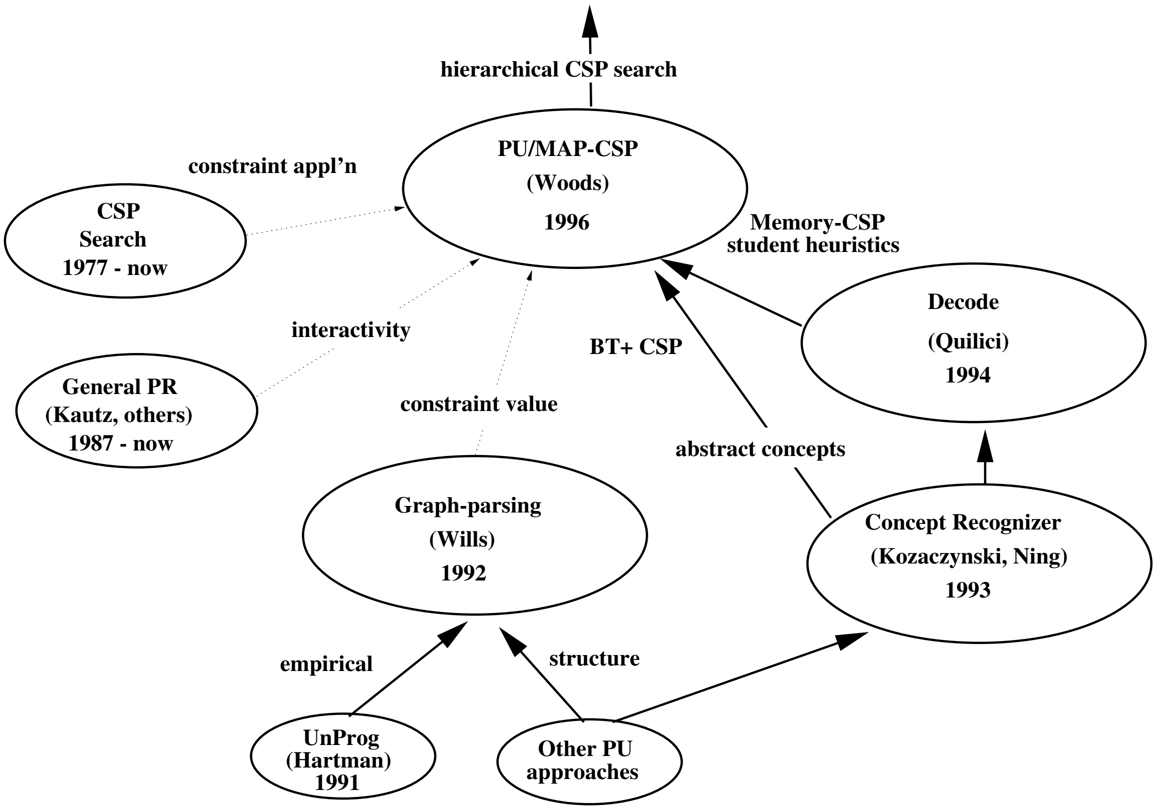


Figure 10.1: The recent Program Understanding world

solutions to the marker constraint and component set results in a set of "index hits". Each of these index hits can now be resolved according to the systematic application of the remaining or completion constraints. This methodology may be thought of as a partial ordering of the application of constraints during search. If one were able to identify a very restrictive (and small) component and constraint set this approach has a very intuitive appeal. For instance, if a particular program construct happened to occur in only a very small portion of the total number of plans in a library, it can be seen as an effective marker which effectively reduces the number of possible completions. Similarly, if one were attempting to match all instances of a particular program plan in a given range of software source, one would select one subset of the template components and constraints as markers and locate these first. In practice, one would likely build indices based on these identified markers in advance of any matching to further improve performance.

This index-based approach has been implemented and tested as a constraint satisfaction heuristic. The advantage of such an approach over previous un-indexed methodologies is now obvious in terms of the portions of the search space that can be effectively skipped over through such indexing. However, It can also be seen that in the absence of a direct indexing advantage to program statement type identification a pre-determined partitioning may not be advantageous in general. In particular, such an approach relies on the specific assumption that the index portions of the template are rare relative to the resolution portions. If this assumption breaks down, then the static ordering imposed by a one-time partitioning can be much less effective than a dynamic ordering of template components that might be undertaken by a generic constraint satisfaction heuristic such as smallest-domain-first variable-ordering.

In just this fashion the CSP model can be utilized to investigate the relative merits of other understanding heuristics with direct reference to the empirical results of such application and interpretation based on the standard search space given by constraint satisfaction problems. The task of identifying portions of the space which are being avoided by a given heuristic is greatly simplified. Previous comparisons of empirical results have been limited to discussions of relative CPU time and other measures of computational work. The problems of comparisons across different machine platforms and highly variable definitions of computational work can be alleviated by making the measure of constraint applications during reasoning the consistent point of comparison between methodologies.

Improved Scalability in Template Matching

This work constitutes the most comprehensive reporting of experimental results for program understanding to date. In particular, I have shown that partial local explanations in generated source examples can be accomplished with less work (and in less time) than previously demonstrated. For example, identification of substantially sized template instances in source fragments of 1,500 lines of code has been demonstrated with about 20 thousand constraint checks in 23 seconds on a shared SparcServer 1000 workstation with the search algorithms prototyped in Allegro Common Lisp. Order of magnitude time increases over these results might be possible with a more powerful, dedicated machine and production-style C implementation. Consequently, recognition of these code fragments can be accomplished in very nearly interactive response times. Reducing the fragment size to half of the 1,500 lines to 750 lines of code results in a search time frame of about five seconds, and a subsequent order of magnitude speed-up could reduce the effective search time to about one half of one second. It should be noted that these reported search results are obtained with *complete* strategies, with negative local results providing equally valuable information to a global explanation process.[1]

Through modeling the CONCEPT RECOGNIZER and DECODE approaches as corresponding constraint satisfaction methodologies, certain generic CSP strategies such as forward-checking with dynamic-rearrangement have been identified as significantly more efficient than domain-specific heuristics over a range of generated program examples and statement distributions. In addition, the experiments undertaken may be thought of as under-constrained relative to a production-level application. Constraint checking of data-flow and control-flow constraints in the prototype is done at constraint-application time,

[1]While I have observed previously that the use of local strategies in place of complete strategies is a promising area for future work, such strategies lose the ability to report necessarily negative results to a global explanation process.

whereas this work could be more profitably performed in-advance of search through application of specialized data-flow extraction routines such as offered in REFINE [Burn, 1992], GEN++ [Devanbu, 1992, Devanbu and Eaves, 1994] or other similar tools. This enhanced representational scheme will reduce the amount of effort required to check a particular constraint by limiting the range of focus around the involved components. While the current implementation of template matching is restricted to data-flow constraint checking, the addition of structural control-flow constraints to the program representation can be exploited directly through analogous representations in the template plans themselves, constraining the matching problem even further.

Unified interactive control

The overall understanding algorithm described in Section 8.3.3 supports a view of understanding as both interactive and iterative. The method of interleaving the discovery of local partial explanations or program plan template instances with the broader task of explaining the function of larger program components by constraint propagation can function with or without expert assistance in the form of selecting, ordering, or rejecting alternative hypotheses during search. This model is capable of integrating both heuristic control strategies and expert observation and suggestion in an interactive, expert-configurable fashion.

10.2 Artificial Intelligence

In the area of **artificial intelligence** the contribution of this work has three primary aspects. First, the program understanding problem is identified as a special case of plan recognition in which software reverse engineering algorithms have been designed to address the restricted plan recognition domain. In particular, these algorithms are able to exploit specific restrictive problem features to empirical advantage. Second, the new application domain of software reverse engineering has been identified as a rich testbed for constraint satisfaction problem (CSP) representation and solution schemes. My work has provided the opportunity for specific application of local, global and hierarchical constraint satisfaction methods. Similarly, program understanding researchers have had the opportunity to see the value of formally representing problems in the CSP framework - increased scalability and standardization of heuristic representations. Finally, through working in the software engineering world with the CSP modeling paradigm, a novel algorithm for propagating consistency in a constraint graph significantly advances the current state of the art. In particular, this algorithm is intended to accommodate domain values situated in a hierarchical structure consisting of both *is-a* and *is-part-of* relationships whereas previous work accommodated only *is-a* relations. This work is easily generalizable to other problems in which domain values can be hierarchically structured.

Program understanding (PU) and plan recognition (PR)

While I have identified the understanding sub-task of software engineering as the focus for my work, it is necessary to make the observation that the plan recognition sub-discipline of artificial intelligence addresses a very similar problem. Plan recognition research has typically focused on both hierarchical representations of plans and methodologies for the (possibly interactive) matching of a sequence of (usually related) observations to a hierarchical plan library. Frequently such research has focused on cooperative plan recognition such as might be encountered with a person interacting with an information-provider such as a course scheduling advising system. Software understanding differs significantly from this type of cooperative and sequential plan recognition.

Program understanding is often viewed as a task of understanding the plans inherent in a software code. Plan recognition research claims a similar goal for plans evidenced in action observation. The results of a comparative study between the foundational plan recognition work of Kautz [Kautz, 1987, Kautz and Allen, 1986] and program understanding is described in detail in Chapter 3, and can be summarized through the understanding that program understanding is a special case of plan recognition. In particular, PR and PU approaches exhibit several marked similarities:

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

It is readily observable that PR and PU also differ in significant ways. In particular, PU makes stronger assumptions about the format of the input and can exploit these assumptions in more restrictive algorithms for mapping between observations (code) and knowledge (library). Some of these assumptions include:

- The Kautz PR strategy assumes a complete library and incomplete observation set, whereas the PU domain of interest in this work focuses on producing as complete a mapping as possible between a complete observation set (program) and a "complete" library (partial set of knowledge).
- PR is necessarily incremental in the sense that knowledge is assumed to be arriving after a "current" reasoning step. Thus, a PR strategy forms minimal coverings or explanations which allow for un-encountered knowledge which may later appear. PU is more restrictive in that un-encountered knowledge can be discounted and any explanations relying upon such knowledge may be discarded.
- PR suffers from a less-restrictive constraint set upon which to limit the combina-

torial problem of disjunctive explanation. While PU may exploit the wealth of structural constraints easily-extracted from the source before recognition, examples in the literature of PR have primarily dealt with only explicit temporal constraints. Consequently, one may expect to solve larger PU problems more efficiently than comparably sized PR problems.

- PU can be thought of as a special, well constrained, case of PR which remains difficult (NP-hard). While it has been seen that general PR approaches are inapplicable to typical PU problem instances, it should be emphasized that one important result of this study is the suggestion that the techniques used in PU be considered for the more general PR problem. In particular, certain PR problem instances could admit pre-processing of the observation set to identify particular causal relationships. These explicit relationships should be applied in conjunction with action representations so as to increase the number and type of constraints available in the problem solution.

This work has demonstrated that PR algorithms have been designed to cover a wider range of less constrained observations and knowledge that is required for PU. Careful analysis of the nature of the restrictive assumptions that program understanding has imposed can result in the specialization of existing plan recognition algorithms such that the restrictions are more readily program explored in addressing partial understanding. This thesis has been concerned with the construction of a variant of a plan recognition algorithm which has been shown to be effective in efficiently recognizing certain classes of plans in real-world programs.

Artificial Intelligence (AI) and Software Engineering Crossover

In any work which seeks to apply techniques across the "borders" of disciplines it is important to note where such boundaries have been breached. In particular, AI techniques have been brought to bear on the software engineering sub-task of program understanding. The modeling of program understanding as constraint satisfaction and the comparison with plan recognition work opens both of these worlds to the software engineering community to some degree. In turn, the program understanding problem is an excellent application domain for AI researchers. The identification of problems in the real world in which AI can be immediately useful frequently is of little use due to the difficulty of the problems found. AI techniques typically based on search have long suffered from criticism based on their relative "slowness" and inability to deal well with interaction with users who frequently needs answers on an "anytime" basis, and who are unused to dealing with problems in a strictly "logical" framework. By structuring CSP solution strategies in such as way as to compartment complete search as a selectable tool in an overall strategy based upon *supporting* an expert in the execution of the understanding task these types of failings are at least partially redressed. The conception of AI subsystems as integral to expert decision support systems is not new; however, as the power of CSP-type algorithms to limit complexity in certain user-selected areas is further identified, the usefulness of such tools will become more obvious.

Hierarchical Arc Consistency

Constraint satisfaction is currently a very active area of research in AI, with new algorithms appearing frequently. Examples of such work includes that of Prosser [Prosser, 1993] and Kondrak and van Beek [Kondrak and van Beek, 1995] among many others. An understanding of both the suitability of particular algorithms to certain problem classes, and recognition of hard classes of given problem instances is being accumulated (see [Cheeseman *et al.*, 1991], [Smith and Grant, 1995], [Gent and Walsh, 1994], [Gent and Walsh, 1993], [Hogg and Williams, 1994], [Crawford and Auton, 1993], and [Mitchell *et al.*, 1992]).

The problem of partially understanding source code through generating correspondences to a hierarchical program plan library requires the ability to create multi-level mappings between code and hierarchy. Previously developed constraint satisfaction algorithms deal primarily with discrete, non-structured domains. One previous algorithm **Hierarchical Arc Consistency** (HAC) [Mackworth *et al.*, 1985] is capable of producing a consistency algorithm for domains which have elements that may be structured hierarchically in **is-a** (or set/subset) relationships. However, a simplified program plan library for program understanding such as has been described involves at least the two relations **is-a** (OR) and **is-subpart** (AND). In response to this requirement, the algorithm **AndOr-HAC** or AO-HAC has been created. This algorithm enforces arc-consistency throughout a constraint graph in which the domain values belong to complete or partial and/or hierarchies. This novel algorithm consequently supports the propagation of partial information about the identity of domain value assignments and reduces the combinatorial space of integrating related partial explanations.

AO-HAC extends the state of the art in constraint satisfaction by fully specifying how to exploit a particularly structured hierarchical breakdown of domain values in a CSP. In Section 8.3.1 I have demonstrated the domain-independent utility of AO-HAC given particularly structured examples, while in Section 8.3.3 I have contextualized the usefulness of AO-HAC in terms of program understanding examples. Chapter 9 details our implementation of AO-HAC .

10.3 Research Extensions and Future Work

The work outlined in this thesis is situated within a diverse and complex landscape of research issues. I have necessarily needed to make many assumptions about where the scope of this work would need to be limited. Consequently, there are many opportunities for fruitful extension and elaboration to my research. In this section I briefly outline some of the primary future research directions that are suggested by my results.

Full Tool Construction

In order to create a fully functional and useful partial recognition tool for commercial-scale reverse engineering, several aspects of my work need to be extended and coupled with other work. These include at least the integration of the matching engine with an interface for visualizing source code and source libraries, and an extension of the extraction of structural constraints of different types from the source.

Visual Interface Construction

Any interactive system requires an easy to use and well configured user interface. Program understanding is a particularly complicated task since the object of scrutiny can be an extremely large source code. Many conceivable views of this code exist other than pure code viewing. Some of these include data-flow, control-flow, and data-based displays indicating how and where data stores are modified. Of course one may wish to view parts of the source in detail and abstract away certain details from other parts. Any combination of these myriad views on code are possible, and it is quite conceivable that different experts will prefer widely varying views of the code.

Visual systems for code interpretation have been presented and researched by many researchers and corporations, a few of which include Müller's RIGI [Müller *et al.*, 1994, Müller, 1995], Quilici's DECODE [Quilici and Chin, 1995] and Reasoning Systems's Refine Language Tools and Software Refinery REFINE [Burn, 1992, Markosian *et al.*, 1994a]. These systems and subsequent efforts surrounding these systems address a myriad of issues which arise from the attempt to present information in a cognitively useful fashion for extremely complex and large domains such as program source code. If one were to attempt an adaptation of my recognition toolset so that it would be useful to a software reverse engineer, it would be critical that these issues be carefully addressed. Even apparently simple issues such as the display and editing of program libraries and program plans require significant effort in understanding how such information is best conveyed to the potential users of such a toolset.

Extended Structural Constraint Extraction

As mentioned in Section 10.1, the more structural constraints that are available to match against library or plan template knowledge constraints, the more efficient my recognition algorithm. Tools for the extraction of such information exist, however they are typically either commercial or proprietary. For instance, AT & T Bell Labs' tool GENOA has been used to create a C parser GEN++ which is capable of producing ASTs annotated with control and data-flow information. AT & T has expressed an interest in future collaborations towards the goal of connecting my recognition system to GEN++ in order to explore the benefit of these additional constraints.

Plan Library Construction

Clearly, any definitive effort at partial program understanding requires a sufficient space of related program plans in which to form the basis for limited matching and hence understanding. Such libraries currently exist only indirectly in the form of diverse shared libraries of code written in particular languages. Work in automatically abstracting such representations and combining these plan libraries to form a diverse basis for recognition is required before any such application can hope to accommodate a sufficiently broad set of programs. While this effort has been outside the scope of this dissertation, a very useful research effort would be to attempt to define (as a starting point) a minimal plan library that remains useful for limited reverse engineering of large programs.

Design Pattern Recognition

Interest in the use of "design patterns" for software development is growing. Design patterns may be thought of as high-level plans for software development, possibly domain independent. These plans, which represent some abstracted programming implementation are essentially selected and modified to fit a particular design goal. It is possible to consider that a design pattern library is analogous to a program plan library. Some recent research [Kazman and Atlee, 1996, Kazman and Reddy, 1996] in architectural re-design is interested in analyzing existing design documents to recognize instances of design patterns in a library, essentially the same task being addressed as program understanding.

One goal of this work is to identify some measure of architectural complexity based on the coverage of a design specification by a given pattern library. For example, if a large percentage of the source is covered by the library, then one may conclude that the specification is relatively less *complex* than a design which is only sparsely covered. Similarly, if a specification is covered by fewer *different* design patterns, it may be considered a *simpler* design than one covered by many different patterns. Analogies to the sub-graph isomorphism problem around which I have modeled program understanding, and to the more generic graph coloring problem are obvious. This work is intended to be applied to architectural re-design efforts. An architectural language is in development [Kazman and Atlee, 1996] which allows for the representation of both design patterns (incorporating architectural styles) and specification. Thus the "intermediate" level matching problem of identifying source to an abstracted plan is greatly clarified. A collaboration has been started in which the CSP model of program understanding is being extended to apply to the problem of design coverage recognition.

Just as my approach to program understanding can be seen as an attempt to exploit intended structure in software during the iterative understanding process, an analysis of the relationship between design patterns, idioms and styles and the rationale for the internal structure of these objects is also presented in [Kazman and Reddy, 1996], including a presentation of a theory of primitive design operations or unit operations which have been derived through a study of software design literature and interviews with expert designers. Unit operations are structure-modifying operations regularly employed by designers such as abstraction, resource sharing, is-a decomposition, and the like. The primary result of this study is a set of design rules clarifying the relationships among unit operations, system requirements, and non-functional qualities such as style.

Concept Cluster Identification

The identification of partial local explanations of source code through use of MAP-CSP has application as a sub-portion of the larger understanding process as I have described. However, this process may also be considered as a stand-alone tool for identifying particular elements or occurrences in source. If one considers that any related set of program components is a *concept cluster*, then it may be useful to identify instances or partial instances of this cluster in a particular source code. In this way one may distinguish a concept cluster from a program plan, in that a cluster might represent several portions of different program plans, or in fact, may not correspond to any know program plan at all. For instance the Year-2000 problem involves locating all instances in a particular source program in which dates are used based on the assumption that the last two digits of a four digit date are always increasing. For example, 1966 comes after 1965. Frequently these dates have been encoded in two-digit fields such as 65 and 66. Unfortunately, as we progress from 1999 to 2000 the last two digits change from 99 to 00. Consequently, a great deal of source code will fail as calculations are invoked which attempt to make reference to the year after 99.

There are a nearly infinite number of ways in which the concept of this date exploitation may appear in source code. If one could, however, represent even some small percentage of these instances as concept clusters and recognize these in large source code segments it would be a great benefit to those experts charged with altering millions of lines of source code. As an example, it would be simple to represent a cluster as the definition of a two digit character field, the initialization of this field through reference to a system clock or date structure, and the later manipulation of this field through addition of a small integer, or perhaps indexing some secondary structure with this value field. Such cases could easily include year-ahead reporting or table-lookup for a particular year. Any reverse engineering for this problem will include these types of reference and a proliferation of others.

Semantics and Syntax

One open question is how to integrate efficient Unix-type syntactic matching mechanisms effectively with the more "relational" view of source programs that MAP-CSP utilizes. MAP-CSP is intended to recognize templates (idioms, clichés) as instances of sets of *related* components in an intermediate representation of source code which makes data-flow and control-flow explicit. PU-CSP is intended to integrate these locally-identified idioms into a broader (or global) view which is consistent with respect to the available library of knowledge about how program plans interconnect. While the existing program plan library contains merely a set of syntactic template specifications, these specification are representative of actual program concepts. In a sense then, the identification of instances of these syntactic templates in a source code representation is an annotation of the *syntax* of the source with the *semantics* encoded into the plan library. Other methods for handling syntax can be complimentary to this approach. For example, Unix tools such as "grep" and "awk" can be configured to recognize a range of string-based text combinations in source code, and are powerful enough to combine these instances with respect to inter-string relationships. While such tools can be complimentary and offer specific, efficient algorithms for string-matching, they cannot deal with information about source code which is textually implicit such as abstract syntax tree structure. This information can be made explicit through the use of a language-specific *parser*, essentially allowing one to deal with the intentionally embedded structure of a programming language, whereas Unix-type matching tools treat source as a flat text-based artifact. Undoubtedly humans make use of both knowledge of the programming language structure and purely text-based keys (such as keyword names and their relation to particular idioms) in understanding source code, and the integration of these paradigms offers an interesting future research area.

Evaluating CSP Effectiveness

A totally unconstrained CSP (one with all existing constraints evaluating to true in all domain value combinations) has a search space in which, for N variables and M domain values per variable, there are M^N solutions. The "constrainedness" of a given CSP is related to the degree to which constraints exist among variables, and also to the degree to which each constraint returns true or false. CSPs range from over-constrained, in which solutions are very rare or non-existent in the vast space of candidate solutions, or under-constrained, in which solutions are readily found in the space. Pathological (very difficult to solve) CSPs tend to have a large number of satisfiable constraints (leading search deeply in the space), but very few total solutions (in which all variables have consistent assignments). An interesting area for future work is to determine, through analysis of CSPs generated from real-world programs, how difficult program understanding (MAP and PU) CSPs really are to solve. In particular, if program understanding CSPs are not that difficult, they may be susceptible to more specific algorithms that exploit the properties or structure of the problems that make them easy.

Experimental Improvements

One of the primary criticisms of my work has been the nature of the generated source data according to the statistical distribution of program statements in real source programs instead of using actual source programs. One advantage of my approach is the ability to study many different program instances of the same size and similar composition. However, in future work it is necessary for us to annotate real programs with data-flow and control-flow constraints and modify my internal representation to use them, rather than the current approximations. This task is complicated by the need for extended code parser/annotaters which are generally proprietary. An agreement between the creator of one such parser, AT & T Bell Labs has been recently arranged, and together with the creator of DECODE, research is moving forward with this important extension. However, for the purposes of this thesis, the goal of our research has been to export the CSP modeling paradigm and demonstrate the feasibility of unified comparisons, and this goal has been met.

Another primary concern from my experiments is that my particular approach to mapping heuristic program understanding methodologies to constraint satisfaction-based problems has led to additional constraint matches or work actually excluded in the original implementations. Despite working in conjunction with the original authors, it is possible that some subtle side-effects have been overlooked. In addition, the CSP models may now be adapted with particular heuristics for even better performance. For example, in the Memory-CSP model of DECODE, the two phase approach first finds all N index matches for a given plan and then generates N resolution problems. The intuition is that as search (refinement) continues, these will either quickly fail or will result in a recognized plan instance. This type of behaviour is observed, however, 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. Consequently, a new, improved version of Memory-CSP may evolve superior to both the original and subsequent generic-CSP solutions.

One important aspect of my continuing research is to attempt to verify these initial results in more complex experimental situations involving a wider breadth of actual source programs and program plans. I intend to do so by comparing the performance of the program understanding algorithms introduced with "real-world" programs as input. In addition, I wish to explore in more detail the specific differences between how various CSP-based understanders perform plan recognition — my goal is to verify that I have accurately captured all of the nuances of indexing with my CSP-implementation of memory-based understanding, and to better understand exactly why the domain-independent CSP algorithm appears to perform so much better. Any such examination will involve careful comparisons of the search spaces and behaviour within the CSP framework, and to this end I wish to work further with previous understanding authors to incorporate more CSP-models of program understanding approaches.

Bibliography

[Agre and Chapman, 1987] P. Agre and D. Chapman. Pengi: An implementation of a theory of activity. In *Proceedings of the 6th AAAI*, pages 268–272, 1987.

[Allen *et al.*, 1990] James Allen, James Hendler, and Austin Tate, editors. *Readings in Planning*. Morgan Kaufmann, 1990.

[Ardissono and Cohen, 1996a] Liliana Ardissono and Robin Cohen. Extending the role of user feedback in plan recognition and response generation for advice-giving systems: an initial report. In *Proceedings of the 1996 Canadian Artificial Intelligence Conference*, May 1996.

[Ardissono and Cohen, 1996b] Liliana Ardissono and Robin Cohen. Improving repsonse generation by using abstract decompositions of actions and redefining relevant plan ambiguity. Draft version, 1996.

[Brooks, 1995] Frederick P. Jr. Brooks. *The mythical man-month: essays on software engineering*. Addison-Wesley Publishing Company, anniversary edition, 1995.

[Burn, 1992] Jules Burn. Overview of software refinery: Product family for automated software analysis and transformation, reasoning. Product description, Reasoning Systems, Palo Alto, CA, jules@reasoning.com, 1992.

[Carberry, 1988] Sandra Carberry. Modeling the user's plans and goals. *Computational Linguistics*, 14(3):23–37, 1988.

[Carberry, 1990a] Sandra Carberry. Incorporating default inferences into plan recognition. *Proceedings of the 8th AAAI*, 1:471–478, 1990.

[Carberry, 1990b] Sandra Carberry. A new look at plan recognition in natural language dialogue. Technical Report 90-08, University of Delaware, 1990.

[Cheeseman *et al.*, 1991] P. Cheeseman, B. Kanefsky, and W.M. Taylor. Where the really hard problems are. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence*, pages 331–337, 1991.

[Chin and Quilici, 1996] D. Chin and A. Quilici. Decode: A cooperative program understanding environment. *Journal of Software Maintenance*, 8(1), 1996.

[Citrin *et al.*, 1996] Wayne Citrin, Carlos Santiago, and Benjamin Zorn. Scalable interfaces to support program comprehension. In *Proceedings of the 4th IEEE Workshop on Program Comprehension (WPC-96)*, Berlin, Germany, March 1996.

[Cohen and Spencer, 1993] Robin Cohen and Bruce Spencer. Specifying and updating plan libraries for plan recognition tasks. Technical Report cs-93-10, University of Waterloo, 1993.

[Cooper, 1989] Martin C. Cooper. An optimal k-consistency algorithm. *Artificial Intelligence*, 41:89–95, 1989.

[Cordy *et al.*, 1991] J.R. Cordy, C.D. Halpern, and E. Promislow. TXL: A rapid prototyping system for programming language dialects. *Computer Languages*, 16(1):97–107, January 1991.

[Crawford and Auton, 1993] J.M. Crawford and L.D. Auton. Experimental results on the crossover point in satisfiability problems. In *Proceedings of the 11th AAAI*, pages 21–27, 1993.

[De Pauw *et al.*, 1994] Wim De Pauw, Doug Kimeham, and John Vlissides. Modeling object-oriented program execution. In *ECOOP '94*, July 1994.

[Dechter and Dechter, 1987] A. Dechter and R. Dechter. Removing redundancies in constraint networks. In *Proceedings of the 6th AAAI*, pages 105–109, 1987.

[Dechter and Meiri, 1989] R. Dechter and I. Meiri. Experimental evaluation of preprocessing techniques in constraint s atisfaction problems. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence*, pages 271–277, Detroit, MI, 1989.

[Dechter and Pearl, 1987] R. Dechter and J. Pearl. Network-based heuristics for constraint-satisfaction problems. *Artificial Intelligence*, 34, 1987.

[Dechter and Pearl, 1989] R. Dechter and J. Pearl. Tree clustering for constraint networks. *Artificial Intelligence*, 38:353–366, 1989.

[Dechter, 1990a] R. Dechter. From local to global consistency. In *Eighth Canadian Conference on Artificial Intelligence*, 1990.

[Dechter, 1990b] Rina Dechter. Enhancement schemes for constraint processing: backjumping, learning, and cutset decomposition. *Artificial Intelligence*, 41:273–312, 1990.

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

[Devanbu and Eaves, 1994] Prem Devanbu and Laura Eaves. **Gen++** - an analyzer generator for c++ programs. Technical report, AT & T Bell Labs, New Jersey, 1994.

[Devanbu, 1992] P. Devanbu. **GENOA/GENII** - a customizable, language- and front-end- independent code analyzer. *Proceedings of the 14th International Conference on Software Engineering*, 1992.

[Freuder and Mackworth, 1992] Eugene Freuder and Alan Mackworth. Introduction to the special volume on constraint-based reasoning. *Artificial Intelligence*, 58:1–2, 1992.

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

[Freuder, 1982] E.C. Freuder. A sufficient condition of backtrack-free search. *Journal of the ACM*, 29(1):23–32, 1982.

[Freuder, 1991] Eugene C. Freuder. Eliminating interchangeable values in constraint satisfaction problems. In *Proceedings of the 9th AAAI*, pages 227–233, 1991.

[Gamma *et al.*, 1993] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In *Proceedings of the 7th European Conference on Object Oriented Programming (ECOOP)*, July 1993.

[Gamma *et al.*, 1995] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Company, 1995.

[Garey and Johnson, 1979] Michael R. Garey and David S. Johnson. *Computers and Intractability : A guide to the theory of NP-Completeness*. W. H. Freeman and Company, Bell Laboratories, Murray Hill, New Jersey, 1979.

[Gent and Walsh, 1993] I.P. Gent and T. Walsh. An empirical analysis of search in GSAT. *Journal of Artificial Intelligence Research*, 1:47–59, 1993.

[Gent and Walsh, 1994] I.P. Gent and T. Walsh. Easy problems are sometimes hard. *Artificial Intelligence*, 70:335–345, 1994.

[Gu, 1992] Jun Gu. Efficient local search for very large-scale satisfiability problems. *SIGART Bulletin*, 3(1), 1992.

[Guan and Friedrich, 1992] Qi Guan and Gerhard Friedrich. Extending constraint satisfaction problem solving in structural design. In F. Belli and F.J. Radermacher, editors, *Lecture Notes in Computer Science*, volume 604, pages 341–350. Springer-Verlag, 1992.

[Hammond, 1990] Kristian J. Hammond. Case-based planning: A framework for planning from experience. *Cognitive Science*, 14:385–443, 1990.

[Haralick and Elliott, 1980] R.M. Haralick and G.L Elliott. Increasing tree-search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.

[Hartman, 1991a] J. Hartman. *Automatic control understanding for natural programs.* Research report ut-ai-tr-91-161, University of Texas at Austin, 1991.

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

[Hartman, 1992a] J. Hartman. Pragmatic, empirical program understanding. In *Workshop Notes, AAAI Workshop on AI and Automated Program Understanding*, Tenth National Conference on Artificial Intelligence, 1992.

[Hartman, 1992b] J. Hartman. Technical introduction to the first workshop on AI and automated program understanding. In *Workshop Notes, AAAI Workshop on AI and Automated Program Understanding*, Tenth National Conference on Artificial Intelligence, 1992.

[Hartman, 1994] J. Hartman. Plans in software engineering - an overview. Technical report, Laboratory for Artificial Intelligence Research, Ohio State University, November 1994.

[Hogg and Williams, 1994] T. Hogg and C.P. Williams. A double phase transition. *Artificial Intelligence*, 69:359–377, 1994.

[Holte *et al.*, 1995] R. Holte, T. Mkadmi, R. Zimmer, and A. MacDonald. Speeding up problem-solving by abstraction: A graph-oriented approach. Technical report TR-95-07, University of Ottawa, March 1995.

[Hubbe and Freuder, 1992] Paul Hubbe and Eugene Freuder. An efficient cross product representation of the constraint satisfaction problem search space. In *Proceedings of the 10th AAAI*, pages 421–427, 1992.

[Johnson and Soloway, 1985] W. L. Johnson and E. Soloway. Proust: Knowledge-based program understanding. *IEEE Transactions on Sofware Engineering*, 11, 1985.

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

[Kautz and Allen, 1986] Henry Kautz and James Allen. Generalized plan recognition. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, pages 32–37, Philadelphia, Pennsylvania, 1986.

[Kautz and Selman, 1995] Henry Kautz and Bart Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. Personal communication, 1995.

[Kautz, 1987] Henry Kautz. *A Formal Theory of Plan Recognition.* PhD thesis, University of Rochester, Department of Computer Science, Rochester, New York, 1987.

[Kazman and Atlee, 1996] Rick Kazman and Joanne Atlee. Design pattern recognition during the re-design process, April 1996. Personal communication.

[Kazman and Reddy, 1996] R. Kazman and K. Reddy. An empirical study of architectural design operations. Personal communication, 1996.

[Kazman *et al.*, 1996] R. Kazman, P. Clements, G. Abowd, and L. Bass. Classifying architectural elements as a foundation for mechanism matching. Personal communication, 1996.

[Kimeham *et al.*, 1994] Doug Kimeham, Bryan Rosenburg, and Tova Roth. Multi-layer visualization of dynamic in software system behaviour. In *Visualization '94*, October 1994.

[Knoblock, 1991a] Craig Knoblock. Search reduction in hierarchical problem solving. In *Proceedings of the 9th AAAI*, volume 2, pages 686–691, 1991.

[Knoblock, 1991b] Craig A. Knoblock. *Automatically Generating Abstractions for Problem Solving.* PhD thesis, School of Computer Science, Carnegie Mellon University, 1991.

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

[Kontogiannis *et al.*, 1995] K. Kontogiannis, M. Galler, and R. DeMori. Detecting code similarity using patterns. *Working Notes of the Third Workshop on AI and Software Engineering : Breaking the Toy Mold (AISE)*, pages 68–73, August 1995.

[Kozaczynski and Ning, 1994] Wojtek Kozaczynski and Jim Q. Ning. Automated program understanding by concept recognition. *Automated Software Engineering*, 1:61–78, 1994.

[Kozaczynski *et al.*, 1992] V. Kozaczynski, J. Ning, and A. Engberts. Program concept recognition and transformation. *Transactions on Software Engineering*, 18(12):1065–1075, December 1992.

[Kumar, 1992] Vipin Kumar. Algorithms for constraint-satisfaction problems. *AI Magazine*, pages 32–44, Spring 1992.

[Mackworth and Freuder, 1985] A.K. Mackworth and E.C. Freuder. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Artificial Intelligence*, 125:65–74, 1985.

[Mackworth and Freuder, 1993] Alan Mackworth and Eugene Freuder. The complexity of constraint satisfaction revistited. *Artificial Intelligence*, 59:57–62, 1993.

[Mackworth *et al.*, 1985] Alan Mackworth, Jan Mulder, and William Havens. Hierarchial arc consistency: Exploiting structured domains in constraint satisfaction problems. *Computational Intelligence*, 1:188–126, 1985.

[Mackworth, 1977] A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.

[Mackworth, 1981] A.K. Mackworth. Consistency in networks of relations. In Webber and Nilsson, editors, *Readings in Artificial Intelligence*, pages 69–78. Morgan Kaufmann Publishers Inc., 1981.

[Mackworth, 1987] A.K. Mackworth. Constraint satisfaction. In S.C. Shaprio, editor, *Encyclopedia of Artificial Intelligence*, volume 1, pages 205–211. John Wiley and Sons, 1987.

[Mackworth, 1992] Alan Mackworth. The logic of constraint satisfaction. *Artificial Intelligence*, 58:3–20, December 1992.

[Markosian *et al.*, 1994a] L. Markosian, R. Brand, and G. Kotik. Customized software evaluation tools: Application of an enabling technology for reengineering. In B. Blum, editor, *Proceedings of the Fourth Systems Reengineering Technology Workshop*, pages 248–255, Johns Hopkins University Applied Physics Laboratory, Feb 1994.

[Markosian *et al.*, 1994b] L. Markosian, P. Newcomb, R. Brand, S. Burson, and T. Kitzmiller. Using an enabling technology to re-engineer legacy systems. *Communications of the ACM*, 37(5):58–71, 1994.

[McGregor, 1979] J. McGregor. Relational consistency algorithms and their application in finding subgraph and graph isomporphisms. *Information Sciences*, 19:229–250, 1979.

[Minton *et al.*, 1990] Steven Minton, Mark Johnston, Andrew Philips, and Philip Laird. Solving large-scale constraint satisfaction and scheduling problems using a heuristic repair method. *Proceedings of the 8th AAAI*, pages 17–24, 1990.

[Minton *et al.*, 1992] Steven Minton, Mark Johnston, Andrew Philips, and Philip Laird. Minimizing conflicts: A heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, 58:161–205, 1992.

[Minton, 1990] Steve Minton. Quantitative results concerning the utility of explanation-based learning. *Artificial Intelligence*, 42:363–391, 1990.

[Mitchell *et al.*, 1992] David Mitchell, Bart Selman, and Hector Levesque. Hard and easy distributions of SAT problems. *Proceedings of the 10th AAAI*, pages 459–465, 1992.

[Mohr and Henderson, 1986] R. Mohr and T.C. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28:225–233, 1986.

[Montanari, 1974] U Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Information Sciences*, 7:95–132, 1974.

[Müller *et al.*, 1993] H. Müller, M.A. Orgun, S.R. Tilley, and J.S. Uhl. A reverse engineering approach to subsytem structure identification. *Journal of Software Maintenance*, 5(4):181–204, December 1993.

[Müller *et al.*, 1994] H. Müller, K. Wong, and S.R. Tilley. Understanding software systems using reverse engineering technology. In *Proceedings of the Colloquim on Object Orientation in Databases and Software Enginering*, pages 88–98, December 1994.

[Müller, 1995] Hausi Müller. Rigi - an extensible system for retargetable reverse engineering. World Wide Web information sheet at tara.uvic.ca/rigi/, 1995.

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

[Nadel, 1990] Bernard Nadel. Representation selection for constraint satisfaction: A case study using n-queens. *IEEE Expert*, pages 16–23, June 1990.

[Newcomb and Markosian, 1993] P. Newcomb and L. Markosian. Automating the modularization of large cobol programs: Application of an enabling technology for reengineering. In *Proceedings of the Working Conference on Reverse Engineering*, pages 222–230, 1993.

[Nilsson, 1980] Nils Nilsson. *Principles of Artificial Intelligence.* Morgan Kaufmann Publishers Inc, 1980.

[Norvig, 1992] Peter Norvig. *Paradigms of Artificial Intelligence Programming, Case Studies in Common Lisp.* Morgan Kaufmann, 1992.

[Pennington, 1987a] N. Pennington. Comprehension strategies in programming. In G.M. Olson, S. Sheppard, and E. Soloway, editors, *Empirical Studies of Programmers: Second Workshop*, pages 100–112, Norwood, N.J., 1987. Ablex Publishing Company.

[Pennington, 1987b] N. Pennington. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, 19:295–341, 1987.

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

[Quilici and Chin, 1994] Alex Quilici and David Chin. A cooperative program understanding environment. In *Proceedings of the Ninth Knowledge-Based Software Engineering Conference*, pages 125–132, Monterey, CA, 1994.

[Quilici and Chin, 1995] Alex Quilici and David Chin. DECODE: A cooperative environment for reverse-engineering legacy software. In *Proceedings of the Second Working Conference on Reverse-Engineering*, pages 156–165. IEEE Computer Society Press, July 1995.

[Quilici *et al.*, 1996] Alex Quilici, Qiang Yang, and Steven Woods. Applying plan recognition algorithms to program understanding. In *Proceedings of the Eleventh Knowledge-Based Software Engineering Conference.* IEEE Computer Society Press, September 1996. To appear.

[Quilici, 1993] A. Quilici. A hybrid approach to recognizing programming plans. In *Proceedings of the Working Conference on Reverse Engineering*, pages 126–133, Baltimore, MD, May 1993.

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

[Quilici, 1995a] Alex Quilici. Reverse engineering of legacy systems: A path toward success. *Proceedings of the 17th International Conference on Software Engineering*, 1995.

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

[Reddy, 1996] Kavita Reddy. The empirical derivation of a design space and design rules for software architecture. Technical Report CS-96-20, University of Waterloo, 1996.

[Rich and Waters, 1988] C. Rich and R.C. Waters. The Programmer's Apprentice: A research overview. *IEEE Comput.*, 21(11):10–25, 1988.

[Rich and Waters, 1990] C. Rich and R.C. Waters. *The programmer's apprentice.* Addison-Wesley, Reading, Mass., 1990.

[Rich, 1987] Charles Rich. *Inspection methods in programming.* PhD thesis, Massachusets Institute of Technology, June 1987.

[Rugaber *et al.*, 1995] Spencer Rugaber, Kurt Stirewalt, and Linda Wills. The interleaving problem in program understanding. In *Proceedings of the Second Working*

Conference on Reverse-Engineering, pages 166–175, 10662 Los Vaqueros Circle, Los Alamitos CA 90720-1264, July 1995. IEEE Computer Society Press.

[Rugaber, 1992] Spencer Rugaber. Program comprehension for reverse engineering. In *Proceedings of the 1992 AAAI Workshop on AI and Automated Program Comprehension*, 1992. San Jose, California.

[Sacerdoti, 1974] Earl Sacerdoti. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 5:115–135, 1974.

[Selfridge, 1994] Peter G. Selfridge. Report on the *first working conference on reverse engineering. Automated Software Engineering*, 1:133–139, 1994.

[Selman and Kautz, 1993] Bart Selman and Henry Kautz. Domain-independent extensions to GSAT: Solving large structured satisfiability problems. *Proceedings of the 13th International Joint Conference on Artificial Intelligence*, pages 290–294, 1993.

[Selman *et al.*, 1994] Bart Selman, Henry Kautz, and Bram Cohen. Noise strategies for improving local search. *Proceedings of the 12th AAAI*, pages 337–343, 1994.

[Sidebottom and Havens, 1992] G. Sidebottom and W.S. Havens. Hierarchical arc consistency for disjoint real intervals in constraint logic programming. *Computational Intelligence*, 8(4):601–623, November 1992.

[Simon, 1968] Herbert Simon. *The Sciences of the Artificial.* M.I.T. Press, Massachusetts, U.S.A., 1968.

[Smith and Grant, 1995] Barbara Smith and Stuart Grant. Sparse constraint graphs and exceptionally hard problems. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, pages 646–651, 1995.

[Soloway and Ehrlich, 1984] E. Soloway and K. Ehrlich. Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, SE-10(5):595–609, 1984.

[Sommerville, 1982] Ian Sommerville. *Software Engineering*. Addison-Wesley Publishing Company, 1st edition, 1982.

[Sommerville, 1996] Ian Sommerville. *Software Engineering*. Addison-Wesley Publishing Company, 5th edition, 1996.

[Song and Cohen, 1991] Fei Song and Robin Cohen. Temporal reasoning during plan recognition. *Proceedings of the 9th AAAI*, pages 247–252, 1991.

[Song, 1990] Fei Song. *A processing model for temporal analysis and its application to plan recognition*. PhD thesis, University of Waterloo, 1990.

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

[Spencer, 1991] Bruce Spencer. *Assimilation in Plan Recognition via Truth Maintenance with Reduced Redundancy*. PhD thesis, University of Waterloo, 1991.

[Storey and Müller, 1995] Margaret-Anne Storey and Hausi Müller. Manipulating and documenting software structures using SHriMP views. In *Proceedings of the 1995 International Conference on Software Maintenance (ICSM '95)*, 1995.

[Tenenberg, 1988] Josh Tenenberg. *Abstraction in Planning*. PhD thesis, University of Rochester, Dept. of Computer Science, Rochester, NY, May 1988.

[Tilley *et al.*, 1993] Scott Tilley, Hausi Müller, Michael Whitney, and Kenny Wong. Domain-retargetable reverse engineering. In *The 1993 Conference on Software Maintenance*, pages 142–151. IEEE Computer Society Press, 1993. Order number 4600-02.

[Tolba *et al.*, 1991] Hany Tolba, Francois Charpillet, and Jean-Paul Haton. Representing and propagating constraints in temporal reasoning. In *Proceedings of the International Conference on Tools for Artificial Intelligence*, pages 181–185, November 1991.

[Tsang, 1993] Edward Tsang. *Foundations of Constraint Satisfaction.* Academic Press Limited, 24-28 Oval Road, London England, NW1 7DX, 1993.

[van Beek *et al.*, 1993] Peter van Beek, Robin Cohen, and Ken Schmidt. From plan critiquing to clarification dialogue for cooperative response generation. *Computational Intelligence*, 9(3), 1993.

[Van Hentenryck *et al.*, 1992a] P. Van Hentenryck, Y. Deville, and C-M. Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57:291–321, 1992.

[Van Hentenryck *et al.*, 1992b] P. Van Hentenryck, H. Simonis, and M. Dincbas. Constraint satisfaction using constraint logic programming. *Artificial Intelligence*, 58:113–159, December 1992.

[Van Hentenryck, 1989] Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming.* The MIT Press, 1989.

[von Mayhrhauser and Vans, 1995] Anneliese von Mayhrhauser and A. Marie Vans. Program comprehension during software maintenance and evolution. *IEEE Computer*, pages 44–55, 1995.

[Waltz, 1975] D. Waltz. Understanding line drawings of scenes with shadows. In P.H. Winston, editor, *The Psychology of Computer Vision*, pages 19–91. McGraw Hill, Cambridge, Massachusets, 1975.

[Wilkins, 1988] David Wilkins. *Practical Planning: Extending the Classical AI Planning Paradigm.* Morgan Kaufmann, CA, 1988.

[Williams and Woods, 1993] Graham Williams and Steven Woods. Representing expectations in spatial information systems: A case study. In Dave Abel and Beng Chin Ooi, editors, *Proceedings of the 3rd International Conference on Large Spatial Databases*, volume 692 of *Advances in Spatial Databases, 3rd edition*, pages 465–476. Springer Verlag, June 1993. Lecture Notes in Computer Science.

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

[Wills, 1992] L. M. Wills. *Automated program recognition by Graph Parsing.* PhD thesis, MIT, July 1992.

[Wills, 1993] Linda Wills. Flexible control for program recognition. In *Proceedings of the First Working Conference on Reverse-Engineering*, pages 124–148, 10662 Los Vaqueros Circle, Los Alamitos CA 90720-1264, May 1993. IEEE Computer Society Press. Baltimore, MD.

[Woods and Quilici, 1996a] Steven Woods and Alex Quilici. A constraint-satisfaction framework for evaluating program-understanding algorithms. In *Proceedings of the 4th IEEE Workshop on Program Comprehension (WPC-96)*, Berlin, Germany, March 1996.

[Woods and Quilici, 1996b] Steven Woods and Alex Quilici. Some experiments in the scalability of program understanding algorithms. In *Proceedings of the Third Working Conference on Reverse-Engineering.* IEEE Computer Society Press, September 1996. To appear.

[Woods and Quilici, 1996c] Steven Woods and Alex Quilici. Toward a constraint-satisfaction framework for evaluating program-understanding algorithms. *Journal of Automated Software Engineering*, 1996. To appear.

[Woods and Yang, 1995a] Steven Woods and Qiang Yang. Constraint-based plan recognition in legacy code. *Working Notes of the Third Workshop on AI and Software Engineering : Breaking the Toy Mold (AISE)*, August 1995.

[Woods and Yang, 1995b] Steven Woods and Qiang Yang. Program understanding as constraint satisfaction. In *Proceedings of the IEEE Seventh International Workshop on Computer-Aided Software Engineering (CASE)*, pages 318–327. IEEE Computer Society Press, July 1995. Also appears in the *Proceedings of the 1995 Second Working Conference on Reverse Engineering (WCRE).*

[Woods and Yang, 1995c] Steven Woods and Qiang Yang. Program understanding as constraint satisfaction: Representation and reasoning techniques. Technical Report CS 95-51, University of Waterloo, Department of Computer Science, 1995.

[Woods and Yang, 1996a] Steven Woods and Qiang Yang. Approaching the program understanding problem: Analysis and a heuristic solution. In *Proceedings of the 18th International Conference on Software Engineering*, Berlin, Germany, 1996.

[Woods and Yang, 1996b] Steven Woods and Qiang Yang. Hierarchical constraint satisfaction and program understanding. Research Note in progress, 1996.

[Woods and Yang, 1996c] Steven Woods and Qiang Yang. Program understanding as constraint satisfaction: Representation and reasoning techniques. Under review, *Journal of Automated Software Engineering*, 1996.

[Woods *et al.*, 1995] Steven Woods, Alex Quilici, and Qiang Yang. Program understanding and plan recognition: reasoning under different assumptions. Technical Report CS 95-52, University of Waterloo, Department of Computer Science, 1995.

[Woods, 1991] Steven G. Woods. An implementation and evaluation of a hierarchical non-linear planner. Masters Thesis available as technical report CS-91-17, Computer Science Department, University of Waterloo, 1991.

[Woods, 1993] Steven Woods. A method of interactive recognition of spatially defined model deployment templates using abstraction. In H. Merklinger, M. Farooq, P. Roberge, J. Grodski, and R. Dobson, editors, *Proceedings of the Knowledge-Based Systems and Robotics Workshop*, pages 665–675. Department of National Defence, Government of Canada, November 1993.

[Woods, 1995] Steven Woods. A method of program understanding using constraint satisfaction for software reverse engineering. Ph.D. thesis proposal, 1995.

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

[Yang, 1990] Qiang Yang. Formalizing planning knowledge for hierarchical planning. *Computational Intelligence*, 6, 1990.

[Yang, 1992] Qiang Yang. A theory of conflict resolution in planning. *Artificial Intelligence*, 58(1-3):361–392, 1992. Special Issue on Constraint-directed Reasoning.

[Yang, 1996] Qiang Yang. *Intelligent Planning - algorithms and analyses for plan reasoning*. 1996. Advance copy of forthcoming book.

[Zaremski and Wing, 1993] Amy Moormann Zaremski and Jeannette M. Wing. Signature matching: A key to reuse. *Proc. ACM SIGSOFT Symp. on the Foundations of Software Engineering*, December 1993.

[Zaremski and Wing, 1995a] Amy Moormann Zaremski and Jeannette M. Wing. Signature matching, a tool for using software libraries. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, April 1995.

[Zaremski and Wing, 1995b] Amy Moormann Zaremski and Jeannette M. Wing. Specification matching of software components. *Proc. ACM SIGSOFT Symp. on the Foundations of Software Engineering*, October 1995.

Appendix A

Constraint Satisfaction Algorithms

A.1 Path and K-consistency

Kumar [Kumar, 1992] introduces the issues underlying discussions of the relative use of constraint propagation in reducing or eliminating search for particular domains. A natural question at this point is whether it is conceivable that one might achieve higher levels of consistency in our CSP in some manner such that one can save even more search effort. Node and arc consistency are only two possible degrees of consistency. A further extension is to make a CSP graph *path consistent* [Montanari, 1974, Mackworth, 1977]. In a most general framework, arc and node consistency may be seen as level 1 and level 2 of what is *N-consistency* in general. It has been shown in [Freuder, 1982] and also outlined in [Kumar, 1992] that any n variable CSP can be completely solved by achieving N-consistency in the CSP graph, without search. Of course, achieving this N-consistency can be at least as expensive in general as straightforward search. The cost of achieving N-consistency is high, and is avoided in most applications since it has been found to be less efficient than combining a smaller degree of consistency application with search. An optimal algorithm for computing N-consistency, where N is an integer, is given by Cooper [Cooper, 1989].

A.2 Utility of constraint propagation

In each of the cases of enforcing *node* and *arc consistency* we are propagating some of the constraints through the graph. Essentially we are just creating a simpler CSP from the original CSP. We might say that given an original problem $CSP_{original}$ we can derive first CSP_{nodeC} and then CSP_{arcC}. Intuitively it would now seem that the simpler CSP should be able to be solved using some heuristic search approach in less computational effort than the more complex original CSP problem, while achieving the same solution. The real issue to resolve though, lies in determining what level of advance consistency computation should be done on a CSP in order to minimize the overall effort of combined preprocessing and search. This question has been addressed empirically in a large body of work including [Kondrak and van Beek, 1995], [Nadel, 1989], [Haralick and Elliott, 1980], and [Dechter and Meiri, 1989]. However, most frequently this work is applied to toy problem domains such as the N-queens and Confused-N-queens problems which involve arranging many queens on a chess board so that either they do not challenge each other (N-queens) or they all challenge others (Confused-N-queens). Good descriptions of these problems are given by Nadel in [Nadel, 1989, Nadel, 1990].

A.3 Partial Arc Consistency

In some experimental work [Nadel, 1989, Dechter and Meiri, 1989], it has been observed that a preprocessing phase accomplishing node consistency for a CSP has been cost effective in terms of overall computational savings for finding *all* solutions in some domains, but also that the cost of achieving arc consistency has not been found to be uniformly effective. As a result of this work, efforts have been made in [Nadel, 1989] to see if the actual optimal benefit point for consistency actually falls somewhere between level 1 (node) and level 2 (arc) consistency. The result has been several algorithms which achieve j-consistency (partial arc consistency), where j is between 1 and 2. Results in some experimental domains have shown that these partial arc consistency algorithms combined with search outperform those utilizing only node consistency as a pre-search step [Nadel, 1989].

Several partial arc consistency algorithms are detailed in [Nadel, 1989], with each representing a progressively higher degree of consistency. Their names reflect the approximate amount (as a fraction) of arc consistency checking performed by each. The algorithms discussed and evaluated in [Nadel, 1989] for the Confused-N-queens problem are listed below. These algorithms can be incorporated into backtracking search to form hybrid versions which guarantee differing levels of arc consistency at each search node.

- Several algorithms are well known for achieving full arc consistency including AC-1, AC-2, AC-3, AC-4, and AC-5. Nadel in [Nadel, 1989] describes the three most well known in common terms. These three are now known as "AC-1" [Mackworth, 1977], "AC-2" [Waltz, 1975], and "AC-3" [Mackworth, 1977]. AC-1 differs from the other two in that the others attempt to avoid unnecessary arc checks performed in AC-1. This difference is elaborated in [Nadel, 1989]. AC-3 differs from AC-2 in that the order in which arcs are each in turn checked for consistency is different. Nadel [Nadel, 1989] points out that the issue of avoiding checks is deserving of more research, and observes that it is indeed an analog of the issue of constraint-check ordering we briefly discuss in this thesis in Section 4.3.5.2. Other full arc consistency algorithms are described in other work, particularly AC-4 [Mohr and Henderson, 1986] and AC-5 [Van Hentenryck *et al.*, 1992a]. An optimal k consistency algorithm is presented in [Cooper, 1989] as a generalization of AC-4.

- AC 1/5 arc consistency, apparently arrived at independently in [Nadel, 1989] and [McGregor, 1979].
- AC 1/4 arc consistency, is identical to "Check Forward" as outlined in [Haralick and Elliott, 1980], and as also presented in [McGregor, 1979].
- AC 1/3 arc consistency, essentially "Partial Look Future" as described in [Haralick and Elliott, 1980].
- AC 1/2 arc consistency, essentially "Look Future" as described in [Haralick and Elliott, 1980].

A.4 Intelligent Backtracking

A.4.1 BackJumping

Consider the case where we have three variables for assignment and consequently a search tree of depth 3. Assume that the variables are selected in the order $V1$, $V2$ and $V3$. Also assume that there is a constraint between $V1$ and $V3$, call it $C_{1.3}$. If $V1$ has a domain of $(1, 2, 3)$, and $V2$ has a domain of (A, B, C), and $V3$ has a domain of $(X1, X2, X3)$, then consider the section of the tree shown in Figure A.1 after we instantiate $V1$ to 1. On the "first" instantiation of $V2$ to A, there are no constraints, so we descend to $V3$, and attempt to instantiate $V3$ to its domain values one at a time. If the constraint $C_{1.3}$ fails $C_{1.3}$ for each domain value of $V3$, a normal backtracking method would retreat and attempt a new value for $V2$. Now, clearly this doesn't make any sense, since the conflict causing the backtrack is unrelated to the current instantiation of $V2$. We can infer now that the assignment of 1 to $V1$, is incompatible with any domain value of $V3$. In a sense we have learned something about arc inconsistency in the CSP graph. We can take advantage of this knowledge now by backtracking to the incompatibility, namely $V1$, and assigning a new domain value to $V1$, and bypassing any other attempts to instantiate $V2$. So not only have we saved ourselves the useless work of re-checking redundant constraints, but we have pruned a considerable portion of the search space involving each domain value of $V2$.

This approach will only be beneficial when there is arc inconsistency present in the CSP we are attempting to solve.

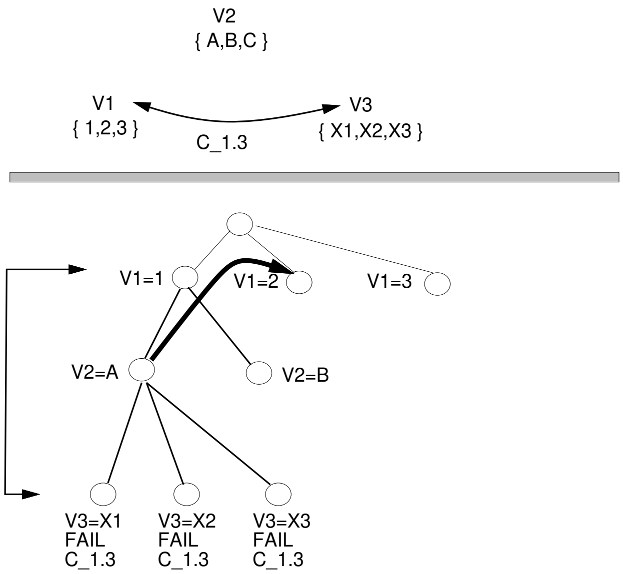


Figure A.1: Example of BackJumping Behaviour

A.4.2 BackMarking

BackMarking is explained in some depth in [Nadel, 1989]. However to put it simply, BackMarking attempts to avoid repeating redundant consistency checks where possible. Whereas BackJumping avoids some cases (and some search space) where a set of consistency checks fail, BackMarking searches the same space as Backtracking, but with fewer consistency checks. The improvement is achieved through a complicated table-save and table-lookup of previously checked constraint values.

A.4.2.1 Sharing AC work in hybrid search

An interesting issue to consider in hybridized search and constraint propagation is whether or not work spent in one part of the search space in locally reducing a particular partially-solved CSP might be shared to other parts of the CSP space, either completely or partially.

One method proposed [Nadel, 1989] for the solution of constraint satisfaction problems (CSPs) is the coupling of traditional breadth-first or depth-first search through the space of all possible variable assignments with some degree of constraint propagation before or during search. Hybrid strategies interleave the work spent on search and the work spent reducing variable domains through some kind of constraint propagation algorithm(s).

Hybrids have been constructed based upon well known search strategies including traditional backtracking, forward checking, backjumping, and backmarking. All of the hybrids utilizing these algorithms have been constructed typically by attempting to achieve some degree of arc consistency during search. For example, forward checking attempts to revise or reduce the domains of remaining variables based upon the most recent instantiated variable value.

It is quite often the case that the work spent in performing constraint propagation is duplicated at various points in the hybrid search space. Essentially identical domain values at disparate points in the search space are removed as a result of identical constraint checks being performed. We present here a brief description of a novel hybrid-based search strategy that can allow for the sharing of constraint propagation work under certain circumstances. In addition, this strategy can be easily encapsulated in a depth-first structure so as to minimize space requirements of search.

This work is derivative of our ongoing research into constraint satisfaction search and hybrid application of search and constraint propagation techniques.

A.4.2.2 Upward Sharing

The general structure of hybrid search may be summarized as selection of a variable, assignment of a domain value for that variable (instantiation), some constraint propagation intended to simplify the remaining problem, and repetition of this sequence until a solution is found. This approach can be nicely formed as a complete depth-first strategy which finds some solution as quickly as possible, and yet which is capable of finding all solutions. Forward checking for instance operates precisely in this manner, where the hybrid aspect appears in that all other variables are revised or have their domains "reviewed" with respect to the latest assignment, and any values no longer satisfying a constraint between a particular variable and the newly instantiated one are discarded.

If we were to take the decision point for the assignment to some variable, call it A, and consider the various subtrees resulting from different domain value selections for A, we can notice that the resulting problems under each subtree are quite similar to the parent (before A) problem. In fact, they differ only in that A no longer has a domain, but rather a single value. Each of the A subtrees differs only in the choice of domain value for A.

Figure A.2 shows a problem with four variables A, B, C and D in which A has been selected for instantiation, and in fact the value assigned to A is 1 in this subtree. If A has a domain of 1, 2 and 3 in the parent, then there will conceivably be three siblings of this nature. In particular, the subgraph consisting of nodes (and domains) B, C, and D plus their associated constraints C3, C4 and C5 will be identical in each of the three subtrees. Further, this subgraph will exist in earlier ancestors of the parent since in our search method we never add variables or domain values as we progressively instantiate

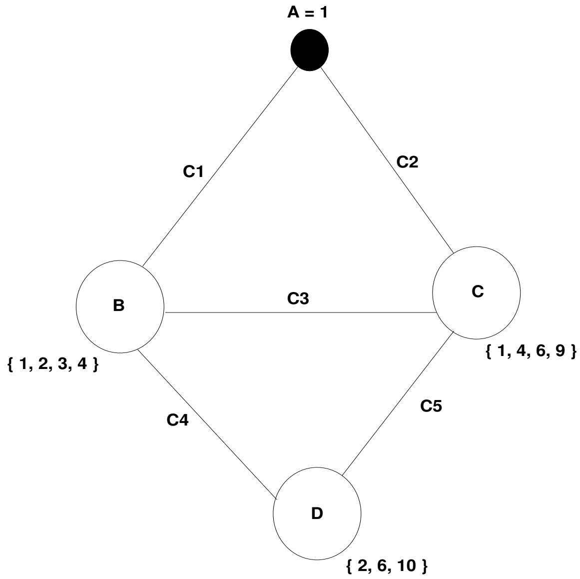


Figure A.2: Partially Instantiated Constraint Graph

variables.

Now, if we were to perform some constraint propagation in which the domain of B is reduced based upon the newly selected value of A, we can say B's domain has become "A-contaminated". This is illustrated in Figure A.3. B's domain has been reduced based upon our new "conjecture" that A's value is 1. Similarly, for any other variable's domain constrained by A, if that domain is reduced based upon an instantiation of A that domain is considered to be "A-contaminated". Now, if D's domain were to be reduced based upon either B or C (each A-contaminated), then D's domain would be said to be A-contaminated also. The basic intuition of contamination in this context is to allow us to be aware of what variable assignment conjectures have been used in our arrival at a particular domain for each variable.

Assume that the order of instantiation in our example is determined prior to any search. For instance, we will say that the order of variable instantiation is always A, B, C and finally D. Therefore, as we progress down our instantiation order and search space,

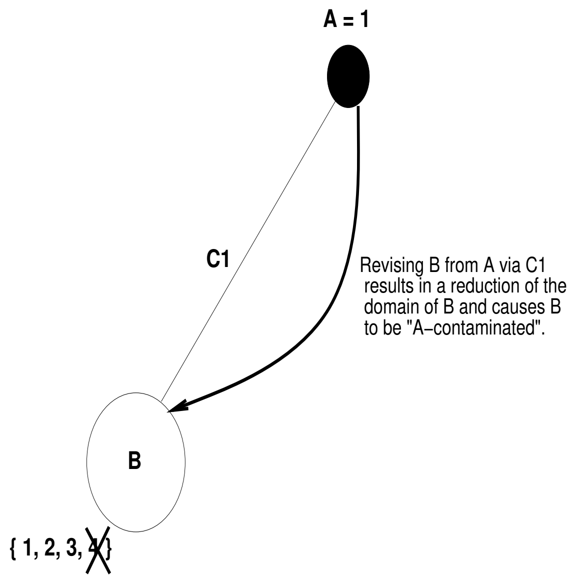


Figure A.3: Contamination trickling through graph

A is instantiated first, and in all subtrees after that instantiation, the assignment of A is a "fact" that can be used in reasoning about other domain values. If we look at the search space as a whole, the variable commitment point spawns child subtrees, where in each subtree the value of that particular fact differs. The parent of these children has no "given" fact for A, only a set of possible domain values.

The concept of contamination allows us to keep track of which "facts" have been used in reducing some possible domain assignments. Since these assignments occur top-down and one at a time, we can see that the point in time where A is assigned always occurs before B's assignment, and further that at any point in the space after B's assignment, A must be assigned to some particular value also. In fact, when an arbitrary variable has just been assigned a particular domain value, all variables earlier in the order have already been assigned, and all variables earlier in the order have a set of possible values.

If, in a particular instance of constraint propagation during search (say after assigning A=1), we reduced the domain of some variable (say D) based upon the current domain

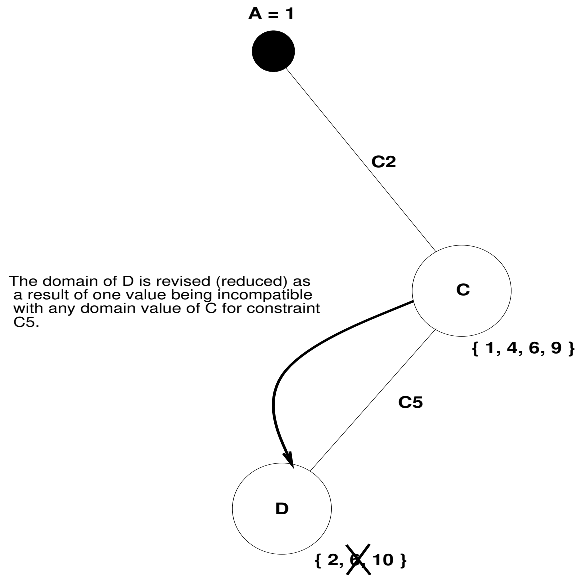


Figure A.4: Constraint propagation between unbound variables

of some other unbound variable (say C) and the constraint set between C and D, then (assuming C is not contaminated) we should be able to "share" this reduction throughout our search space. Figure A.4 details this reduction. The reasoning that resulted in a reduction of D's domain was not dependent on any instantiation explicitly or implicitly, and this reduction could have as easily been done *prior* to the start of any search.

In a backtracking paradigm, sharing this information can be thought of as propagating the domain reduction upwards in the search space. In this particular example, we have just assigned A=1, and the parent is the original (top) problem formulation without assignments. If we propagate the D reduction to each parent up to and including the top problem, any further successor creations through other instantiations will carry the reduction. The D reduction will therefore have been made locally in the ancestor tree of the node where the constraint propagation occurred. It will, however, take global affect as search progresses.

The key to this upward propagation is in the lack of contamination of the domain being propagated. We cannot reduce a domain based upon some variable assignment (directly or indirectly), and propagate this result above the "highest" variable which has taken part in the "contamination". Figure A.5 and A.6 show the upward propagation to the highest level. Figure A.6 shows how this revision of D's domain is later used in a different instantiation of A.

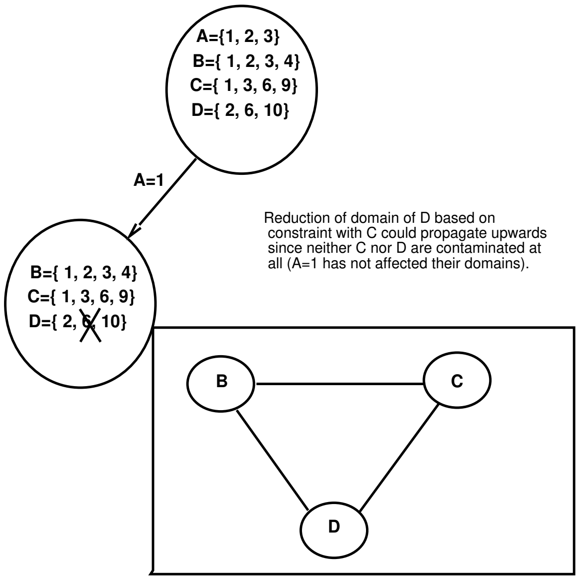


Figure A.5: Problem space before upward propagation

For example consider Figure A.7. If we instantiated A=1 and B=2, then reduce the domain of D based only on C where neither D nor C have been contaminated by either A or B, then we may propagate the new D domain to all parents up to and including the whole problem. All subsequent children resulting from backtracking will include this reduction. If we instantiated A=1, and B=3, then reduced the domain of C based on A's value, then C has become A-contaminated. C's new domain may only propagate up to the level where A was instantiated. All subsequent children of the node where A was instantiated will now reflect the reduced C domain. We must be careful since if we attempt to propagate a contaminated reduction past its point of instantiation, we

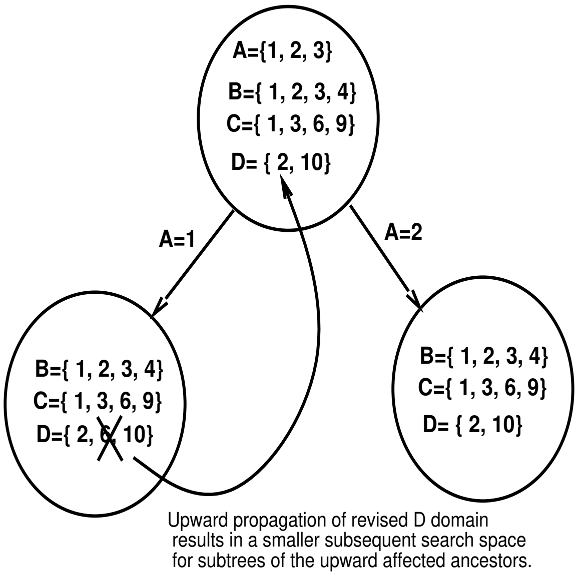


Figure A.6: Problem space after upward constraint

are essentially carrying inferences based upon assignments past where they occur, and thus would force a reduction in a domain based upon an erroneous assumption, and completeness will have been sacrificed.

A.4.2.3 Implications

As a result of propagating these constraints upward in the depth first path where possible, we can reduce the size of the search space in subsequent instantiation paths. However, we should note that the work we are sharing could have been performed as part of more elaborate initial consistency checking prior to the start of search. The advantage of this approach is that we may be uncertain initially as to how much effort to spend on preliminary problem reduction. If some factors appear or are noticed during hybrid search, we can take advantage of these conditions locally via application of further consistency propagation, and achieve global improvement for only the cost of replacing ancestor domains where reduction occurs and where the domain contamination determines upward

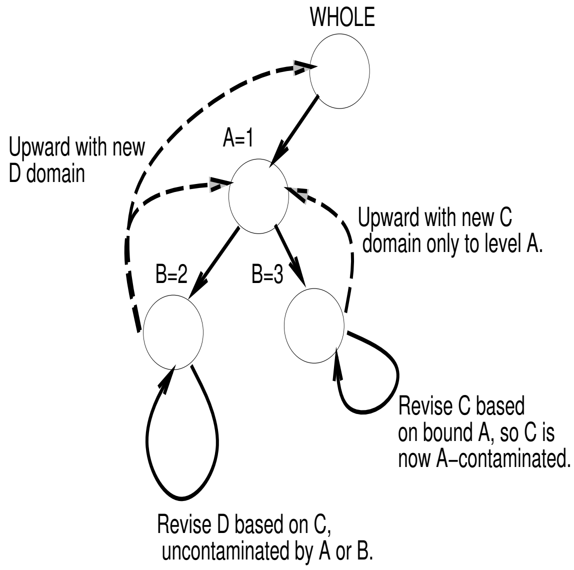


Figure A.7: Problem space after upward constraint

propagation is permissible.

This approach will result in savings only if:

1. A constraint propagation algorithm(s) is utilized which at some point successfully revises some domain based upon constraints among uninstantiated variables. For this point, the tighter the constraints the more likely it is that at least one value will be removed. If no values are removed, there is nothing to propagate.

2. In at least where the above (1) is successful, we are able to propagate the reduction at least one level above the current instantiated variable. For instance, if we have just instantiated B, then in order for a revision to be useful, we must propagate it to at least A so that A's later children (the current node's siblings), reflect this improvement. Of course, the more levels that a removal is propagated, the greater the potential for savings as the scope of the removal is widened.

Note also :

1. This method evolved from thinking about how to share constraint propagation *across* the search space in an A^* [Nilsson, 1980] or breadth-first search.

2. Since upward propagations occur along the ancestor path of the current node, the upward propagation of several domain reductions can combine combinatorially within ancestor nodes to produce larger reduction in the size of the search space expanded following upward propagation. These reductions should form a multiplicative reduction in the search required.

3. Search strategies that do not pre-order the variable instantiations can still benefit from this upward ancestor propagation strategy. Since we are careful to keep track of the "contamination" level of each domain during search, we never propagate an erroneous inference above the level in the space where the "trigger instantiation" was first made.

4. This method seems to relate to propagating hypothetical inferences. If identical or highly similar work has been done, we have been unable to locate it.

5. In Chapter 8 we elaborate a hierarchical AC algorithm based loosely on this conception. In particular, an effort is made to carefully propagate "earlier" constraint applications in an effort to reduce repetitive constraint checks.

A.5 Partitioning and Hierarchical Methods

One drawback with the approach to solving constraint satisfaction problems based upon interleaving search and constraint propagation described in the preceding sections is that a solution or set of solutions is found for a particular CSP in a particular world situation or set of domain values, but those "near" misses for CSP solutions are ignored. Essentially, some particular sets of variable and domain pairs that fails a single constraint, or is unable to match a single variable to a domain value in the world situation does not appear to the user as a candidate solution. In many domains it is easy to see that this hard and fast "yes/no" response to a CSP solution is inadequate. It is quite conceivable that there is a certain degree of uncertainty in the formulation of the CSP itself, to say nothing of possible inaccuracies in the data expressed as domain values. Certainly some of this uncertainty may be addressed in the formulation of constraints themselves, but it seems obvious that some way of specifying the "degree" of match in terms of some predefined "idea" or "desired" match state would be desirable. An ordering or heuristic of this type could serve also as a means of focusing search effort on particularly promising candidates. In addition to uncertainty or error present in either the template specification or in the data, we must also recognize that we are attempting with many CSP problems to model the way in which some agent performs a particular task. The modeling process itself can easily have missed or been unable to capture certain aspects of the reasoning process underlying the solution strategy, and a method of interaction with an expert allowing for dynamic shaping of the search process would be particularly valuable in terms of both improved search performance and in terms of perceived utility of the problem solving system itself. In many problem instances, the amount of time allowed for searching for a potential solution is limited. In many cases, if a solution cannot be found in that time period, then a "best" partial solution or set of "best" candidates that could possibly be solutions, would be much more valuable than no solution at all.

In viewing constraint satisfaction as search, each progression down in the search tree is as a result of the instantiation of a variable with a particular domain value. Further, if we are performing consistency checking as we go, only consistent assignments will be reached at each level. As a result, any given node in the search space represents a partial solution. A total solution is reached only at a leaf node. In fact, if we were to consider stopping this search before ever reaching a leaf node successfully, many partial solutions were seen, some of which were "closer" to a total solution than others. It is possible to keep track of these partial solutions and then utilize them in the event of failure to find a total solution. Essentially what this would "relax" the remaining constraints, suggesting that the resulting partial solution(s) are "good enough". Ideally though, we would like to perform this "relaxation" in a well conceived and systematic fashion so as to obtain partial solutions that fulfill all of what we deem "crucial" constraints, and perhaps relax some of the other less important ones only.

Essentially, we need to shape the constraint satisfaction process so that it can:

- Provide a degree of modeling of uncertainty by ordering or allowing a certain subset of failed total solutions as "partial solutions".
- Provide an ability for interactive ordering or preferring of partial solutions *during* the search for total solutions.
- Allow for "anytime" behaviour, where a demand can be made of the problem solving system for its' best solution or set of solutions at an arbitrary point in time, *before* the search is completed. Ideally, the solutions given would be the "best" possible (in terms of some predetermined defining measure), given the time spent. Further, subsequent time spent would "improve" the quality of the solutions. It seems reasonable that if we are attempting to deal with uncertainty in some measure, that the "best" quality could be "certainty" or "confidence", thus we could achieve greater confidence in a particular solution or set of solutions by spending more time on search.

In subsequent subsections we will outline three different approaches to ordering and preferring potential partial solutions. The first, *partitioning*, will be concerned with attempting to break down CSPs into several component parts, solving each independently and then re-composing the solutions to each component into a solution or set of solutions to the total problem. The second, *abstraction*, will outline how we may arrange a problem hierarchically into progressively more constrained version of the problem, and attempt to search through the space of all simple or constrained solutions in a way in which we can achieve both timely solutions and are guaranteed increasing solution "quality" over time. The third subsection will discuss possible *hybrid approaches* for combining partitioning and abstraction. In addition, we will look specifically at how these approaches relate with an interactive problem solving paradigm, and also what kinds of heuristics will be of particular use to a decomposed problem solving approach.

A.5.1 Partitioning CSP

A.5.1.1 Simple partitions

Some problems seem to "naturally" decompose into parts that may be solved independently first, with the final solution being composed of the completed independent solutions. A simple example of this type of decomposition would be in solving mathematical expressions. Bracketed sections may be solved first in an expansion tree structure, with the results being accumulated upward from the leaves of the tree to the root, with the root being the ultimate solution. Other problems are less "structured" and yet can also be decomposed readily. For instance, the problem of constructing an airplane might be seen as too complex a task. Breaking the task down into manageable pieces might first suggest that we need to "construct a pair of wings", "construct a body", and only then assemble the wings and the body to form an airplane[1].

Similarly if we were attempting to recognize an airplane in an image, we might first consider trying to recognize portions of the plane first that are in some sense more "ele-

[1] Airplane analogy due to Qiang Yang

mental", for instance the wings, and the body. Once we had recognized what we believed were wings and bodies in a picture, we could then attempt to put them together according to the known definition of what a plane looks like in terms of the relationships between wings and the body.

Partitioning implies a divide-and-conquer methodology which relies on the fact that the divided problems are each simpler than the original problem and which are simpler to solve as a set than the original problem. Essentially this type of decomposition and recomposition could only be justified as a sensible approach if the amount of effort spent solving the individual subproblems plus the time spent decomposing and re-composing was less that the effort that would be spent in solving the entire problem all at once.

A.5.1.2 Embedded CSPs using partitions

We commonly look at a CSP as a problem in which we have a set of variables each with a domain set, and with a set of constraints on the assignment of domain values to each variable. Consider now that each "variable" in the CSP could represent a sub-problem, and each domain value for that variable a possible solution. Figure A.8 outlines this "embedded" problem structure.

If we consider that the "outside" or "external" CSP is made up of the "rules" which constrain how the individual "internal" CSP solutions may be combined, we see a convenient representation for CSP decomposition.

A.5.2 Abstraction and CSP

A.5.2.1 What abstraction means in this context

In attempting to apply techniques of abstraction to a constraint satisfaction problem (CSP), we must first understand what abstraction means in the context of constraint

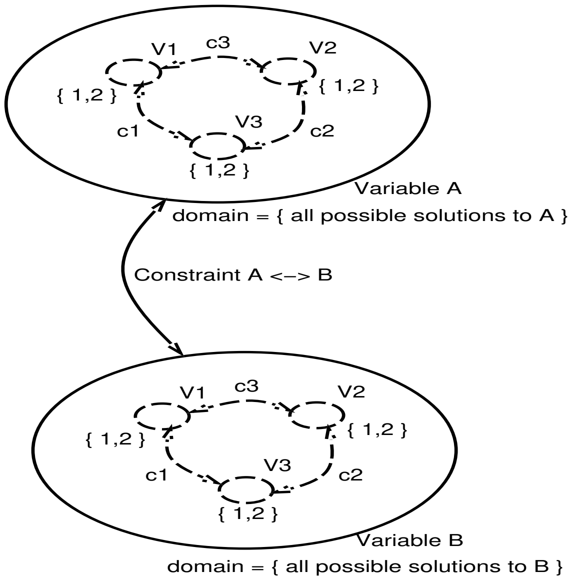


Figure A.8: Embedded Constraint Satisfaction

satisfaction. Some basic questions must be addressed at this point, such as "How can abstraction improve problem solving efficiency ?", "How would abstraction be encapsulated into our problem specification ?", and "How must we modify the way we search for solutions in order to take this new specification into account ?".

`''How can abstraction improve problem solving efficiency ?''`

In order to be of any use, one would expect that abstraction would result in an overall simplification of either the problem itself, and in tighter control of the process of solving the problem. It is apparent that simplification of the problem could arise from encoding more domain information into our representation. Given this additional information, we would wish to modify our problem solving process so that we basically remove from consideration effort-consuming dead ends earlier in our search. The intuitive appeal of this use of abstraction lies in the fact that we are attempting to tailor our search for a problem solution according to specific problem attributes. The effectiveness of abstraction as a strategy is not well understood theoretically, and in the context of this work the utility of abstraction is addressed empirically only. The next obvious question in our specification of abstraction is what domain information can we encode, and how ?

`''How would abstraction be encapsulated into our problem specification ?''`

Typically abstraction is applied to a particular problem by creating what is known as abstraction hierarchies. Hierarchies basically attempt to identify for each aspect of a particular problem how "integral", "key", or "important" that aspect is. These structural problem aspects are organized into hierarchies from most to least "important". In terms of CSPs, problems are structures in terms of N variables, M constraints amongst those N variables, and a domain of some size for each of the N variables. Consequently, the specification of an abstraction hierarchy for a particular CSP would involve the identification of the "important" variables, and constraints. In addition, abstraction is often to taken to mean detailing the difference between "more specific" and "less specific" problem representation. In CSPs, constraints are essentially the guardians of specificity and so a less specific representation of a problem instance could in fact be adjusted to have less constraining constraints. We then have at least 3 ways of formulating the differences among levels of an abstraction hierarchy for, or of relaxing, a particular CSP.

1. Ordering Variables by "importance".

   In ordering variables, the intent will be that "high" level representation of a particular problem will include only those variables of "high" importance. At each successively more specific layer of the abstraction hierarchy, more variables would be progressively added to the representation.

2. Ordering Constraints by "importance".

As for variables, the intent is that a "high" level representation of a particular problem includes only important constraints. Clearly the definition of important constraints depends to some extent on which variables are applicable at each level. In fact there are two classes of constraints which differ fundamentally. Node constraints or reflexive constraints affect only a single variable and as such may be ordered in terms of importance either with respect to a particular variable, or categorized in some other way by level for all variables. Arc constraints affect a set of variables and may be categorized with respect to either that set of variables or to the abstraction level.

3. Relaxing Constraints.

   In addition to constraint ordering, constraints may themselves be relaxed or have their restrictiveness softened. As for constraint ordering, only constraints among variables which exist at a particular have any meaning in the representation.

In some manner we must formulate the problem representation hierarchy utilizing these tools. Work has been done in other domains in terms of attempting to automatically create these hierarchies [Knoblock, 1991b, Knoblock, 1991a] but we assume only the manual construction of these hierarchies. In conjunction with a domain expert we must identify the key features of the problem, and in some number of layers represent what that expert expects will be successive problems whose solution would suggest a progressively increasing degree of confidence in the accuracy of the solution. For instance, if a solution to the "most abstract" representation of a problem is considered to be "quite likely" true, then the solution of the next "least abstract" representation should result in an overall increase in confidence in that solution.

We have now identified one immediate benefit of applying abstraction to CSP. Problems may be represented using hierarchies so as to specify a solution strategy for identifying an approximate-first approach. In such an approach, the lack of a complete least-abstract representation solution will not leave the user without some "best-guess" near-solutions. Further, these near-solutions are arrived at systematically in that the user has implicitly indicating what constitutes progressively improving confidence for a particular problem domain.

`''How must we modify the way we search for solutions in order to take this new specification into account ?''`

Essentially the hard work is done in the representation phase. What we are now presented with is a succession of CSPs which may be solved according to the techniques discussed earlier in this paper. The advantage now lies in that since the hierarchy specifies "increasing specificity", any solution to the most specific representation will also be a solution to any of the more abstract representations. Thus, once we have solved an abstract representation completely, we need only filter the abstract solution set between levels according to the changes we detail in terms of constraints or variables at that level change.

In fact, at each level of abstraction in the abstract solution space we essentially have the problem neatly partitioned. In [Woods, 1993] these partitions are described in terms of representation changes. Since each abstract solution is a candidate to become a next least-abstract level solution, we may apply the constraints between levels to each candidate independently of the others. Failure of the new constraint set application indicates the failure of the candidate to become accepted as a solution at the next level.

Further, other heuristics essentially independent of the abstraction representation may be applied at each successive abstraction level shift. For instance, a heuristic specific to problems with spatial orientation (Spatial Cohesion) is described on page 105 and can be used as a means to simplify extended abstract solutions before applying the next level constraint set.

A.5.2.2 Abstraction as partial solution of CSP

We have until this point discussed the concept of "partial solutions" quite loosely, essentially labeling any consistent partial assignment of domain values to variables a partial solution. To some extent this is true, since the portion of the problem associated with those assigned variables has been solved. However, such a "partial solution" may be as useless as no solution at all. What would be valuable is a predefined portion of the whole problem, whose solution indicates something about the entire solution in itself. [Freuder and Wallace, 1992] introduce the concept of Partial Constraint Satisfaction Problems (PCSP) in order to systematically define the relaxation of constraint satisfaction problems. In essence, there are three ways in which a CSP may be "relaxed" or simplified:

1. The domain of a particular variable can be extended to include more domain values.

2. A variable or constraint may be removed from the original CSP.

3. The domain of a constraint may be enlarged.

In Figure A.9 we consider the original CSP as the common root of a graph representing all possible relaxations of that CSP. Each node in the graph are simpler CSPs obtained by applying a sequence of relaxations. This graph is not a tree since application of relaxation steps in different orders can give the same PCSP as a result. Since there is a very large space of possible applications of the three relaxation steps at any stage of creating a PCSP space, a domain dependent method is assumed to detail appropriate relaxations, and indicating bounds on how much relaxation is acceptable for this particular problem. If we now were to consider the problem as searching through this problem space attempting to

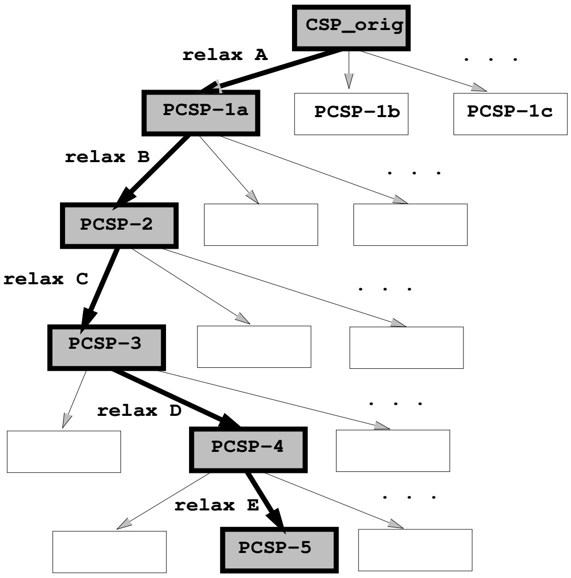


Figure A.9: One abstraction hierarchy in a PCSP space

find one PCSP that could be solved, a great deal of work must be expended in determining which ones could not be solved. An appropriate method might interleave search through the PCSP space, thus not over-committing to either too strict or too relaxed versions of the CSP.

In a very real sense in building the PCSP space, we have constructed an abstraction space in which the least constrained PCSPs are the most abstract, and the most constrained PCSPs are the most specific instances of this CSP. However, there are many possible versions of the least constrained PCSPs. How do we differentiate amongst these? We would like to to arrange these PCSPs into a more systematic hierarchy where some domain knowledge can be used to structure the hierarchy.

Consider that we could extract the “key” or “critical” information from a CSP, and form a “most abstract” PCSP (call it PCSP-3) which encapsulates this knowledge. Next, we extract several common distinguishing features or identifiers which reflects how the set of partial solutions is limited. Each of these sets of distinguishing features is applied to PCSP-3 as an inverse instance of the previous three relaxation techniques. Specifically, we can now add a variable, restrict a variable domain, or restrict a constraint domain. We now construct the more constrained PCSP-2, then PCSP-1, and finally our complete problem, CSP. Figure A.9 on page 345 outlines this process. The number of stages or the groupings of the sets of "making stricter" actions can be dependent on a particular problem, and on the typical features or keys of that domain. The advantage of this systematic structuring of the hierarchy of PCSPs is that a constraint satisfaction approach which solved PCSP-3 first resulting in a set S1 of solutions, then attempted to solve PCSP-2 for S2 given the initial starting point as each of the solutions in S1, and so on, would essentially have the quality of iteratively refining rough or partial solutions into specific or total ones. The effort spent arriving at the set S1 is not spent again in solving for set S2, as we build on the work done before.

The process described above works very much in a top-down manner, solving PCSP-3 giving set S3, then refining S3 in solving PCSP-2 giving S2, finally refining S2 in solving CSP, giving our final solution set. One obvious difficulty with such an approach is that it finds all of the PCSP-3 solutions in S3 before attempting to find any solution elements for S2, for example. Basically each level is completely solved before the subsequent level in a breadth-first manner. This approach has the advantage that if search were halted at an arbitrary point in time, all of the "best-so-far" answers are on the same level. It has the disadvantage that time spent searching the less constrained CSPs might have resulted in more concrete solutions. In [Woods, 1991], a search strategy known as 'Left-Wedge" is described for exploring the multiple level of abstraction solution space which progresses downwards towards more concrete solutions, *and* from left to right through the solutions at each level at the same time. This strategy is a variation on A^* [Nilsson, 1980] search which (with the expenditure of additional search effort) can provide us with both concrete solutions and abstract solutions.

To some extent, the selection of an approach for this type of search depends highly on how we would wish to take advantage of interaction with the user during search. In a certain case a user may wish to view and prioritize or eliminate higher level solutions in order to guide the discovery of concrete solutions. It is also possible that a user may wish to interleave the various levels of search and interaction.

A.5.3 Combining partitions and abstraction

We have outlined how CSPs could be decomposed through either partitioning or abstraction approaches. An interesting point for further investigation is the issue of interleaving these concepts in decomposing CSPs. For instance, a complex problem abstracted into a hierarchy of several layers could have further decomposition within each level, in the form of a partition, or even further abstraction inside that level. This flexibility allows a very wide range of definition of a particular CSP, and suggests many possible options for the designer to consider in deciding how to decompose a particular problem.

This type of problem structuring has fairly broad implication to the manner in which search for solutions could be performed. A global strategy controlling where effort is spent across hierarchies and between partitions would be required. In addition, the very nature of partitioning suggests that multiple processor or distributed solutions may offer many more possible extensions to the control strategy.

A.5.4 Decomposition and user interaction

As we have stated earlier, one advantage of partitioning or abstracting a large CSP problem is that it provides a useful model for solving the CSP in which the user may interactively guide the search. This guidance can occur in several forms for both partitioning and abstraction. We consider abstraction as an interaction strategy in detail in [Woods, 1993], and only touch on the important issues in this short summation.

During the search of the PCSP space detailed in the abstraction section, high level solutions are found for abstracted CSPs, and these candidates are then refined if possible to lower level solutions. This process is repeated until a solution set is found at the lowest or base level of abstraction. Justifications for this approach include:

1. Elimination of high level solutions from the set of solutions will simplify the search, and can also account for the ability of the user to understand the domain better than has been captured effectively in any pure search pruning heuristic. Basically, the user is allowed to view and remove candidates before they are passed between levels. This process can be modified to suit the user and the application in terms of the degree of interactiveness required or desired.

2. Ordering of high level solutions by "likelihood" or some other user based preference heuristic that may not be easily captured in an automatic heuristic. If the user can see and evaluate the possible high level solutions based on his own experience or preference, the solutions that are eventually found at lower levels may suit better the user requirements or needs.

An interesting side-effect of this user interaction is that observation or analysis of the selections or ordering of the user may assist in the creation of more automatic methods of filtering that can be employed in future versions of the search process. These methods may become apparent to the user through his mechanized selection process, or perhaps to other independent analysts observing the filtering process as employed by the user.

Looking at CSP search as several disjoint CSP searches for separate partitions, and then re-composing these partitions into complete solutions allows for a similar level of user interaction in the process. Justifications for this approach include:

1. Elimination of elements from *any* of the partitions will reduce the number of re-composition attempts that need to be attempted without computational effort. De-

pending on the problem domain, the identification of some or even one part might be enough to dramatically reduce the possible explanations of that deployment. In this way partial solutions may be very valuable.

2. While many real world problems may often be solved using a form of reasoning similar to abstraction, others may have no easily identifiable abstract quality, while possessing several sub-parts of equal importance.

Appendix B

Mechanism Matching

There is a broad range of research in this area. For example, [Kazman *et al.*, 1996] describes how architectural components may be classified - particularly in terms of component interactions. Component instantiations include objects, programs, processes, tasks; interactions include RPCs, shared memory, sockets, pipes or rendezvous. In this work specific distinctions are made among different types of conceptual matching. In particular, *mechanism matching* may be defined as instantiating components/interactions into language parts that facilitate a particular type and communication, *signature matching* as agreements on the form of data flowing between matched elements (types, structures, parameters), and *semantic matching* which assures that computations together satisfy the behavioural and resource utilization system requirements. This division of functionality is closely analogous to a breakdown of structural constraints in our PU-CSP approach, except at a varying degree of abstraction and with emphasis on a variety of programming devices where we have focused more on static (COBOL-like) interactions as opposed to RPCs and the like. In essence, the structural analysis work has been considerably expanded in this work and could be adapted in our model as additional constraint information. In fact, the primary purpose of the work reported in [Kazman *et al.*, 1996] is to create a system-building paradigm independent of language which enhances re-usability of components and allows system builders to think in terms of application rather than programming language. The architectural features described in the paper are meant to add a semantic element to the previous purely syntactic idiom descriptions in the literature. In Section 10.3 we briefly discuss how some of the work presented in this thesis could be formulated with this as a methodology for mechanism matching.

Appendix C

Details of Hierarchical CSP Algorithms

For these algorithms, a set of functionality is assumed as follows.

- Linkage accessor functions are available as follows: *Link.srcDom* returns the source domain value of the link, *Link.srcVar* returns the source variable of the link, *Link.targetlist* returns the set of justifying target values in the link.
- The expression $A+=B$ is shorthand for $A = append(A, B)$.
- The function $resetMarking(x, fn)$ resets the marking to nil of the hierarchy rooted at x for function fn.
- The function $GetNextNotDeletedSibling(x, source)$ returns the next undeleted sibling of x for variable source in order to attempt to re-establish a linkage in stepped revision. The next function relies on an arbitrary but predictable ordering for domain values.

C.1 Algorithm DeleteSourcePropagateAggressive

The $DSPA$ algorithm propagates a deletion of a target domain value through any justification links which utilize this value. If the target value T belongs to a justification link L for a source value S, S is deleted in the case that no other target justifiers exist. The existence of a link in the aggressive case indicates that all siblings of T have already been checked against S, and the search for justifiers relies solely on the linkage structure. If the deletion causes a linkage for S to contain only one target $T2$, then the relation is re-asserted with the hope of reducing the hierarchy of the source based on the single remaining target.

Algorithm $DSPA$(Val:x_i,Var:$Source$,int:$MaxD$,relation:γ);

```
DeleteDomainValue(x_i); AffectList := (x_i);
Tlinks := Justification Links which for which x_i is target;

ForAll Link in Tlinks do
  newTlist := Link with x_i removed;
  if (not null newTlist)
  then
    if Length(newTlist) = 1
    then
      reAssert :=
        ApplyR(Link.srcDom, newTlist.first, γ, initial, MaxD, 0);
      if FAIL_3(reAssert)
      then
        AffectList + =
          DSPA(Link.srcDom, Link.srcVar, MaxD, γ);
      else
        if Simplify(Link.srcDom, 1, γ)
        then
          AffectList + =
            KSPA(Link.srcDom, Link.srcVar, MaxD, γ);
    else
      Link.targetList := newTlist;
  else
    Affectlist + =
      DSPA(Link.srcDom, Link.srcVar, MaxD, γ);
End ForAll;

Return AffectList;
```

Table C.1: The $DeleteSourcePropagateAggr$ propagation algorithm

C.2 Algorithm KeepSourcePropagateAggressive

The $KSPA$ algorithm propagates a revision of a target domain value through justification links. In particular, any linkage which involves this target needs to be re-checked for the target's membership as a justifier. If the target is no longer a justifier due to the hierarchical change, it must be removed from the link. This removal may trigger subsequent calls to $DSPA$. As is the case for $DSPA$, in the event that a justification link declines to a single justifier, the source hierarchy may be simplified.

Algorithm $KSPA$(Val:x_i,Var:$Source$,int:$MaxD$,relation:γ);

```
AffectList := (x_i);
Tlinks := Justification Links which for which x_i is target;

ForAll Link in Tlinks do
  newTlist := Link with x_i removed;
  if (null newTlist)
  then resetMarking(x_i, γ);

  reAssert := ApplyR(Link.srcDom, x_i, γ, initial, MaxD, 0);
  if FAIL_3(reAssert)
    then
  if (null newTlist)
  then
    AffectList+ = DSPA(Link.srcDom, Link.srcVar, MaxD, γ);
  else
    if (Length(newTlist) = 1)
    then
      resetMarking(x_i, γ);
      lastAssert := ApplyR(Link.srcDom, newTlist.first, γ, initial, MaxD, 0);
      if FAIL_3(lastassert)
      then
        if Simplify(Link.srcDom, 1, γ)
        then AffectList+ = KSPA(Link.srcDom, Link.srcVar, MaxD, γ);
        else AffectList+ = DSPA(Link.srcDom, Link.srcVar, MaxD, γ);
      else
        if ((null newTlist) and Simplify(Link.srcDom, 1, γ))
        then
          AffectList+ = KSPA(Link.srcDom, Link.srcVar, MaxD, γ);
End ForAll;

Return AffectList;
```

Table C.2: The $KeepSourcePropagateAggr$ propagation algorithm

C.3 Algorithm DeleteSourcePropagateStepped

The *DSPS* algorithm propagates a deletion of a target domain value through any justification links which utilize this value. In this case, the linkage structure is not known to be complete, and consequently, this deletion triggers an attempt to find a sibling of the deleted target justifier to keep justifying the particular source value. If no such justifier is found, the deletion propagates to the source recursively. If a found justifier is the last sibling, then the source hierarchy may be simplified. The simplification of a justifier implies the need to re-establish any affected link.

Algorithm $DSPS$(Val:x_i,Var:$Source$,int:$MaxD$,relation:γ);

```
DeleteDomainValue(x_i); AffectList := (x_i);
Tlinks := Justification Links which for which x_i is target;

ForAll Link in Tlinks do
  x_iSibling := GetNextNotDeletedSibling(x_i, Source);
  if (null x_iSibling)
  then
    AffectList+ = DSPS(Link.srcDom, Link.srcVar, MaxD, γ);
  else
    foundReEst := FALSE;
    Loop until ((null X_iSibling) or foundReEst)
      reAssert := ApplyR(Link.srcDom, x_iSibling, γ, initial, MaxD, 0);
      if (not FAIL_3(reAssert))
      then foundReEst := TRUE;

      if (not foundReEst)
      then
        x_iSibling := GetNextNotDeletedSibling(x_iSibling, Source);
        lastSibling := TRUE if found last sibling;
    End Loop;

    if foundReEst
    then
      if (lastSibling and Simplify(Link.srcDom, 1, γ))
      then AffectList+ = KSPS(Link.srcDom, Link.srcVar, MaxD, γ);
    else AffectList+ = DSPS(Link.srcDom, Link.srcVar, MaxD, γ);
End ForAll;

Return AffectList;
```

Table C.3: The *DeleteSourcePropagateStep* propagation algorithm

C.4 Algorithm KeepSourcePropagateStepped

The $KSPA$ algorithm propagates a revision of a target domain value through justification links. In particular, any linkage which involves this target needs to be re-checked for the target's membership as a justifier. If the target is no longer a justifier due to the hierarchical change, it must be removed from the link. This removal results in the need to find a sibling replacement justifier, and failure to do this will trigger subsequent calls to $DSPS$. As is the case for $DSPS$, in the event that a justification link declines to a single justifier, the source hierarchy may be simplified .

Algorithm $KSPS$(Val:x_i,Var:$Source$,int:$MaxD$,relation:γ);

```
AffectList := (x_i);
Tlinks := Justification Links which for which x_i is target;
ForAll Link in Tlinks do
  x_iSibling := GetNextNotDeletedSibling(x_i, Source);
  if (null X_iSibling)
  then resetMarking(x_i, γ);

  reAssert := ApplyR(Link.srcDom, x_i, γ, initial, MaxD, 0);
  if FAIL_3(reAssert)
    then
      if (null x_iSibling)
      then
        if Empty(Link.targetlist)
        then AffectList+ = DSPS(Link.srcDom, Link.srcVar, MaxD, γ);
      else
        foundReEst := FALSE; lastSibling := FALSE;
        Loop until ((null X_iSibling) or foundReEst)
        reAssert := ApplyR(Link.srcDom, x_iSibling, γ, initial, MaxD, 0);
        if (not foundReEst)
        then
          x_iSibling := GetNextNotDeletedSibling(x_iSibling, Source);
          lastSibling := TRUE if found last sibling;
        End Loop;
        if foundReEst
        then
          if (lastSibling and Simplify(Link.srcDom, 1, γ))
          then AffectList+ = KSPS(Link.srcDom, Link.srcVar, MaxD, γ);
          else AffectList+ = DSPS(Link.srcDom, Link.srcVar, MaxD, γ);
    else
      if ((null X_iSibling) and Simplify(Link.srcDom, 1, γ))
      then AffectList+ = KSPS(Link.srcDom, Link.srcVar, MaxD, γ);
End ForAll;
Return AffectList;
```

Table C.4: The $KeepSourcePropagateStep$ propagation algorithm