

Fall 1999

A constraint-based framework for configuration

Daniel Sabin

University of New Hampshire, Durham

Follow this and additional works at: <https://scholars.unh.edu/dissertation>

Recommended Citation

Sabin, Daniel, "A constraint-based framework for configuration" (1999). *Doctoral Dissertations*. 2100.
<https://scholars.unh.edu/dissertation/2100>

This Dissertation is brought to you for free and open access by the Student Scholarship at University of New Hampshire Scholars' Repository. It has been accepted for inclusion in Doctoral Dissertations by an authorized administrator of University of New Hampshire Scholars' Repository. For more information, please contact nicole.hentz@unh.edu.

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI[®]

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

A CONSTRAINT-BASED FRAMEWORK FOR CONFIGURATION

BY

DANIEL SABIN

M.S. in Computer Science, Bucharest Polytechnic Institute, ROMANIA, 1984

DISSERTATION

Submitted to the University of New Hampshire
in Partial Fulfillment of
the Requirements for the Degree of

Doctor of Philosophy

in

Computer Science

September 1999

UMI Number: 9944003

UMI Microform 9944003

Copyright 1999, by UMI Company. All rights reserved.

**This microform edition is protected against unauthorized
copying under Title 17, United States Code.**

UMI

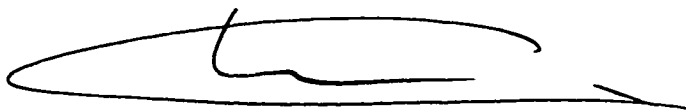
**300 North Zeeb Road
Ann Arbor, MI 48103**

This dissertation has been examined and approved.


Dissertation Director, Eugene C. Freuder
Professor of Computer Science



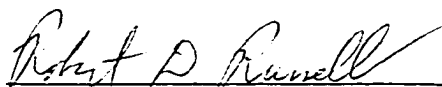
Christian Bessière
Laboratoire d'Informatique, de Robotique et de Mi-
croélectronique de Montpellier
Centre National de la Recherche Scientifique, France



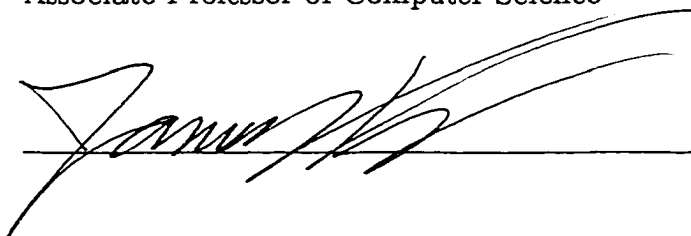
Philip J. Hatcher
Professor of Computer Science



Robert D. Russell
Associate Professor of Computer Science



James L. Weiner
Associate Professor of Computer Science



July 20th, 1999

Date

DEDICATION

To Andreea, Daniel Jr. and Mihaela

TABLE OF CONTENTS

DEDICATION	iii
ABSTRACT	xi
1 INTRODUCTION	1
1.1 Sample Configuration Problem	2
1.2 Thesis Contribution	6
1.3 Thesis Overview	8
2 CONFIGURATION BACKGROUND	10
2.1 Comparison with Other Reasoning Tasks	12
2.2 Problem Description	13
2.2.1 Specification	14
2.3 Current Solutions	16
2.3.1 Rule-Based Systems	16
2.3.2 Model-Based Reasoning	18
2.4 Chapter Conclusions	26
3 BRIEF CONSTRAINT SATISFACTION BACKGROUND	29
3.1 Definitions and Notations	30
3.2 CSP Solving techniques	32
3.2.1 Backtrack Search	32
3.2.2 Consistency Inference	34

3.2.3	Arc Consistency	34
3.2.4	Higher Order Consistency	37
3.3	Advanced Search Methods	39
3.3.1	Consistency Inference as Preprocessing	39
3.3.2	Prospective Search Algorithms	40
3.3.3	Ordering Heuristics	43
4	COMPOSITE MODEL FOR CONFIGURATION	45
4.1	Aggregation and Context Independence	46
4.2	Abstraction	47
4.3	Anatomy of a Component Type	48
4.4	Chapter Conclusions	53
5	CONFIGURATION AS COMPOSITE CONSTRAINT SATISFACTION	54
5.1	The Composite Constraint Satisfaction Problem	54
5.1.1	Composite Values	55
5.1.2	Port Variables	56
5.1.3	Related Work	57
5.2	Component Type Models as Composite CSPs	61
5.2.1	Capturing Structural Information	64
5.2.2	Capturing Hierarchical Information	64
5.2.3	Capturing Relationships	65
5.3	Chapter Conclusions	67
6	TAKING ADVANTAGE OF PROBLEM STRUCTURE	68
6.1	Introduction	68

6.2	Example	70
6.3	Related Work	77
6.4	Algorithm	79
6.5	Experimental Evaluation	84
6.6	Chapter Conclusions	96
7	OPTIMIZATION METHODS FOR CONSTRAINT RESOURCE PROBLEMS	97
	LEMS	97
7.1	Introduction	97
7.2	Problem Definition	98
7.2.1	Example 1	99
7.2.2	Example 2	101
7.2.3	Problem representation	103
7.3	Algorithms	104
7.3.1	Port variables instantiation	105
7.4	Achieving Optimality through Constraint Propagation	107
7.4.1	Improved lower bound computation	108
7.5	Equivalent Partial Solutions	109
7.5.1	Eliminating equivalent partial solutions through interchangeability	110
7.5.2	Abstraction and context-dependent interchangeability	111
7.6	Experimental Evaluation	112
7.7	Chapter Conclusions	116
8	CONCLUSION	118

List of Tables

List of Figures

1-1	Generic server system architecture	3
2-1	Reasoning tasks hierarchy	12
3-1	Sample CSP problem	32
3-2	Typical search tree	33
3-3	Enforcing arc consistency (example 1)	36
3-4	Enforcing arc consistency (example 2)	37
3-5	Search tree produced by Forward Checking	41
3-6	Search tree produced by Maintaining Arc Consistency	43
5-1	Composite CSP representation for SERVER	65
5-2	Composite CSP representation for SUPERSERVER	66
6-1	Sample constraint network	71
6-2	The constraint network after instantiating variable U	72
6-3	The constraint network after instantiating variable U	73
6-4	The constraint network after finding a solution	74
6-5	The constraint network after eliminating the cycle-free variables	77
6-6	Cycle-cutsets for the example in Figure 6-1 (first heuristic)	82
6-7	Cycle-cutsets for the example in Figure 6-1 (second heuristic)	83
6-8	Cycle-cutsets for the example in Figure 6-1 (third heuristic)	84
6-9	Comparison between the cycle-cutset method and MACE	86

6-10 Comparison between the cycle-cutset method with arc-consistency preprocessing and MACE	87
6-11 Performance ratio between the cycle-cutset and MACE	88
6-12 Comparison MAC-7ps to MACE on problems with 20 variables, density=0.05	89
6-13 Comparison MAC-7ps to MACE on problems with 20 variables, density=0.15	89
6-14 Comparison MAC-7ps to MACE on problems with 20 variables, density=0.25	90
6-15 Comparison MAC-7ps to MACE on problems with 20 variables, density=0.35	90
6-16 Comparison MAC-7ps to MACE on problems with 20 variables, density=0.45	91
6-17 Comparison MAC-7ps to MACE on problems with 20 variables, density=0.55	91
6-18 Comparison MAC-7ps to MACE on problems with 20 variables, density=0.65	92
6-19 Comparison MAC-7ps to MACE on problems with 20 variables, density=0.75	92
6-20 Comparison MAC-7ps to MACE on problems with 20 variables, density=0.85	93
6-21 Comparison MAC-7ps to MACE on problems with 20 variables, density=0.95	93
6-22 Performance ratio between MAC-7ps and MACE on problems with 20 variables	94
6-23 Comparison between MAC-7ps and MACE on problems with 40 variables .	95
6-24 Performance ratio between MAC-7ps and MACE on problems with 40 variables	95
7-1 Snapshot of the search tree for an optimal solution (example 1).	100
7-2 Snapshot of the search tree for an optimal solution (example 2).	102
7-3 Snapshot of the search tree for an optimal solution (example 1 – after adding the redundant constraint).	110
7-4 Snapshot of the search tree for an optimal solution (example 2 – after adding the redundant constraint and using context-dependent interchangeability. .	113
7-5 Comparison with the original Solver code.	114
7-6 Search effort in terms of number of failures	114

7-7	Search effort in terms of CPU time.	115
7-8	The effect of each optimization method on algorithm performance.	116

ABSTRACT

A CONSTRAINT-BASED FRAMEWORK FOR CONFIGURATION

by

DANIEL SABIN

University of New Hampshire, September, 1999

The research presented here aims at providing a comprehensive framework for solving configuration problems, based on the Constraint Satisfaction paradigm. This thesis is addressing the two main issues raised by a configuration task: modeling the problem and solving it efficiently. Our approach subsumes previous approaches, incorporating both simplification and further extension, offering increased representational power and efficiency.

Modeling

We advance the idea of local, context independent models for the types of objects in the application domain, and show how the model of an artifact can be built as a composition of local models of the constituent parts. Our modeling technique integrates two mechanisms for dealing with complexity, namely composition and abstraction. Using concepts such as locality, aggregation and inheritance, it offers support and guidance as to the appropriate content and organization of the domain knowledge, thus making knowledge specification and representation less error prone, and knowledge maintenance much easier.

There are two specific aspects which make modeling configuration problems challenging: the complexity and heterogeneity of relations that must be expressed, manipulated and

maintained, and the dynamic nature of the configuration process. We address these issues by introducing Composite Constraint Satisfaction Problems, a new, nonstandard class of problems which extends the classic Constraint Satisfaction paradigm.

Efficiency

For the purpose of the work presented here, we are only interested in providing a guaranteed optimal solution to a configuration problem. To achieve this goal, our research focused on two complementary directions.

The first one led to a powerful search algorithm called Maintaining Arc Consistency Extended (MACE). By maintaining arc consistency and taking advantage of the problem structure, MACE turned out to be one of the best general purpose CSP search algorithms to date.

The second research direction aimed at reducing the search effort involved in proving the optimality of the proposed solution by making use of information which is specific to individual configuration problems. By adding redundant specialized constraints, the algorithm improves dramatically the lower bound computation. Using abstraction through focusing only on relevant features allows the algorithm to take advantage of context-dependent interchangeability between component instances and discard equivalent solutions, involving the same cost as solutions that have already been explored.

CHAPTER 1

INTRODUCTION

It is no longer a question of whether AI technologies will have an impact on manufacturing, but one of better understanding and exploiting the broad potential of AI in this domain. New manufacturing concepts and philosophies [...] place increasing emphasis on the need for more intelligent manufacturing systems, and there is general consensus that AI technologies will play a key role in the manufacturing enterprise of the future.

Call for Papers for the
1996 Artificial Intelligence and Manufacturing Research Planning Workshop

Configuration systems have a long history in AI, of almost two decades, starting with the landmark R1/XCON system (McDermott 1982), used in the configuration of computer systems at Digital Equipment Corporation. Since then, many configuration systems have been built for configuring computers, communication networks, cars, trucks, operating systems, buildings, circuit boards, keyboards, printing presses, insurance policies, *etc.* This work has lead to techniques for representing and solving configuration problems.

Based on the information we have from both industry and research groups, current configurators either fail to address some of the issues raised by configuration problems, or their performance in doing so is inadequate. We propose to develop a constraint-based configuration framework which overcomes these limitations. Our goal is to provide the basis for a faster and more flexible configuration system.

1.1 Sample Configuration Problem

We present in this subsection part of a hypothetical configuration problem. Our goal is to introduce the reader to the issues raised by configuration tasks and present the type of problems the work presented in this thesis focuses on. The problem is neither complete nor real. It is inspired from a set of real specifications, but due to the proprietary nature of the information, the actual architecture and values have been slightly modified. However, the problem preserves the structure and main characteristics specific to real configuration problems.

The task is to configure customized server systems. All server systems share the same generic *architecture*. Two other pieces of knowledge complete the specification of a server system: the set of *functional features* and *properties*, which identify the system and its constituent parts, and the set of *relations* among components and/or their features and properties. The system architecture is presented in Figure 1-1

Structurally, a SERVER SYSTEM consists of a set of racks, in which we plug servers and disk arrays, a system console, and optionally, a server console and a laser printer. The type of a particular server system is specified by the value of the *type* property, which can take the value *Mini* or *Super*. The values for the number of racks, *rack-count*, and the number of servers, *server-count*, are determined functionally. The type of the system defines for both properties a minimum value required and a maximum value allowed. For a *Mini* system, $rack-count = 1$ and $1 \leq server-count \leq 3$. For a *Super* system, $1 \leq rack-count \leq 3$, $2 \leq server-count \leq 4$. Choices for the *country* property are *US*, *UK*, *France*, *Europe* and *WorldWide*. Country selections restrict certain other product selections, such as power supply, system and server consoles.

Each RACK contains 9 drawers, numbered from bottom to top starting at 1. According

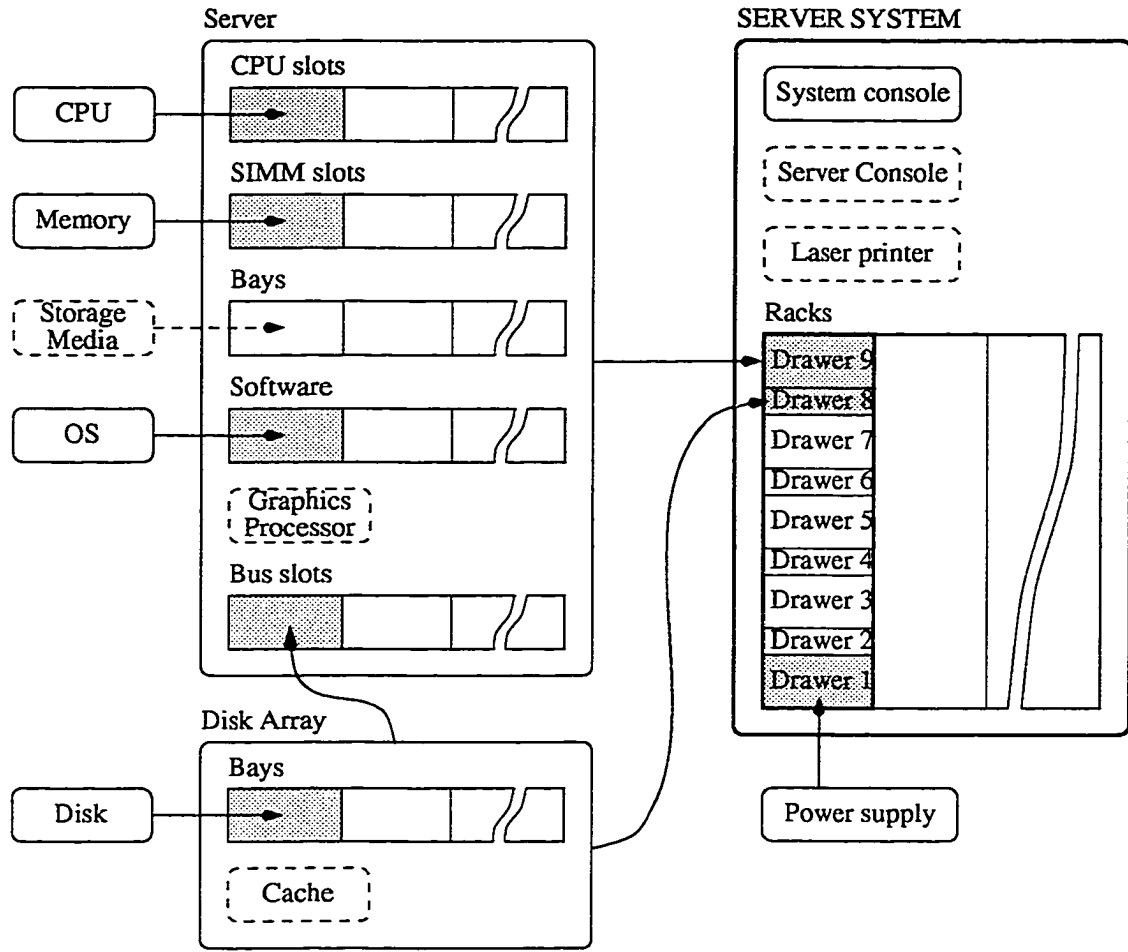


Figure 1-1: Generic server system architecture

to their *height*, there are two types of rack available: *tall* and *short*. An *uninterrupted power supply* takes always the bottom drawer. Either servers or disk arrays can be plugged in any of the other drawers. A *Mini* server requires one tall drawer, while a *Super* server requires two adjacent drawers, one tall and one short. First server takes drawer(s) 7(8). Additional servers take drawers 5(6), 3(4), 9, in this order. Disk arrays take drawers 2, 4, 6, 8, in that order. In a *Mini* system with more than 4 disk arrays, move the server in drawer 5 (if any) to drawer 7 from the next rack. In a *Super* server with more than 2 disk arrays, move the server in drawers 5(6) (if any) to drawers 7(8) from the next rack.

A SERVER provides several sets of *resources*: *CPU slots*, *SIMM slots*, *bus slots*, and *bays*. The *type* of a server is required to be the same as the type of the system. Each type requires minimum values for the number of CPUs, *CPU-count*, and total amount of internal memory, *memory-size*. At the same time, the type dictates the lower and upper bounds on the number of bays and different slot categories: *bay-count*, *CPU-count*, *SIMM-count* and *bus-count*. Server resources are used both by constituent parts and by other components in the system to which the server connects. Each CPU consumes one CPU slot, and each MEMORY board requires one SIMM slot.

Property *type* of type DISK-ARRAY identifies one of the two types available, *DA-1300* and *DA-2900*. The maximum number of servers supported, *server-count*, is 4 for both models, and the required minimum number of servers connected is 1, independent of the boolean value of *daisy-chained*. Additional features are the number of bays, *bay-count*, the amount of disk space, *disk-size*, and the amount of cache memory, *cache-size* provided, as well as *SCSI-type*, which can be either *mono-SCSI* or *dual-SCSI*. The number of DISK ARRAY units a server can connect to is limited by the number of bus slots it has available. A *mono SCSI* disk array consumes one bus slot, while a *dual SCSI* disk array requires two bus slots. Disk arrays also consume resources provided by racks. A daisy chained disk array consumes two drawers in a rack, and a not daisy chained one consumes one drawer. The type of the disk array restricts the amount of cache memory and disk size, as well as the number of bays. For model *DA-1300*, $3 \leq \text{bay-count} \leq 10$, $\text{disk-size} \geq 12.6 \text{ Gb}$, $\text{cache-size} \in \{0, 32\text{Mb}\}$. For a *DA-2900*, $5 \leq \text{bay-count} \leq 15$, $\text{disk-size} \geq 21 \text{ Gb}$, and $\text{cache-size} \in \{0, 32\text{Mb}, 64\text{Mb}\}$.

The types of MEMORY boards available are *64Mb*, *128Mb*, *256Mb*, *512Mb* and *1024Mb*. Each board consumes one SIMM slot. There are two DISK models, one providing 4.2 Gbyte

and the other one 9.1 Gbyte. Each of them consumes 1 bay.

In addition to the generic architecture of the artifact and a detailed description of the components available, the specifications for a configuration problem contain also several optimization criteria that will guide the search for a particular solution. The optimization criterion can be expressed locally, at the level of a component or group of components, as well as globally, at the level of the entire artifact. Given the total amount of internal memory for a server, the number and type of memory boards selected must provide the memory required while minimizing the number of memory slots used. Similarly, in determining the individual disk drives required to provide the total disk size selected, the number of disks will be minimized. Finally, the total price of the system, calculated as the sum of the price for each of its constituent parts, has to be minimal.

The configuration of a particular instance of server system is based on a set of needs and preferences specified by the user. Some of the input categories and possible choices are the following:

- Total number of users that will have access to the system: < 100 , $100-250$, $251-500$, > 500 ;
- User storage quota: $< 50\text{ Mb}$, $5-100\text{ Mb}$, $> 100\text{ Mb}$;
- Industry selling information: *Financial*, *Industrial products*, *High Technology*, *Aerospace*, *Automotive*;
- Budget, $< \$100,000$, $\geq \$100,000$;
- Enterprise software packages (select all that apply): *Oracle ERP*, *BAAN*, *Peachtree Accounting*, *CISC*, *SAP*;

- Groupware software: *Lotus Notes*, *Microsoft Exchange*;
- ...

1.2 Thesis Contribution

The original contribution of this thesis is twofold, addressing the main issues raised by a configuration task: modeling the problem and solving it efficiently. Our approach subsumes previous approaches, incorporating both simplification and further extension, offering increased representational power and efficiency.

Modeling

The solution we propose is based on the idea of local, context independent models for the types of objects in the application domain. We build the model of an artifact as a composition of local models of the constituent parts. Part of the type description is a well specified interface through which specific requirements, imposed on particular instances by the context in which these are used, can be expressed. All the specific aspects of using the constituent parts in the particular context of a component/product are specified in the model of that component/product.

Finding a suitable representation for these models is difficult for two reasons. One is the complexity and heterogeneity of relations that must be expressed, manipulated and maintained. Inheritance, aggregation, producer-consumer or compatibility are just a few of the relationships among object types, while combinations of specific instances can be restricted by arithmetic, geometric or structural constraints. The second aspect is the dynamic nature of configuration tasks. Information controlling the evolution of the model, as well as mechanisms for enforcing it, must be contained in the model as well. We address these is-

sues by introducing Composite Constraint Satisfaction Problems, a new, nonstandard class of problems which extends the classic Constraint Satisfaction paradigm. Generalization and aggregation are captured directly by the definition of a composite CSP, while relations among instances are expressed through port variables. The dynamic aspect is handled through the set of constraints posted on port variables and the instantiation mechanism.

Another critical requirement for the knowledge representation mechanism used in configuration is the ability to cope with the high rate of change of the domain knowledge. Changes in a component type specification should not propagate across the knowledge base, affecting other component types. Adding or eliminating component types should be handled without any disruption and should not require any special user intervention. Our modeling technique integrates two mechanisms for dealing with complexity, namely composition and abstraction. Using concepts such as locality, aggregation and inheritance, it offers support and guidance as to the appropriate content and organization of the domain knowledge, thus making knowledge specification and representation less error prone. Furthermore, based on a declarative paradigm, our framework provides complete separation between domain knowledge and control strategy, and this makes knowledge maintenance much easier.

Efficiency

For the purpose of the work presented here, we are only interested in complete search methods, that produce a guaranteed optimal solution to a configuration problem. To achieve this goal, our research focused on two complementary directions.

The first one led to a powerful search algorithm called Maintaining Arc Consistency Extended (MACE). By maintaining arc consistency and taking advantage of the problem structure, MACE turned out to be one of the best general purpose CSP search algorithms

to date.

The second research direction aimed at reducing the search effort involved in proving the optimality of the proposed solution by making use of information which is specific to configuration problems. Our strategy in achieving this goal was to make the algorithm automatically tune up the model based on the information already available in the specifications, without putting any additional burden on the user. First, by adding redundant specialized constraints, the algorithm improves dramatically the lower bound computation. Using abstraction through focusing only on relevant features allows the algorithm to take advantage of context-dependent interchangeability between component instances and discard equivalent solutions, involving the same cost as solutions that have already been explored. Experimental evaluation shows large increase in performance when both techniques are combined together.

1.3 Thesis Overview

The thesis consists of three parts. The first part, Chapters 2 and 3 present background information. Chapter 2 introduces the motivation for this work and gives an overview of existing approaches to representing and solving configuration problems. Chapter 3 makes a brief presentation of the CSP paradigm, defining concepts and algorithms relevant to our work. Our original contribution is presented in the rest of the thesis. The second part, Chapters 4 and 5, discusses main aspects involved in modeling configuration tasks. Chapter 4 identifies issues to be solved, and provides a powerful and intuitive composite model for representing configuration problems. Chapter 5 introduces a new class of nonstandard CSPs, composite CSPs, and shows how we implement configuration models as composite CSPs. The last part, Chapters 6 and 7, presents the algorithms that we use in our framework.

Chapter 6 presents the MACE algorithm. In Chapter 7 we show how by adjusting the model dynamically we can improve overall efficiency when searching for optimal solutions. Both chapters offer experimental evidence supporting our claims. Chapter 8 contains concluding remarks.

CHAPTER 2

CONFIGURATION BACKGROUND

The place of configuration. Comparison with other reasoning tasks. Problem description. Problem specification. Current solutions. Rule-based systems. Model-based reasoning. Logic-based approaches. The resource-based approach. Constraint-based approaches.

Today's business environment is competitive on a global scale. To compete effectively on the rapidly changing market, manufacturers must differentiate their products by focusing on individual customer needs. To face this challenge, companies must move into a new manufacturing paradigm. The classic paradigm, *mass production*, achieves economies of scale by reducing the unit costs of nearly identical products by using high-volume manufacturing techniques. In *custom manufacturing*, at the other end of the spectrum, products are tailored according to specific customer requirements.

Mass customization aims at combining the previous paradigms, in achieving economies of scale of mass production while offering the flexibility of custom manufacturing at the same time. The impact of this strategy on organizations is twofold, affecting both the product realization process and the order realization process.

At the product realization level, using a mass-customization strategy translates into having a new perspective on design. The goal of design has to shift from designing individual products towards designing families of products. The solution to a design problem must thus produce a generic product architecture, which represents a set of different specific

product architectures. The generic product architecture is described in terms of generic parts, which refer to sets of alternative components. The variety required by customers is achieved by “plugging in” various parts, implementing different functionality, in the generic architecture. In particular this means establishing standard interfaces between parts, such that different components could be combined systematically in order to cover the desired range of possible functions. Designing a product family can thus be seen as the process of capturing multiple product variants within a single data model (Mannisto, Peltonen, & Sulonen 1996). And the same principle applies recursively at the parts level.

At the order realization level, the requirement is to understand accurately the customer’s needs and to create a complete description of a product variant that meets those needs. This is the *configuration* step. Given a set of customer requirements and a product family description in the form of the generic product architecture, the task of configuration is to find a valid and completely specified product structure from among the alternatives described by the generic architecture.

A side effect of using flexible product lines, which can involve hundreds or thousands of configurable parts, is the increased possibility for errors. *The configuration step thus becomes crucial for the success and efficiency of the entire enterprise.* Errors made during this early phase, when requirements are captured, functional specifications are created and the appropriate architecture is selected, create major problems in meeting the schedule and lead to costly iterations that occur in later phases downstream. The later the error is discovered, at assembly time, at testing time, or by the customer, the higher the costs associated with it are.

This explains the renewed industrial interest in configuration and the large number of configuration related research projects in the past decade, which lead to techniques for

representing and solving configuration problems.

2.1 Comparison with Other Reasoning Tasks

Configuration is a special case of a reasoning task. Reasoning tasks can be classified in several ways (Parunak, Kindrick, & Muralidhar 1988) and (Queyranne 1990). Based on the type of the outcome, reasoning tasks can be divided in two classes, *synthesis* and *analysis*, as presented in Figure 2-1. In synthesis, the task is to derive the structure of a system starting from a set of specifications and requirements. In contrast, in analysis the structure of the system is known and, based on it, we predict the behavior of the system (*simulation*), or compare the expected behavior with the actual one (*diagnosis, monitoring*).

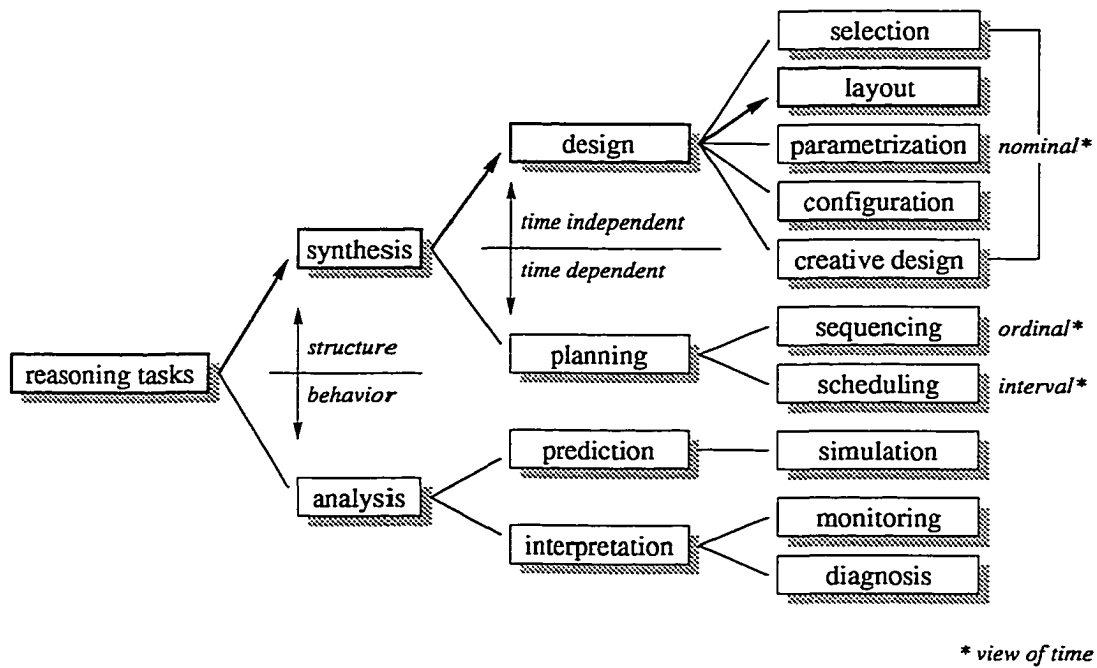


Figure 2-1: Reasoning tasks hierarchy

If *time* is taken into account, synthesis tasks can be further refined in *design* and *plan-*

ning. Design tasks do not reason about time: the specifications describing the artifact (structure, behavior) do not imply any time dependency. The solution to a design problem must satisfy all the requirements simultaneously, and its structure and composition do not change over time. On the other hand, planning involves imposing a (partial) order over time on a set of steps or events. If the duration and the distance between two steps or events is not important, the planning task is a *sequencing* task, where if we consider duration and distance, we have a *scheduling* task.

2.2 Problem Description

A configuration can be defined informally as a special case of design activity, with the following key features (Mittal & Frayman 1989):

- the artifact being configured is assembled from instances of a fixed set of well-defined component types, and
- components interact with each other in predefined ways.

Selecting and arranging combinations of parts which satisfy given specifications is the core of a configuration task. During this process, no new component types can be created and the interface of the existing component types cannot be modified. The solution has to produce the list of selected components, and, as important, the structure and topology of the product.

According to this definition, besides the activity of creating technical products, configuration fits a broad set of everyday tasks: developing a therapy as a composition of repair actions, synthesizing problem solving strategies, or composing qualitative models to explain a phenomena. All these tasks can be reduced to the generic task of “assembling” some “ar-

tifact” from a fixed set of available “building blocks”: actions in therapy, knowledge sources in problem solving synthesis, or models in qualitative reasoning.

2.2.1 Specification

The specification of a configuration task involves at least two distinct phases: the description of the *domain knowledge*, and the *specification of the desired product*. The domain knowledge describes the *types (classes) of objects* available in the application domain and the *relations* among object instances. The specification of the desired product includes *requirements* that must be satisfied by the product, a description of the *environment* in which the product has to operate, and, possibly, the *optimization criteria* that should be used to guide the search for a solution.

There are several aspects which differentiate between configuration tasks and other problem solving activities. The most important ones, with deep implications from both the representational and algorithmic point of view, are pointed out below.

Closed-world Assumption

Selecting and arranging combinations of parts which satisfy given specifications is the core of a configuration task. During this process, no new component types can be created and the interface of the existing component types cannot be modified. The implication of this assumption is twofold. First, configuration problems are well-structured and completely specified: the description of all the components is complete and all the relations and constraints among different components can be stated explicitly. Second, components have a well-defined internal structure, which cannot change during the search for a solution.

Hierarchical Organization

Each application object is uniquely identified within the application universe by the set of its properties, which characterize its function and performance. It is a common practice to organize application objects in an abstraction/generalization hierarchy. The hierarchy contains both abstract and physical entities. The physical entities are concrete parts, or components, available in the application domain. By generalizing the properties of a class of physical parts we develop abstract parts. During this process, common structure and functionality are factored out, while specific aspects are identified.

Dynamic Nature

According to a recent survey (Stefik 1995), most of the reported configuration systems rely on the previous characteristics, and work through well-defined phases: user specifications lead to an abstract configuration, where goals are represented in terms of the desired functions the system has to provide. This abstract solution then undergoes an expansion and refinement process until a complete, detailed configuration is obtained. This process is dynamic in nature. During the search for a solution new components will be created and added to the current configuration as needed for maintaining consistency with the requirements ¹.

Exactly what is an abstract configuration depends on the representation choice, ranging from a set of desired functions the system has to provide to a minimal set of required parts, both abstract and concrete. Examples are presented in the following sections.

¹ We emphasize here the difference between a component type and its instances. Although the number of component types is fixed, there is no predefined limit on the number of instances of a component type that can be created.

2.3 Current Solutions

As we have already pointed out, configuration systems have a long history in AI, starting with the R1/XCON system from Digital, and continuing with a long list of systems developed by other organizations. Currently, there are four main approaches to configuration. One of them uses heuristic reasoning, while the other three are model-based.

2.3.1 Rule-Based Systems

This type of system, also known under the generic name of *expert system*, uses *production rules* as a uniform mechanism for representing both domain knowledge and control strategy. The first expert system in daily use in industry (McDermott 1981), (McDermott 1982), (Bachant & McDermott 1984), XCON became a typical example of successful application of expert systems technology.

XCON uses OPS5, a production rule programming language. Prior to XCON, Digital attempted to tackle the configuration task using traditional procedural programming languages, but without success. Procedural languages, with their commitment to predetermined sequences of activities with limited branching, did not provide the flexibility at run time that is needed to cope with two key aspects of configuration tasks:

- *what* actions need to be performed to obtain a valid configuration, and
- *when* can an action appropriately occur in relation to other actions.

The production rule programming paradigm explicitly provides the dynamic, run-time decision making that is essential for coping with the above characteristics.

The fundamental activity of any program written in OPS5 is the execution of the *recognize/act* cycle (Bachant & Soloway 1989). A production rule has the form *if condition*

then consequence. A *working memory* holds global state information: inputs and results from executing actions. The conditional part of the rule specifies tests on the working memory. If the conditional part is satisfied, then the consequence part of the rule is executed, possibly modifying the working memory.

Solutions are derived in a forward-chaining manner. At each step, the system examines the entire set of rules and considers only the set of rules which can be executed next. Each rule carries its own complete triggering context, which identifies its scope of applicability. One of the rules under consideration is then selected and executed, by performing its action part.

Although it was claimed that the rule-based reasoning allows incremental development, it became soon apparent that one has to deal with enormous maintenance problems for large rule-based systems (Golden, Siemens, & Ferguson 1986) (Barker & O'Connor 1989). Rules specify both *directed relationships* and *actions*. A directed relationship represents domain knowledge (compatibilities, dependencies, *etc.*), while an action represents (procedural) knowledge controlling the computation of a solution. It is exactly this lack of separation between domain knowledge and control strategy and the spread of knowledge about a single entity over several rules that make the knowledge maintenance task so difficult.

Consider the problem of updating rules in light of changes in the component types specification. It is very hard even to know if one has found all the rules that need changes. Furthermore, if the condition does not trigger, the consequence part of the rule is not considered. It is the rule base developer's responsibility to ensure that the set of rules covers completely the context of each desired consequence. To get an idea about the complexity of the knowledge maintenance task, XCON had in 1989 in its knowledge base more than 31,000 components and approximately 17,500 rules (Barker & O'Connor 1989). The change

rate of this knowledge averaged about 40% per year.

This challenge led to the development of a programming methodology, RIME, that provides structuring concepts for rule-based programming (Bachant 1988). Meta rules are used to control and order context specific decisions. The user can force, at development time, the firing of rules in a fixed sequence, thus decomposing the problem into steps. Upon entering a step, the satisfied rules are activated and the process proceeds to the next step. Indirectly, this provides some guidance in organizing the rule base, but due to the sheer quantity of rules and their size, the maintenance problem remains still unsolved.

2.3.2 Model-Based Reasoning

Unfortunately, most of this early work offers only a limited understanding of the configuration process, the existing systems being designed to solve specific instances of configuration tasks. As an effort to address the limitations of expert systems, mainly the maintenance problem (Barker & O'Connor 1989), *model-based reasoning* emerged as a new field of Artificial Intelligence research, around 1980. The main assumption behind model-based reasoning is the existence of a *model* of the system, which consists of *decomposable entities* and *interactions* between their elements. The most important advantages of model-based systems are, as presented in (Hamscher 1992):

- a better separation between *what* is known and *how* the knowledge is used,
- enhanced robustness (increased ability to solve a broader range of problems),
- enhanced compositionality (increased ability to combine knowledge from different domains within a single model), and
- enhanced reusability (increased ability to use existing knowledge to solve related

classes of problems).

There are several model-based approaches to configuration. We will present in the following subsections the most relevant ones, characterizing them along the lines described earlier.

Logic-Based Approaches

One prominent family of logic-based approaches is based on *Description Logic (DL)*. Description Logic is not monolithic. There is an entire assortment of description logics and extensions implemented by the DL community, each one better suited for a specific type of application. Description Logics are formalisms for representing and reasoning with knowledge, unifying and giving a logical basis to the well-known traditions of frame-based systems, semantic networks and KL-ONE-like languages, object-oriented representations, semantic data models, and type systems. There are three fundamental notions in DLs:

- *individuals*, representing objects in the application domain,
- *concepts*, representing sets of individuals, and
- *roles*, which are binary relations between individuals.

These systems reason about the intensional description of concepts and their instances (individuals). Complex, composite descriptions can be created using *constructors* (e.g. *and*, *or*, *at-least*, *all*, *etc.*).

The main inference mechanism in DLs is *subsumption*, i.e. decide whether one concept (description) is more general than another concept (description). The semantics of subsumption is defined by the subset relationship between the two concepts as sets of individuals. A concept C_1 subsumes another concept C_2 when every instance of C_2 is also

an instance of C_1 . Most of the other forms of inference performed in DLs systems can be expressed using subsumption. Examples are classification and recognition. Classification is the process of integrating a new concept into the concept hierarchy, and recognition determines if an individual instantiates a particular concept.

The clear semantics and simple logical operations made DLs popular in theoretical studies as well as practical applications. Description logic-based configuration applications have been used within AT&T since 1990 (Wright *et al.* 1995). They are based on CLASSIC, an object-centered knowledge representation and reasoning tool. DLs can offer support for such applications both during the knowledge acquisition phase and the problem solving phase.

The advantages of DLs during the knowledge acquisition phase are clear. The classification facility automatically organizes descriptions into an explicit taxonomy, based on subsumption inferences. In addition, consistency is automatically maintained over time, as new descriptions are classified and added to the knowledge base, or existing descriptions are modified and reclassified.

During the problem solving phase, DLs can offer support in one of two ways. The first possibility is for the DLs system to provide run-time support for other configuration engines. The organization of the taxonomy, based on subsumption, enables efficient retrieval of descriptions. Another advantage of DLs is their ability to deal with partial, incomplete descriptions (of systems, component types, functionality, *etc.*). Furthermore, there exist different extensions to DLs, that have been designed to provide special types of reasoning, for example *explanation* (McGuinness & Borgida 1995).

The second possibility is for the DLs system to solve the entire configuration problem. An example is presented in (Wright *et al.* 1995). The terminological knowledge base con-

tains descriptions of the classes (component types) as well as rules. These rules, attached to concepts on different levels of abstraction, can extend and refine an individual configuration as required (in more sophisticated systems, an external, domain-specific control mechanism may be used for increasing the efficiency). After the interface has guided the user through some simple questions, the system uses these inputs, and the application, through follow up questions guided by CLASSIC, arrives at a complete (abstract) solution.

This approach has one potential drawback: the tradeoff between the efficiency of the reasoning tasks and the expressiveness of the knowledge representation tool is crucial. If the formalism aims to a certain level of expressiveness (by allowing existential quantifications or disjunctions, for example), subsumption becomes \mathcal{NP} -complete. On the other hand, restricting expressiveness to ensure tractability makes the formalism unable to represent complex systems, which is often the case in representing practical configuration tasks.

Other examples of logic-based systems for configuration include *Prose* (Wright *et al.* 1993) and *Beacon* (Searls & Norton 1988). The *constructive problem solving*, described in (Klein 1996), is a different logic-based approach, centered around the idea of model construction.

The Resource-Based Approach

In the resource-based approach, the interfaces through which technical systems, their components, and the environment interact are represented as *abstract resources* (Heinrich & Jungst 1991). A resource-based system offers a producer-consumer model of the configuration task, in which each technical entity is characterized by the amount and type of resources it *supplies*, *uses*, and *consumes*. The description of the environment, similarly expressed in terms of the amount of resources demanded from and supplied for the technical

system, represents the requirement specifications. The goal is to find a set of components that bring the overall set of resources in a balanced state, where all the demands are fulfilled. A configuration is acceptable only if the the resources demanded by environment and different components are each balanced by the resources the environment and components can maximally supply.

The algorithm for solving a configuration task in the context of this model is straightforward (Heinrich & Jungst 1991). Start with the set of resources demanded by the environment in the requirement specifications. Then, select one resource type which is not balanced yet and create the list of component types that can supply that resource. Select one component type from the list and add an instance of that component to the current (partial) configuration. Repeat this process until for every resource the required amount is less than or equal to the amount supplied by the environment or by components. In case of dead-end, backtrack to the last choice point.

Domain knowledge is organized on three levels

1. *system knowledge*, describing the types of resources associated with a given application domain. It is represented as a taxonomy of resource types.
2. *catalog knowledge*, embedded in the description of the available component types. It can be organized in a hierarchy based on component similarity from the point of view of the resources they use/consume or supply.
3. *heuristic knowledge*, used for guiding the resource and component selection process during the search for a solution. This type of knowledge can be further divided into three categories, represented in the form of value slots attached at different levels of abstraction in the resource and component taxonomies:

- *evaluation knowledge*: a measure of how good a component is. It is represented by the cost entailed by including the component in the configuration.
- *performance knowledge*: sequencing of decisions that are known to lead quickly to a solution. Represented by an order imposed on the resource types based on static priorities.
- *exception knowledge*: rules on resource types and amounts. It is part of the performance knowledge.

The approach has been implemented in the COSMOS system, an expert system shell for configuration. The inference engine used in COSMOS has a *blackboard architecture*. A *decision record* keeps track at each decision step of the list of all viable component types, in decreasing order of their utility. *Balance sheets* tally for each type of component the amount required versus the amount supplied by the components already selected. The *agenda* contains one of two types of action, either component selection (creation of the next entry in the decision record) or component positioning (instantiating a resource by associating it with the selected component). The process starts with the environment as the first selected component.

The approach is well-suited for configuration tasks where the main concern is in covering a desired functionality, especially when single components satisfy only partially that functionality. For example, the system offers an elegant solution to the configuration of modular systems, where all constraints are resource-based. The method loses its elegance and simplicity when forced to deal with structural and specific placement requirements.

Constraint-Based Approaches

The first attempt to define a generic domain-independent model for configuration tasks, and one of the most important works towards a general model of configuration, was presented in (Mittal & Frayman 1989). The paper identifies the main characteristics of configuration tasks and introduces the definition presented in section 2.2.

Each component is defined by a set of properties and a set of ports for connecting to other components. Constraints among components restrict the ways in which various components can be combined to form a valid configuration. In addition to component descriptions, the specification for a configuration task also includes a description of the desired product and optimization criteria.

Given a specification, the goal is to build one or more configurations (i.e. set of components and a description of the connections between them) that satisfy both the specification and optimization criteria, in case such solutions exist, or detect inconsistencies in the requirements otherwise.

The problem solving method described in (Mittal & Frayman 1989) is based on two simplifying assumptions about the domain knowledge. First, configuration is usually a purposeful activity, i.e. the kinds of functional roles that have to be fulfilled by the artifact are known ahead of time. Second, for each functional role one or more components can be identified as the *key component*, i.e. any implementation of that function will include one of these components. This restricted form of configuration task can be represented as a *constraint satisfaction problem (CSP)* (Tsang 1993) in which components and their ports are variables, and constraints restrict the way components can be integrated in a solution.

Because the mapping between functional roles and the set of components available is typically *many-to-many*, the configuration task is more of a dynamic nature. Different com-

ponents for the same functional role may need nonidentical sets of additional components or functional roles. Also, multifunction components often provide nonidentical sets of functions. To cope with this situation, (Mittal & Falkenhainer 1990) introduces an extension to the classical CSP paradigm, called *dynamic constraint satisfaction problems (DCSP)*. The additional requirements are then expressed by activity constraints, which make possible that new variables and constraints be dynamically introduced in the solution as the result of choosing a particular component to (partially) implement a functional role.

The main advantage of this extension over the standard CSP is that inferences can now be made about variable activity, based on the conditions under which variables become active, avoiding irrelevant work during search.

A formal, mathematically well-founded, treatment of the conditional existence of variables is presented in (Bowen & Bahler 1991) where the constraint network is viewed as a set of sentences in first-order free logic.

The dynamic CSP model described so far has, however, several limitations. The most severe one is the requirement that all the variables (and constraints among them) be declared from the beginning, although only a subset of them will be active, and thus part of a solution. It is not only a matter of efficiency, considering that sometimes it may be possible to come up with an upper bound on the set of variables needed to represent a particular artifact. There are cases when it is not practical (or even possible) to anticipate the set of variables to cover all possible solutions of a configuration problem.

The constraint-based configuration framework CONFCS (Haselbock 1993) extends the previous model to handle an indefinite number of variables. This is accomplished by specifying generic constraints on a meta level, as relations among component types, instead of component instances, and is based on the fact that the set of different component types

is finite and all components of a given type behave in the same manner. Since the set of variables that exist in a specific configuration is not predefined anymore, generic constraints involve meta-variables, which are place holders for the component variables. The approach eliminates the need for declaring the complete set of variables beforehand, and allows the treatment of multiple occurrences of components with similar behavior.

CONFCS also offers support for specifying constraints on sets of variables which are not known before hand. A special case are *resource constraints*, which express cumulative relations on resource properties of a set of components. The only restriction is that the aggregate functions have to be monotonic.

Although not really part of the framework, type abstraction can also be integrated in CONFCS and the author gives some ideas of how hierarchical reasoning can be integrated in the constraint-based schema. In addition, interchangeability (Freuder 1991) can be used to improve search efficiency.

2.4 Chapter Conclusions

Each of the approaches presented in the previous sections uses a different technique for representing the knowledge base and specific reasoning algorithms, but all agree that most of the complexity of solving a configuration task lies in the *domain knowledge* representation. This is due mainly to the heterogeneity of relations that one has to be able to express: component taxonomies, composition, arithmetic, geometric and structural constraints, and resource balancing (Klein 1996). Furthermore, just being able to represent the domain knowledge is not enough. The knowledge representation mechanism must also support the maintenance and evolution of the knowledge base.

To summarize, the challenges that configuration frameworks face are the following:

- provide an expressive and flexible knowledge representation formalism,
- provide efficient knowledge application in a highly combinatorial context,
- integrate in a natural way different types of reasoning, *e.g.* hierarchical and resource-balancing, and
- provide mechanisms for coping with the high rate of change of the knowledge base.

None of the existing frameworks is able to address all these requirements. The knowledge embedded in rule-based systems can be very complex and, in general, these systems have very good performance, but the lack of separation between domain and control knowledge leads to insurmountable maintenance problems. Logic-based systems make a clear tradeoff between expressiveness and efficiency. The resource-based approach is not powerful enough to represent and reason with topology information, and therefore there are certain classes of configuration problems which cannot be expressed at all.

Although expressive and powerful, the knowledge representation mechanism used by the existing constraint-based approaches does not structure well the knowledge base. In particular, because the model does not differentiate between structural (*e.g.* aggregation) and arbitrary, user-defined, relationships, it offers no support and guidance as to the appropriate content and organization of the knowledge, which makes modeling difficult and error prone (Sabin & Freuder 1998). Current systems do not capture explicitly the aggregate structure and hierarchical organization of the component types. This prevents them from further improving search efficiency and maintenance by taking advantage of these particular characteristics of a configuration task.

A second aspect that should be taken into account by any configuration framework is optimality. Most of the existing approaches disregard the fact that the solution to a

configuration problem, besides being valid, must also be optimal according to some user-defined optimization criteria. Even the few approaches that acknowledge this aspect do so without providing a concrete solution.

CHAPTER 3

BRIEF CONSTRAINT SATISFACTION BACKGROUND

Constraint satisfaction problems. Constraint network. Local and global consistency. Consistency inference. Constraint propagation. Backtracking search. Arc consistency. Advanced search methods. Forward checking. Maintaining arc consistency. Variable ordering heuristics.

Constraint satisfaction has established itself as a well founded formalism with wide application in artificial intelligence. Based on a simple mathematical model, domain independent and completely declarative, the *constraint satisfaction problem* (CSP) paradigm provides an elegant and natural framework for representing and solving a large variety of reasoning tasks, and lately has been widely accepted, both in academia and industry, as the formalism of choice for dealing with optimization problems as well.

This is the result of a sustained research effort that started two decades ago, with the work presented by Waltz (Waltz 1975), and continued with landmark contributions by Mackworth and Freuder (Mackworth 1977), Haralick and Elliot (Haralick & Elliot 1980) and Dechter and Pearl (Dechter & Pearl 1988).

This chapter is not intended to be a survey of the state-of-the-art in CSP solving methods. Instead, it provides CSP definitions and techniques relevant for the work presented in the rest of the thesis. For a more detailed presentation of the field, the reader should consider (P 1991) or (Tsang 1993).

3.1 Definitions and Notations

A CSP \mathcal{P} is formally defined as a triplet $\mathcal{P} = \langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle$, where

- $\mathcal{V} = \{V_1, \dots, V_n\}$ is the set of problem *variables*, of cardinality $n = |\mathcal{V}|$.
- $\mathcal{D} = \{D_{V_1}, \dots, D_{V_n}\}$ is the set of *domains*. For each $V_i \in \mathcal{V}$, $D_{V_i} = \{d_{i_1}, \dots, d_{i_m}\}$ represents the finite set of possible values that can be assigned to variable V_i . Let d be the cardinality of the largest domain, $d = \max |D_i|, D_i \in \mathcal{D}$.
- $\mathcal{C} = \{C_{\mathcal{U}} \mid \mathcal{U} = \{V_{i_1}, V_{i_2}, \dots, V_{i_k}\} \subset \mathcal{V}, C_{\mathcal{U}} \subseteq D_{V_{i_1}} \times D_{V_{i_2}} \times \dots \times D_{V_{i_k}}\}$ is the set of problem *constraints*. Each tuple in $C_{\mathcal{U}}$ defines an ordered set of values that variables in the ordered set \mathcal{U} are allowed to take simultaneously. Let e be the total number of constraints, $e = |\mathcal{C}|$.

Two parameters are commonly used to characterize CSPs. The problem *density* represents the ratio between the number of problem constraints and the total possible number of constraints. The *tightness* of a constraint is the ration of the number of tuples disallowed to the total possible number of tuples.

The task of solving a CSP involves finding values for problem variables subject to constraints that are restrictions on which combinations of values are allowed (Tsang 1993).

To *instantiate* a variable means to assign that variable one of the values in its domain. A *partial instantiation* on an ordered set of variables $\mathcal{U} = \{U_{i_1}, U_{i_2}, \dots, U_{i_k}\} \subset \mathcal{V}$ is a tuple $I_{\mathcal{U}} = (d_{i_1}, d_{i_2}, \dots, d_{i_k})$ such that each variable $U_i \in \mathcal{U}$ is assigned the corresponding value $d_i \in I_{\mathcal{U}}$. If $\mathcal{U} = \mathcal{V}$, the set of problem variables, *i.e.* all variables have been assigned one value, the partial instantiation becomes a *complete instantiation*.

The *projection* $C_{\mathcal{U}}|_{\mathcal{T}}$ of a constraint $C_{\mathcal{U}}$ to a set of variables $\mathcal{T} \subseteq \mathcal{U}$, is a set of tuples obtained from tuples in $C_{\mathcal{U}}$, by keeping only the values that correspond to variables in \mathcal{T} .

Given a constraint C_U and a subset of variables \mathcal{T} , a partial instantiation $I_{\mathcal{T}}$ *satisfies* the constraint C_U iff $U \cap \mathcal{T} \neq \Phi \Rightarrow I_{U \cap \mathcal{T}} \in C_U|_{U \cap \mathcal{T}}$.

A partial instantiation I_U which satisfies all the constraints involving variables in U is called a *partial solution*. In this case, the partial instantiation I_U is said to be *locally consistent*.

Similarly, a complete instantiation which satisfies all the constraints of a CSP is called a *solution*, and the complete instantiation is said to be *globally consistent*.

A partial instantiation I_U is *locally consistent* iff it satisfies all the constraints involving variables in U . According to the previous definition, a partial solution is locally consistent. A partial instantiation I_U is *globally consistent* iff it can be extended to a solution.

A CSP $\mathcal{P} = \langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle$ can also be represented in the form of a hypergraph $\mathcal{H} = \langle \mathcal{X}, \mathcal{E} \rangle$ (also known as *constraint graph* for short, or *constraint network*), by associating a node X_i with each variable $V_i \in \mathcal{V}$ and a hyper-edge $E_U = \{X_{i_1}, X_{i_2}, \dots, X_{i_k}\}$ with each constraint $C_{U=\{X_{i_1}, X_{i_2}, \dots, X_{i_k}\}} \in \mathcal{C}$.

We use the notation \mathcal{N}_V to denote the set of variables to which a variable V is connected by constraints, called the *neighborhood* of V .

In case all the constraints of a CSP involve only pairs of variables, the problem is called a *binary CSP*, and its constraint graph becomes a regular graph. An example of binary CSP is presented in Figure 3-1. The problem has three variables, X , Y and Z . Each domain contains three values, $D_X = \{a, b, c\}$, $D_Y = \{d, e, f\}$, and $D_Z = \{g, h, i\}$. The binary constraints C_{XY} , C_{XZ} , and C_{YZ} , describe the set of allowed pairs of values.

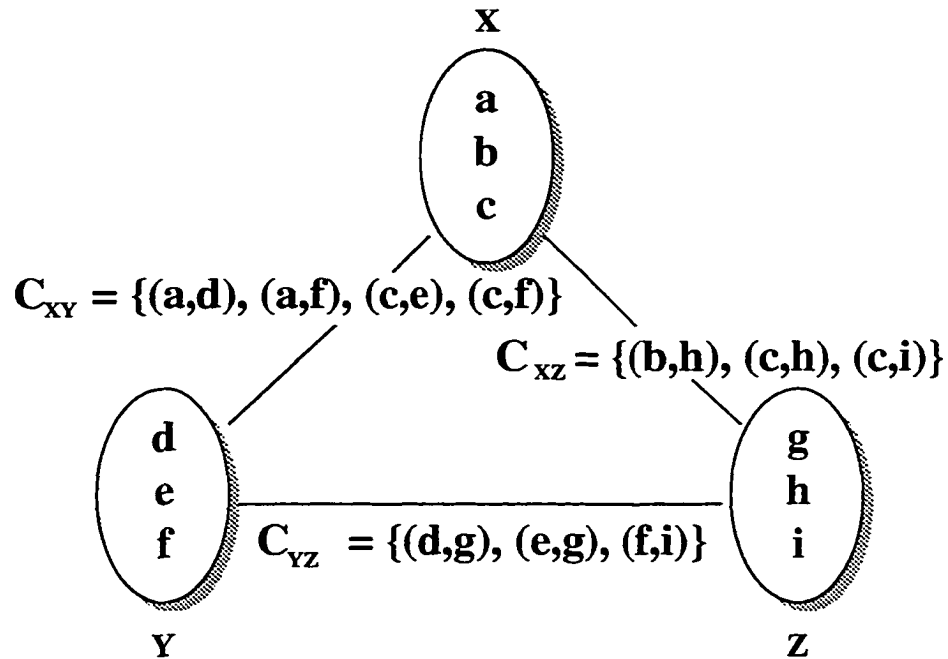


Figure 3-1: Sample CSP problem

3.2 CSP Solving techniques

Two standard techniques used in solving CSP problems are backtrack search and consistency inference.

3.2.1 Backtrack Search

Backtrack search is the standard procedure for solving CSPs. The idea behind backtracking is very simple. Considering that variables are ordered according to some criteria, the algorithm will try to instantiate each variable in turn. Each time a value is assigned to a new variable, the resulting partial instantiation is tested for consistency. If the test succeeds, the algorithm moves to the next variable. If the test fails, the variable is tentatively assigned the next value in the domain. Backtracking occurs when there are no more values in the domain to try. The algorithm then backtracks to the immediately (chronologically)

preceding variable and replaces its current value with the next value in the domain. If there is no such value, the algorithm backtracks one more step.

Assuming that we are only looking for one solution, the algorithm can end in one of two situations. If all variables have been instantiated successfully, the problem has a solution. However, if backtracking continues until all the values in the domain of the first variable have been exhausted, the problem is *inconsistent*, *i.e.* it has no solution.

The search space explored by backtracking algorithms can be represented as a tree. Each time a variable is instantiated, a node is added to the search tree. The root does not correspond to any instantiation. Instead, it is the common ancestor of all the nodes associated with the first variable. Each path from the root to any node represents a partial instantiation. A depth-first traversal of the search tree produces the sequence of steps taken by the algorithm. Figure 3-2 presents a sample search tree for the problem in Figure 3-1.

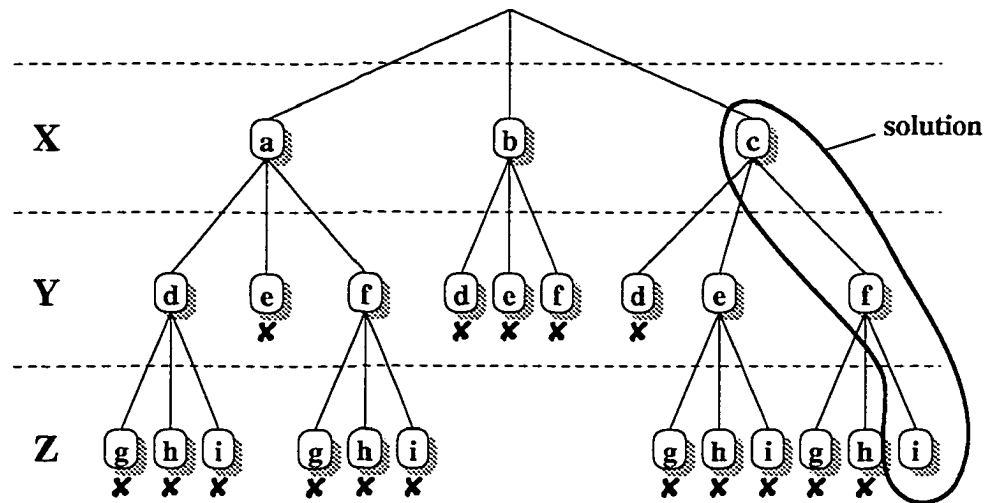


Figure 3-2: Typical search tree

For this example, the order in which we consider variables and values is the order in which they have been introduced. We start by assigning value *a* to variable *X*. There is no

previously assigned variable, so there is no constraint we have to check yet. We then assign value d to variable Y and check the constraint C_{XY} . Since pair $(a, d) \in C_{XY}$ the constraint is satisfied, and we move further. We try first value g for variable Z . Pair $(x, g) \notin C_{XZ}$ and this instantiation fails. Values h and i also fail. Because there is no other value in the domain of Z to try, we must backtrack. This means that we must try another value in the domain of Y . The next value, e , violates constraint C_{XY} . This process continues until we eventually find the solution $X = c, Y = f, Z = i$.

3.2.2 Consistency Inference

Consistency inference, sometimes referred to as *constraint propagation*, denotes a set of techniques used for transforming a CSP problem into an equivalent one, with exactly the same set of solutions, but easier to solve. The set of transformations includes eliminating values from domains, tightening constraints (i.e. reducing the number of tuples allowed), or even structural changes in the constraint network.

3.2.3 Arc Consistency

There are several consistency inference techniques available. The most widely used one is based on the notion of *arc-consistency* (AC) (Waltz 1975). Arc-consistency is a form of local consistency which requires that each value x_j from the domain D_{V_i} of variable V_i be compatible (i.e. *satisfies* all the constraints involving V_i). Given a CSP $\mathcal{P} = \langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle$, a domain $D_{V_i} \in \mathcal{D}$ is *arc consistent* iff $D_{V_i} \neq \Phi$ and $\forall d_j \in D_{V_i}, \forall C_U, V_i \in U \Rightarrow d_j \in C_U \mid_{V_i}$ (P 1991).

A binary CSP $\mathcal{P} = \langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle$, is arc-consistent iff $\forall X \in \mathcal{V} \Rightarrow D_X \neq \Phi$ and $\forall a \in D_X, \forall C_{XY} \in \mathcal{C}, \exists b \in D_Y$ s.t. $(a, b) \in C_{XY}$. In other words, for a CSP to be arc-consistent, each

value in the domain of each variable must be supported by at least one value in the domain of every variable in its neighborhood. If a certain value in the domain of some variable has no support on some constraint, that value is *inconsistent*, and is eliminated from the domain because it can never be part of any solution. If during this process the domain of any variable becomes empty, it means that the problem is *arc-inconsistent*, and has no solution.

Arc-consistency enforcing algorithms are recursive in nature. The idea is to successively eliminate from domains those values which are inconsistent. If a value that has just been eliminated was the only support on some constraint for another value, the latter becomes now inconsistent and the process continues recursively. This phase is called the *propagation* phase of the algorithm.

Figure 3-3 presents the actions required for making the CSP in Figure 3-1 arc-consistent. Values connected by lines in the picture represent pairs of values allowed by constraints. First we check all the values in every domain for consistency, and eliminate the inconsistent ones. We eliminate value a from the domain of X because there is no value in domain of Z to support it on constraint C_{XZ} (1). Similarly, we eliminate b at X because it lacks support on constraint C_{XY} (2). Value d at Y remains without support on C_{XY} , and is eliminated (3). Also eliminated are values g and h at Z , because they have no support on constraint C_{XZ} (4), and C_{YZ} (5), respectively. The last action, (6), is an example of how eliminating one value, g , from the domain of one variable, Z , propagates and leads to the removal of another value, e , at another variable, Y .

The first notable arc-consistency algorithm, *AC-3*, was presented by Mackworth (Mackworth 1977). The time complexity of *AC-3* is $O(ed^3)$ (Mackworth & Freuder 1985). Since then, a lot of research effort has been spent on providing more efficient algorithms. Mohr

There are two ways in which consistency inference can be used to improve search efficiency. Constraint propagation techniques can prune values from consideration either before or during search. The idea in both cases is the same. Since the size of the search tree is exponential in the size of the domains, smaller domains translate directly into smaller search trees. We will present this implication in more detail in the next section.

3.2.4 Higher Order Consistency

Arc consistency cannot eliminate all the inconsistencies present in a CSP problem. Consider the example in Figure 3-4. After eliminating values f and c , although the constraint network is arc-consistent, there is no combination of values which satisfies all the constraints, *i.e.* the problem has no solution.

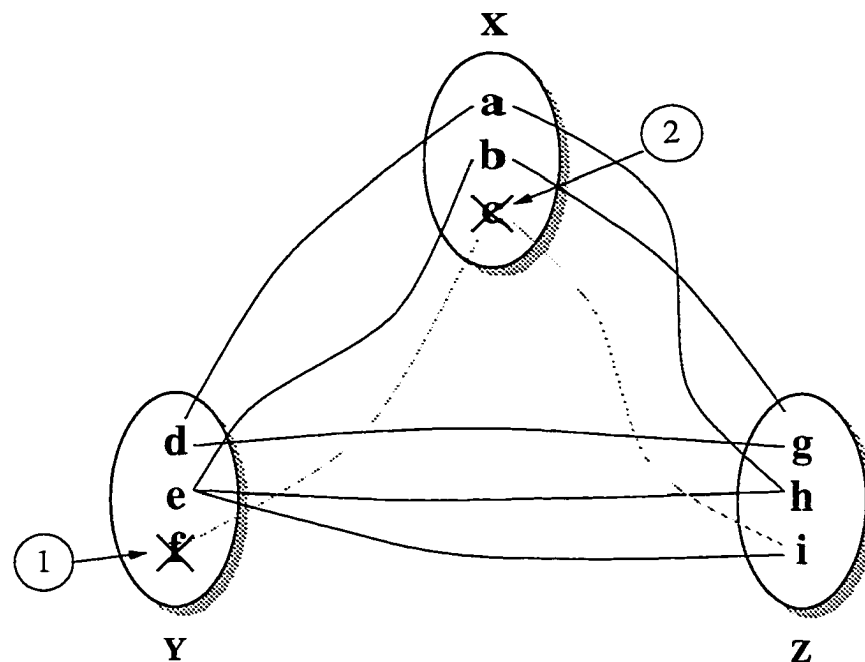


Figure 3-4: Enforcing arc consistency (example 2)

As a result, researchers started looking for stronger local consistency properties of con-

straint networks. The first one to formalize this was Montanari by defining the notion of *path-consistency* (PC) (Montanari 1974). But the most important work in this direction was presented by Freuder in (Freuder 1978). He defined the notion of *k-consistency*, which generalizes and extends previous forms of consistency.

A CSP $\mathcal{P} = \langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle$ is *k-consistent*, $1 < k \leq |\mathcal{V}|$ iff $\forall \mathcal{U} = \{U_{i_1}, U_{i_2}, \dots, U_{i_{k-1}}\} \subset \mathcal{V}$, $\forall I_{\mathcal{U}}$ a consistent partial instantiation, $I_{\mathcal{U}} = (d_{i_1}, d_{i_2}, \dots, d_{i_{k-1}})$ and $\forall X \in \mathcal{V} \setminus \mathcal{U} \Rightarrow \exists x \in D_X$ such that $(d_{i_1}, d_{i_2}, \dots, d_{i_{k-1}}, x)$ is a consistent partial instantiation on $\mathcal{U} \cup \{X\}$.

According to the new definition, arc-consistency is 2-consistency, while path-consistency is 3-consistency. Informally, a CSP is k-consistent if any consistent partial instantiation on a subset of $k - 1$ variables remains consistent when extended to any k^{th} variable.

Reading carefully the definition, the reader will realize that k-consistency with $k = |\mathcal{V}|$ is not equivalent to global consistency. This is because if there is no consistent partial instantiation on a subset of $n - 1$ variables, although the previous condition is satisfied, no solution is possible. This is why Freuder gave a second definition of consistency, which links together successive levels of consistency.

A CSP \mathcal{P} is *strong k-consistent* iff for any $1 \leq i \leq k$, \mathcal{P} is i-consistent.¹ The first consistency inference algorithm for achieving k-consistency was proposed by Freuder in (Freuder 1978). It was later improved by Cooper in (Cooper 1989) using an approach similar to the one used in AC-4.

Now we can make a direct equivalence between strong consistency and global consistency. If a CSP $\mathcal{P} = \langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle$ is strong n-consistent, $n = |\mathcal{V}|$, then it is also global consistent.

¹ Although we did not give a definition for it, *node consistency*, which corresponds to 1-consistency, involves simply instantiating variables only with values which satisfy all unary constraints.

However, the cost of achieving k -consistency is exponential in k . From a practical point of view this is too expensive and prohibits the use of k -consistency for k larger than 3.

3.3 Advanced Search Methods

The running time of the backtrack search algorithm is proportional to the size of the underlying search tree. For a CSP $\mathcal{P} = \langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle$, this is $O(d^n)$. Since the CSP problem is known to be \mathcal{NP} -complete, and thus it is unlikely that there exists a polynomial version of backtracking, various methods have been developed for improving the average case performance of CSP solving algorithms by concentrating on three main directions:

- Reducing the search space using consistency inference as preprocessing to eliminate values from consideration before search.
- Reducing the search space dynamically, during search, by using techniques which allow the algorithm to prune entire branches of the search tree. Algorithms produced by this line of research can be divided into two categories:
 - prospective (*look-ahead*) algorithms, and
 - retrospective (*look-back*) algorithms, also known as *intelligent backtracking* algorithms.
- Reduction of the search effort by guiding the search procedure on the shortest path to a solution.

3.3.1 Consistency Inference as Preprocessing

Conventional CSP wisdom says that using consistency inference in a preprocessing step, to prune values before search, will reduce the subsequent search effort. There has been

some question as to the degree of consistency preprocessing that is desirable — additional preprocessing effort may outweigh subsequent search savings (Dechter & Meiri 1989). However, it seems an obvious article of faith that removing values from consideration during a preprocessing step will lead to savings during the subsequent search step or, at the very least, do no harm. A counterintuitive demonstration that pruning values can increase search effort, was obtained recently by Prosser. He showed that pruning values can degrade performance for algorithms that employ “intelligent backtracking” (though the actual exhibited effects were small) (Prosser 1993a). We demonstrated in (Sabin & Freuder 1994) that there are circumstances in which pruning values by consistency preprocessing can in fact *greatly increase* subsequent search effort.

3.3.2 Prospective Search Algorithms

We present only prospective variants of backtracking in this section because our work is entirely based on look-ahead schemes. For details on retrospective schemes consult (Prosser 1993b).

Prospective search algorithms perform a limited amount of computation for enforcing some level of consistency after each variable is instantiated. The combination of consistency pruning with backtrack search has a long history (Gaschnig 1974), (Golumb & Baumert 1965), (Mackworth 1977). Various degrees of consistency processing interleaved with backtrack search were studied experimentally in (Haralick & Elliot 1980), (McGregor 1979), (B. 1989). A variety of algorithms were considered that alternate choosing a value for a variable with “looking ahead”, via a constraint propagation process, to infer the consequences of that choice for pruning the values available for the as yet uninstantiated variables. The algorithms differed in how much constraint propagation they performed, and thus in the

degree of consistency they achieved.

Forward Checking (FC) is an algorithm which does a minimal amount of constraint propagation, in the sense that it performs the minimal amount of lookahead needed to avoid having to “look back”, i.e. to avoid the need to check new choices against previous ones. It combines backtrack search with a very limited form of arc consistency maintenance. The main idea is to project forward the consequences of variable assignments during search. When a variable V is assigned a value $x \in D_V$, the new value is checked against the domains of each variable in the neighborhood of V that is as yet unassigned. All values inconsistent with x are removed. This way a limited form of arc consistency is maintained. (If, during this process, the domain of some variable becomes empty, then no complete extension of the current assignment set to a solution is possible, and the current assignment for V must be discarded). For details on forward checking consult (Haralick & Elliot 1980). Figure 3-5 presents the search tree for FC on the example in Figure 3-1.

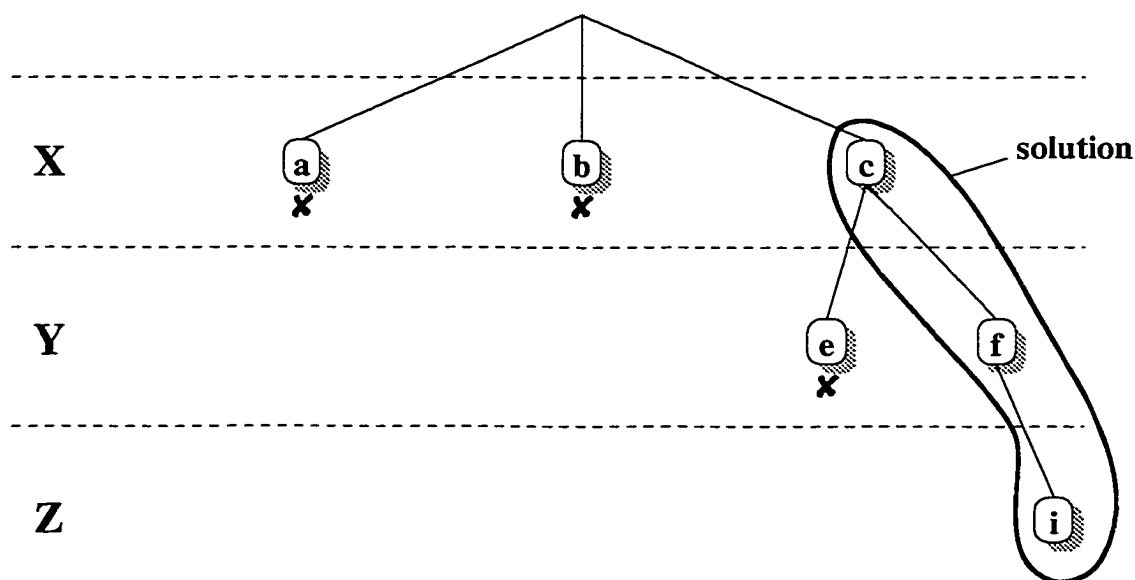


Figure 3-5: Search tree produced by Forward Checking

In experimental studies forward checking repeatedly proved superior to algorithms interleaving more constraint propagation. Of course, the limitations of these experiments were recognized. However, the repeated success of forward checking began to bias the conventional wisdom in the CSP community in the direction of “less is more”: using consistency inference during search, to prune values that become inconsistent after making search choices, is best limited to the minimal inference embodied in the forward checking algorithm. The feeling is that additional search savings produced by pruning more values will be offset by the additional inference cost. For example, in a recent survey of CSP algorithms (Kumar 1992), the section on “How Much Constraint Propagation Is Useful?” concludes: “Experiments by other researchers [in addition to Nadel] with a variety of problems also indicate that it is better to apply constraint propagation only in a limited form”.

In our laboratory several studies began to suggest that “more could be more”. Gevecker studied full arc consistency maintenance (Gevecker 1991) and Freuder and Wallace studied a range of hybrid algorithms based on a notion of “selective” or “bounded” constraint propagation (Freuder & Wallace 1991). However, these results were still limited in their understanding of the random problem space. Eventually, by introducing *Maintaining Arc Consistency* (MAC), an algorithm that efficiently maintains arc consistency during search, in (Sabin & Freuder 1994), we proved that maintaining *full* arc consistency during search is in fact very cost effective. Subsequent work done by Regin (Regin 1995) lead to the development of MAC7-PS, a variant of MAC based on AC7, which is the best general CSP search algorithm to date (Bessiere & Regin 1996). Figure 3-6 shows the search tree produced by MAC for the example in Figure 3-1.

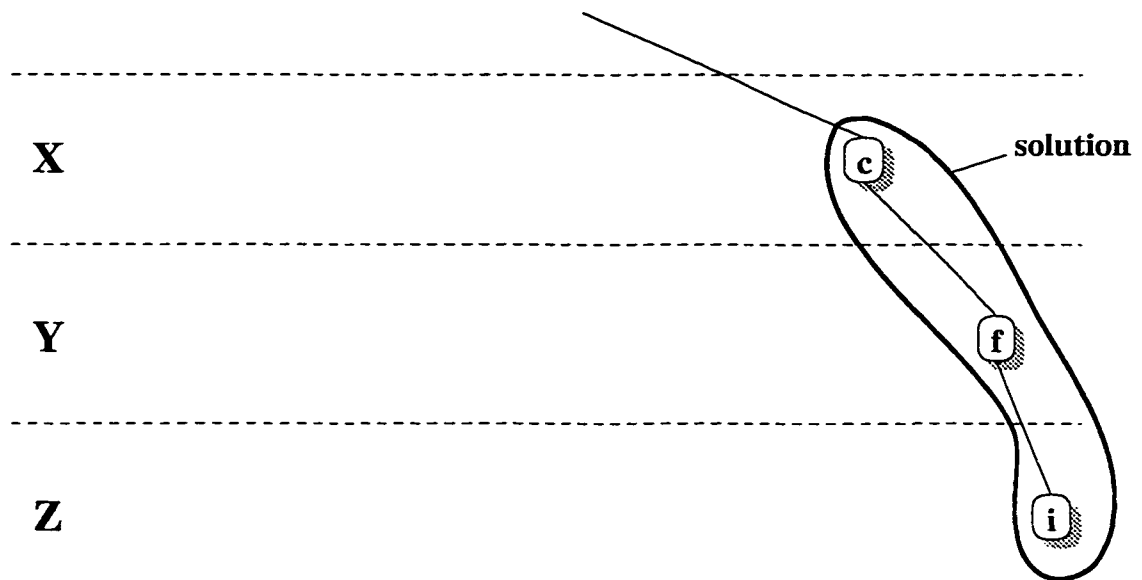


Figure 3-6: Search tree produced by Maintaining Arc Consistency

3.3.3 Ordering Heuristics

As mentioned before, the search algorithm assumes that variables, and their domains, are ordered according to some criterion. During search, the algorithm selects the next variable in the ordered sequence and assigns to it the next value available. When we are interested in finding either the first or the optimal solution, the ordering we choose can make a tremendous difference in performance. A static ordering is obviously less expensive, but research shows that the results produced by using a dynamic ordering are far superior.

There are many good variable ordering heuristics in the literature and it is beyond the scope of this section to present them in detail here. Instead, we single out two heuristics that perform very well in conjunction with prospective algorithms. The *minimum domain* heuristic, selects the variable with the least number of values still present in the domain as the next variable to be instantiated. It is a popular ordering heuristic. In a probabilistic

analysis, it was shown optimal under certain assumptions in (Haralick & Elliot 1980). It has proven particularly useful in conjunction with forward checking. A similar heuristic, *minimum domain/degree* (Bessiere & Regin 1996), which performs especially well in conjunction with MAC, selects as the next variable to be instantiated the variable with the smallest ratio between the size of the remaining domain and its degree in the constraint graph.

CHAPTER 4

COMPOSITE MODEL FOR CONFIGURATION

Aggregation and Context Independence. Composite models as aggregation of local models. Anatomy of a component type. The Descriptive Section. The Structural Section. The Internal Relationships Section.

Our research was partially inspired by the work presented in (Mittal & Frayman 1989). We adopt the same *component-oriented* definition of configuration because it is general enough to cover a broad range of configuration tasks. The two simplifying assumptions, the fixed set of well-defined component types and the predefined ways of interconnecting components, do not limit the scope of the definition. Instead, they reduce the complexity of the task and offer guidance during the search for a solution, thus increasing the efficiency.

Since a typical configuration application can involve hundreds or thousands of different, configurable components, the search space can be very large. To manage a task of such complexity, it is imperative to first organize the domain knowledge in terms of models. A *model* is a simplification of a real entity, capturing only the aspects which are relevant for a specific task. By using models, we can describe a problem in a formal manner, restating it at an abstract level more amenable to finding a solution. Modeling usually offers two techniques for dealing with complexity: *aggregation* and *abstraction*, and as we will see, both techniques appear naturally in configuration problems.

Additional motivation for using model-based configuration systems lies in the nature of configuration, which is, as presented earlier, a synthesis task. The ability to cover the entire range of possible solutions in a systematic way is a must, and this is exactly what using models allows us to do. Although experience in a particular domain can be gained and used to improve efficiency, the system should still work for application domains in which there is no prior experience.

4.1 Aggregation and Context Independence

A typical example of a configuration task is the configuration of technical systems, and there are two important observations we can make about this domain:

- (a) a product is assembled from components from a finite set of available component types.

We use here the term *component* in a generic manner, to designate an arbitrarily complex artifact, which can be used as a building block for obtaining other components and/or the desired product.

- (b) whether we are successful or not in solving a configuration task depends on the success of engineering configurable components that can be interconnected systematically, covering the entire range of desired functions. We should therefore bear in mind that a component was designed with a specific purpose in mind, to provide a certain function, and not to be used in a specific context.

Observation (b) above leads to the idea that domain knowledge can be expressed declaratively at the level of each component. More precisely,

- each component should be modeled **locally**,

in a context-free manner, independent of the context in which it may be used.

A natural solution to modeling a configuration task would then be, according to point (a) above, to

- build the model of the desired product as a **composition of local models** of the constituent components.

This can be done only if the components are represented by pure local models. But not all the domain knowledge is context-free. An important part of this knowledge refers exactly to how components relate to each other when assembled together in a particular environment. Apparently, by adding this type of contextual information to our models we destroy their context-free property. In fact, we can preserve this desirable property if all the specific aspects of using the constituent parts in the particular context of some component/product are specified in the model of that component/product.

4.2 Abstraction

Conceptually, a *type* is a description of all the properties, both behavioral and structural, a set of objects has. As mentioned before, all relevant properties are captured by a model. Thus a type is the formal representation of a model.

All the possible instances of a particular type form a *class*. By identifying common structure and functionality among the elements of a class, we can group similar instances in *subclasses*. We keep in the type only the information associated with properties common to all the instances of the class. We associate with each subclass a *subtype*, which contains only the properties which are specific to that particular subclass. We continue this process recursively, at the level of each subclass, until no more common aspects can be identified.

As a result, the domain knowledge is organized in the form of a tree, in which each

subtype is the descendent of its type ¹. Instances of a particular type inherit all the properties specified by the ancestors of that type. Because types which are internal nodes in the tree contain only a subset of the properties associated with real objects, their instances are abstract concepts, having no real-world counterpart. Types which are leaf nodes, on the other hand, describe completely objects in the application domain and their instances are concrete parts. The one-to-one relationship between a subclass (subtype) and its class (type) is called *generalization*, usually known as the “is_a” relationship. The inverse of generalization, a one-to-many relationship between a class (type) and its subclasses (subtypes), is called *specialization*.

4.3 Anatomy of a Component Type

The type associated with a particular class describes the entire part of the domain knowledge that refers to the class. This information is split into three sections:

- Descriptive information, consisting of properties based on which we can differentiate individual instances of the type,
- Structural information, specifying the composition of the object, in the form of a set of interconnected objects.
- Internal relationships established among constituent parts and/or their properties.

¹ For the purpose of modeling configuration tasks, we consider simple inheritance between classes sufficient, but nothing prevents us from using a different organization of the domain knowledge, based on multiple inheritance, if required by the application.

The Descriptive Section

An object is an instance of a type and has all the properties characteristic for that type. Besides common properties though, instances are distinct. They have their own identity according to the values of their properties. The collection of all the properties based on which instances can be differentiated forms the descriptive section of a type. Based on their semantics, properties can be further divided in:

- *Attributes*, which specify descriptive features, like functional and technical characteristics, visual properties, physical dimensions, *etc.* An attribute can:
 - have a single value (*e.g.* $rack-count = 1$, $height = 1''$, *etc.*) or
 - take values from a predefined domain (*e.g.* $cache-size \in \{ 0, 32Mb, 64Mb \}$, $5 \leq bay-count \leq 15$, $disk-size \geq 21\text{ Gb}$, *etc.*).
- *Resources*, which specify (functional) characteristics that can be either supplied or used by different components in the system. Each resource in a system must be balanced, *i.e.* the amount of resource produced must be at least equal to the amount of resource consumed. This requirement can be expressed globally, at the level of the entire system, and/or locally, at the level of a particular component. Resource balancing is the main mechanism that controls what new parts are created and added to the system.

There are several dimensions along which we can classify resources. From a semantic perspective, resources can be either:

- *qualitative*, *e.g.* *electrical power*. A power supply will provide the resource electrical power, while an electronic device will use the resource electric power,
or

- *quantitative*, for example the amount of power supplied by a power supply.

From the point of view of how the resource is being used, we identify the following cases:

- the resource is *produced*, *e.g.* the power supply supplies 2,000 watts of electrical power.
- the resource is *shared*, *e.g.* internal memory is shared among programs running in a multitasking system.
- the resource is *used*, *e.g.* the cabinet uses 1,000 watts out of the 2,000 watts of electrical power supplied by the power supply.
- the resource is *consumed*, *e.g.* a printer consumes the data-path resource provided by a cable.

In general, qualitative resources can be either shared or consumed, while quantitative resources are either produced or used in predefined units (*e.g.* Kbyte, inch, dollar, etc.).

The Structural Section

As described at the beginning of this chapter, we view a component as an aggregation of objects which interact under the control of a design. The model is recursive in that each object is a component in its own right and thus may have its own internal composition.

The second section in the specification of a component type enumerates the set of objects (sub-components) which are part of the component. Conceptually, all these objects exist only if the component exists. The one-to-one relationship between a sub-component and its component is called “part_of”. The inverse relationship, between a component and

its sub-components is a one-to-many relationship called “is_composed”. For example, the composition of a server system includes a system console, and a power supply. In the process of configuring an instance of SERVER SYSTEM, exactly one instance of SYSTEM CONSOLE and one instance of POWER SUPPLY will be automatically created and added to the configuration.

In addition to sub-components, both abstract and concrete, a component may also contain homogeneous collections of objects which can be seen as a whole. We refer to such a collection by the generic term *port*. Conceptually, a port represents an *association*, which describes a relationship between a set of objects (the members) and the component to which the port belongs.

A port is described by its own properties and may be connected with other objects. Two special properties, which are found in the description of any port, are the attributes *cardinality* and *base-type*. Base-type represents the type of the objects allowed to “connect” to the port. Cardinality controls the minimum and maximum number of these objects. The server system described earlier contains a set of racks, expressed in the model as a port of type RACK of cardinality minimum 1 and maximum 3.

Ports are also used for representing optional components in the model. An optional sub-component can be seen as a set of at most one element. For example, in addition to the set of racks, a system console and a power supply, the server system may also contain a server console and a laser printer. The server console and the laser printer are represented by two ports, each with cardinality between 0 and 1.

The Internal Relationships Section

The design of a component defines the interactions between its sub-components and describes the connections among them. The main source of complexity in modeling a con-

figuration task is the complexity and heterogeneity of these inter-object relationships. We have already presented one type of relationship, generalization, which describes the relation between component types. The structural section of a component type introduces another type of relationship, this time between component instances, namely aggregation.

All the other inter-object relationships among sub-component are specified in the internal relationships section. Some describe functional dependencies (*e.g.* the functional relation between electrical currents expressed by Kirchoff's laws), performance requirements (*e.g.* CPU speed \geq 400 MHz), compatibility information (*e.g.* country = France \rightarrow power_supply.voltage = 220 V), producer-consumer relations (server_system.power = \sum racks.power), *etc.* These can be seen as restrictions imposed by design on how sub-components can be interconnected.

Other relationships describe arbitrary associations between objects that otherwise can exist independently of each other. These are expressed using ports. For example, a server is "mounted_on" a rack of a server system. This relationship is expressed in our model by adding a port *racks* of type RACK to the model for SERVER SYSTEM.

Conceptually, a port offers a dual point of view on a set of objects. First, a port can be viewed as an object in its own right, and thus can participate in relationships with other objects, *e.g.* the port *SIMM_slots* is "part_of" a SERVER.

Second, a port can be viewed as a collection of objects which are in a certain relationship with the component to which the port belongs. The relationship can be defined by restricting the objects allowed in the collection. The port is used as a vehicle for introducing restrictions on the attributes of the candidate objects. For example, *SIMM_slots.pin_type = gold* implies that only MEMORY instances which have the value *gold* for the attribute *pin_type* can be in the relation "mounted_on" with an object of type SERVER.

Furthermore, by viewing a port as a collection of objects we can express producer-consumer relationships between sub-components. Again the port is the vehicle. It allows us to define cumulative expressions over attributes of all the instances in the collection. For example, $\sum CPU_slots.size$ represents the sum of the value of attribute *size* over all the instances of MEMORY “mounted_on” a server. We can now define the relationship between the attribute *memory_size* of an instance *serverA* of SERVER and the total amount of memory “mounted_on” it: $serverA.memory_size = \sum serverA.CPU_slots.size$.

4.4 Chapter Conclusions

Using local models for components allows us to separate completely the component description from the environment in which particular instances are used. There are two benefits associated with this. The first one is *reusability*. The same model can be used for each instance of a given type. The second benefit is *increased maintainability*. Changes in the specification of a component type are local to the component model, and require no modification outside it.

To preserve these benefits, all the interactions between any component instance and a particular environment in which the instance is used (*i.e.* larger component, artifact, *etc.*) must not be part of the component’s model. Instead, they are part of the model of that environment. It is this locality that allows us to increase the efficiency of the search algorithm by isolating, and then making inferences on, contextual information, as we will show later in Chapter 7.

CHAPTER 5

CONFIGURATION AS COMPOSITE CONSTRAINT SATISFACTION

The Composite Constraint Satisfaction Problem. Composite values. Port variables. Related work. Hierarchical Domain CSPs. Dynamic Constraint Satisfaction Problems. Meta Constraint Satisfaction Problems. Component Type models as Composite CSPs. Composite CSPs as knowledge representation mechanism.

Highly declarative, domain independent, and simple to use, the CSP paradigm offers a good starting point for a configuration framework. Unfortunately, the classic CSP formalism is not powerful enough to capture:

- The internal structure of components and the hierarchical organization of component types.
- The large variety of complex relationships among components, and
- The dynamic nature of the configuration process.

5.1 The Composite Constraint Satisfaction Problem

After exploring several possible extensions, we came to a surprisingly simple solution. Enhancing the definition of CSP, we provide a natural extension of the constraint satisfaction

paradigm, offering an elegant and efficient solution to the above issues. We call this new, non-standard class of CSPs, *composite constraint satisfaction problems*.

As mentioned in Chapter 3, a constraint satisfaction problem \mathcal{P} is completely specified by the triplet $\langle \mathcal{V}, \mathcal{D}_{\mathcal{V}}, \mathcal{C}_{\mathcal{V}} \rangle$. A composite CSP is defined in a similar manner, with the following important differences.

5.1.1 Composite Values

The values a variable can take are not restricted to be atomic values, as in the classic case. Instead, a value can be an entire problem $\mathcal{P}' = \langle \mathcal{V}', \mathcal{D}_{\mathcal{V}'}, \mathcal{C}_{\mathcal{V}'} \rangle$. After instantiating a variable $U \in \mathcal{V}$ to a composite value \mathcal{P}' , the problem \mathcal{P} is modified dynamically and becomes $\mathcal{P} = \langle \mathcal{V} \cup \mathcal{V}', \mathcal{D}_{\mathcal{V}} \cup \mathcal{D}_{\mathcal{V}'}, \mathcal{C}_{\mathcal{V}} \cup \mathcal{C}_{\mathcal{V}'} \rangle$.

The impact of this change on the existing CSP algorithms is minimal. As the result of instantiating a variable V to value val , \mathcal{V} , \mathcal{D} and \mathcal{C} change dynamically, according to the definition. In case this instantiation does not lead to a solution, these changes will be undone as a result of backtracking. Since no other modifications are necessary, all existing filtering and search algorithms can thus be easily adapted and used, without the need for introducing specialized, conceptually different methods.

Examples of composite values, extracted from the representation of the problem introduced in Chapter 1, include *MiniServer* and *SuperServer* for variable *type* in the model of SERVER, *64Mb*, *128Mb*, *256Mb*, *512Mb* and *1024Mb* for variable *type* in the model of MEMORY, etc.

5.1.2 Port Variables

We introduce a special type of variable, called *port variable*. The domain associated with a port variable is a finite set of composite values. What is special about port variables is that their domains have an *unspecified, though finite, cardinality*.

All the values in the domain of a port are instances of the same component type, *i.e.* they all have exactly the same structure. Remember, composite values are actually composite CSPs, so every member of the domain will have the same number and type of variables, interconnected by the same number and type of constraints. This common structure is captured by a generic value, called *wildcard*. The wildcard anticipates all the possible members of the domain. Acting like a generator, new members can be created on demand by duplicating it. Initially, the domain of the port variable contains only the generic value¹.

The value a port variable can take is a subset of its domain. Despite having identical internal structure, the members of the domain are distinct. Consider a port variable V with domain $D_V = \{ p_1, \dots, p_n, wildcard_V \}$. Each composite CSP $p_i = \langle \mathcal{V}_i, \mathcal{D}_{\mathcal{V}_i}, \mathcal{C}_{\mathcal{V}_i} \rangle$, $p_i \in D_V$, has its own identity, according to the domains of variables in \mathcal{V}_i , which can be restricted by constraints with variables outside \mathcal{V}_i .

A large variety of constraints can be posted on port variables. We divide them in three main categories.

- A lower and upper bound can be imposed on the number of domain elements which can be assigned to a port, *i.e.* the cardinality of the port. Constraints restricting the

¹ This representation is, at some extent, the result of work done together with Daniel Mailharro from ILOG S.A., as part of a joint research effort. A similar description is presented in (Mailharro 1998)

cardinality of a port are called *cardinality constraints*.

- Constraints in the second category refer to individual elements of the domain and act like filters. We call these *filtering constraints*.

Consider again a port variable V with domain $D_V = \{ p_1, \dots, p_n, wildcard_V \}$, where each $p_i \in D_V$ is a composite CSP, $p_i = \langle \mathcal{V}_i, \mathcal{D}_{\mathcal{V}_i}, \mathcal{C}_{\mathcal{V}_i} \rangle$. Filtering constraints are defined between variables of $wildcard_V$ and variables in $\mathcal{V} - \bigcup \mathcal{V}_i, 1 \leq i \leq n$. As mentioned before, all members of the domain have identical structure, in particular all sets $\mathcal{V}_i = \{ V_{i_1}, \dots, V_{i_m} \}$ have the same number and types of variables. This allows us, using the constraints on $wildcard_V$ as templates, to generate and post a similar set of constraints on individual elements of the domain, as they are added to the current value of the port.

- The last category consists of constraints that refer to the current value of the port as a whole. Using again the fact that the elements in the domain of a port variable have identical structure, we define *cumulative expressions* on the current value of a port. A cumulative expression can be any monotonic function applied to the same (in terms of the common structure) numeric variable of each element in the current value. Examples include *sum*, *product*, *Min*, *Max*, etc.. Cumulative expressions participate in arithmetic constraints, which restrict the values expressions can take, and provide a mechanism for expressing producer-consumer relationships.

5.1.3 Related Work

For more than a decade now, various extensions to the classic CSP paradigm have been proposed, in an attempt to increase its representational power, and thus make it able to address issues raised by practical application. Some of these extensions are particularly

relevant to our work and the most important ones are presented in this section.

Hierarchical Domain CSPs

As we pointed out earlier, in many applications, in general, and in configuration, in particular, objects naturally cluster into sets with common properties and relations, which can be organized in a specialization/generalization (IS-A) hierarchy. This type of information can be represented as a CSP by organizing hierarchically the domains of the variables. By doing so, we obtain a *hierarchical domain CSP*. The first hierarchical arc-consistency, HAC, based on AC3, is presented in (Mackworth, Mulder, & Havens 1985). An improved algorithm, HAC6, based on AC6, is presented in (Kokeny 1994). The idea behind HAC6 is the following. For every variable, the domain of values contains all the nodes (internal, as well as leaves) in the corresponding hierarchy. A breadth-first linearization of the directed-acyclic graphs representing the hierarchical domains is used to impose a partial order on these flattened domains. Using the relative position of domain values in the partial order, inferences can be made about their viability without having to perform actual constraint checks. This can lead to significant performance increase.

Hierarchical domain CSPs are clearly a special case of composite CSP. We represent internal nodes in the hierarchy as composite values. There are also additional advantages associated with our representation. Since only direct descendants on one level in the hierarchy are required to be available at any time, domains of the variables in the composite CSP are smaller. And as a bonus, any regular consistency algorithm, including, but not limited to, arc consistency, can be used without modifications.

Dynamic Constraint Satisfaction Problems

According to the CSP definition, the set of variables and the set of constraints have to be exactly specified, as part of the problem. On the other hand, in many reasoning tasks, especially in synthesis tasks, the set of variables that participate in a solution cannot be determined *a priori*. To cope with this problem, (Mittal & Falkenhainer 1990) introduce the notion of *dynamic constraint satisfaction problem (DCSP)*. The main idea is that only a subset of all the variables need to be instantiated and, thus, part of a solution. Each variable can be in one of two states: *active* or *inactive*. Constraints are also divided into two specialized types: *activity* constraints, that specify conditions on the values of already active variables, under which new variables become active, and *compatibility* constraints, which specify relations on consistent values for variables. Inferences can now be made about the status of a variable, based on the activity constraints, avoiding irrelevant work during search.

A DCSP is described by the set of all variables that may potentially become active, a nonempty initial set of active variables, the set of domains and the two sets of activity and compatibility constraints. The search process starts with the initial set of active variables. Additional variables are introduced as the result of satisfying activity constraints. The problem is thus changing dynamically as the search progresses. The set of active variables induces the set of “active” constraints, in the sense that a constraint which does not have all its variables active is by definition trivially satisfied, and thus, becomes “inactive”. A formal, mathematically well-founded treatment of the conditional existence of variables is presented in (Bowen & Bahler 1991) where the constraint network is viewed as a set of sentences in first-order free logic.

A first limitation of dynamic CSPs is the requirement that the set of all variables in

the problem must be known from the beginning. As mentioned earlier, this is not only inefficient, but often even impossible to achieve. Another major disadvantage, especially when modeling very complex systems, is the lack of guidance as to the appropriate content and organization of the domain knowledge. As shown in (Sabin & Freuder 1998), even simple dynamic CSPs can contain inconsistencies which impact not only search performance, but can also produce erroneous results.

Composite CSP overcomes both limitations. The effect of instantiating a variable V with a composite value $\mathcal{P} = \langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle$ is that all variables in \mathcal{V} are added to the current problem, thus having the same effect as a set of activity constraints with condition $V = \mathcal{P}$ and list of variables to be activated covering \mathcal{V} . However, as we will see in the next section, the instantiation mechanism for port variables supports dynamic creation of variables, on request, thus eliminating the need for knowing in advance the set of all variables.

A second advantage of using composite CSP is the support it offers implicitly in organizing and maintaining the domain knowledge based on abstraction/generalization and aggregation.

Meta Constraint Satisfaction Problems

In the CSP context, when we talk about structure in general, we can position ourselves on one of three levels: *macro*, *micro* and *meta* (Freuder 1992). At the meta level, we decompose the problem into subproblems, and view this decomposition as a *meta-problem*.

Each *metavariable* of the meta-problem is a subproblem of the original problem. The domain of a metavariable is the set of solutions to the subproblem. Each subproblem includes a subset of the variables in the original problem, together with the values for these variables and the constraints relating variables within this subset. Meta-variables can

overlap by sharing common variables. A *meta-constraint* between two meta-variables must enforce all the original constraints, involving variables from the corresponding subproblems. Furthermore, if the same variable appears in both subproblems, the meta-constraint must ensure that this variable receives the same value in the solution chosen as *meta-value* for each of them.

The goal of this decomposition is to deal with the complexity of solving a problem by solving an equivalent problem, represented at a different level of abstraction. This is desirable because either the meta-problem or the metavariable subproblems may present a special structure, which can be solved more efficiently.

Composite CSPs support a somewhat opposite approach. The initial problem is described at an abstract level. During search only relevant subproblems are expanded and added to the original problem, thus keeping the complexity to a minimum.

5.2 Component Type Models as Composite CSPs

Each component type is modeled in our framework as a composite CSP. The translation process is almost a one-to-one mapping. Properties are represented in the model as variables. Restrictions imposed by the design are expressed as constraints. The internal structure of the component type is captured by the structure of composite values. The generalization relationship between component types is also expressed through composite values. Associations between components are described using port variables. Producer-consumer relationships are expressed as arithmetic constraints involving cumulative expressions on port variables (resource constraints). Finally, cardinality and resource constraints control the dynamic evolution of the model.

Before moving further to a more detailed presentation, we introduce some notations.

Models are described using a pseudo-language providing some object-oriented constructs. The types of variables can be deduced from the description of their domain. Ranges of numeric values are enclosed in square brackets. Character '*' as upper bound stands for infinity. As mentioned in the definition of a port variable, all the elements in the domain have identical structure. In other words, they are instances of the same type. The notation for a port P of type \mathcal{T} and cardinality at least m and at most M is $P<\mathcal{T}>[m..M]$. The expression $|P|$ represents the cardinality of the current value of the port. To denote attribute a of an object x we use $x.a$. Similarly, to refer to attribute a of any member of the current value of the port, we use the notation $P.a$.

As an example, the following is the model for component type SERVER, from the problem described in Chapter 1.

```

type SERVER {

  // variables

  type: {MINISERVER, SUPERSERVER}

  bay-count: [0..4]

  CPU-count: [1..4]

  SIMM-count: {2, 4, 6, 8}

  bus-count: [2..4]

  memory-size: [128..8096]

  // ports

  CPU-slots<CPU>[1..4]

  SIMM-slots<MEMORY>[2..8]

  Bays<MEDIA>[0..4]

  Software<SOFTWARE>[2..*]

```

```

    Bus-slots<BUS>[2..4]

// constraints

C1bay-count,Bays: bay-count = |Bays|

C2CPU-count,CPU-slots: CPU-count = |CPU-slots|

C3SIMM-count,SIMM-slots: SIMM-count = |SIMM-slots|

C4Bus-count,Bus-slots: Bus-count = |Bus-slots|

C5memory-size,SIMM-slots: memory-size =  $\sum$  SIMM-slots.size

C6memory-size,Software: memory-size  $\geq$  Software.memory
}

type MINISERVER {

// constraints

C7CPU-count: CPU-count  $\leq$  2

C8memory-size: memory-size  $\leq$  1024

}

type SUPERSERVER {

// constituent parts

graphics: GRAPHICS-PROCESSOR

// constraints

C9CPU-count: CPU-count  $\geq$  2

C10memory-size: memory-size  $\geq$  512

}

```

As mentioned previously, none of the existing constraint-based approaches to configura-

tion addresses the issues raised by configuration problems in an integrated manner. In the following sections we will show how this is accomplished in our framework in a natural way.

5.2.1 Capturing Structural Information

A composite value is an instance of some component type. A particular component type describes the composition of that component (aggregation). Whenever a new composite value is created, as the result of either a request for generating and adding a new value to the domain of a port, or an instantiation action, a new component instance is created. Accordingly, all the appropriate variables and constraints, as specified by the type description, get created and added to the existing composite CSP (*i.e.* the existing model). For example, the constraint graph corresponding to a composite value of type SERVER is presented in Figure 5.2.1.

5.2.2 Capturing Hierarchical Information

According to the problem description, there are two types of server available: miniserver and superserver. Both types have a similar structure and set of features. However, the number of CPUs and memory size are dictated by the type of server. We capture this information by factoring the common part into an abstract component type SERVER, and adding two concrete types, MINISERVER and SUPERSERVER, which specialize the abstract type and extend it with specific properties. We express this specialization relationship by adding the composite variable *type* to the model of SERVER, with domain $D_{server} = \{miniserver, superserver\}$. The instantiation mechanism will insure that this variable will be eventually assigned one of the two possible values, thus refining the server type. As a result, the appropriate set of variables and/or constraints, as specified by the model of the

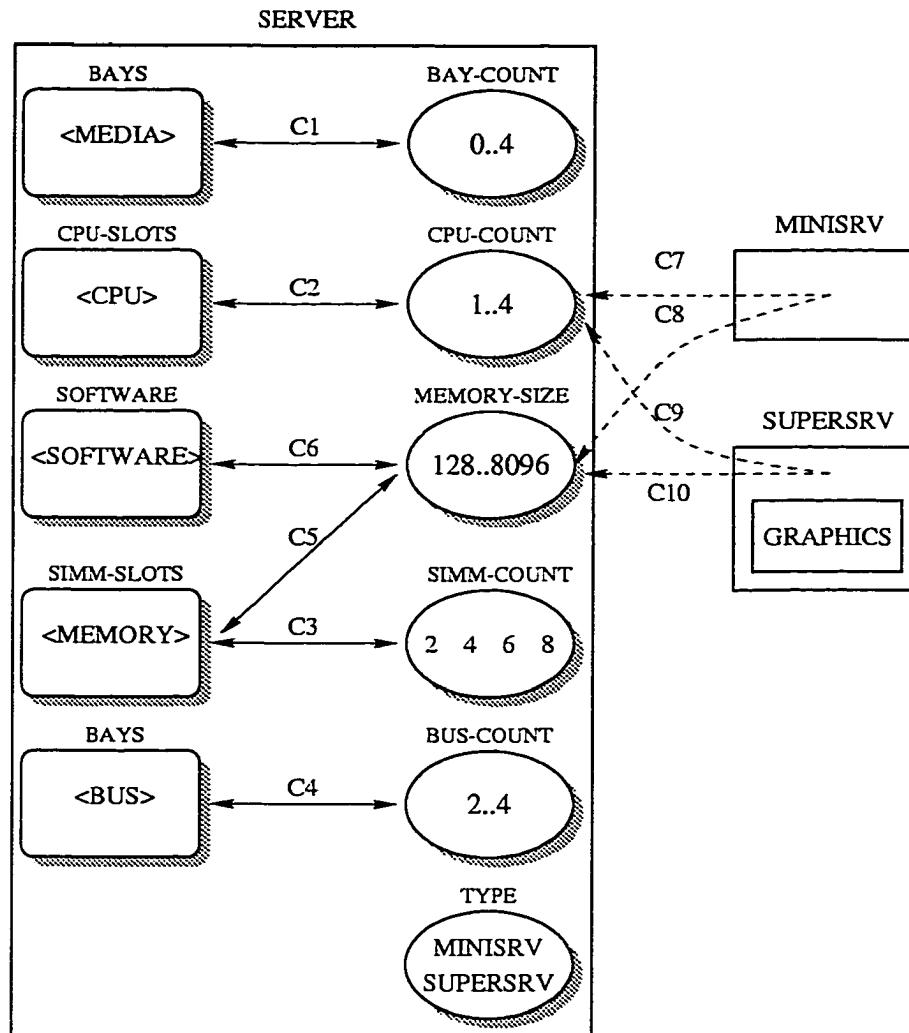


Figure 5-1: Composite CSP representation for SERVER

value that was selected for instantiation, will be created and added to the model. The new model becomes the one presented in Figure 5.2.2.

5.2.3 Capturing Relationships

We have already presented two types of relationship which can be expressed by a composite CSP: abstraction and aggregation. Other types of relationships, describing associations between objects which otherwise can exist independently, are expressed through the use of

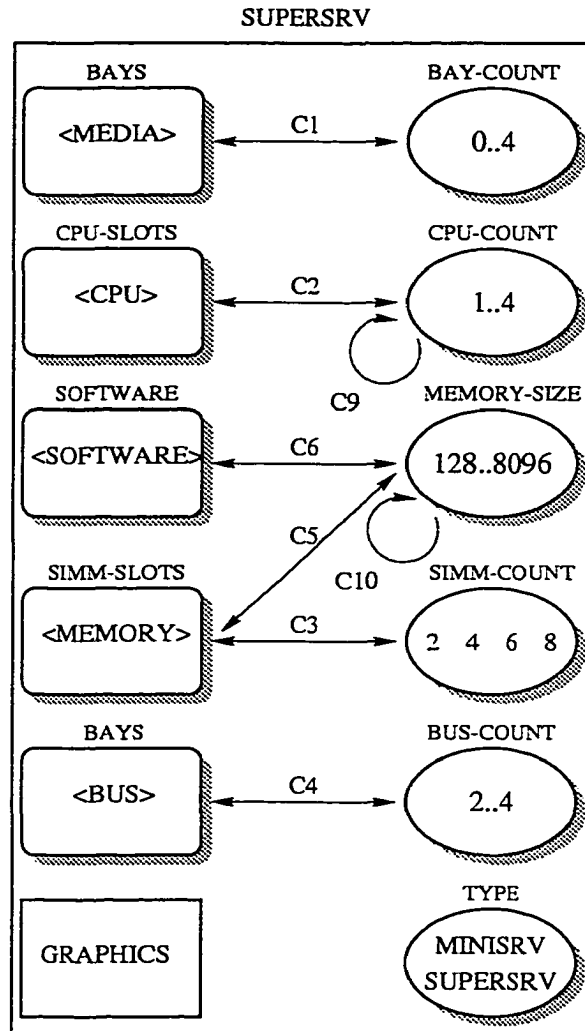


Figure 5-2: Composite CSP representation for SUPERSERVER

port variables. For example the relationship between instances of SOFTWARE and SERVER. A server is not composed of software, among other things. Instead, the software is in the relation “runs-on” with a server. This is represented in our model through the port variable *Software*. The set of SOFTWARE instances assigned to the value of this port by a particular solution represents the set of programs that are in the relation “runs-on” with this server machine.

Another important set of relationships are producer-consumer relationships which gov-

ern the use of resources. For example, we express the fact that the amount of memory available on a server is equal to the total amount of memory provided by the memory chips mounted on the machine through the constraint $memory-size = \sum SIMM-slots.size$. Similarly, $memory-size \geq Software.memory$ rejects all software packages which require more memory than the maximum amount which is available on the machine.

5.3 Chapter Conclusions

In this chapter we have introduced Composite Constraint Satisfaction Problems, a new, nonstandard class of problems which extends the classic Constraint Satisfaction paradigm. The new things composite CSPs bring are composite, structured values, and port variables, with domains which can be extended dynamically. Two standard types of relationships, generalization and aggregation are captured directly by the notion of composite value, while various association relations among application objects are expressed through port variables. The dynamic aspect of the configuration process is handled through the set of constraints posted on port variables and the instantiation mechanism. Because it is based on a declarative paradigm, this knowledge representation mechanism provides complete separation between domain knowledge and control strategy, which makes both knowledge specification and knowledge maintenance much easier.

CHAPTER 6

TAKING ADVANTAGE OF PROBLEM STRUCTURE

Maintaining arc consistency(MAC). Why it works. The cycle-cutset method. How to improve MAC. Instantiate less. Propagate less. MAC extended (MACE).

6.1 Introduction

As we briefly showed in Chapter 3, in order to improve the efficiency of CSP algorithms, very often search is interleaved with consistency inference (constraint propagation) which is used to prune values during search. The basic pruning technique involves establishing or restoring some form of *arc consistency*, pruning values that become inconsistent after making search choices. Recent research on finite domain CSPs suggests that *Maintaining Arc Consistency (MAC)* (Sabin & Freuder 1994) is the most efficient general CSP algorithm (Grant & B.M. 1995) (Bessiere & Regin 1996). Using implementations based on AC-7 or AC-Inference (Christian, Freuder, & Regin 1995) (Regin 1995), which have a very good space and worst case running time complexity, and a new dynamic variable ordering heuristic (Bessiere & Regin 1996), MAC can solve problems which are both large and hard.

The enhanced look ahead allows MAC to make a much more informed choice in selecting the next variable and/or value, thus avoiding costly backtracks later on during search.

However, additional search savings will be offset by the additional costs if proper care is not taken during the implementation. There are two sources of overhead in implementing MAC:

- the cost of restoring arc consistency after a decision has been made during search (either to instantiate a variable or to delete a value)
- the cost of restoring the problem to the previous state in case the current instantiation leads to failure.

Specifically, most of the effort is spent in deleting inconsistent values, during the propagation phase, and adding them back to the domains, after backtracking. Accordingly, we propose two ways to lower these costs:

- *Instantiate less.* In the context of maintaining full arc consistency, the search algorithm can focus on instantiating only a subset of the original set of variables, yielding a partial solution which can be extended, in a backtrack free manner, to a complete solution. Depending on the problem's density, the size of this subset, and thus the effort to find a solution, can be quite small.
- *Propagate less.* Instead of maintaining the constraint network in an arc consistent state, we propose to maintain an equivalent state, less expensive to achieve because it requires less propagation, which is:
 - only partially arc consistent, but
 - guaranteed to extend to a fully arc consistent state.

6.2 Example

(Grant & B.M. 1995) presents a major study of the performance of MAC ¹, over a large range of problem sizes and topologies. The results demonstrate that the size of the search trees is much smaller for MAC than for FC and that MAC produces backtrack-free searches over a considerably larger number of problems across the entire range of density/tightness values commonly used to characterize random problem space.

If we expected MAC to do better than FC, due to its enhanced look-ahead capabilities, our own experiments showed an unexpected result: that on problems with low and medium constraint densities (up to 0.5 – 0.6) a static variable ordering heuristic, instantiating variables in decreasing order of their degree in the constraint graph, is in general more effective in the context of MAC than the popular dynamic variable ordering based on minimal domain size. In the majority of cases the gain in efficiency was due to a lower number of backtracks, very large regions of the search space being backtrack-free.

Trying to understand how a static variable ordering can be better than a dynamic one is what led us to the ideas we will present next via an example. We restrict our attention here to *binary* CSPs, where the constraints involve two variables. One way of representing a binary CSP is in the form of a constraint graph. Nodes in the graph are the CSP variables and the constraints form the arcs.

Let us now consider the example represented by the constraint graph in Figure 6-1, and see what happens during the search for a solution. For the sake of simplicity, assume that all constraints are *not-equal* and all domains are equal to the set $\{r, g, b\}$.

¹ The authors describe a “weak” form of MAC; we believe that the results would have been even stronger if the experiments had been done with the MAC algorithm described later in this chapter.

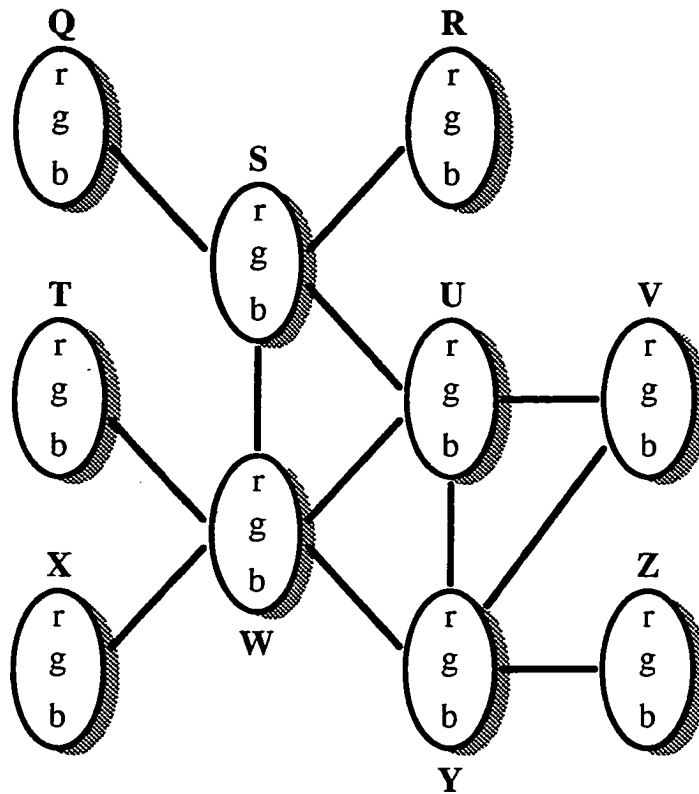


Figure 6-1: Sample constraint network

We are ready now to explain what we mean by *instantiate less*. Suppose that MAC will choose for some obscure (for now) reason, U as the first variable to be instantiated. After selecting value r , the algorithm will eliminate all the other values in the domain of U and will propagate the effects of these removals, restoring arc consistency, as shown in Figure 6-2.

At this point the reader can verify that no matter which variable is next instantiated, and no matter which value is selected for the instantiation, MAC will find a complete solution without having to backtrack. Furthermore, we claim that if we had been interested only in finding out whether the problem is satisfiable or not, the algorithm could have stopped after having successfully instantiated variable U and have returned an affirmative answer.

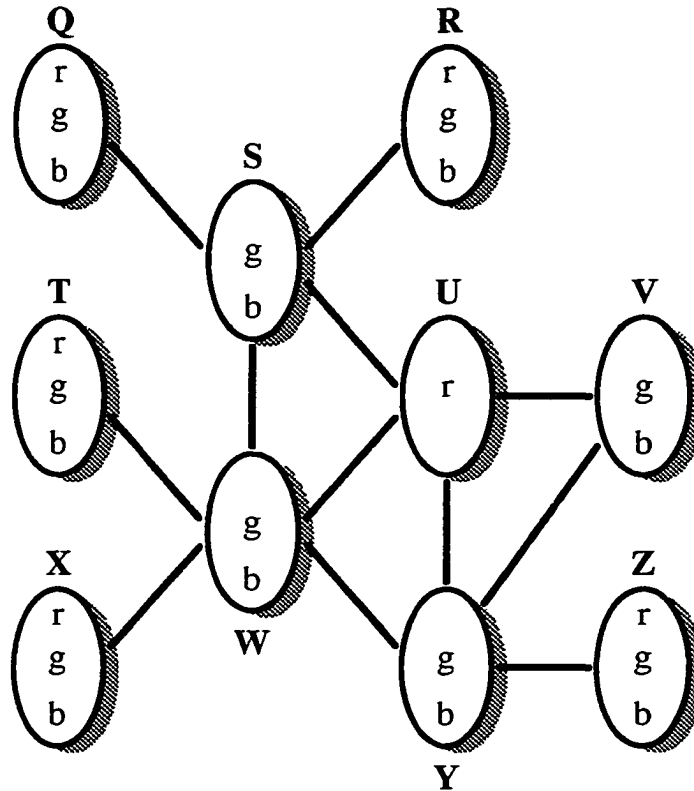


Figure 6-2: The constraint network after instantiating variable U

Why? Take a look at Figure 6-3, which presents the state of the problem after instantiating variable U .

Intuitively, since r is the only value left in the domain of U and it supports (is consistent with) all the values remaining in the domains of neighbor variables, it will itself always have a support as long as these domains are not empty. U thus becomes irrelevant for the search process trying to extend this partial solution, and we can temporarily “eliminate” from further consideration both U and all constraints involving it.

If we ignore the grayed part of the constraint net in Figure 6-3, the constraint graph becomes a tree. This, plus the fact that every value in the domain of any variable is supported by at least one value at each neighbor (the network is arc consistent), implies

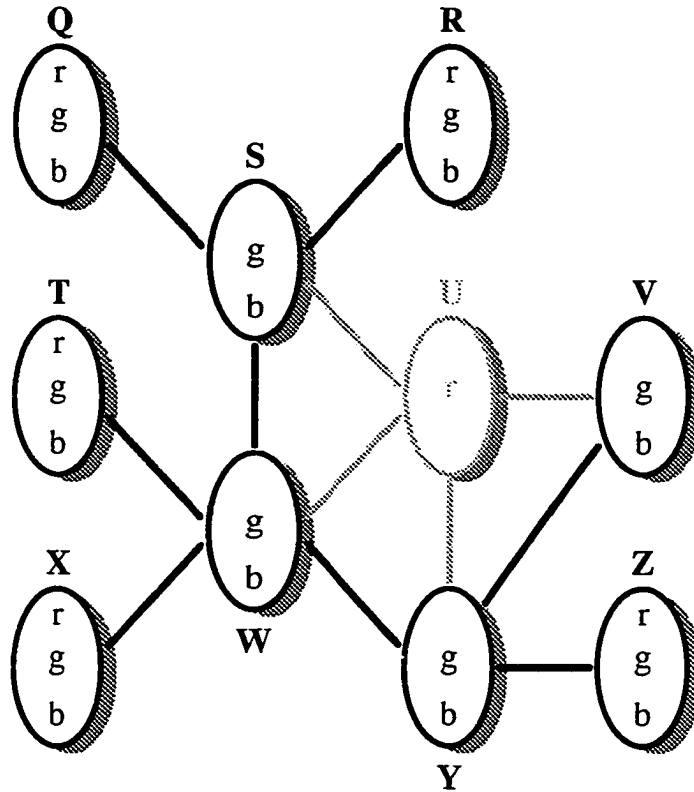


Figure 6-3: The constraint network after instantiating variable U

that the problem is globally consistent and makes it possible to find a complete solution in a backtrack-free manner, for example the one in Figure 6-4.

In fact, for this reason, MAC is able to find all the solutions involving $U = r$ without having to backtrack. But what makes U so special ? If we look again to the graph in Figure 6-3, we see that

- all the cycles in the graph have one node in common, the one corresponding to variable U , and
- by eliminating this node and all the edges connected to it we obtain an acyclic graph.

A set of nodes which “cut” all the cycles in a graph is called *cycle cutset*. In our case, the set $\mathcal{C} = \{ U \}$ represents a minimal cycle cutset for the graph in Figure 6-3. It is obvious

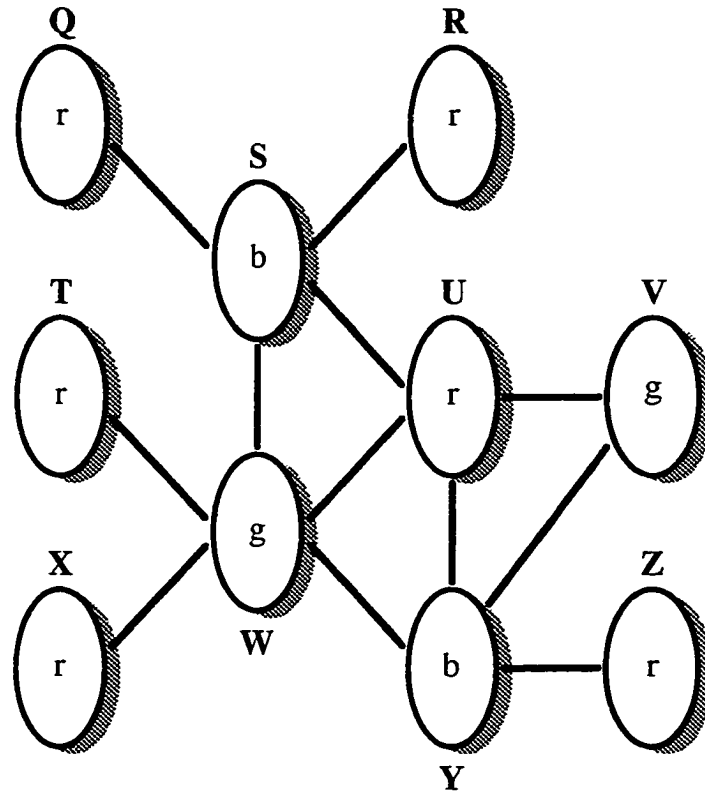


Figure 6-4: The constraint network after finding a solution

that the graph obtained from the original one by eliminating the nodes in any cycle cutset and the related edges is acyclic.

The observations made on the graph in our example are directly supported by research on discrete domain CSPs (Dechter & Pearl 1988) (Dechter 1990) (Freuder 1982), and are similar to the following two theorems presented in (Hyvonen 1992):

- (1) An acyclic constraint net is globally consistent iff it is arc consistent.
- (2) If the variables of any cutset of a constraint net S are singleton-valued, then S is globally consistent iff it is arc consistent.

If after all the variables in some cutset are instantiated the net is locally consistent, we can “eliminate” these variables and their related constraints from the problem, as shown

above. This cuts the loops and makes the constraint net acyclic. In this case, according to Theorem (1), local consistency is equivalent to global consistency. In addition, regardless of the order in which variables are instantiated, MAC can find a complete solution without any backtracking. We can now present a first modified version of MAC, in the form of the following algorithm:

1. Enforce arc consistency on the constraint network. If the domain of any variable becomes empty, return failure.
2. Identify a cycle-cutset \mathcal{C} of the constraint graph.
3. Instantiate all variables in \mathcal{C} while maintaining full arc consistency in the entire constraint network. If this is not possible, return failure.
4. Use MAC to extend the partial solution obtained in step 3 to a complete solution, in a backtrack-free manner.

So far we showed that, in order to guarantee the existence of a complete solution in the context of maintaining arc consistency, it is sufficient to obtain a partial solution, by successfully instantiating only a subset of the variables, namely the cycle-cutset of the constraint graph. A simple heuristic to find a cycle-cutset (not necessarily minimal) is to order the variables in decreasing order of their degree, which explains why this static ordering performed so well in our tests.

Let us see if we can do better by *propagating less*. As we indicated earlier, after each modification MAC tries to restore the network to an arc consistent state. We claim that it is sufficient to bring the network to a partially arc consistent state only. More exactly, we need to maintain arc consistency just in part of the constraint graph, involving only some of the variables and constraints of the original CSP.

Figure 6-5 presents the constraint graph of our example, in which variables not involved in any cycle have been grayed. Once arc consistency is established, these variables become irrelevant for the search process. If the problem is inconsistent, none of these variables can be the source of the inconsistency. If there is a partial solution instantiating any of the normal variables in Figure 6-5, we are guaranteed to be able to extend it to a complete solution in a backtrack-free manner. Therefore, they can be disconnected from the constraint network, until we decide whether it is possible to instantiate successfully the variables which are left. During search the algorithm will propagate any change, and restore consistency accordingly, only in a (potentially small) part of the network. This partially arc consistent state is equivalent with the fully arc consistent state in the sense that both lead to exactly the same set of complete solutions.

Once all the variables which are still part of the network are instantiated, it is enough to reconnect the variables previously disconnected and to enforce arc consistency (or directed arc consistency) in order to obtain global consistency and to extend the partial solution to a complete solution in a backtrack-free manner.

The two ideas, *instantiate less* and *propagate less*, can now be combined under the name of *MACE (MAC Extended)*, which instantiates only a subset of the CSP variables while maintaining only a partially arc consistent state of the constraint network. The gain in efficiency is twofold. Instantiating a smaller number of variables aims at reducing the number of backtracks (and, accordingly, the number of constraint-checks, nodes visited, values deleted, etc.). Since the values disconnected are not part of any cycle-cutset, and hence, will not be instantiated in the first phase of the algorithm, the limited propagation implied by the second idea does not influence at all the number of backtracks or nodes visited, but reduces the number of constraint checks and values deleted.

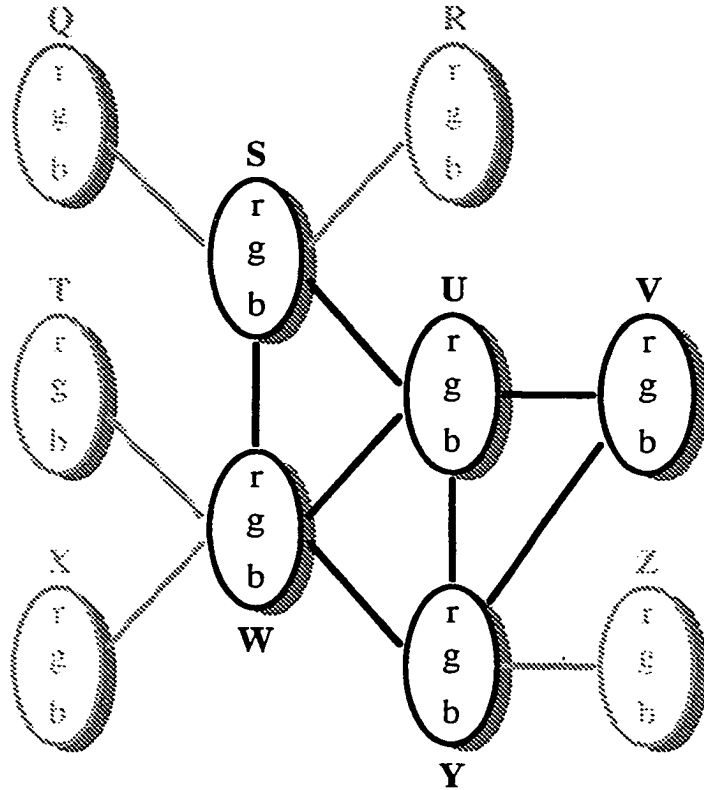


Figure 6-5: The constraint network after eliminating the cycle-free variables

6.3 Related Work

The idea of using the cycle-cutset of a constraint graph to improve the efficiency of CSP algorithms was used in (Dechter & Pearl 1988) as part of the *cycle-cutset method (CC)* for improving backtracking on discrete domain CSPs. (Hyvonen 1992) uses it for *interval CSPs*. A related idea is used in (Solotorevsky, Gudes, & Meisels 1996) for solving *distributed CSPs*.

Dechter and Pearl's cycle-cutset method can be described by the following scheme.

1. Partition the variables into two sets: a cycle-cutset of the constraint graph, \mathcal{C} , and \mathcal{T} , the complement of \mathcal{C} .
2. Find a(nother) solution to the problem with variables in \mathcal{C} only, by solving it inde-

pendently. If no solution can be found, return failure.

3. Remove from the domain of variables in \mathcal{T} all values incompatible with the values assigned to variables in \mathcal{C} and achieve directed arc consistency at variables in \mathcal{T} . If the domain of any variable becomes empty, restore all variables in \mathcal{T} to their original state and repeat step 2.
4. Use a backtrack-free search for extending the partial solution found in step 2 to a complete solution.

The major problem with the cycle-cutset method is its potential for thrashing. One type of thrashing is illustrated by the following example. Suppose the variables in \mathcal{C} are instantiated in the order X, Y, \dots . Suppose further that there is no value for some variable Z in \mathcal{T} which is consistent, according to constraint C_{XZ} , with value a for X . Whenever the solution to the cutset instantiates X to a , step 2 will fail. Since this can happen quite often, the cycle-cutset method can be very inefficient. We can eliminate this type of thrashing if we make the constraint network arc consistent before search starts, in a preprocessing phase.

A different type of thrashing, which cannot be eliminated by simply preprocessing the constraint network, is the following. Suppose that after making the network arc consistent initially, the domain of variable Z contains two values, c and d . Furthermore, value a for X supports value c on C_{XZ} , but does not support d . On the other hand, value b for Y supports d and not c on C_{YZ} . The cycle-cutset method will discover the inconsistency only while trying to instantiate Z , and this failure will be repeated for each solution of the cutset problem instantiating X to a and Y to b .

Our approach maintains arc consistency during the search (in fact, it maintains an

equivalent state, as explained above). This eliminates both sources of thrashing and leads to substantial improvements over the cycle-cutset method.

6.4 Algorithm

The goal of this section is to present the description of three algorithms: MAC, the cycle-cutset method (CC), and the new algorithm we propose, MACE. The following is a high level description of the basic MAC algorithm.

```

MAC ( in: Var ; out: Sol ) return boolean

1   consistent  $\leftarrow$  INITIALIZE( )
2   while consistent do
3       (X, valx)  $\leftarrow$  SELECT( Var, 0 )
4       if SOLVE( (X, valx), Var  $\setminus$  {X}, Sol, 1 ) then
5           return true
6       Dx  $\leftarrow$  Dx  $\setminus$  {valx}
7       consistent  $\leftarrow$  Dx  $\neq \emptyset$  and PROPAGATE( Var  $\setminus$  {X}, 1 )
      □
8   return false
      □

SOLVE( in: (X, valx), Var, Sol, level ; out: Sol ) return boolean
9   Sol  $\leftarrow$  Sol  $\cup$  {(X, valx)}
10  if level = N then
11      return true
12  for each a  $\in$  Dx, a  $\neq$  valx do

```

```

13       $D_x \leftarrow D_x \setminus \{a\}$ 
14      consistent  $\leftarrow$  PROPAGATE( Var, level )
15      while consistent do
16           $(Y, val_y) \leftarrow$  SELECT( Var, level )
17          if SOLVE(  $(Y, val_y)$ ,  $Var \setminus \{Y\}$ , Sol, level+1 )
18              return true
19           $D_y \leftarrow D_y \setminus \{val_y\}$ 
20          consistent  $\leftarrow D_y \neq \emptyset$  and PROPAGATE(  $Var \setminus \{Y\}$ , level )
21       $Sol \leftarrow Sol \setminus \{(X, val_x)\}$ 
22      RESTORE( level )
23      return false

```

□

It is worth stressing the differences between MAC and another algorithm that restores arc consistency, called *Really Full Lookahead (RFL)* (Nadel 1988). Once the constraint network is made arc consistent initially (line 1), MAC restores arc consistency after each instantiation, or forward move, (lines 12–14), as RFL does, and, in addition:

- whenever an instantiation fails, MAC removes the refuted value from the domain and restores arc consistency (lines 6–7 and 19–20);
- after each modification of the network, both after instantiation and refutation, MAC chooses a (possible new) variable, as well as a new value (lines 3 and 16).

For our experiments we implemented a slightly improved version of MAC, called MAC-7ps (Regin 1995). According to the results presented in (Christian, Freuder, & Regin

1995), (Bessiere & Regin 1996) and (Regin 1995), MAC-7ps is the best general-purpose CSP algorithm to date. It is an AC-7 based implementation of the basic MAC, with one notable improvement: special treatment of singleton variables. The idea is roughly the following. After restoring arc consistency, singleton variables can be disconnected temporarily from the network. The goal is to avoid studying the constraints connecting other variables to the singletons. A detailed description of the implementation can be found in (Regin 1995).

MACE and CC need an algorithm to find a cycle-cutset. There is no known polynomial algorithm for finding the minimum cycle-cutset. There are several heuristics which yield a good cycle-cutset at a reasonable cost. The simplest sorts first the variables in decreasing order of their degree. Then, starting with the variable with the highest degree, as long as the graph still has cycles, add the variable to the cycle-cutset and remove it, together with all the edges involving it, from the graph. Assuming that lexical ordering is used to break ties, this method yields for our example the cycle-cutset presented in Figure 6-6. Variables are added to the cutset in the order W , S and U . On a problem with n variables and e constraints, the worst case run time complexity for this heuristic is $O(ne)$.

A smaller cutset can be obtained if, before adding a variable to the cutset, we check whether it is part of any cycle or not. For example, after removing W from the graph, S is not involved in a cycle anymore, and, with the new algorithm, we find the cycle cutset in Figure 6-7. The worst case time complexity for this heuristics is $O(ne)$.

Additional work leads to an even smaller cutset. The cutset shown in Figure 6-8 is obtained by a third heuristic, which determines dynamically the number of cycles in which each variable is involved and adds to the cutset at each step the variable participating in the most cycles. The worst case time complexity of this heuristic is $O(n^2e)$.

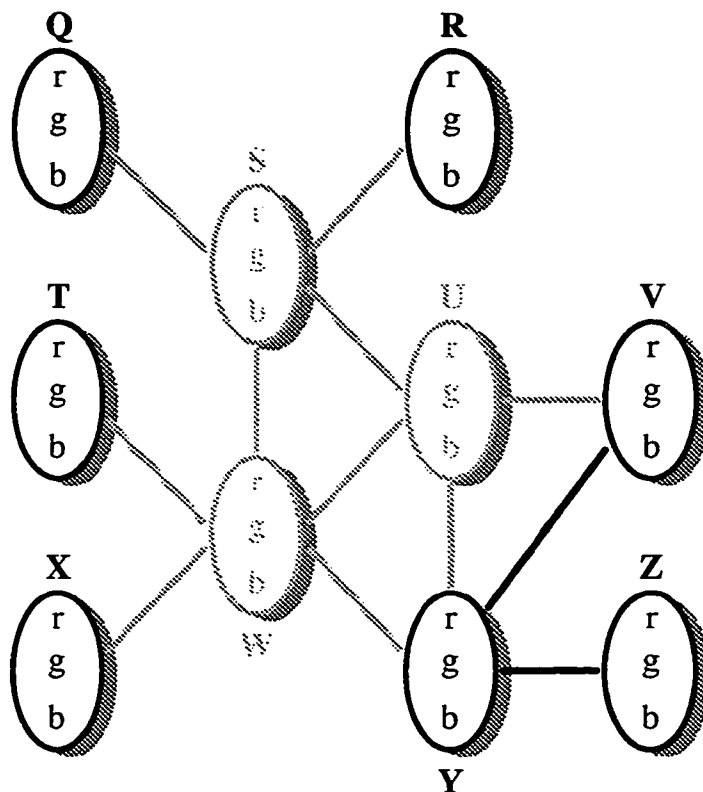


Figure 6-6: Cycle-cutsets for the example in Figure 6-1 (first heuristic)

We used exactly the same algorithm in implementing both CC and MACE. We performed the tests using the third heuristic presented above. The implementation of CC is straightforward. Since there is no requirement on the algorithm to solve the cutset subproblem, to keep the comparison with MACE as fair as possible, we used the basic MAC as our choice.

To implement MACE, we modified the algorithm presented earlier as follows.

- After enforcing arc consistency, procedure INITIALIZE (line 1) partitions the set of variables into two sets, one of which is the cycle-cutset \mathcal{C} . Disconnect from the constraint network all variables which are not involved in any cycle and add them to the set of disconnected variables, \mathcal{U} .

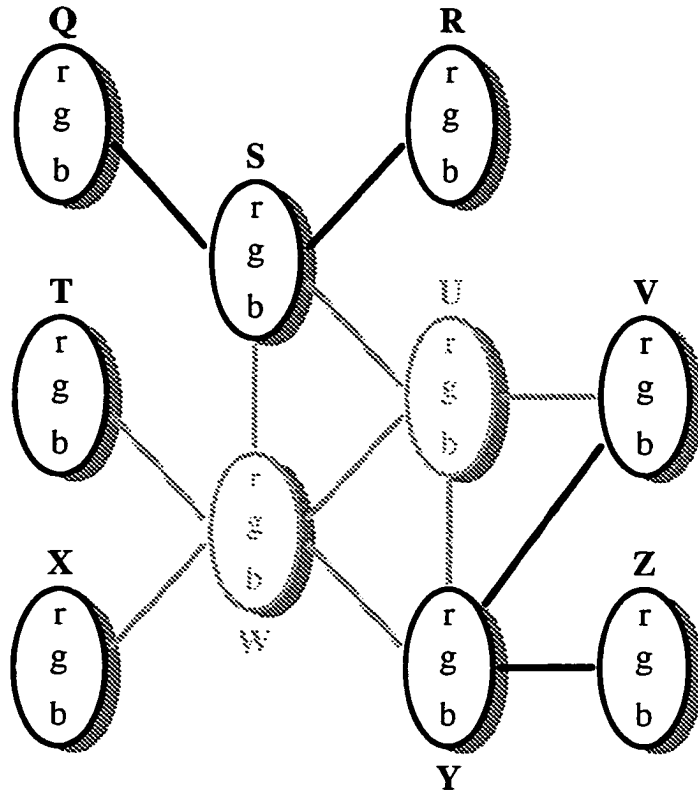


Figure 6-7: Cycle-cutsets for the example in Figure 6-1 (second heuristic)

- Restrict procedure SELECT (lines 3 and 16) to choose only from among variables in \mathcal{C} .
- Whenever a variable becomes a singleton disconnect it from the network and add it to \mathcal{U} . If this makes other variables “cycle-free”, disconnect them and add them to \mathcal{U} as well. Continue this process until no more variables can be disconnected.
- Once all variables in \mathcal{C} have been successfully instantiated, reconnect all variables in \mathcal{U} and eliminate from their domains all values incompatible with the values assigned to variables in the cutset. Enforce directed arc consistency with respect to some width-1 order on the problem containing only variables in the complement of \mathcal{C} and conduct a backtrack-free search for a complete solution.

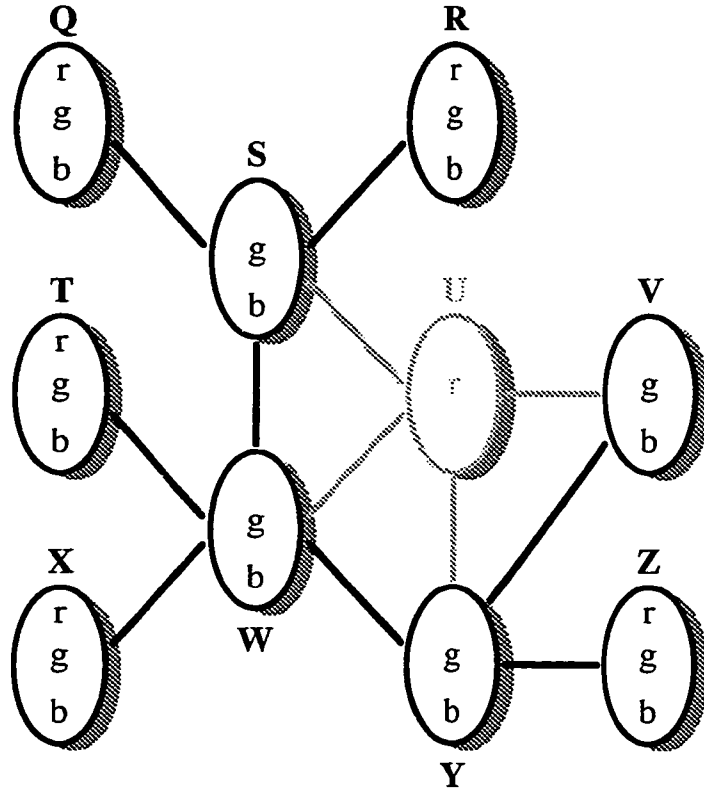


Figure 6-8: Cycle-cutsets for the example in Figure 6-1 (third heuristic)

6.5 Experimental Evaluation

We tested our approach on random binary CSPs described by the usual four parameters: number of variables, domain size, constraint density and constraint tightness. We generate only connected constraint graphs (connected components of unconnected components can be solved independently). Therefore the number of edges for a graph with n vertices is at least $n - 1$ (for a tree, density=0) and at most $n(n - 1)/2$ (for a complete graph, density=1). Constraint density is the fraction of the possible constraints beyond the minimum $n - 1$, that the problem has. Thus, for a problem with constraint density D and n variables, the exact number of constraints that the problem has is $\lfloor n - 1 + D(n - 1)(n - 2)/2 \rfloor$.

Constraint tightness is defined as the fraction of all possible pairs of values from the

domains of two variables, that are not allowed by the constraint. So, for a domain size of d and a constraint tightness of t , the exact number of pairs allowed by the constraint is $\lfloor (1 - t)d^2 \rfloor$.

The tests we conducted addressed the problem of finding a single solution to a CSP (or determining that no solution exists). We ran three sets of experiments on hard random problems, situated on the ridge of difficulty in the density/tightness space.

For the first two sets we generated problems with 20 variables and domain size of 20. The density of the constraint graph varies between 0.05 and 0.95, with a step of 0.05, while the tightness varies between $T_{crit} - 0.08$ and $T_{crit} + 0.08$, with a step of 0.01. For each pair of values (density, tightness) we generated 10 instances of random problems, which gives us roughly a total of 3,200 problems per set.

The problems in the third set have 40 variables and domain size of 20. We expected the problems in this set to be much harder than the ones in the previous sets. Therefore the constraint density varies only between 0.05 and 0.30. The tightness varies between $T_{crit} - 0.08$ and $T_{crit} + 0.08$, with a step of 0.01. We generated again 10 instances of random problems for each (density, tightness) pair, which gives us almost 1,000 problems for this set.

We present the results of the experiments using two types of plots. One type represents, on the same graph, the performance of two algorithms in terms of constraint checks, as a function of tightness.

The second type of plots represents the ratio between the performance of two algorithms as a function of tightness, in the form of a set of points. Again, results from different sets of problems, with different densities, are plotted on the same graph. Each point on the graph represents the average over the 10 problems generated for the corresponding (density,

tightness) pair.

It is very important what measure is used to judge the performance of algorithms. The usual measure in the literature is the number of constraint checks performed by an algorithm during the search for a solution. Whenever establishing that a value a for a variable X is consistent with a value b for a variable Y , a single consistency check is counted. Constraint checks are environment independent, but are highly dependent on the efficiency of the implementation. In our case, since we use more or less the same implementation for all the algorithms, we choose this measure as being representative for the search effort.

We ran experiments comparing the performance of three algorithms: the cycle-cutset method, MAC-7ps and MACE. All algorithms used the dynamic variable ordering heuristic proposed in (Bessiere & Regin 1996), choosing variables in increasing order of the ratio between domain size and degree.

The first set of experiments compares the performance of the cycle-cutset method and MACE on the first set of test problems. Figure 6-9 shows the relative average performance of the algorithms in terms of constraint checks.

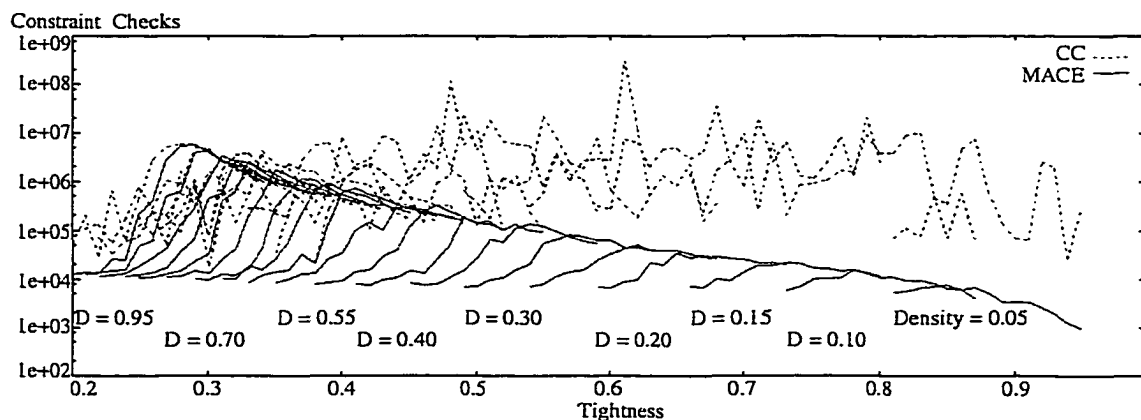


Figure 6-9: Comparison between the cycle-cutset method and MACE

As we can see, MACE outperforms substantially the cycle-cutset method on problems with densities up to 0.90–0.95, when they have approximately the same performance. The size of the cycle-cutset varies almost linearly with the density, from 3 for density 0.05 to almost 18 for density 0.95. For problems in the high density area the cutset is almost the entire set of variables (these are 20-variable problems) and therefore the behavior of the two algorithms is almost identical.

As suggested in Section 3, we added an arc consistency preprocessing phase to CC and ran this combination on the same problem sets. The results are presented in Figure 6-10. As we can see, the preprocessing improves the performance of CC only in the very sparse region, by discovering the arc inconsistent problems. On the rest of the problems the preprocessing had practically no effect.

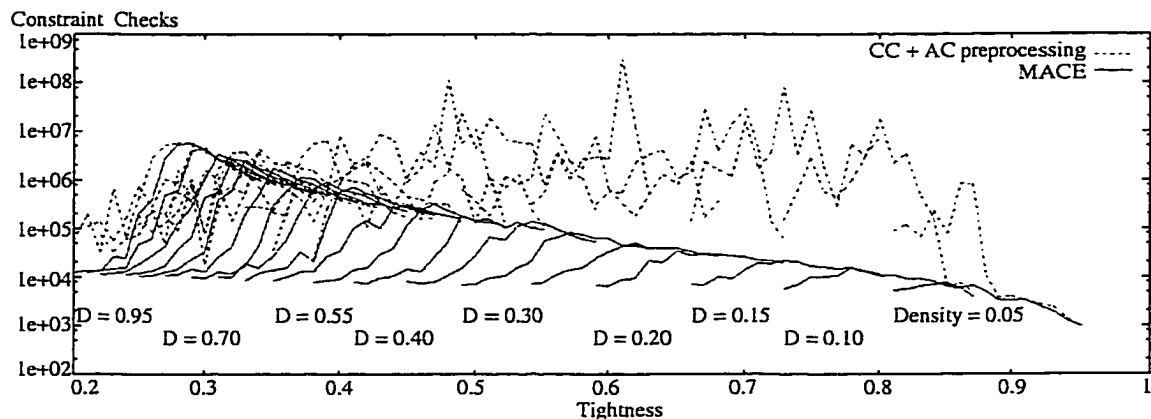


Figure 6-10: Comparison between the cycle-cutset method with arc-consistency preprocessing and MACE

The same results are presented from a different perspective in Figure 6-11, which shows the ratio between the constraint checks performed by the cycle-cutset method and MACE. The advantage of MACE is very clear.

The second set of experiments compares the performance of MAC-7ps and MACE. Figures 6-12 – 6-21 show the relative average performance of the two algorithms in terms of constraint checks on the second set of problems, with 20 variables.

As we can also see from the plot in Figure 6-22, which shows the ratio between the number of constraint checks for MAC-7ps and MACE on the same set of problems, MACE performs better than MAC-7ps. For problems with high densities (0.9 – 0.95) although MACE still dominates, MAC-7ps wins a few times. Again, the explanation consists in the size of the cycle-cutset, which increases with the density. In this particular area the sets of variables instantiated by the two algorithms become almost the same. Therefore, both algorithms exhibit similar behaviors, MACE being still slightly better than MAC-7ps on average.

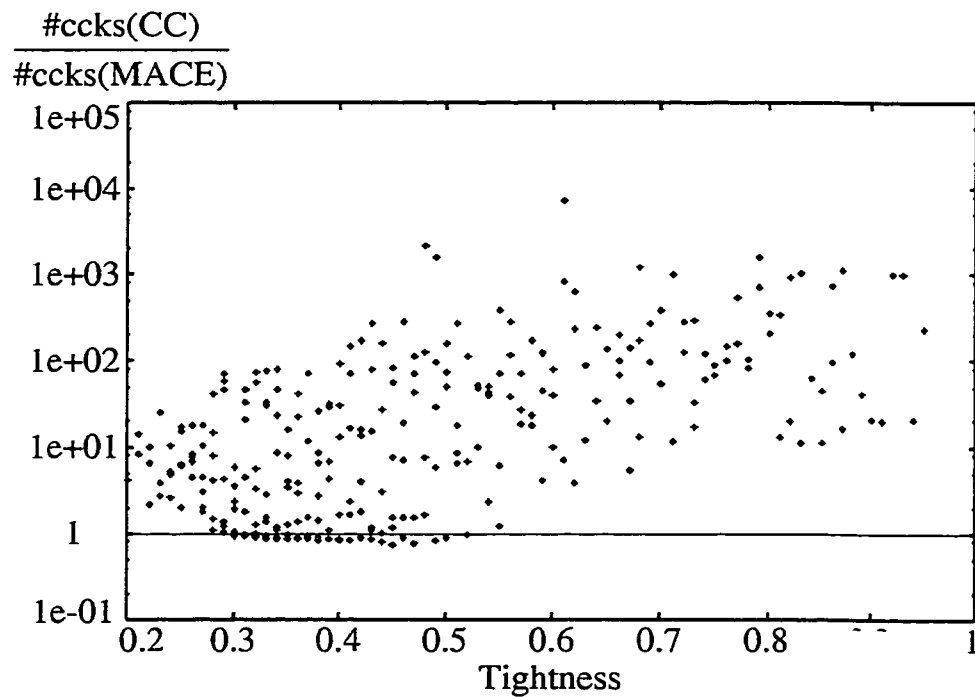


Figure 6-11: Performance ratio between the cycle-cutset and MACE

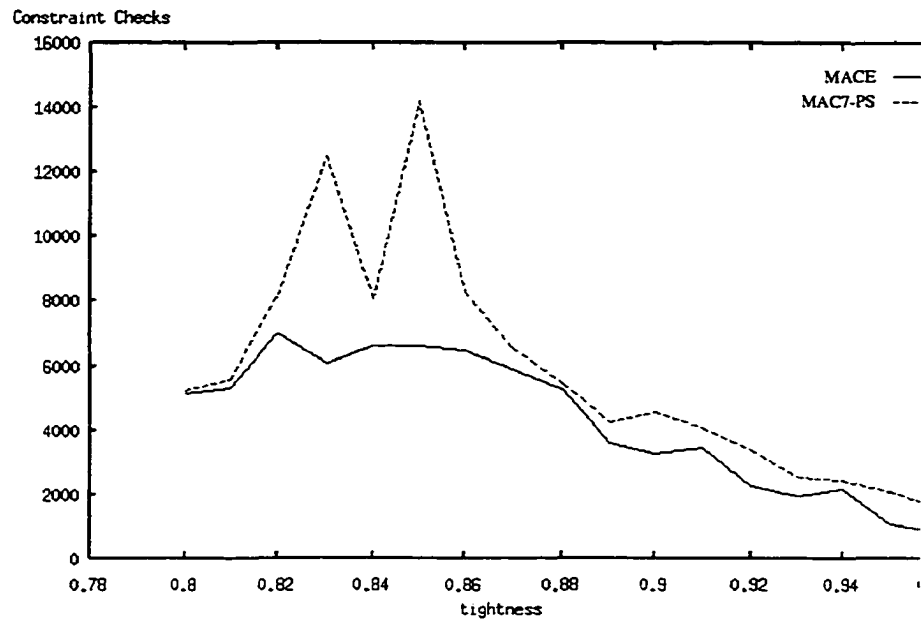


Figure 6-12: Comparison MAC-7ps to MACE on problems with 20 variables, density=0.05

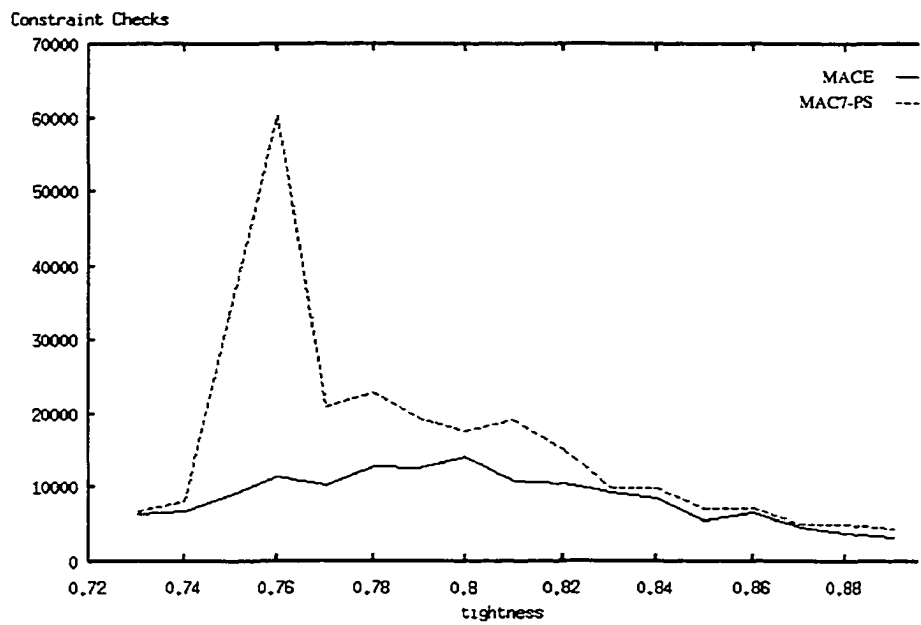


Figure 6-13: Comparison MAC-7ps to MACE on problems with 20 variables, density=0.15

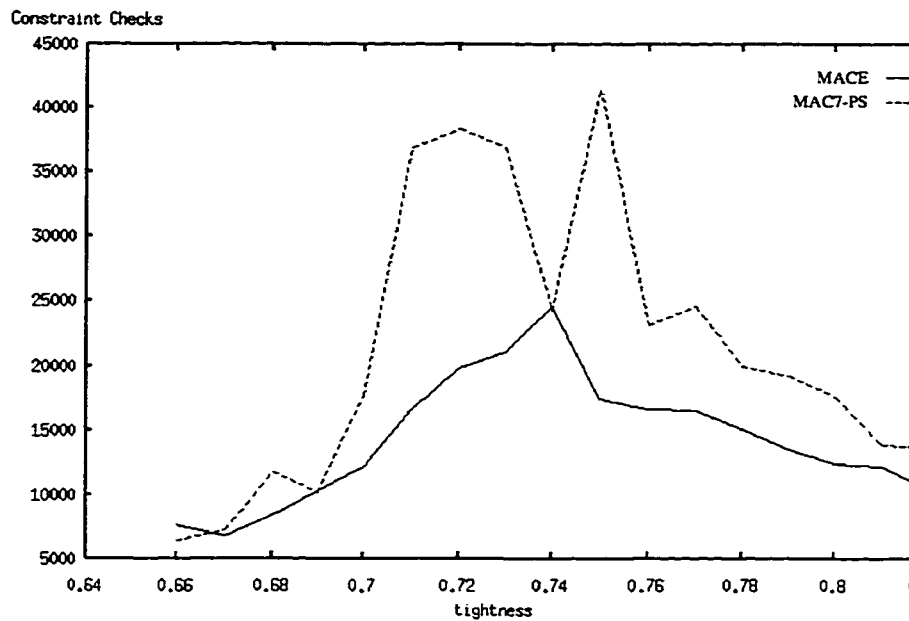


Figure 6-14: Comparison MAC-7ps to MACE on problems with 20 variables, density=0.25

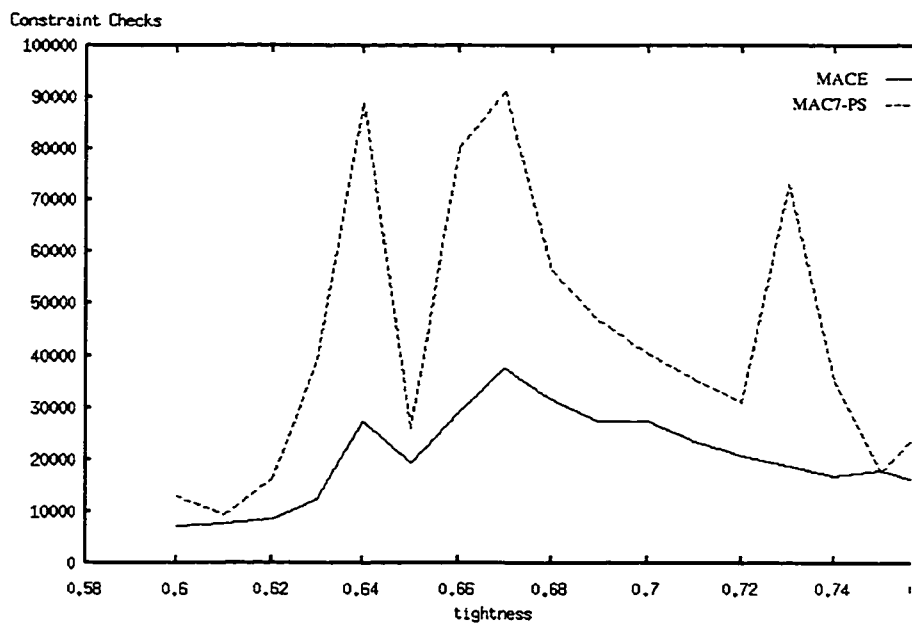


Figure 6-15: Comparison MAC-7ps to MACE on problems with 20 variables, density=0.35

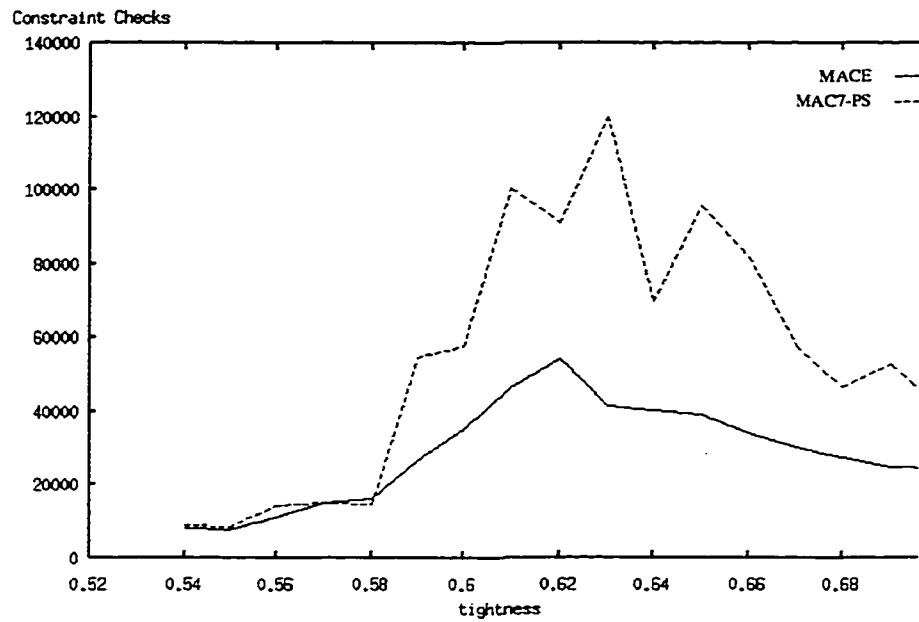


Figure 6-16: Comparison MAC-7ps to MACE on problems with 20 variables, density=0.45

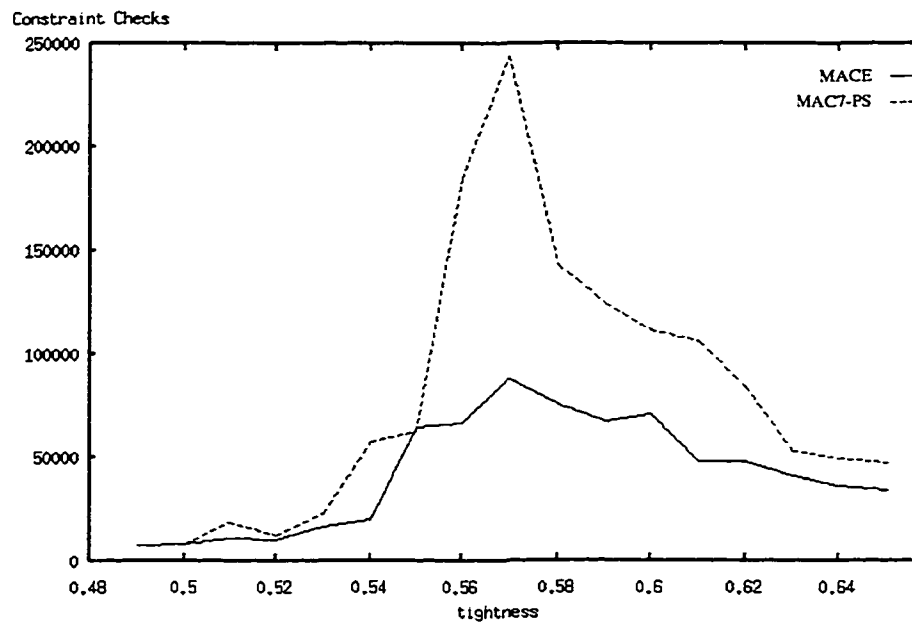


Figure 6-17: Comparison MAC-7ps to MACE on problems with 20 variables, density=0.55

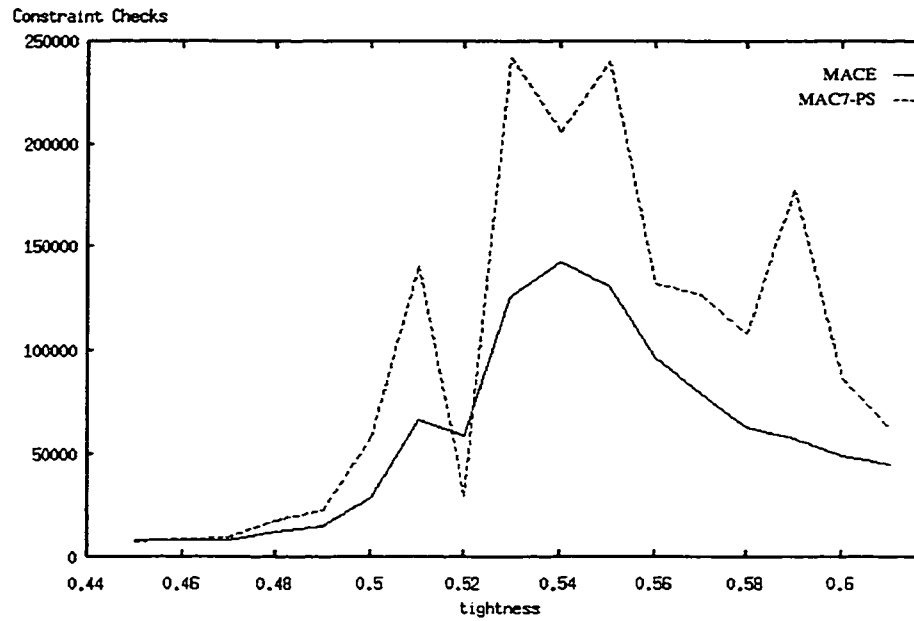


Figure 6-18: Comparison MAC-7ps to MACE on problems with 20 variables, density=0.65

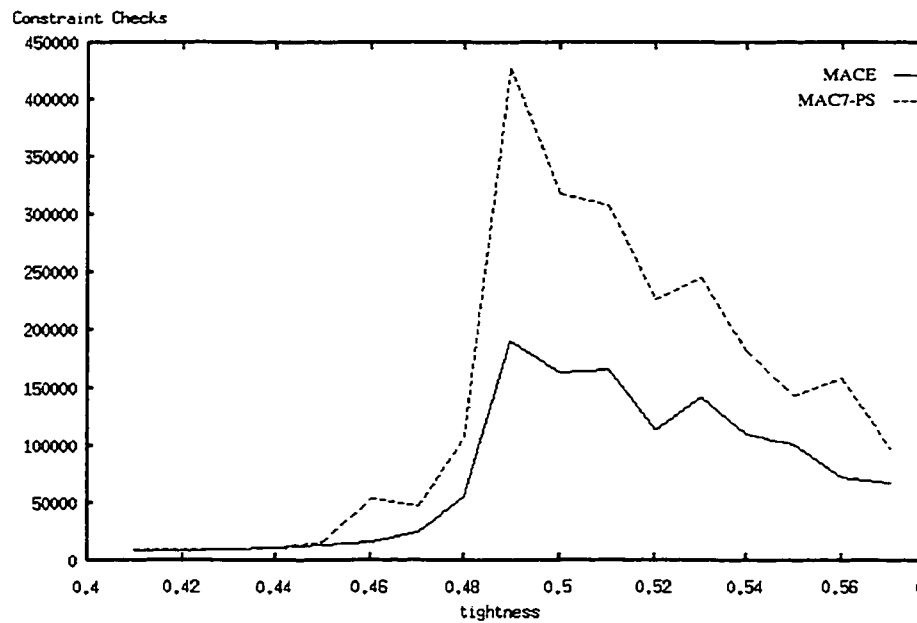


Figure 6-19: Comparison MAC-7ps to MACE on problems with 20 variables, density=0.75

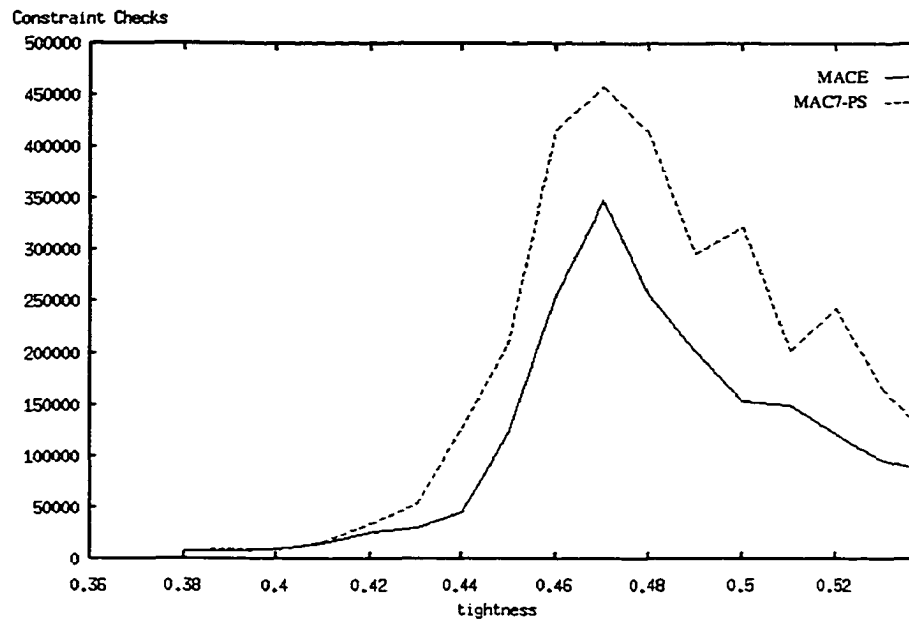


Figure 6-20: Comparison MAC-7ps to MACE on problems with 20 variables, density=0.85

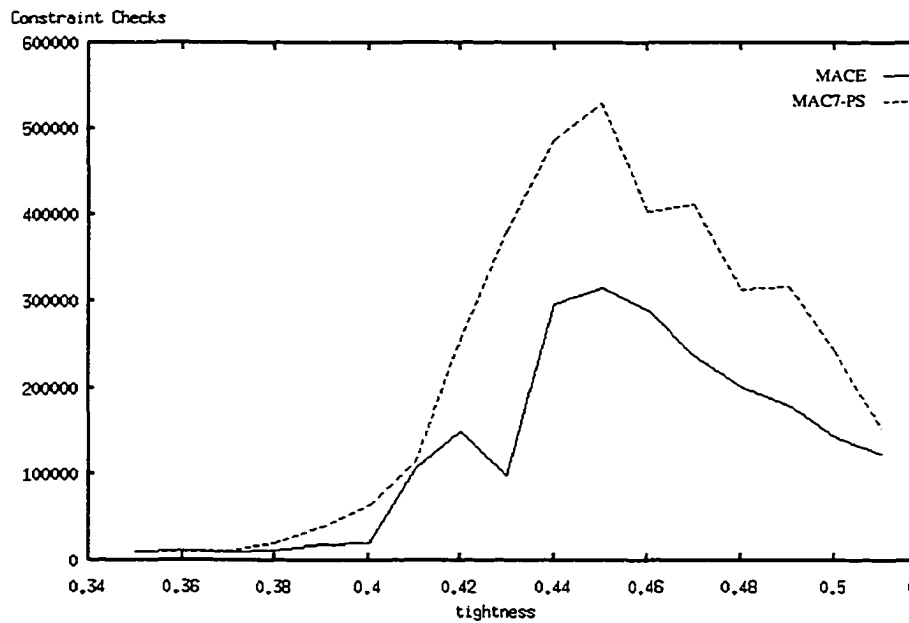


Figure 6-21: Comparison MAC-7ps to MACE on problems with 20 variables, density=0.95

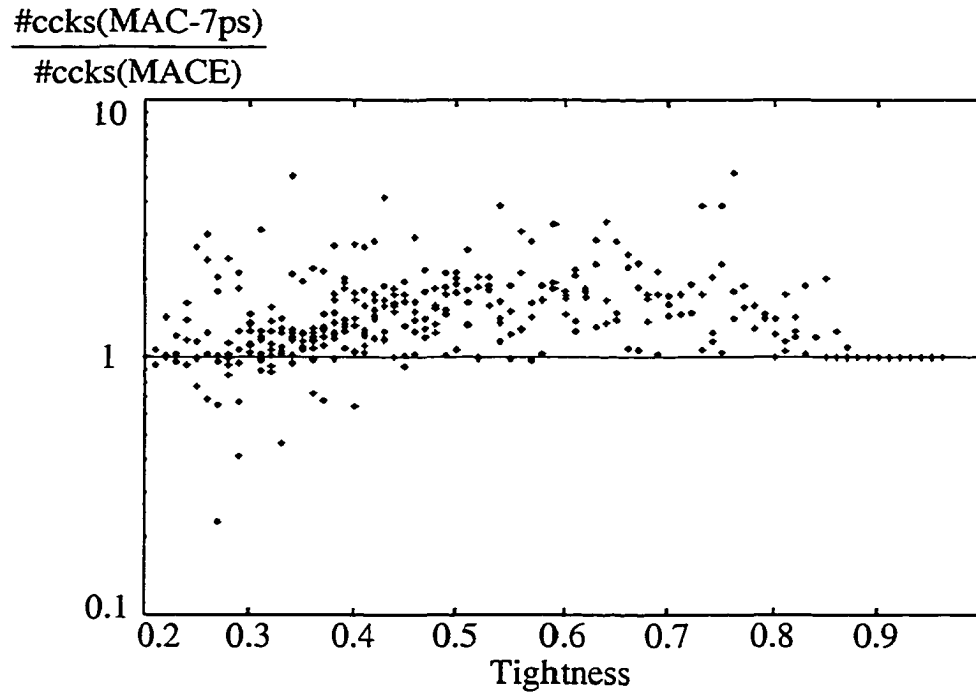


Figure 6-22: Performance ratio between MAC-7ps and MACE on problems with 20 variables

The last set of experiments studies the scalability of our approach as problem size increases. We therefore compared the performance of MAC-7ps and MACE on problems with 40 variables, using the third set of random problems. Figure 6-23 shows again the relative average performance of the two algorithms in terms of constraint checks, while Figure 6-24 presents the same data, but in the form of the ratio between the number of constraint checks for MAC-7ps and MACE. Both plots show again that MACE outperforms MAC-7ps significantly. The data also suggests that MACE scales well, the relative gain in efficiency increasing as the problems become larger.

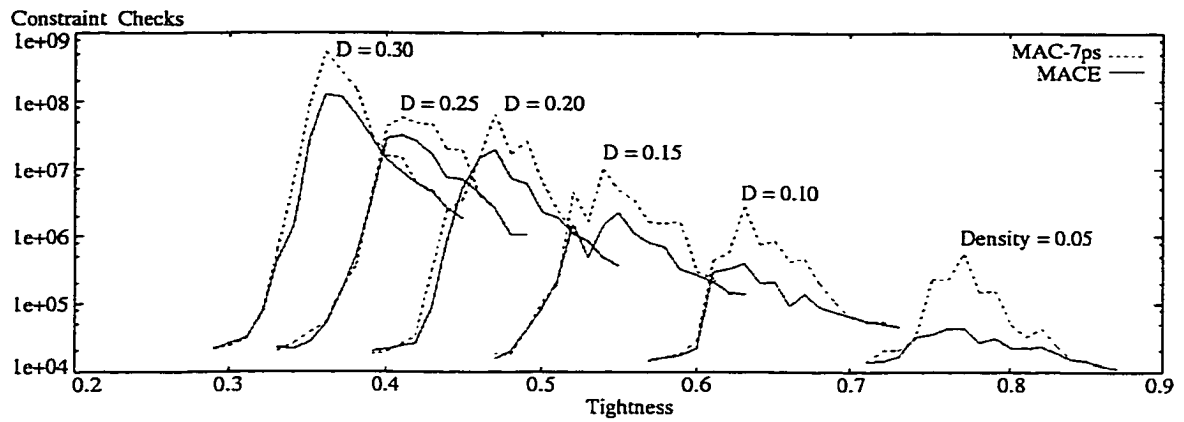


Figure 6-23: Comparison between MAC-7ps and MACE on problems with 40 variables

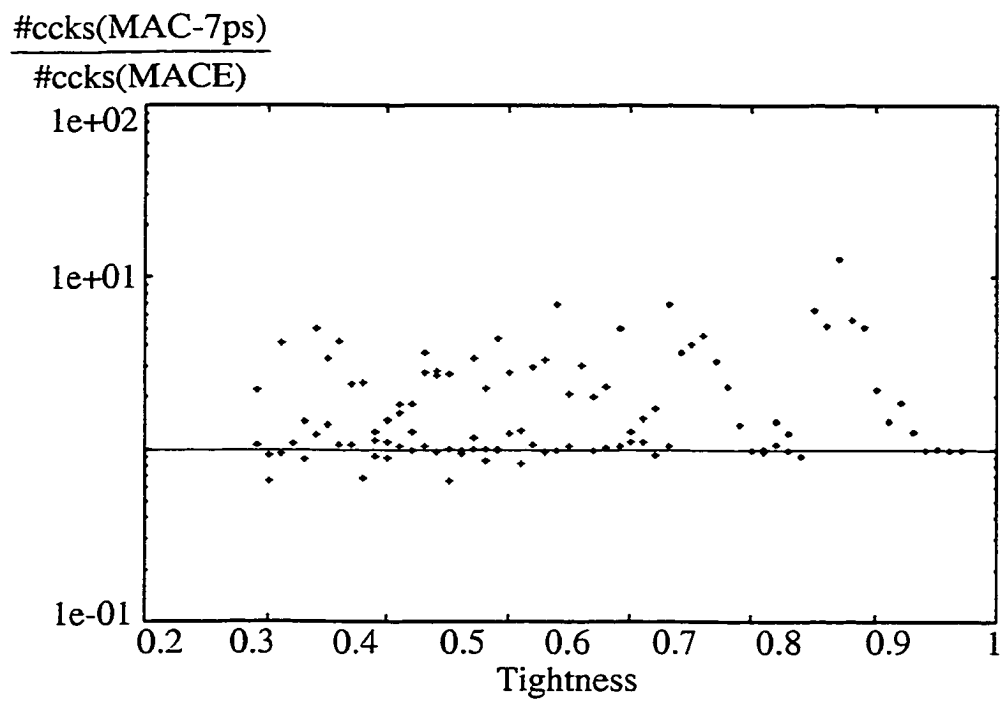


Figure 6-24: Performance ratio between MAC-7ps and MACE on problems with 40 variables

6.6 Chapter Conclusions

Recent research on finite domain constraint satisfaction problems suggest that Maintaining Arc Consistency (MAC) is the most efficient general CSP algorithm for solving large and hard problems. In the first part of this chapter we explain why maintaining full, as opposed to limited, arc consistency during search can greatly reduce the search effort. Based on this explanation, in the second part of the chapter we show how to modify MAC in order to make it even more efficient. Experimental results prove that the gain in efficiency can be quite important.

CHAPTER 7

OPTIMIZATION METHODS FOR CONSTRAINT RESOURCE PROBLEMS

Constraint resource problems in synthesis tasks. Examples. Problem representation. Algorithms. Port variables instantiation. Achieving optimality through constraint propagation. Improved lower bound computation. Eliminating partial solutions through interchangeability. Abstraction and context-dependent interchangeability. Experimental evaluation.

7.1 Introduction

Many synthesis tasks can be reduced, on an abstract level, to the generic task of “assembling” some “artifact” from a set of “building blocks” (*e.g.* components in configuration and design, actions in planning, repair actions in therapy, qualitative models in model synthesis, *etc.*).

Central to synthesis is the notion of resource. An important part of the knowledge associated with a particular application domain is represented by *producer-consumer* relations between various parts of the artifact. They introduce cumulative restrictions on resource properties of a set of objects. All the resources in the model must be balanced, *i.e.* the amount of resource produced should be equal or greater than the amount used.

In the majority of synthesis tasks, the optimization criterion implies the minimization or maximization of some resource and this is what eventually dictates the structure of the artifact.

In this chapter we present resource optimization methods for efficiently solving synthesis problems in a constraint-based framework. Our original contribution is twofold:

- We show how to obtain a tighter *lower bound* of the problem optimum by adding redundant constraints that take into account the “wastage” in a partial solution.
- We show how abstraction through focusing on relevant features permits added interchangeability to deal with equivalent sets of partial solutions.

In Section 2 we describe a class of problems which is representative for most synthesis tasks. Section 3 describes briefly our algorithms. Each of the following two sections, on the lower bound computation and on the use of abstraction and interchangeability, have a subsection presenting a running example, demonstrating that these techniques can significantly reduce the search effort for finding the optimal solution and proving its optimality. Section 6 presents additional experimental evidence to support our claims. We end with some concluding remarks.

7.2 Problem Definition

The problem we are interested in is very general. We are given a set of consumers, each characterized by the amount of resources it consumes. Available are several types of producers, each described by the amount of resources it can provide. A cumulative expression on some of the resources is designated as the *cost* of a solution. The task is to find the optimal set of producers such that:

- all the resources are balanced, and
- the cost of the solution is minimal.

Instances of this problem appear as subproblems in any synthesis task. Because the motivation of our work lies mainly in solving configuration tasks, the concrete examples used come from the configuration domain. Although we use simplified versions of real problems, the main aspects are preserved.

7.2.1 Example 1

Consider this problem, adapted from (ILOG 1998). A control system consists of a set of racks with electrical connectors in which one can plug different types of electronic cards. A rack has 3 connectors, and each connector can receive exactly one card. In addition to the number of connectors it provides, each rack is characterized by the maximal power it can supply. Cards are characterized only by the power they use. Available are two types of racks, capable of providing 90 and 110 units of power, and four types of cards, consuming 20, 45, 50, and, respectively, 65 units of power. The number and type of cards which can be connected to a rack is limited by two factors: the number of electrical connectors the rack has, and the maximal power the rack can provide.

The cost of a solution is represented by the maximal power supplied by all the racks in the system. The problem asks for the number and type of racks which can accept a particular set of cards, such that the cost is minimal.

Assume we are required to configure a control system that must accommodate four cards, $\{C_{20}, C_{45}, C_{50}, C_{65}\}$, one of each type available. We start by creating one instance of RACK, R_1 , in which we plug cards C_{20} and C_{45} . None of the other two cards can use R_1 anymore, this would require more than the maximum 110 units of power a rack can

provide. We add a new rack to the system, R_2 and plug in it C_{50} . The power limitation again prevents us from using the same rack for C_{65} , so we end up by using three racks. Since we are interested in minimizing the total maximal power, we choose for each of the three racks the lowest-power variant able to satisfy the request, thus obtaining a solution with cost 270. This gives us an upper bound for the optimal solution.

Continuing the search in a backtracking manner, we eliminate C_{45} and plug C_{20} and C_{50} in R_1 , which allows us to use the same rack for both C_{45} and C_{65} , making complete use of the power provided by R_2 . In fact this new solution, of cost 200, is optimal. To prove its optimality we have to show that a solution of cost lower than 200 is not possible. Actually, we can restrict the new solution even more: the next possible combination of lower cost, two small racks, has a cost of 180.

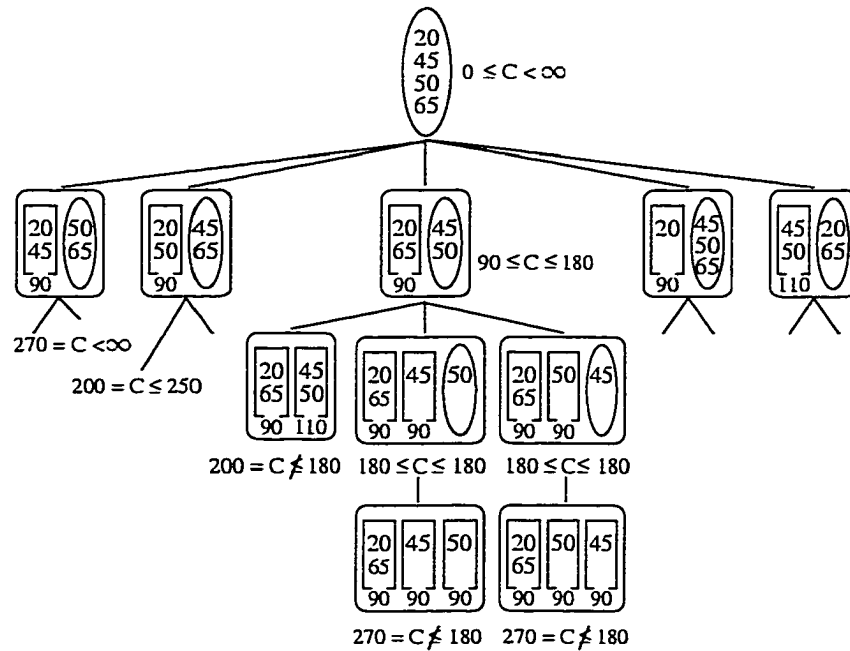


Figure 7-1: Snapshot of the search tree for an optimal solution (example 1).

We use *Branch and Bound* to reduce the amount of unnecessary work performed. The algorithm abandons a search path when the cost of the partial solution, *i.e.* the lower bound, exceeds the upper bound. On our example though, it turns out that the lower bound is not tight enough to really do any pruning. As we can see in Figure 7-1, it exceeds the upper bound too late, only after two racks have already been added to the system. This is because the lower bound computation is based solely on racks. To account for the amount of power left unused in each rack, the lower bound should consider both the maximal power of the existing racks, and the amount of power required by the cards which have not been plugged in yet. This would allow the algorithm to discover immediately after plugging C_{20} and C_{65} in R_1 that this partial solution actually incurs a minimum cost of 185 units, and therefore cannot lead to a solution of cost 180. In Section 4 we show how to achieve this by using specialized redundant constraints.

7.2.2 Example 2

Let us change the problem slightly. The two types of racks available provide 150 and 200 power units and the four types of cards require 20, 40, 50, and, respectively, 75 power units. The set of cards which must be plugged into racks is $\{ C_{20}^1, C_{20}^2, C_{20}^3, C_{20}^4, C_{20}^5, C_{20}^6, C_{20}^7, C_{20}^8, C_{40}^9, C_{40}^{10}, C_{40}^{11}, C_{40}^{12}, C_{50}^{13}, C_{50}^{14}, C_{75}^{15} \}$. The lower index represents the amount of power the instance requires. Instances requiring the same amount of power are of the same type. A snapshot of the search tree associated with this example is shown in Figure 7-2. We start again by creating an instance R_1 of rack, in which we plug cards C_{20}^1 through C_{40}^9 . The power provided by R_1 , 200 power units, is consumed entirely. A new rack instance, R_2 , receives cards C_{40}^{10} through C_{50}^{13} , which consume 170 power units of the maximum 200 it provides. Finally, the last two cards, C_{50}^{14}, C_{75}^{15} , are plugged in the third rack, R_3 , and use

125 power units out of the maximum 150 the rack provides. The cost of this first solution is 550, and gives us an upper bound for the optimal solution. Since the increment for the cost is 50, the next solution will be better only if it has a cost of at most 500.

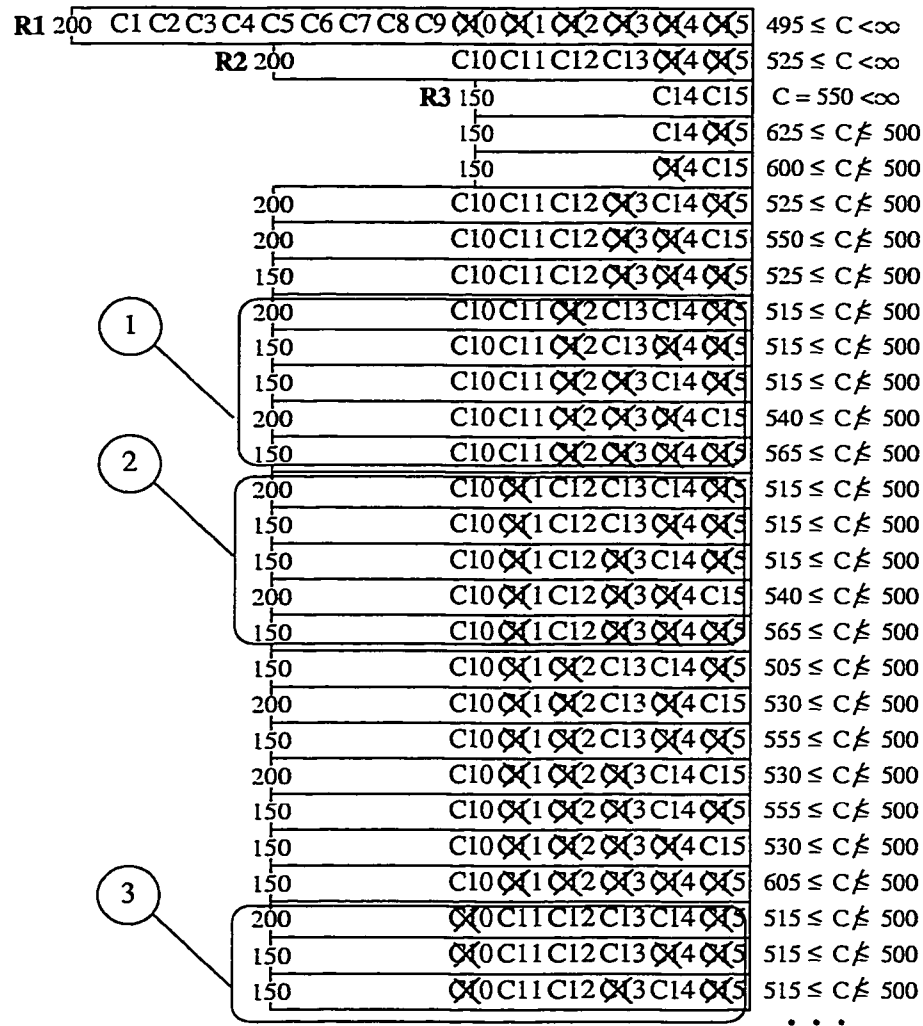


Figure 7-2: Snapshot of the search tree for an optimal solution (example 2).

The optimal solution actually has a cost of 500 power units. It turns out that finding it and proving its optimality is more difficult than in the previous example. This is due to the fact that most of the search effort is spent on exploring sets of equivalent partial solutions,

introduced in the search space by the use of multiple instances of the same type of card. In the figure we point out an example. Since the only restriction imposed on the cards is on their power consumption, although involving different values (card instances), the partial solutions in the three sets (1), (2) and (3) are equivalent. After the algorithm already investigated a solution assigning two 40 power unit cards to R_2 (set (1)), there is no point in trying other combinations of two similar cards. Therefore we can prune from the search tree the regions (2) and (3) without losing any problem solution. We show in Section 4 how to eliminate equivalent partial solutions efficiently using abstraction and interchangeability.

7.2.3 Problem representation

A producer-consumer relation implies a bidirectional connection between the objects involved in the relation. We capture this by adding a *port variable* to the model of each object that has resource properties. Ports are characterized by *base type* and *cardinality*. The domain of a port variable $P < \mathcal{T} > [m..n]$ is a set of objects of type \mathcal{T} , and the values the port can take are subsets of the domain, of cardinality at least m and at most n . We use the notation $|P|$ to refer to P 's cardinality.

There are several types of constraints that can be posted on port variables, two of which are relevant in the context of our presentation:

- cardinality constraints, imposing a lower and/or upper bound on the number of objects that can be assigned to the port, *e.g.* $|P| \leq 2$, $|P| > 0$, *etc.*, and
- cumulative constraints on attributes of the objects assigned to the port, *e.g.* $\sum P.x \leq 100$, where x is a numeric attribute of instances of type \mathcal{T} .

The model for the two examples described in the previous section is the following:

- The model for *system* consists of two variables:
 - integer variable $power_{SYSTEM}$ with domain $\{0..\infty\}$, and
 - port variable $racks_{SYSTEM} \langle RACK \rangle [1..\infty]$.
- RACK is described by three variables:
 - integer variable $power_{RACK}$ with domain $\{90, 110\}$ and $\{150, 200\}$, respectively;
 - port variable $system_{RACK} \langle SYSTEM \rangle [1..1]$;
 - port variable $cards_{RACK} \langle CARD \rangle [1..3]$.
- CARD instances are described by two variables:
 - integer variable $power_{CARD}$ with domain $\{20, 40, 50, 65\}$ and $\{20, 40, 50, 75\}$, respectively;
 - port variable $racks_{CARD} \langle RACK \rangle [1..1]$.
- In addition, the model for objects of type SYSTEM and RACK contains constraints expressing producer-consumer relations:
 - $power_{SYSTEM} = \sum racks_{SYSTEM} \cdot power$
 - $power_{RACK} \geq \sum cards_{RACK} \cdot power$
- The cost of a solution is represented by the value of the variable $power_{SYSTEM}$.

7.3 Algorithms

For the purpose of this thesis, we consider only complete search methods because we are interested in proving the optimality of the solution. A lot of research effort has been invested lately in the study of branch and bound variants of CSP search algorithms (Freuder &

Wallace 1992) (Cabon, De Givry, & Verfaillie 1998). Branch and Bound keeps track of an upper and lower bound for the cost of the solution. The upper bound is the cost of the best solution found so far, and can be updated when a new solution is discovered. The lower bound represents an estimate of the cost implied by the current (partial) solution, and gets monotonically updated as the algorithm advances on the solution path. These bounds are used for pruning entire branches from the search tree. At each step of the algorithm, the two bounds are compared against each other, and once the lower bound becomes at least as large as the upper bound ¹, it is clear that the current search path cannot lead to a better solution, and is abandoned. Obviously, the better (tighter) the bounds are, the more pruning the algorithm achieves. Although it is fairly easy to come up with a good upper bound, in the majority of cases this is not true for the lower bound (De Givry, Verfaillie, & Schiex 1997).

7.3.1 Port variables instantiation

One way to implement a port variable $V \langle T \rangle [m..M]$ is to maintain internally two sets of T instances, one representing the current value, the other one the domain. When the port is created, its domain consists of the set of all T instances which exist at that time in the model plus a *wildcard* instance $*_{\mathcal{T}}$, accounting for any future instance of T that might be created. This representation is similar to the one presented in (Mailharro 1998), although the implementation details seem to be different.

The instantiation process we propose is fairly straightforward. Using the set of constraints posted on the port as a filter, inconsistent instances are eliminated from the domain.

¹ We consider here that the objective is to minimize the cost variable, but the same principle applies when we try to maximize it.

All instances which passed the filter, except for the wildcard, are moved to the current value set. When the filtering phase ends, there are two possibilities.

1. The cardinality of the current value is at least m . In this case the port has been successfully instantiated.
2. The cardinality does not satisfy the lower bound requirement. Again we are left with two possibilities.
 - (a) The domain is empty, *i.e.* the wildcard has been rejected by the filter. In this case the port is considered to be *closed*. What this means is that no instance of type T can satisfy (anymore) the requirements imposed by the port, and therefore the instantiation fails.
 - (b) The wildcard is still in the domain. The procedure will first create a new instance of T and add it to the domain of all ports with base type \mathcal{T} which have not been closed yet ². Then, the instantiation process continues, with the new instance in the domain.

However, there is another aspect of the algorithm that we would like to point out. A connection established through ports is bidirectional. We capture this aspect in our model by using pairs of complementary ports. Assume that objects of type \mathcal{U} have a port of type \mathcal{T} , say $P<\mathcal{T}>$. Objects of type \mathcal{T} must then have a port of type \mathcal{U} , call it $Q<\mathcal{U}>$. Consider two instances, x and y , of type \mathcal{U} and \mathcal{T} , respectively. Connecting y to the port P of x means adding y to the current value of P . This happens during the process of instantiating

² We will show in Section 5 that this step can fail as well due to global limitations on the total number of instances of a given type.

P . Due to bidirectionality, x then must be added to the current value of port Q of y as well. The implication of this step is twofold. First, if adding x to Q would lead to a constraint violation, then it is not possible to add y to P either. Second, instances can be added to a port's current value even after the port has been instantiated, as long as the port has not been closed yet.

7.4 Achieving Optimality through Constraint Propagation

The search algorithm we use is not a Branch and Bound algorithm, but achieves the same effect through constraint propagation on redundant specialized constraints.

Our algorithm is based on a powerful CSP algorithm, *MAC* (Sabin & Freuder 1994) (Sabin & Freuder 1997). *MAC* uses constraint propagation for maintaining *arc-consistency* during search. Every time the domain of a variable is modified, the constraints in which the variable is involved are responsible for propagating the change to related variables. For more details on how this can be done efficiently see the original papers.

MAC is a general-purpose CSP search algorithm. In particular, it has no provision for finding optimal solutions. However, we do not need to change the algorithm for making it search for the optimal solution, we update the problem instead. Each time a solution of cost C is found, the constraint $cost < C$ is added to the problem to reflect the new upper bound, and then simulate a failure, thus forcing the algorithm to look for a better solution. A similar technique can be found in (ILOG 1998). It is obvious that the value C is the upper bound of the solution and that by adding the new constraint the updated upper bound becomes actively involved in the search.

The lower bound is integrated in the model through the use of resource constraints. In our problem the value of the cost variable $power_{SYSTEM}$ is controlled by the equality

constraint with $\sum racks_{system}.power$. Internally, the lower bound of the \sum is updated incrementally, as new elements are added to $racks_{SYSTEM}$. Through the equality constraint, the change propagates and updates the lower bound of the $power_{SYSTEM}$ variable.

Before moving further, we want to mention briefly that when deciding which variable to instantiate next, port variables are always preferred, and among several port variables, we choose first the ones belonging to a producer.

7.4.1 Improved lower bound computation

Let us get back to Example 1. We can observe from the beginning that the amount of power the racks have to provide must be at least 180 power units, the amount of power required by the four cards. The current model does not provide any way of directly relating this information to the cost variable. We will add a redundant constraint which, through propagation, will provide the connection.

To be able to keep track of the power requirement for all the cards in the system, we need a global point of view. We associate with each type \mathcal{U} a special type of port variable, called *metaport*. A metaport variable associated with type \mathcal{T} , $M\langle\mathcal{T}\rangle$, contains all the instances of \mathcal{T} that have been created and are currently part of the model.

Cardinality constraints on metaports allow us to put a limitation on the total number of instances of a given type that can be created. In addition to the usual constraints that can be posted on regular port variables (resource, cardinality, *etc.*), metaports offer a special type of resource constraint, called a *balancing* constraint. The constraint is described by a 4-tuple $\langle P, C, x, y \rangle$, where

- P is a metaport variable associated with type \mathcal{T} ,
- C is a metaport variable associated with type \mathcal{U} ,

- x and y are attributes representing the amount of resource r produced by an instance of \mathcal{T} and used, respectively, by an instance of \mathcal{U} .

A balancing constraint implies the existence of a producer-consumer relation between instances of the two types, \mathcal{T} and \mathcal{U} , on resource r , *i.e.* any instance t of \mathcal{T} has a port $U\langle\mathcal{U}\rangle$ and $t.x \geq \sum t.U.y$. Its semantics is the following:

- The initial lower bound for $\sum P.x$ is the lower bound of $\sum C.y$.
- The lower bound of $\sum P.x$ is updated incrementally as the result of:
 - Creating a new instance u of type U : the value of $\sum P.x$ is increased by $u.y$.
 - Closing an instantiated port $t.U$ on attribute y : the lower bound of $\sum P.x$ is increased by the difference $t.x - \sum t.U.y$.

We extend now the model for `SYSTEM`. We add two metaports, $P\langle RACK \rangle$ and $C\langle CARD \rangle$, as well as two constraints: $Balance(P, C, power_{RACK}, power_{CARD})$ and, since all instances of `RACK` must be part of the system, $power_{SYSTEM} = \sum P.power_{RACK}$. The results of this change are presented in Figure 7-3.

7.5 Equivalent Partial Solutions

Let us get back to Example 2. Although we did not mention this before, the instantiation algorithm considers an implicit ordering among the elements in the domain, thus avoiding symmetries introduced by permutation of values. For example, once the algorithm discovers the solution of cost 550 which assigns the value $\{ C_{40}^{10}, C_{40}^{11}, C_{40}^{12}, C_{50}^{13} \}$ to rack R_2 , it will never consider trying permutations of this set as values for R_2 .

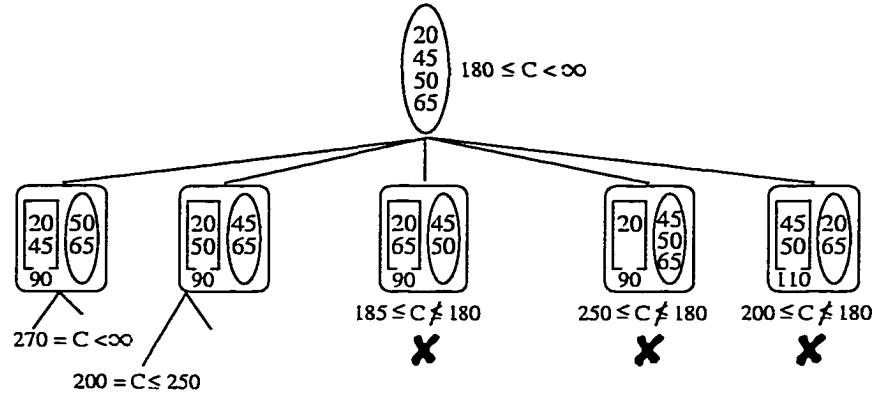


Figure 7-3: Snapshot of the search tree for an optimal solution (example 1 – after adding the redundant constraint).

This cuts down some of the search effort, but we are still left with partial solutions that are equivalent in the sense that they all participate exactly with the same amount to the final solution cost.

7.5.1 Eliminating equivalent partial solutions through interchangeability

The simplest type of equivalence is introduced by multiple instances of the same type. Take a look at Figure 7-2. Exchanging C_{40}^{12} for C_{40}^{11} in the partial solution that includes two instances of 40 power unit cards in the value of R_2 , C_{40}^{10} and C_{40}^{11} , will lead to a solution of equal cost. This is because in our model any two card instances of the same type are identical in all respects.

By analogy with (Freuder 1991), we say that two instances are *interchangeable* if replacing one by the other in any solution produces another solution of equal cost. According to this definition, two card instances of the same type are interchangeable.

The process we propose for eliminating equivalent partial solutions is the following. Once an instance is rejected from a domain during port variable instantiation, we look for

all the other instances of components of the same type and reject them as well. The effect of doing this on problem in Example 2 is shown in Figure 7-4.

Although true for cards, it is not always the case that instances of the same type are interchangeable. Here is a simple example. We have two racks of the same type, R_{150}^1 and R_{150}^2 . Due to the different sets of cards already connected to the two racks, R_1 has 30 units of power still available, while R_2 has 50 left. Suppose the two racks are in the domain of card C_{40}^{11} which must be instantiated next. R_1 is rejected because of the power requirement, but rejecting R_2 based on the fact that the two instances have the same type would be wrong, since R_2 satisfies the power requirement.

The question is then how to decide when two instances are interchangeable. Remember that they are modeled as composite CSPs. Since all instances from the domain of a port have the same type, the corresponding composite CSPs have the same sets of variables and internal constraints. Then a sufficient, but not necessary, condition for two instances to be interchangeable is that pairs of corresponding variables have the same domain in both problems. In case the domains are the same, the two instances are clearly interchangeable.

7.5.2 Abstraction and context-dependent interchangeability

But this method might prove to be too restrictive. Assume that type CARD can be refined to several specialized types, each with additional features and providing non-identical functionality. Some cards requiring equal amounts of power are not instances of the same type anymore. Their models may differ, both in structure (*i.e.* number and type of variables and constraints) and in the domain of the variables. According to the above definition, these instances are not interchangeable anymore. However, because the only relevant aspect for deciding whether a card can be connected to a rack or not is the amount of power it re-

quires, solutions involving the same number of cards with equal power requirements are still equivalent.

We abstract the model for CARD and RACK through focusing on relevant common features only. Considering only the abstracted model permits added interchangeability. The decision on what features are relevant is made based on the set of constraints imposed on the port variable.

As shown before, constraints on ports involve attributes of the instances in the domain of the port, which in our model are represented by variables. It is this restricted set of variables which will be checked for domain identity in deciding whether two instances are interchangeable or not. In our example, the set of variables contains only the variable $power_{CARD}$.

According to the new definition of interchangeability, cards with equal power requirements are interchangeable. Applying the algorithm presented earlier on the problem instance in Example 2 produces the results presented in Figure 7-4.

7.6 Experimental Evaluation

In order to test the performance of our approach, we used a set of randomly generated test problem instances similar to the one presented in Example 2. Each instance is characterized by the cardinality of the set of cards. We generated problems having between 10 and 200 cards, with an increment of 10. For each number of cards we generated 50 problem instances. The types of the cards were assigned randomly among the four types.

We conducted two sets of experiments in which we addressed the problem of finding the optimal solution and proving its optimality. First, we compared our algorithm with a program implemented specifically for solving this problem, presented in (ILOG 1998). The

R1	200	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13	C14	C15	$495 \leq C < \infty$
											C10	C11	C12	C13	C14	C15	$525 \leq C < \infty$
																	$C = 550 < \infty$
																	$625 \leq C \neq 500$
															C14	C15	$600 \leq C \neq 500$
																	$550 \leq C \neq 500$
															C13	C14	$525 \leq C \neq 500$
															C12	C13	$515 \leq C \neq 500$
															C13	C14	$515 \leq C \neq 500$
															C12	C13	$540 \leq C \neq 500$
															C12	C13	$565 \leq C \neq 500$
															C12	C13	$505 \leq C \neq 500$
															C12	C13	$530 \leq C \neq 500$
															C12	C13	$555 \leq C \neq 500$
															C12	C13	$530 \leq C \neq 500$
															C12	C13	$605 \leq C \neq 500$
																	...

Figure 7-4: Snapshot of the search tree for an optimal solution (example 2 – after adding the redundant constraint and using context-dependent interchangeability.

results, in terms of CPU time, are presented in Figure 7-5. The advantage of our method is obvious. For example for problems with 30 cards, we limited the running time for the Solver code to two hours, while our algorithm completed on average in 0.5 seconds.

For the second set of experiments we used only our algorithm and compared the search effort spent for finding the first solution with the search effort required for finding the optimal solution and proving its optimality. The results are presented in Figures 7-6 and 7-7. We report two different measures of the search effort: number of failures (backtracks) and CPU time. Both figures consist of two plots, one for the first solution (the plot name is prefixed by *first*), the other for finding the optimal solution and proving its optimality (the plot name is prefixed by *optimal*). Each point of the plot was computed as the average over the 50 problem instances generated for each value of the number of cards.

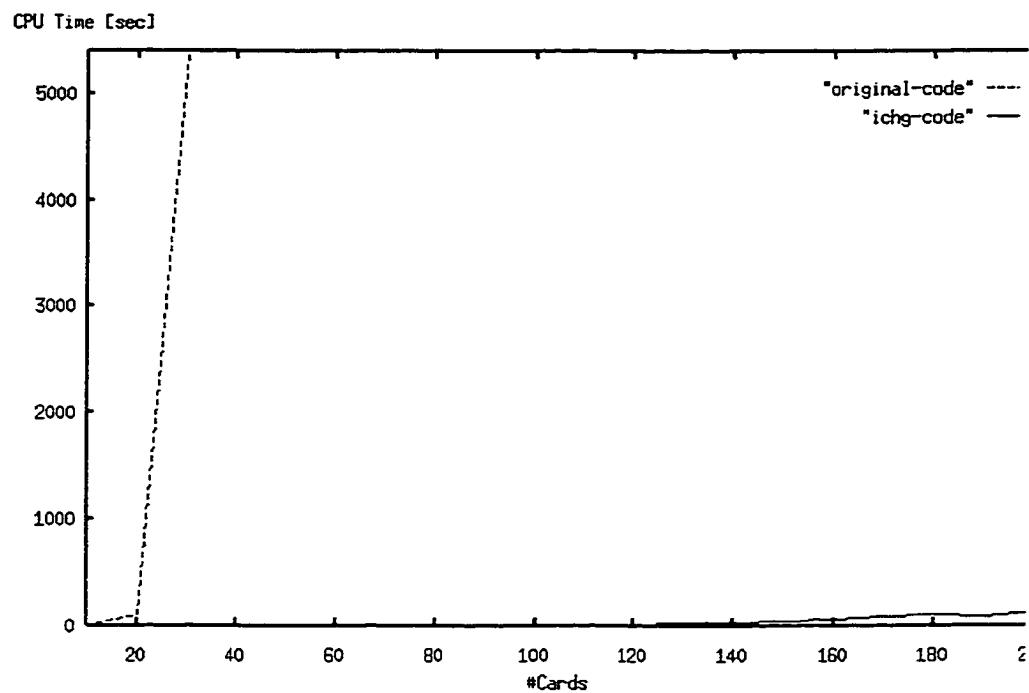


Figure 7-5: Comparison with the original Solver code.

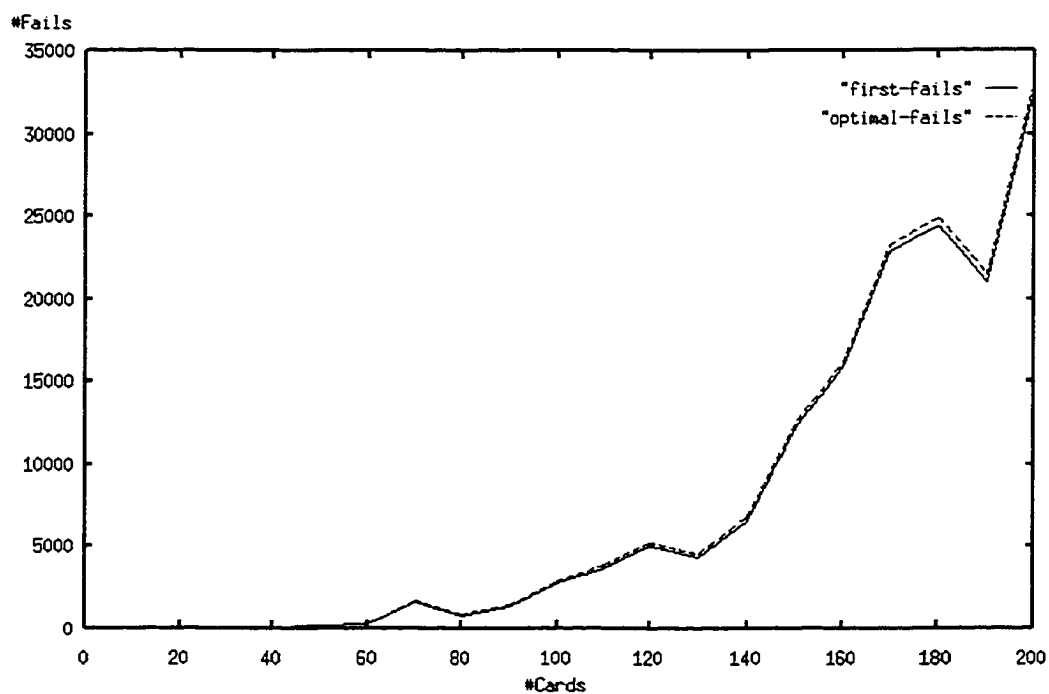


Figure 7-6: Search effort in terms of number of failures

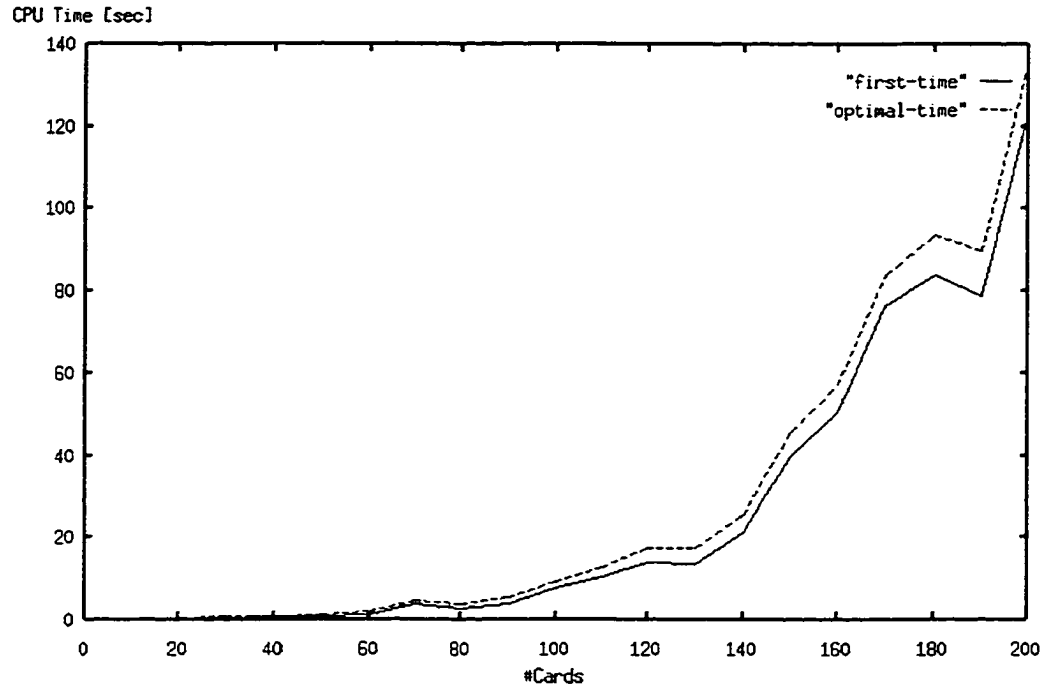


Figure 7-7: Search effort in terms of CPU time.

As we can observe, the two plots are very close to each other, which proves the advantages of our method: we not only discover quickly the optimal solution, but we are also able to prove very quickly its optimality.

To realize what is the impact of each of the two methods on the algorithm performance, we present in Figure 7-8 the results of a third set of experiments. We show the running time of the base algorithm, the base plus the improved lower bound computation, the base plus the context-dependent interchangeability method, and the base plus the two methods combined. Each point of the plot was computed as the average over the 50 problem instances generated for each value of the number of cards.

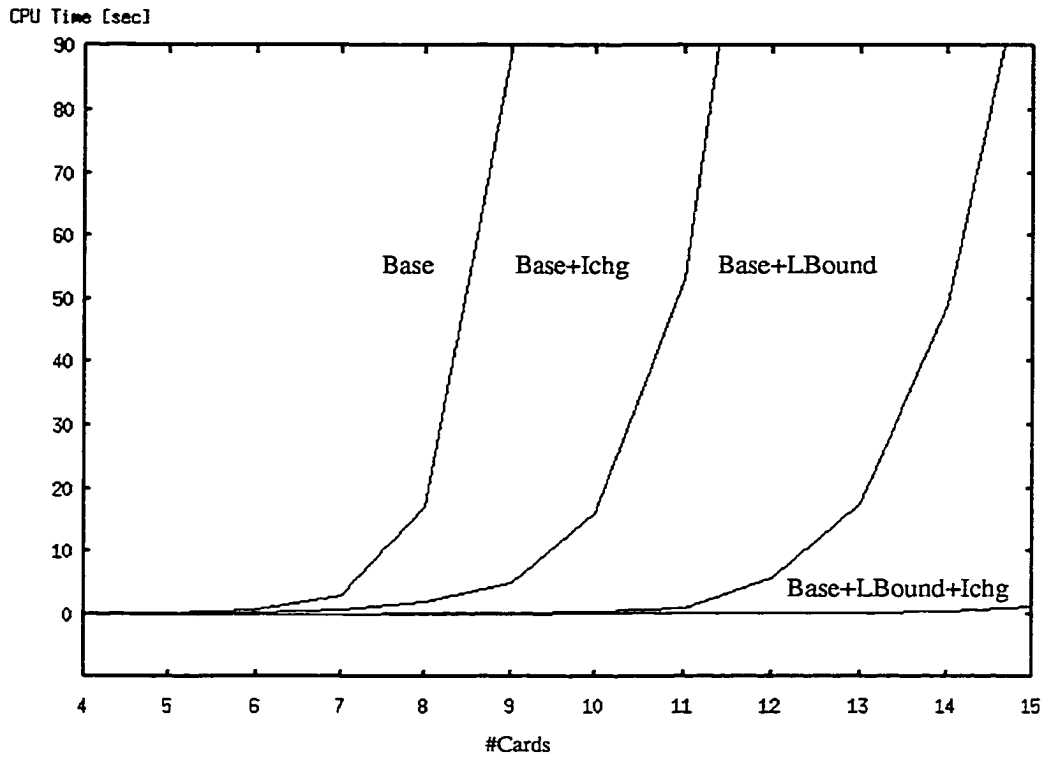


Figure 7-8: The effect of each optimization method on algorithm performance.

7.7 Chapter Conclusions

The specification of most configuration problems requires that the solution be optimal according to some criterion. Most often, this is either to maximize or to minimize a certain resource produced, respectively consumed, by the artifact. Taking advantage of the locality imposed by our modeling methodology and of the composite CSP representation, we have developed two specific optimization methods, which outperform by orders of magnitude previous methods. More specifically, we obtain a tight lower bound of the problem optimum by adding redundant constraints that take into account the “wastage” in a partial solution, while abstraction through focusing on relevant features permits added interchangeability

to deal with equivalent sets of partial solutions. Combining these two ideas allows us to discover quickly the optimal solution, and also to prove very quickly its optimality.

CHAPTER 8

CONCLUSION

We have presented a constraint-based framework for configuration. Our work was motivated by the configuration of technical products, but the results we present can be applied to nontechnical domains as well. It addresses the two main issues raised by any reasoning task, in general, and by configuration, in particular, namely modeling and efficient solving.

Our approach offers a component-oriented view of configuration tasks. The knowledge associated with a particular application domain is modeled by a generic product architecture, described in terms of generic parts, which captures multiple product variants within a single data model. The solution we propose combines object-orientation with constraint-based reasoning. Modeling concepts like abstraction and aggregation provide the support for a natural and compact organization of the domain knowledge, while the underlying constraint-based representation offers powerful solving techniques. In particular, the essential contributions of this thesis are the following:

- We have compared several approaches for modeling configuration tasks, based on different paradigms, presenting both their strengths and weaknesses. We have identified the main aspects raised by configuration tasks, that need to be addressed by any configuration framework: hierarchical organization of the domain knowledge, the intrinsic internal structure of the application objects, and the dynamic nature of the configuration process.

- The modeling methodology we propose, promoting a composite model of the artifact, obtained by aggregation of local, context independent models of its constituent parts, offers support and guidance as to the appropriate content and organization of the domain knowledge, thus making knowledge specification and representation less error prone.
- To be able to provide the powerful and flexible representation mechanism required by this modeling methodology, we have introduced Composite Constraint Satisfaction Problems, a new, nonstandard class of problems which extends the classic Constraint Satisfaction paradigm. Generalization and aggregation are captured directly by the notion of composite value, while relations among application objects are expressed through port variables. The dynamic aspect is handled through the set of constraints posted on port variables and the instantiation mechanism. Furthermore, based on a declarative paradigm, our framework provides complete separation between domain knowledge and control strategy, which makes both knowledge specification and knowledge maintenance much easier.
- Once the representation mechanism is in place, the second main concern of a configuration framework is to provide efficient search techniques, able to cope with the size and complexity of the knowledge base. In 1994 we proposed a search algorithm which maintains full arc consistency during search, MAC, and showed that it can be very effective. Based on subsequent advances in constraint propagation algorithms and dynamic variable ordering heuristics, a recent implementation of MAC (Bessiere & Regin 1996) became the best general search algorithm to date. By taking advantage of the problem structure, we have developed MACE, an improved version of MAC,

which consistently outperforms the previous version.

- The specification of most configuration problems requires that the solution be optimal according to some criterion. Most often, this is either to maximize or to minimize a certain resource produced, respectively consumed, by the artifact. Taking advantage of the locality imposed by our modeling methodology and of the composite CSP representation, we have developed two specific optimization methods, which outperform by orders of magnitude previous methods. More specifically, we obtain a tight lower bound of the problem optimum by adding redundant constraints that take into account the “wastage” in a partial solution, while abstraction through focusing on relevant features permits added interchangeability to deal with equivalent sets of partial solutions. Combining these two ideas allows us to rapidly discover the optimal solution, and also to prove very quickly its optimality.

References

- Bachant, J., and McDermott, J. 1984. R1 revisited: Four years in the trenches. *AI Magazine* 5(3).
- Bachant, J., and Soloway, E. 1989. The engineering of XCON. *Communications of the ACM* 32(3).
- Bachant, J. 1988. RIME: Preliminary work toward a knowledge-acquisition tool. In Marcus, S., ed., *Automating Knowledge Acquisition for Expert Systems*. Boston, MA: Kluwer Academic Publisher.
- Barker, V., and O'Connor, D. 1989. Expert systems for configuration at Digital: XCON and beyond. *Communications of the ACM* 32(3).
- Bessiere, C. 1994. Arc-consistency and arc-consistency again. *Artificial Intelligence* (65).
- Bessiere, C., and Cordier, M. 1993. Arc-consistency and arc-consistency again. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*.
- Bessiere, C.; Freuder, E.; and Regin, J.-C. 1995. Using inference to reduce arc consistency computation. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, volume I.
- Bessiere, C., and Regin, J.-C. 1996. MAC and combined heuristics: Two reasons to forsake FC (and CBJ?) on hard problems. In *Second International Conference on Principles and Practice of Constraint Programming - CP96*, number 1118 in Lecture Notes in Computer Science. Springer.
- Bowen, J., and Bahler, D. 1991. Conditional existence of variables in generalized constraint networks. In *Proceedings of the Ninth National Conference on Artificial Intelligence*.
- Cabon, B.; De Givry, S.; and Verfaillie, G. 1998. Anytime Lower Bounds for Constraint Violation Minimization Problems. In *Proceedings of the 4th International Conference on Principles and Practice of Constraint Programming*, number 1520 in Lecture Notes in Computer Science. Springer.
- Cooper, M. 1989. An Optimal k-consistency Algorithm. *Artificial Intelligence* (41).
- De Givry, S.; Verfaillie, G.; and Schiex, T. 1997. Bounding the Optimum of Constraint Optimization Problems. In *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming*, number 1330 in Lecture Notes in Computer Science. Springer.
- Dechter, R., and Meiri, I. 1989. Experimental evaluation of preprocessing techniques in constraint satisfaction problems. *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*.
- Dechter, R., and Pearl, J. 1988. Network-based heuristics for constraint satisfaction problems. *Artificial Intelligence* (34).
- Dechter, R. 1990. Enhancement Schemes for Constraint Processing: Backjumping, Learning and Cutset Decomposition. *Artificial Intelligence* (41).
- Freuder, E., and Wallace, R. 1991. Selective relaxation for constraint satisfaction problems. In *Proceedings of the Third International IEEE Computer Society Conference on Tools for Artificial Intelligence*.

- Freuder, E., and Wallace, R. 1992. Partial Constraint Satisfaction. *Artificial Intelligence* (58).
- Freuder, E. 1978. Synthesizing constraint expressions. *Communications of the ACM* 21(11).
- Freuder, E. 1982. A sufficient condition for backtrack-free search. *Journal of the ACM* 29(1).
- Freuder, E. 1991. Eliminating Interchangeable Values in Constraint Satisfaction Problems. In *Proceedings of the Ninth National Conference on Artificial Intelligence*.
- Freuder, E. 1992. Constraint solving techniques. In Mayoh, B.; Tyngu, E.; and Penjaen, J., eds., *Constraint Programming*, volume 131 of *Series F: Computer and Systems Sciences*. NATO ASI Series. 51–74.
- Gaschnig, J. 1974. A constraint satisfaction method for inference making. In *Proceedings of the Twelfth Annual Allerton Conference on Circuit and System Theory*.
- Gevecker, K. 1991. Relating the utility of relaxation in constraint satisfaction algorithms to the structure of the problem. Master's thesis, University of New Hampshire.
- Golden, M.; Siemens, R.; and Ferguson, J. 1986. What's wrong with rules? In *Proceedings of WESTEX-86*.
- Golumb, S., and Baumert, L. 1965. Backtrack programming. *Journal of the ACM* (12).
- Grant, S., and Smith, B. 1995. The phase transition behaviour of maintaining arc consistency. Technical Report 95.25, University of Leeds, School of Computer Studies.
- Hamscher, W. 1992. Model-based reasoning in financial domains. Technical Report TR-27, Price Waterhouse Technology Center.
- Haralick, R., and Elliot, G. 1980. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence* (14).
- Haselbock, A. 1993. *Knowledge-based Configuration and Advanced Constraint Technologies*. Ph.D. Dissertation, Institut für Informationssysteme, Technische Universität Wien.
- Heinrich, M., and Jungst, E. 1991. A resource-based paradigm for the configuring of technical systems from modular components. In *Proceedings of the Seventh IEEE Conference on Artificial Intelligence Applications*.
- Hyvonen, E. 1992. Constraint reasoning based on interval arithmetic: the tolerance propagation approach. *Artificial Intelligence* (58).
- ILOG, S. A. 1998. *ILOG Solver User's Manual*.
- Jegou, P. 1991. *Contribution à l'Etude des Problèmes de Satisfaction de Contraintes: Algorithmes de Propagation et de Résolution, Propagation de Contraintes dans les Réseaux dynamiques*. Ph.D. Dissertation, Université de Montpellier II.
- Klein, R. 1996. A logic-based description of configuration: the Constructive Problem Solving approach. In *Configuration – Papers from the 1996 Fall Symposium. AAAI Technical Report FS-96-03*.
- Kokeny, T. 1994. A new arc-consistency algorithm for csp's with hierarchical domains. In *Workshop Notes of the ECAI'94 Workshop on Constraint Satisfaction Issues Raised by Practical Applications*.

- Kumar, V. 1992. Algorithms for constraint-satisfaction problems: a survey. *AI Magazine* 13(1).
- Mackworth, A., and Freuder, E. 1985. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Artificial Intelligence* (25).
- Mackworth, A.; Mulder, J.; and Havens, W. 1985. Hierarchical arc consistency: exploiting structured domains in constraint satisfaction problems. *Computational Intelligence* (1).
- Mackworth, A. 1977. Consistency in networks of relations. *Artificial Intelligence* (8).
- Mailharro, D. 1998. A Classification and Constraint based framework for configuration. *AIEDAM*. Special issue on Configuration.
- Mannisto, T.; Peltonen, H.; and Sulonen, R. 1996. View to product configuration knowledge modelling and evolution. In *Configuration – Papers from the 1996 Fall Symposium. AAAI Technical Report FS-96-03*.
- McDermott, J. 1981. R1: The formative years. *AI Magazine* 2(2).
- McDermott, J. 1982. R1: A rule-based configurer of computer systems. *Artificial Intelligence* (19).
- McGregor, J. 1979. Relational consistency algorithms and their applications in finding subgraph and graph isomorphism. In *Information Science*.
- McGuinness, D., and Borgida, A. 1995. Explaining subsumption in description logics. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*.
- Mittal, S., and Falkenhainer, B. 1990. Dynamic constraint satisfaction problems. In *Proceedings of the Eighth National Conference on Artificial Intelligence*.
- Mittal, S., and Frayman, F. 1989. Towards a generic model of configuration tasks. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*.
- Mohr, R., and Henderson, T. 1986. Arc and path consistency revisited. *Artificial Intelligence* (28).
- Montanari, U. 1974. Networks of constraints: fundamental properties and applications to picture processing. *Information Sciences* (7).
- Nadel, B. 1988. Tree search and arc-consistency in constraint satisfaction algorithms. In Kanal, L., and Kumar, V., eds., *Search in Artificial Intelligence*. Springer-Verlag.
- Nadel, B. 1989. Constraint satisfaction algorithms. *Computational Intelligence* (5).
- Parunak, H.; Kindrick, J.; and Muralidhar, K. 1988. MAPCon: a case study in a configuration expert system. *AI EDAM* 2(2).
- Prosser, P. 1993a. Domain filtering can degrade intelligent backtracking. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*.
- Prosser, P. 1993b. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence* (9).
- Queyranne, M. 1990. *Production planning and scheduling*, volume 26 of *Annals of the Mathematical Sciences. Serie A, Annals of Operations Research*. Baltzer Scientific Publishing.
- Regin, J.-C. 1995. *Développement d'outils algorithmiques pour l'Intelligence Artificielle. Application à la chimie organique*. Ph.D. Dissertation, Université Montpellier II.

- Sabin, D., and Freuder, E. 1994. Contradicting Conventional Wisdom in Constraint Satisfaction. In *Proceedings of the 11th European Conference on Artificial Intelligence*. Wiley.
- Sabin, D., and Freuder, E. 1997. Understanding and Improving the MAC Algorithm. In *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming*, number 1330 in Lecture Notes in Computer Science. Springer.
- Sabin, M., and Freuder, E. 1998. Detecting and resolving inconsistency and redundancy in conditional constraint satisfaction problems. In *Workshop Notes of the CP'98 Workshop on Constraint Problem Reformulation*.
- Searls, D., and Norton, L. 1988. Logic based configuration with a semantic network. *Journal of Logic Programming* 8.
- Solotorevsky, G.; Gudes, E.; and Meisels, A. 1996. Modeling and Solving Distributed Constraint Satisfaction Problems (DCSPs). In *Second International Conference on Principles and Practice of Constraint Programming - CP96*, number 1118 in Lecture Notes in Computer Science. Springer.
- Stefik, M. 1995. *Introduction to Knowledge Systems*. San Francisco, CA: Morgan Kaufmann Publishers Inc.
- Tsang, E. 1993. *Foundations of Constraint Satisfaction*. London: Academic Press.
- Wallace, R. 1993. Why AC-3 is almost always better than AC-4 for establishing arc consistency in CSPs. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*.
- Waltz, D. 1975. *The Psychology of Computer Vision*. McGraw-Hill. chapter Understanding line drawings of scenes with shadows.
- Wright, J.; Weixelbaum, E.; Brown, K.; Vesonder, G.; Palmer, S.; Berman, J.; and Moore, H. 1993. A knowledge-based configurator that supports sales, engineering, and manufacturing at AT&T Network Systems. In *Proceedings of the Innovative Applications of Artificial Intelligence Conference*.
- Wright, J.; McGuinness, D.; Foster, C.; and Vesonder, G. 1995. Conceptual modeling using knowledge representation: Configurator applications. In *Proceedings of the Artificial Intelligence in Distributed Information Networks Workshop, IJCAI-95*.