Finite Domain Constraints: Declarativity meets Efficiency Theory meets Application

Vom Fachbereich Informatik
der Universität Kaiserslautern
zur Verleihung des akademischen Grades
Doktor der Naturwissenschaften (Dr. rer. nat.)
genehmigte Dissertation

von

Dipl.-Inform. Manfred Meyer

Berichterstatter:	Prof. Dr. Michael M. Richter
	Prof. Dr. Werner Dilger
Dekan:	Prof. Dr. Otto Mayer

Tag der wissenschaftlichen Aussprache: 25. November 1994

D 386

To Kerstin

Zusammenfassung

Techniken zur Constraint-Verarbeitung über endlichen Wertebereichen (*finite domain constraints*) werden seit über 20 Jahren untersucht und haben sich gerade in den letzten Jahren ein immer breiter werdendes Anwendungsfeld geschaffen. Obwohl diese Techniken in ihrem Kern auf ideale Weise Deklarativität bei der Problem*beschreibung* mit Effizienz bei der Problem*lösung* vereinbaren, wird in vielen praktischen Anwendungen das Ziel der Deklarativität zugunsten einer verbesserten Effizienz aufgegeben.

Die vorliegende Arbeit zeigt, daß dies nicht notwendigerweise der Fall sein muß: Nach einer Einführung in die grundlegenden Verfahren der Constraint-Verarbeitung werden zunächst Techniken untersucht, die den Umgang mit gewichteten Randbedingungen und somit die Relaxierung eines sonst unlösbaren Problems ermöglichen. Damit eröffnet sich ein sehr viel weiteres Spektrum von Anwendungen für die Constraint-Verarbeitung. Da Gewichtungen deklarativ formuliert werden können, trägt diese Erweiterung auch zur Erhöhung der Deklarativität dieses Ansatzes bei. Das gleiche gilt ebenfalls für die Verwendung hierarchisch strukturierter Wertebereiche (Taxonomien) bei der Constraint-Verarbeitung, wobei dieser Ansatz auch bereits deutliche Effizienzvorteile bietet. Für die Realisierung praktischer Anwendungen hat es sich außerdem als nützlich herausgestellt, die Techniken der Constraint-Verarbeitung in einer allgemeinen Programmiersprache nutzen zu können. Eine solche Integration in eine logik-orientierte Programmiersprache (Prolog) wird deshalb ebenfalls untersucht, sowie abschließend, als weitere Möglichkeit zur Effizienzsteigerung, auch deren Parallel-Verarbeitung diskutiert.

Alle beschriebenen Techniken sind in Form der Constraint-Systeme Contax bzw. FiDo implementiert und praktisch erprobt worden. Daß die mit diesen Arbeiten untermauerte These, wonach sich Deklarativität und Effizienz in der Constraint-Verarbeitung sehr gut vereinbaren lassen, nicht nur auf kleine Beispielanwendungen beschränkt bleibt, zeigt der abschließende Anwendungsteil der vorliegenden Arbeit: Hier werden zwei praktische Anwendungen in der Fertigungsplanung im CIM-Bereich, sowie in der Sportstättenbelegung des Hochschulsports vorgestellt. Der Erfolg beim Einsatz des Constraint-Systems Contax zur Lösung dieser Probleme beruht ganz wesentlich auf den Möglichkeiten zum effizienten Umgang mit gewichteten Constraints und hierarchisch strukturierten Wertebereichen.

Somit wird in dieser Arbeit nicht nur gezeigt, daß die beschriebenen Techniken und Erweiterungen die sonst einander ausschließenden Ziele der Deklarativität und Effizienz in Einklang bringen (*declarativity meets efficiency*), sondern es wird auch gezeigt, daß diese Arbeiten neben einem theoretischen Interesse auch die Anforderungen realer Anwendungen befriedigen (*theory meets application*).

Summary

Constraint processing techniques over finite domains (*finite domain constraints*) are studied for more than 20 years and have entered more and more application fields. Although these techniques ideally combine declarative problem *formulation* with efficient problem *solving*, most practical applications abandon declarativity for the sake of higher efficiency.

The present thesis shows that this must not necessarily be the case: After introducing the basic constraint processing methods, techniques are studied for dealing with weighted constraints and thus allowing for relaxing problems that otherwise would be unsolvable. This opens a much broader range of application areas for constraint processing. As weights can be formulated declaratively, this extension also improves the declarativity of our approach as a whole. The same is true for making use of hierarchically structured domains (taxonomies) in constraint processing which already offers significant advantages as concerns efficiency.

In practical applications it has also appeared to be useful having constraint processing techniques embedded in a universal programming language. Therefore, we will also study an integration of consistency techniques in a logic programming language (Prolog) and will finally discuss further improving efficiency by using parallel processing.

All techniques described are implemented in the constraint systems Contax and FiDo and have been tested in practical use. The claim that as far as finite domain constraints are concerned, declarativity does not compete efficiency, is furthermore supported by the second part of the thesis where two practical applications to computer integrated manufacturing (CIM) and to university sports scheduling are presented. The success of using the Contax system to solve these problems basically rests on the ability to efficiently deal with weighted constraints and hierarchically structured domains.

Hence, the present thesis shows that the techniques and extensions presented here do not only contribute to bridge the gap between declarativity and efficiency—and let *declarativity meet efficiency*— but also tackle interesting theoretical issues as well as practically relevant problems—and let *theory meet application*.

Acknowledgements

First and foremost, I would like to thank my thesis advisors, Professor Dr. Michael M. Richter and Professor Dr. Werner Dilger. Professor Dilger initially drew my attention to the field of constraint processing when I attended his lectures on "Constraint Systems" at the University of Kaiserslautern in 1987. After joining the research group of Professor Richter at the German Research Center for Artificial Intelligence (DFKI) in 1989, I continued to investigate constraint processing techniques, how they can be used for solving real-world application problems in computer-integrated manufacturing, how the available techniques have to be extended and how the resulting technology can be applied to numerous application problems from various domains. I am very grateful to Professor Richter for his constant encouragement, for many productive discussions, and for his general support during all these years; but most important, I am indebted to him for urging me to collect, focus, and finally write down all this material which then resulted in the present thesis.

Parts of the research presented in this thesis are based on work several students did under my supervision as part of their master's thesis or *Projektarbeit* at the University of Kaiserslautern: Bernd Bachmann's HieraCon system already made use of hierarchical domain structures for the configuration of Hewlett-Packard computer systems. Frank Steinle implemented, improved, and maintained the Contax system over several years. The various implementational approaches for the FiDo system were realized by Stefan Schrödl, Jörg Müller, Hans-Günther Hein, and Werner Stein. Jörg Müller also formalized the lathe-tool selection problem as a CSP and realized a first prototype, and Enno Tolzmann implemented the lathe-tool selection module of the ARC-TEC planning system using Contax. Michael Kreinbihl showed that Contax can be used in various application domains by building the CoSyPlan system in close cooperation with Gunnar Krug. Without the practical work of all these people, this thesis would never have taken its present form. Finally, they all helped to support the second claim of this thesis: theory meets application.

I am also grateful to all my colleagues within the ARC-TEC, TooCon, and VEGA projects at the DFKI for fruitful discussions, cooperation, and a stimulating atmosphere.

Numerous other people, including Philippe Codognet, Gene Freuder, Jérôme Gensel, Michael Jampel, Andrei Mantsivoda, Gustaf Neumann, Andreas Podelski, and Dan Vlasie also influenced my work and spent time discussing ideas.

Thanks also to Andreas Abecker, Bernd Bachmann, Hans-Günther Hein, Gerd Kamp, Marlene Miller, Jörg Müller, Frank Steinle, and Holger Wache for their thorough proofreading of various drafts of this thesis. Their hints and suggestions contributed much to the present form. Of course, I take full responsibility for all remaining orthographic, stylistic or factual mistakes.

Special thanks are also due to Klaus Elsbernd for his technical support, in general, during all my time at the DFKI, and especially for his instant help when our system crashed just the day before the thesis had to be submitted to the computer science department.

Most of all, my loving thanks to my fiancee Kerstin for putting up with me during the final weeks of working on my thesis.

Kaiserslautern, November 1994 Manfred Meyer

Contents

1

Introduction

It is wrong to think of Waltz's work only as a statement of the epistemology of line drawings of polyhedra. Instead I think it is an elegant case study of a paradigm we can expect to see again and again.
– Patrick Winston
The Psychology of Computer Vision (1975)

When more than two decades ago David Waltz presented his now well-known *filtering algorithm* for labeling three-dimensional line-diagrams [Waltz, 1972], one could hardly expect that the basic principle he introduced, namely the technique of constraint propagation, would become the basis of an emerging research area which has already outgrown the field of Artificial Intelligence and started to influence many other disciplines.

Today, an increasing body of researchers from Logic Programming, Knowledge Representation, Expert Systems, Theoretical Computer Science, Operations Research and other related fields are now investigating the use of constraint satisfaction techniques, their foundations and their applications to real-life problems. Over the recent years, constraint satisfaction has come to be seen as the core problem in many applications, for example temporal reasoning [Allen, 1983, Dechter *et al.*, 1989, Frühwirth, 1993, Hrycej, 1993, Tolba, 1993], spatial reasoning [Güsgen, 1989, Kopisch, 1993, Charman, 1993], configuration [Frayman and Mittal, 1987, Havens and Rehfuss, 1989, Mittal and Falkenhainer, 1990, Dauboin, 1993], planning [Descotte and Latombe, 1985, Müller, 1990, Liu and Popplestone, 1990, Boizumault *et al.*, 1993], and scheduling [Fox *et al.*, 1983, Fox, 1986, Fox and Sadeh, 1990, Duncan, 1990, Evans, 1992]. Its role in logic programming has also been recognized [Jaffar and Lassez, 1987, Rossi, 1988, Van Hentenryck, 1989] as well as its relation to Operations Research [Van Hentenryck and Carillon, 1988, Jaakola, 1990, Lee *et al.*, 1993] and Databases [Morgenstern, 1985, Deßloch, 1993].

The importance of constraint satisfaction is also reflected by the abundance of publications made at recent conferences such as ECAI-92, AAAI-92, and IJCAI-93, the dominance of constraint processing topics in the list of accepted papers at this year's major conferences like AAAI-94 and ECAI-94, as well as by the recent appearance of dedicated workshops on this topic such as the PPCP series[1] in the United States and the Constraint Processing Workshops in Europe[2]. A special volume of *Artificial Intelligence* was also dedicated to constraint processing in 1992 (Volume 58, Nos 1–3).

The main reason for the increasing attraction and the success of constraint processing techniques is that they appear to allow for both a *declarative problem formulation* and an *efficient problem solving.* However, in most real-life applications it turned out that the constraint processing techniques at hand were too weak either to formulate (lack of expressiveness) or to solve the given application problem (lack of efficiency). Thus, except for 'toy applications' or problems from a very limited class the power of constraint processing could be used only to a small extent; when faced with complex real-life applications, specialized systems were built incorporating specific features of the given application, and thus declarativity was often abandoned for the sake of efficiency, or for being able to solve the problem at all.

In contrast to this observation, the present thesis claims that a wide range of real-life problems can still be declaratively formulated as well as solved efficiently when extending the currently available constraint processing techniques to meet significant requirements appearing in almost all practical applications, e.g. the need for dealing with weighted constraints (priorities) or for making use of hierarchical domain structures (taxonomies). This way, the thesis contributes to make the constraint processing approach applicable to a wider class of application problems while still preserving both its most attractive properties: declarativity **and** efficiency.

1.1 The World of Constraint Satisfaction Problems

Constraint problems can naively be defined as consisting of a set of variables and a set of constraints, each specifying a relation on a particular subset of the variables. Thus, the relation *constrains* the values that the variable may take. By sharing variables among different constraints, networks of constraints can be built. The problem which then has to be solved is to find the relation on the set of all variables that simultaneously satisfies all the given constraints—respectively, the underlying relations. This problem is referred to as the *Constraint Satisfaction Problem* (CSP). Typically, unary relations for each variable specify their domains as a set of possible values and thus the required solution relation is a subset of the Cartesian product of the variable domains. If each domain is finite, we are dealing with *finite domain constraints* and the CSP then is a finite constraint satisfaction problem (FCSP) [Mackworth, 1992b].

[1]In 1993, an annual workshop on the *Principles and Practice of Constraint Programming* has been established and meetings have been held in Newport, Rhode Island, in April 1993 and on Orcas Island, Washington, in May 1994.

[2]Having recognized that the Constraint Processing community appeared to be very heterogeneous, the need became obvious for bringing together researchers from different areas dealing with various aspects of constraint processing regarded as a general paradigm of computation. Therefore, in July 1993 we initiated and organized a two-days *Workshop on Constraint Processing* at CSAM'93 in St. Petersburg (Russia) [Meyer, 1993]. Motivated by the success of this first European Constraint Processing meeting we are currently going to continue these efforts by organizing another *Workshop on Constraint Processing* to be held in August 1994 at ECAI-94 in Amsterdam [Meyer, 1994].

The Waltz filtering algorithm at the root of CSP research can be seen as solving a typical finite domain constraint problem, the *scene labeling problem* in computer vision which is probably the first constraint problem to be formalized. In vision, the scenes are normally captured as images by cameras. After some preprocessing, lines can be recognized from the images, then scenes like the one shown in Figure 1.1 are generated.

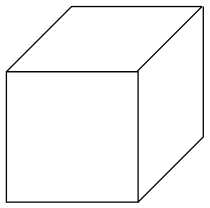
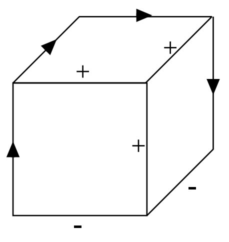


Figure 1.1: Example of a scene to be labeled

To recognize the objects in the scene, one has first to interpret the lines in the drawings. Therefor, [Huffman, 1971] and [Clowes, 1971] categorize the lines in a scene as follows:

- **convex edges:** A *convex edge* is an edge formed by two planes, both of which extend (from the edge) away from the viewer. Convex edges are marked by '+'.
- **concave edges:** A *concave edge* is an edge formed by two planes, both of which extend (from the edge) towards the viewer. Concave edges are marked by '–'.
- **occluding edges:** An *occluding edge* is a convex edge where one of the planes is hidden behind the other and therefore not seen by the viewer. Occluding edges are marked by either '→' or '←', depending on the situation. If one moves along an occluding edge following the direction of the arrow, the area on the right represents the face of an object which can be seen by the viewer, and the area on the left represents the background or some faces of the objects at the back.

Using these labelings, the scene in Figure 1.1 can, for example, be labeled as follows:



The key observation for recognizing the scene-labeling problem as a CSP, is that, given any junction independent of the scene, there are only limited choices of labels as only a few combinations of labels are physically possible. These combinations *constrain* the possible labelings for the lines in the scene and are shown in Figure 1.2.

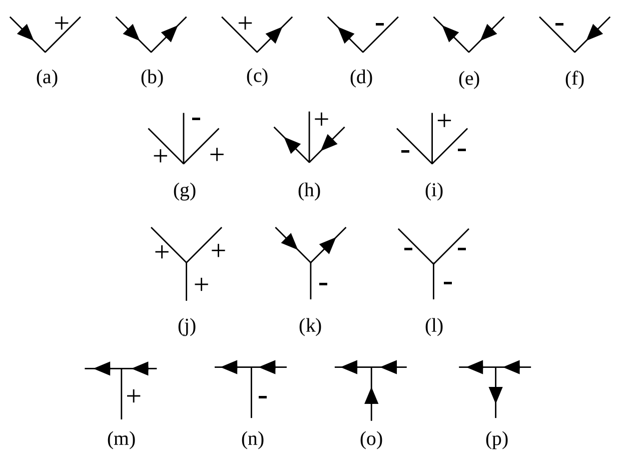


Figure 1.2: Legal labels for junctions

Thus, it is straightforward to formalize the scene labeling problem as a CSP: One uses one variable to represent the value of a line in the scene. For example, in the scene in Figure 1.1 we have the following variables: {A, B, C, D, E ,F, G, H, I}, as shown in Figure 1.3. The domain of each variable is therefore the *finite* set $\{+, -, \rightarrow, \leftarrow\}$.

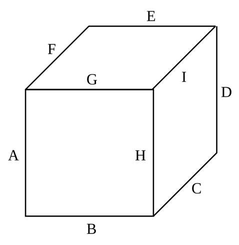
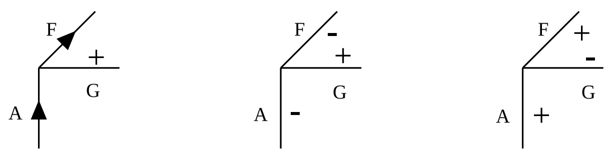


Figure 1.3: Variables in the scene labeling problem

The limited choices of label combinations in the junctions as shown in Figure 1.2 impose constraints on the variables. Since the lines represented by the variables A, F, and G form an arrow, according to (g)–(i) in Figure 1.2, the values that these three variables can take simultaneously are restricted to the following combinations:



Similarly, every other junction poses constraints to the labeling of the lines which form it. The task then remains to label all the variables, satisfying all the constraints. One may find one or all solutions to this problem, depending on the need of subsequent processing.

Applying a backtracking algorithm in order to compute a solution which satisfies the given constraints turned out to be a desperate approach for practical examples. Thus, Waltz introduced an alternative which is now known as the *Waltz filtering algorithm* [Waltz, 1972] and is presented here mainly for historical reasons as it constitutes the first constraint propagation algorithm and thus states the birth of constraint processing research.

```
Form a queue consisting of all junctions.
Repeat
    Remove the first element from the queue. Call it the current junction.
    If the current junction has never been visited,
    then create a pile of junction labels for it consisting of all possible
          junction labels for the junction type involved.
          Note that a pile change has occured.
    endif
    If any junction label from the current junction's pile is incompatible
       with all the junction labels in any neighboring junction's pile,
    then eliminate that incompatible label from the current junction's pile.
          Note that a pile change has occured.
    endif
    If a pile change has occured,
    then for each neighboring junction with a pile that is not on the
          queue, add that junction to the front of the queue.
    endif
until the queue is empty.
```

Figure 1.4: The Waltz filtering algorithm for scene labeling

The basic idea of this algorithm is that constraints are evaluated locally and the restrictions for labelings are *propagated* to neighbor constraints (junctions) without constructing the set of all possible combinations of labelings. Thus, by using *constraint propagation* techniques, consistent labelings could be found efficiently.

Many other problems can be formulated as constraint satisfaction problems, although researchers who are not familiar with this field sometimes fail to recognize them, and consequently, fail to make use of specialized techniques for solving them.

However, most real-life problems do not come in the concise form as the scene-labeling problem does. Often it requires some thought to recognize how a given problem can be formalized as a CSP. And, even more often the problem exhibits some characteristics that excludes it from being solved using currently available constraint solving techniques. For example, consider the types of requirements occuring in a production planning application which are shown in Figure 1.5.

The technological requirements stored in a database can be easily transferred into a representation such that CSP techniques can use them appropriately. The same is true for the geometrical requirements although the domains may no longer be finite and thus the most powerful class of constraint-solving techniques cannot be applied. However, the hardest problem for most state-of-the-art constraint solvers comes with the economical requirement which obviously needs not to be satisfied in any case. Therefore, techniques are needed to handle different degrees of how important or restrictive a constraint is, or

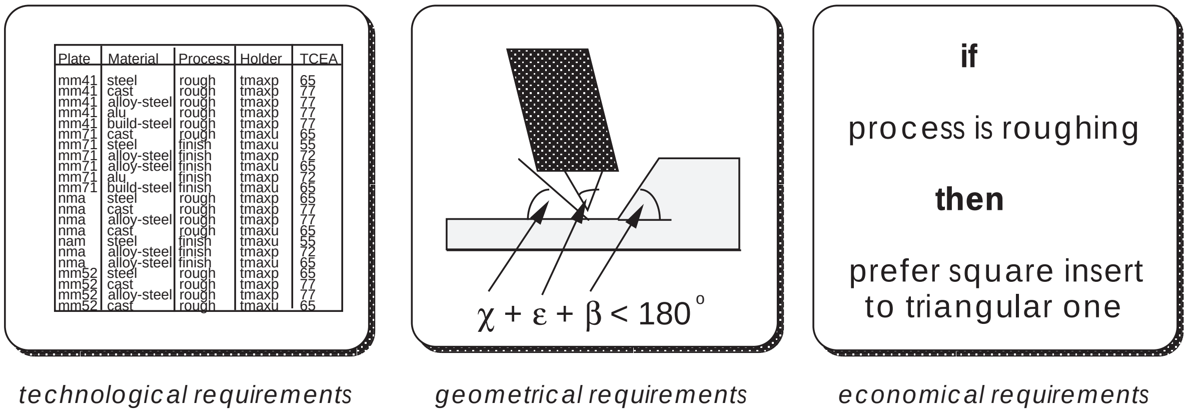


Plate	Material	Process	Holder	TCEA
mm41	steel	rough	tmaxp	65
mm41	cast	rough	tmaxp	77
mm41	alloy-steel	rough	tmaxp	77
mm41	alu	rough	tmaxp	77
mm41	build-steel	rough	tmaxp	77
mm71	cast	rough	tmaxu	65
mm71	steel	finish	tmaxu	55
mm71	alloy-steel	finish	tmaxp	72
mm71	alloy-steel	finish	tmaxu	65
mm71	alu	finish	tmaxp	72
mm71	build-steel	finish	tmaxu	65
nma	steel	rough	tmaxp	65
nma	cast	rough	tmaxp	77
nma	alloy-steel	rough	tmaxp	77
nma	cast	rough	tmaxu	65
nam	steel	finish	tmaxu	55
nma	alloy-steel	finish	tmaxp	72
nma	alloy-steel	finish	tmaxu	65
mm52	steel	rough	tmaxp	65
mm52	cast	rough	tmaxp	77
mm52	alloy-steel	rough	tmaxp	77
mm52	cast	rough	tmaxu	65

$\chi + \varepsilon + \beta < 180^\circ$
if
process is roughing
then
prefer square insert to triangular one
technological requirements
geometrical requirements
economical requirements

Figure 1.5: Requirements for Lathe-Tool Selection in a CIM Environment

to *relax* a problem appropriately if it appears that not all constraints (requirements) can be satisfied simultaneously.

In the same application CIM application we are also faced with large domains of values that are structured hierarchically in a way as shown in Figure 1.6.

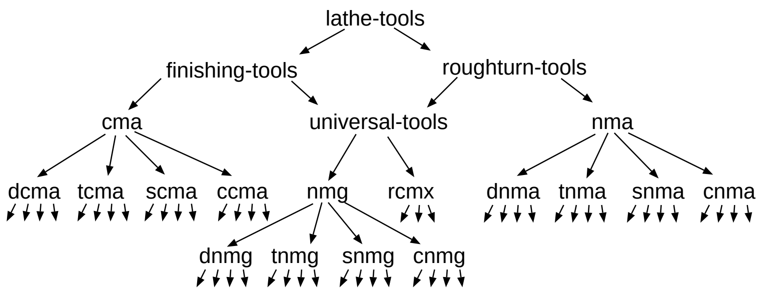


Figure 1.6: A sample hierarchical domain of lathe tools

As most state-of-the-art constraint solvers do not exploit this hierarchical structure, they perform rather inefficient on problems with large hierarchical domains. Therefore, investigating techniques for dealing with these characteristics that are apparent in most practical applications, will be one of the central issues for this thesis.

1.2 Declarativity competes Efficiency

Given a problem like the scene labeling problem, there are several ways to represent and solve it using a computer. Besides other criteria, they can be compared concerning the declarativity and efficiency of the resulting implementation.

- **Declarativity** as a goal in Software Engineering traces back to Bob Kowalski who proposed the separation of the logical (declarative) and control (procedural) parts of an algorithm when he stated his now famous equation:

Algorithm = **Logic** + **Control** [Kowalski, 1979].

Today, declarativity plays a more and more important role in knowledge representation and reasoning as well as in programming language design. Declarative representation, meaning that knowledge is being represented without anticipating its later use, is now believed to be the key issue

- for achieving *software/knowledge sharing and reuse*,
- for allowing *formal verification* of knowledge bases or programs,
- for supporting *software or knowledge-base maintenance*, and finally
- for *increasing the acceptance* of AI applications by developing programs or knowledge-bases that are much easier to understand.

- **Efficiency** can naively be measured by the final run-time of a given application. Although this often is what the user of a system is ultimately interested in, its absolute value will change from one hardware generation to the next and thus efficiency in this sense would just be a matter of the development of more powerful hardware systems. In contrast, efficiency can be better defined in terms of how efficiently a system performs its task, i.e., how many of the operations that it performs are really necessary to solve a particular problem.

One possible approach for solving a combinatorial problem, would of course be to develop a specialized computer program using some *imperative* programming language, that will be hard-coded to analyze the given line drawings in the scene labeling problem. Such a program can especially be tuned to solve its one given problem. Thus, the programmer can take advantage of any optimizations and codings that may help to solve the problem faster, e.g. by using a bit-vector coding for representing connections between the edges in a scene which has to be labeled. Such a program would solve its task very efficiently. However, as soon as it has to be adapted to an only slightly different problem specification, we are faced with serious problems:

- First, the program will be hard to understand—sometimes even for the programmer himself or herself, respectively.

- Second, it will be very difficult to identify those parts of the program that can be reused for a new application[3].

- Finally, such a program will be very hard or even impossible to verify, i.e. to prove its correctness and completeness. Therefore, it may not be used in any security-relevant application where it has to be guaranteed that it will always behave as specified.

[3]This is one of the hardest problems in Software Engineering in general and has led to the development of abstractions and encapsulation concepts for enabling the sharing and reuse of software modules (cf. e.g. [Boehm, 1988], [Wegner, 1990]).

Thus, the price that one has to pay for getting highest possible efficiency will be a loss in *understandability*, *reusability*, *maintainability*, and in *applicability* of formal verification methods.

On the other extreme, as the scene labeling problem comes with a very concise and logical definition, one could also try to represent it as a first-order theory, formulate the existence of a consistent line labeling as a theorem, and then run a general theorem prover on it. The resulting program, i.e. the theory, will be much easier to *understand*. As it will be represented as a logical formula, parts of it will be easily *reusable*, and the theory/program itself will be easy to *maintain* when the problem specification changes. Finally, due to the formal logical representation, *formal verification methods* are also easily applicable. However, these benefits of a purely declarative problem representation as a logical formula also have to be paid for: First-order theorem provers turn out to be very inefficient due to their generality. Thus, they have only rarely been used for solving real-life problems if problem-specific heuristics have been available to guide the proof search.

These observations show that usually declarativity and efficiency compete each other. Therefore, one may start to search for a compromise between these counterparts: *declarative representation* on the one hand and *efficient problem-solving* on the other hand: Coming from the *procedural* representation side, higher-level programming languages can be used to build abstractions by hiding low-level details and to support the encapsulation of software modules. This approach obviously increases the declarativity—for the cost of efficiency. On the other side, restricting oneself from first-order logic to Horn logic, more efficient theorem provers from logic programming, i.e. PROLOG systems, can be used. However, PROLOG does also introduce some constructs that violate the declarative approach, e.g. the *cut operator* and the *negation as failure* principle. Hence, by using PROLOG instead of a general theorem prover, we increase efficiency—for the cost of declarativity. Thus, it seems that all approaches, i.e. programming paradigms, can be qualitatively drawn on a convex curve as shown in Figure 1.7.

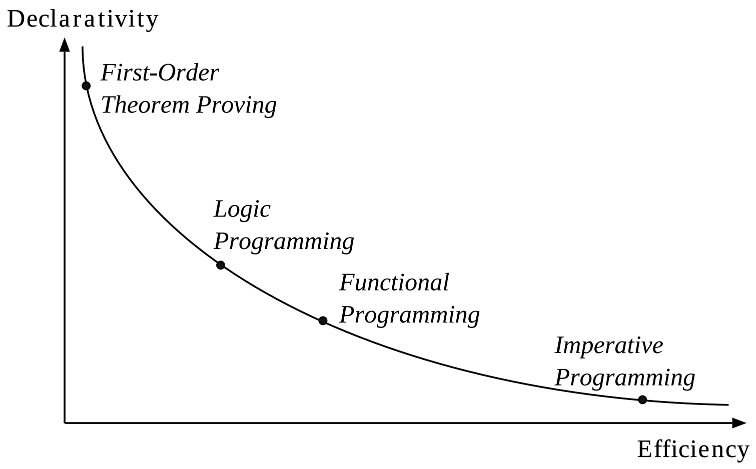


Figure 1.7: Declarativity competes Efficiency

1.3 The Role of Finite Domain Constraints

For combinatorial problems like the scene labeling problem, we recognize that declarativity and efficiency do not necessarily compete. Instead, representing these problems as finite domain constraint satisfaction problems allows to preserve a very declarative representation while at the same time gaining an efficient problem solving. This is especially important as constraint satisfaction problems belong to the class of $\mathcal{NP}$-*complete* problems, meaning that no algorithm for solving them in acceptable time[4] has been found, and it is most unlikely, according to accepted mathematical conjecture $\mathcal{NP} \neq \mathcal{P}$, that any such algorithm will ever emerge. The only theoretical way of solving an $\mathcal{NP}$-*complete* problem is to enumerate all possible solutions and wait for an acceptable one to appear. This exhaustive exploration of the search space can, in the worst case, take an amount of time that increases *exponentially* with the size of the data. This implies that we should give up any hope of tackling these problems exhaustively.

Fortunately, we do **not** need to give up hope because the worst case (which requires exponential time) is not likely to appear in reality. This is because for finite domain CSPs we can make use of the specific characteristics of each problem to restrict as much as possible the number of potential solutions that need to be exhaustively explored. In the sense of the above interpretation of the term 'efficiency' this directly improves the efficiency of the approach. The techniques for doing this are basically *constraint propagation* techniques which feature flexibility and data-driven computation in order to avoid treating all cases as the worst case. Thus, for finite domain constraints we have a situation like that shown in Figure 1.8.

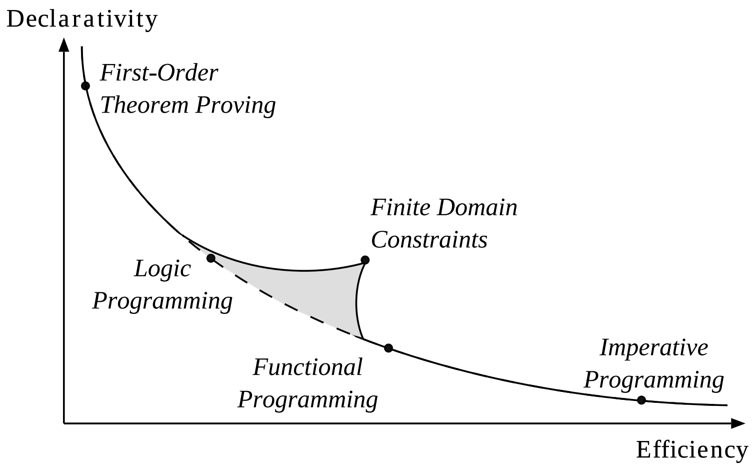


Figure 1.8: The Role of Finite Domain Constraints

This means that for problems which can be represented as FCSPs we have techniques at hand—basically, constraint propagation—that allow for a problem representation at least as declarative as when using a purely relational representation (e.g., pure PROLOG) but at the same time offering much better problem solving efficiency.

[4]The term *acceptable time* is taken to mean that the time to find a solution is no worse than a polynomial function of the size of the data, as opposed to, say, an exponential function.

However, the majority of real-life problems cannot be directly represented as a CSP. Therefore, we have first to investigate how constraint satisfaction techniques can be extended in order to become applicable to a wider class of problems, and second, how we can still achieve both declarative representation and efficient problem solving even for this wider class of problems. The present thesis contributes to both of these goals.

1.4 Outline of the Thesis

Being faced with various real-life application problems at DFKI, we recognized that although these problems can in some sense be regarded as 'finite domain constraint satisfaction problems', current CSP techniques turned out either to be not applicable at all or to be too weak to solve the given problems efficiently.

Thus, in this thesis we focus on developing extensions to current CSP techniques in order to cope with the characteristics of the problems at hand. Although the extensions presented are motivated by concrete applications, throughout the first part we will use a simple but rich enough map-coloring example to illustrate the key problems and how to extend the framework in order to solve them.

In Chapter 2 we will first formally introduce the constraint satisfaction problem and set up a logical framework for the discussion of constraint satisfaction techniques that also helps to clarify the relationship to other representation and reasoning systems.

Chapter 3 will then introduce and discuss constraint propagation techniques and present an algorithm for efficiently solving general CSPs which constitutes the kernel of the CONTAX system. Motivated by incrementally extending the map-coloring example, we will then present two main extensions to the classical CSP framework, that have also been implemented in the CONTAX system:

Most real-life problems cannot be solved optimally. Regarding such problems as constraint satisfaction problems, this means that no formally consistent solution can be found. However, humans do solve such problems every-day by 'computing' the 'best sub-optimal solution. The key issue is that normally not all given constraints have to be satisfied with the same strength: some definitely need to be fulfilled, others can be omitted if necessary. Therefore, in Chapter 4 we will extend the classical CSP framework by introducing weighted constraints which will then allow us to relax a given CSP as much as needed in order to find the 'best sub-optimal solution'.

Moreover, for most real-life problems, especially in technical applications, the domains (classes) we are dealing with can be arranged hierarchically. As the constraints are also mostly dealing with entire sub-classes instead of individual elements, constraint propagation techniques should also be able to perform propagation on this more abstract level of domain subsets. Therefore, in Chapter 5 we will extend the classical CSP framework by introducing hierarchical consistency-techniques.

Having extended our CSP framework by allowing for weighted constraints and hierarchical domains, we will be able to both declaratively represent and efficiently solve a much larger class of constraint satisfaction problems. However, again motivated by the need to go beyond isolated toy problems and to tackle real-life problems, we soon recognize that only few application problems can be completely mapped to a CSP. A large part of any application is dealing with, e.g. user interfaces, I/O, arithmetics, and overall control. Thus, the constraint solver will either have to be linked to an application program written e.g. in Lisp via an *interface*, or be embedded in a universal relational programming language like e.g. Prolog. Therefore, in Chapter 6 we will discuss how to integrate finite domain constraint satisfaction techniques into Prolog. Obviously, the resulting FiDo language combines and thus enhances the declarative representation features of both constraint satisfaction and logic programming. However, special care has to be taken not to loose the efficiency gained for each of these approaches separately. Thus, we will also study different implementation approaches for FiDo in detail.

For improving the efficiency of any computer application, one approach that has received increasing attention over the last years, is to split a given problem into independent subtasks which can be solved in parallel, and then to construct the final solution by combining the results of the subtasks. Therefore, in Chapter 7 we will finally discuss how to parallelize constraint satisfaction within the FiDo system.

Figure 1.9 summarizes the research issues to be tackled in this thesis and how they contribute to improving declarativity and efficiency of finite domain constraint solving.

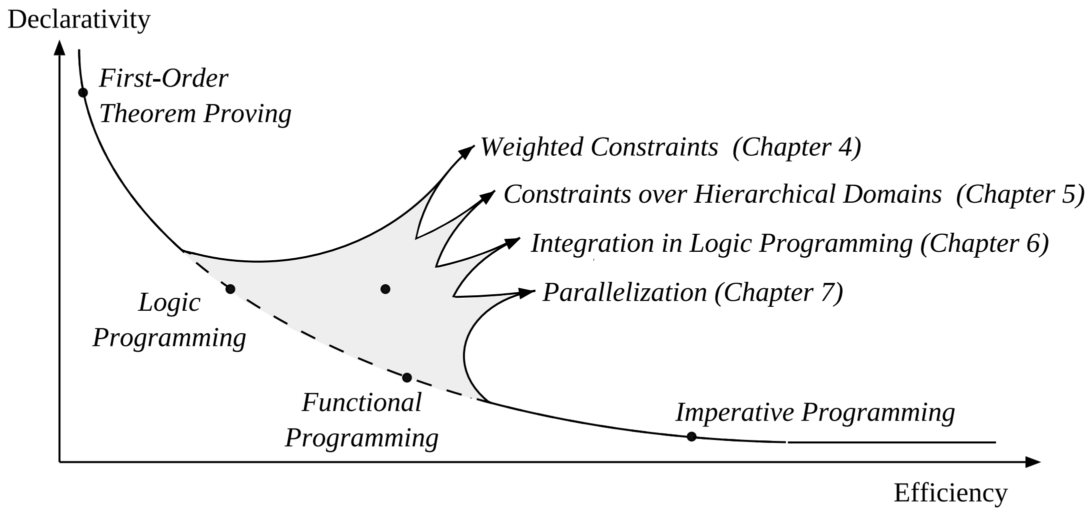

Declarativity
First-Order Theorem Proving
Weighted Constraints (Chapter 4)
Constraints over Hierarchical Domains (Chapter 5)
Integration in Logic Programming (Chapter 6)
Parallelization (Chapter 7)
Logic Programming
Functional Programming
Imperative Programming
Efficiency

Figure 1.9: Contributions to Finite Domain Constraints

Summarizing, the techniques that will be presented in this thesis enable us to solve a wider class of problems, allow for a declarative problem representation, and can be implemented very efficiently. Thus, we will show that for a significant and increasing class of application problems improving both declarativity and efficiency is not a contradiction and hence we have:

Declarativity meets Efficiency.

While all these more theoretical issues will be illustrated using one simple to understand, but nevertheless rich enough 'toy example', we will show in Chapter 8 how the theory presented in the previous chapters can be used to solve real-life application problems in different domains, e.g. computer-aided process planning and university sports planning and will thus show that:

Theory meets Application.

2

Basic Concepts and Views on Constraint Satisfaction

Applying constraint satisfaction techniques for solving a combinatorial problem requires two tasks to be performed: First, the problem must be formulated in terms of a constraint satisfaction problem (CSP). Once this has been done we can run appropriate constraint-solving techniques on the CSP in order to obtain a solution for the given problem.

In this chapter, we will first focus on CSP *representation* before we will discuss techniques for *solving* them. Section 2.1 will introduce the constraint satisfaction problem as a *declarative problem formulation* according to [Amarel, 1970] and will present some useful terminology which enables us to give a formal definition of the constraint satisfaction problem in general and what is meant by solving a CSP. We will then focus on finite constraint satisfaction problems (FCSPs), i.e. CSPs whose variable range over finite domains, because, as we will see later, for solving FCSPs efficient algorithms can be developed which exploit these features. In Section 2.2 we will take a logical view on the finite constraint satisfaction problem and will present the FCSP framework as a highly restricted logical calculus within a space of logical representation and reasoning systems. Establishing connections amongst the different logical views of constraint satisfaction problems will allow algorithms and results from these desperate areas to be imported, and specialized, to FCSPs. Finally, in Section 2.3 we will then focus on the aspect of *efficiently solving* an FCSP, present a generic search algorithm for FCSPs, and will discuss directions towards improving its efficiency which have been realized in the CONTAX and FIDO systems to be presented in the next chapters.

The formal framework and most algorithms will be illustrated using the following instance of the map-coloring problem as a simple and easy to understand example throughout the next chapters:

Problem 1 (German-map coloring problem) *The German-map coloring problem is to decide how to color the political map of West Germany*[1] *as shown in Figure 2.1 using only three colors:* red, blue, *and* green. *Each state should be colored differently from its neighbours, and some colors have already been fixed: Hessen should be colored* green *while Hamburg, Bremen and Saarland should be* red.

[1]For the moment we will restrict ourselves to coloring the political map of the Federal Republic of Germany as of before the re-unification in 1990. We will later extend and 'update' this example by including the new (eastern) states (cf. Chapter 4).

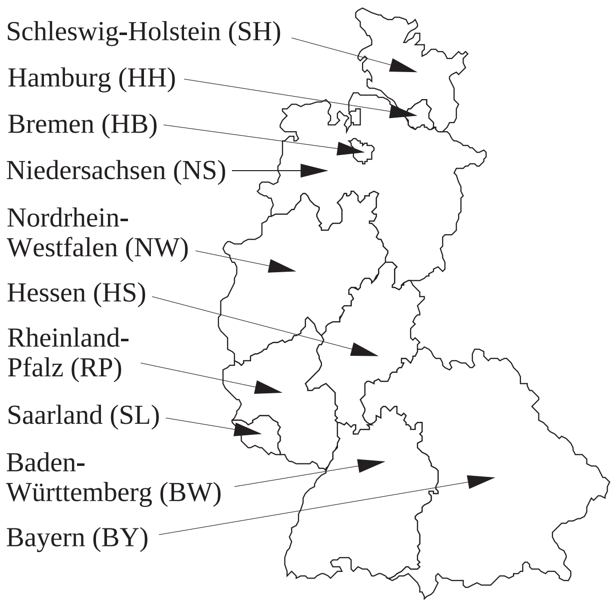


Figure 2.1: An example FCSP: coloring the political map of Germany

This problem seems quite trivial and its solution requires only little thought, but it serves our purpose here. It will later be extended and used to motivate and illustrate extensions of our framework towards constraint relaxation methods (cf. Chapter 4) and techniques for efficiently dealing with hierarchically structured domains (cf. Chapter 5).

2.1 Formulating Constraint Satisfaction Problems

Problem solving is a central phenomenon studied in AI. It is a process that involves finding, or constructing, a solution to a given problem. The idea of using a purely declarative formulation when posing a given problem to a problem-solving system dates back to the early days of AI research. In [Amarel, 1970] we can already find the following definition of a declarative problem formulation:

Definition 1 (declarative problem formulation [Amarel, 1970]) *A* declarative problem formulation *takes the following general form: "Given a domain specification D, find a solution x such that x is a member of a set of possible solutions X and it satisfies the problem conditions C."*

This kind of a declarative problem formulation is very close to the formulation of a constraint satisfaction problem (CSP). The following table shows that the notions used in the *declarative problem formulation* can be easily mapped to terms in a *CSP formulation*:

domain specification (D)	=	set of possible values for the variables
set of possible solutions (X)	=	set of all value combinations for the variables
problem conditions (C)	=	set of constraints

This mapping also shows that formalizing a problem as a CSP yields a very declarative representation. In the following, we will now more formally discuss the CSP formulation. Therefore, we will first introduce some basic concepts that are needed for a formal definition of the constraint satisfaction problem.

2.1.1 Variables and domains

When representing a given problem as a CSP, the objects of the problem statement can be represented as variables. A variable in a CSP ranges over a set of possible values that will be referred to as its domain:

Definition 2 (variable domain) *The* domain *of a variable is a set of all possible values that can be assigned to the variable. If X is a variable, then we use D_X to denote the domain of X.*

When the domain contains numbers only, the variable is called *numerical variable*. The domain of a numerical variable may be further restricted to integers, rational numbers or real numbers. For example, the domain of an integer variable is an infinite set {1,2,3,...}. When the domain contains boolean values only, the variable is called *boolean variable*; when it contains values from an enumerated type of objects, the variable is called *symbolic variable*.

Example 1 (variable domain) *The color assigned to a state in the map-coloring problem can be represented in a CSP formulation as a symbolic variable of which the domain is the finite set* {red, green, blue}.

2.1.2 Assigning values to variables

We will now introduce the concept of *labels* that represent the assignment of a value to a variable.

Definition 3 (label) *A* label *is a variable-value pair that represents the assignment of a value to a variable. We use $\langle X \leftarrow v \rangle$ to denote the label of assigning the value v to the variable X. $\langle X \leftarrow v \rangle$ is only meaningful if v is in the domain of X, i.e. $v \in D_X$.*

Definition 4 (compound label) *A* compound label *is the simultaneous assignment of values to a set of variables. We use $(\langle X_1 \leftarrow v_1 \rangle, \langle X_2 \leftarrow v_2 \rangle, \ldots, \langle X_n \leftarrow v_n \rangle)$ to denote the compound label of assigning v_1, v_2, ..., and v_n to X_1, X_2, ..., and X_n respectively.*

Since compound labels are regarded as sets, the ordering of the labels in this representation is insignificant.

Definition 5 (k-compound label) *A* k-compound label *is a compound label which assigns k values to k variables simultaneously.*

Definition 6 (projection) *If m and n are integers such that $m \leq n$, then an m-compound label $M = (\langle X_1 \leftarrow v_1\rangle,\ldots,\langle X_m \leftarrow v_m\rangle)$ is a* projection *of an n-compound label $N = (\langle Y_1 \leftarrow w_1\rangle,\ldots,\langle Y_n \leftarrow w_n\rangle)$, written as* projection(N,M)*, if the labels in M all appear in N:*

$$\forall M, N : M = (\langle X_1 \leftarrow v_1\rangle,\ldots,\langle X_m \leftarrow v_m\rangle),\ N = (\langle Y_1 \leftarrow w_1\rangle,\ldots,\langle Y_n \leftarrow w_n\rangle) :$$
$$\mathsf{projection}(N,M) \equiv \{\langle X_1 \leftarrow v_1\rangle,\ldots,\langle X_m \leftarrow v_m\rangle\} \subseteq \{\langle Y_1 \leftarrow w_1\rangle,\ldots,\langle Y_n \leftarrow w_n\rangle\}$$

Example 2 (projection) *Let* HS, HH *and* HB *be variables in the German-map coloring problem. Then the compound label* (⟨HS ← green⟩,⟨HB ← red⟩) *is a projection of the compound label* (⟨HS ← green⟩,⟨HH ← red⟩,⟨HB ← red⟩)*, which means the proposition* projection((⟨HS ← green⟩,⟨HH ← red⟩,⟨HB ← red⟩), (⟨HS ← green⟩,⟨HB ← red⟩)) *is true.*

Definition 7 (variables of a compound label) *The set of* variables of a compound label L*, written as* vars*(L), is the set of all variables which appear in that compound label:*

$$\mathsf{vars}((\langle X_1 \leftarrow v_1\rangle,\langle X_2 \leftarrow v_2\rangle,\ldots,\langle X_n \leftarrow v_n\rangle)) \equiv \{X_1, \ldots, X_n\}$$

2.1.3 Constraints

The *problem conditions* in the problem statement of [Amarel, 1970] can be formulated as constraints in a CSP. A constraint on a set of variables is a restriction on the values that they can take simultaneously.

A constraint can be represented in a number of ways. Constraints on numerical variables can be represented by equations or inequalities; for example, a binary constraint $C_{\{X,Y\}}$ may be $X + Y < 10$. Conceptually, a constraint can be seen as the set of all legal compound labels for the subject variables, showing the values that are mutually compatible for the set of variables. This representation is taken here as it helps to formulate the concept of *constraint relaxation* where relaxing a constraint means adding elements to the set of legal compound labels. Alternatively, a constraint may also be viewed as a function which maps every compound label on the subject variables to true or false; or it can be seen simply as a relation on the subject variables. This logical representation is taken in Section 2.2 when viewing constraint satisfaction as a logical representation and reasoning framework. However, the choice of representation does not affect the generality of our discussions.

Definition 8 (constraint) *Let $V = \{X_1, \ldots, X_n\}$ be a finite set of variables with values from their domains $D_{X_1}, \ldots, D_{X_n}$. A* constraint *C on the set of variables V denotes a subset of the Cartesian product of the domains, i.e. $C \subseteq D_{X_1} \times \ldots \times D_{X_n}$, represented as a set $C_{\{X_1,\ldots,X_n\}}$ of compound labels for the variables $X_1, \ldots, X_n$. It is also identified with the formula $\mathsf{p}_{X_1,\ldots,X_n}(X_1, \ldots, X_n)$ which is satisfied by a tuple $\bar{v} = (v_1, \ldots, v_n)$ with $v_i \in D_{X_i}, 1 \leq i \leq n$ iff $\bar{v} \in C$, or $(\langle X_1 \leftarrow v_1\rangle,\ldots,\langle X_n \leftarrow v_n\rangle) \in C_{\{X_1,\ldots,X_n\}}$, respectively.*

Example 3 (constraint) *Let $V = \{\mathsf{NS}, \mathsf{SH}\}$ be a set of variables in the German-map coloring problem representing the colors assigned to Niedersachsen and Schleswig-Holstein, respectively. The domain of both variables is the finite set* $\{\mathsf{red}, \mathsf{green}, \mathsf{blue}\}$.

As the states Niedersachsen and Schleswig-Holstein have a common border (cf. Figure 2.1), the colors assigned to the variables NS *and* SH *have to be different. This constraint* $\mathsf{p}_{NS,SH}(\mathsf{NS},\mathsf{SH}) \equiv \mathsf{NS} \neq \mathsf{SH}$ *is represented by the following set of compound labels* $C_{\{\mathsf{NS},\mathsf{SH}\}} = \{\ (\langle\mathsf{NS} \leftarrow \mathsf{green}\rangle,\langle\mathsf{SH} \leftarrow \mathsf{red}\rangle),\ (\langle\mathsf{NS} \leftarrow \mathsf{green}\rangle,\langle\mathsf{SH} \leftarrow \mathsf{blue}\rangle),\ (\langle\mathsf{NS} \leftarrow \mathsf{red}\rangle,\langle\mathsf{SH} \leftarrow \mathsf{green}\rangle),\ (\langle\mathsf{NS} \leftarrow \mathsf{red}\rangle,\langle\mathsf{SH} \leftarrow \mathsf{blue}\rangle),\ (\langle\mathsf{NS} \leftarrow \mathsf{blue}\rangle,\langle\mathsf{SH} \leftarrow \mathsf{red}\rangle),\ (\langle\mathsf{NS} \leftarrow \mathsf{blue}\rangle,\langle\mathsf{SH} \leftarrow \mathsf{green}\rangle)\ \}$ *which enumerates all legal combinations of colors for the two states.*

Definition 9 (variables of a constraint, arity) *The* variables *of a constraint* C*, written as* $\mathsf{vars}(C)$*, are the variables of the members of the constraint* C*:*

$$\mathsf{vars}(C_{\{X_1,\ldots,X_n\}}) \equiv \{X_1,\ldots,X_n\}$$

The arity *of a constraint* C *is the cardinality of the set* $\mathsf{vars}(C)$*.*

A constraint $C_{\{X_1,X_2,\ldots,X_k\}}$ restricts the set of compound labels that X_1, X_2, ..., and X_k can take simultaneously. Therefore, taking $k = 1$ allows to represent the domain D_X of a variable X as a unary constraint $C_{\{X\}}$ on X.[2]

Example 4 (domains as unary constraints) *In the German-map coloring problem the variable* NS *can only take the values* red*,* green*, and* blue*, i.e.* $D_{\mathsf{NS}} = \{\mathsf{red},\mathsf{green},\mathsf{blue}\}$*. We can now represent the domain of* NS *also as a unary constraint* $C_{\{\mathsf{NS}\}} = \{(\langle\mathsf{NS} \leftarrow \mathsf{red}\rangle),\ (\langle\mathsf{NS} \leftarrow \mathsf{green}\rangle),\ (\langle\mathsf{NS} \leftarrow \mathsf{blue}\rangle)\}$*.*

2.1.4 Satisfying a constraint

Compound labels represent value assignments to a set of variables. As the values that a variable may take are restricted by constraints, we need a notion to express whether a compound label satisfies a constraint. We shall define a binary relationship satisfies between a compound label and a constraint:

Definition 10 (satisfies, for compound labels) *Let* $L = (\langle X_1 \leftarrow v_1\rangle,\langle X_2 \leftarrow v_2\rangle,\ldots,\langle X_n \leftarrow v_n\rangle)$ *be a compound label and* $C_{\{X_1,X_2,\ldots,X_n\}}$ *be a constraint. Then,* L satisfies C *if and only if* L *is an element of* C*.*

$$\mathsf{satisfies}(L, C_{\{X_1,X_2,\ldots,X_n\}}) \equiv L \in C_{\{X_1,X_2,\ldots,X_n\}}$$

For convenience, satisfies is also defined between labels and unary constraints:

Definition 11 (satisfies, for labels) $\mathsf{satisfies}(\langle X \leftarrow v\rangle, C_{\{X\}}) \equiv (\langle X \leftarrow v\rangle) \in C_{\{X\}})$

This allows us to write $\mathsf{satisfies}(\langle X \leftarrow v\rangle, C_{\{X\}})$ as well as $\mathsf{satisfies}((\langle X \leftarrow v\rangle), C_{\{X\}})$. Following [Freuder, 1978], the concept of $\mathsf{satisfies}(L, C)$ is extended to the case when C is a constraint on a subset of the variables of the compound label L.

[2]Note the difference between $C_{\{X\}}$ and D_X: $C_{\{X\}}$ is a set of labels while D_X is a set of values.

Definition 12 (satisfies) *Given a compound label L and a constraint C such that the variables of C are a subset of the variables of L, the* compound label L satisfies constraint C *if and only if the projection of L onto the variables of C is an element of C.*

$$\begin{aligned}&\forall X_1, X_2, \ldots, X_k : \forall v_1 \in D_{X_1}, v_2 \in D_{X_2}, \ldots, v_k \in D_{X_k} :\\ &\quad \forall S \subseteq \{X_1, X_2, \ldots, X_k\} :\\ &\quad\quad \mathsf{satisfies}((\langle X_1 \leftarrow v_1\rangle, \langle X_2 \leftarrow v_2\rangle, \ldots, \langle X_k \leftarrow v_k\rangle), C_S)\\ &\quad\quad\quad \equiv (\exists cl \in C_S : \mathsf{projection}((\langle X_1 \leftarrow v_1\rangle, \langle X_2 \leftarrow v_2\rangle, \ldots, \langle X_k \leftarrow v_k\rangle), cl))\end{aligned}$$

In other words, when C is a constraint on the variables $\{X_1, X_2, \ldots, X_k\}$ or its subset, then L satisfies C if the labels for those variables in L are legal as far as C is concerned.

Example 5 (satisfies) *Consider the compound label* $L = (\langle \mathsf{HS} \leftarrow \mathsf{green}\rangle, \langle \mathsf{NS} \leftarrow \mathsf{blue}\rangle, \langle \mathsf{SH} \leftarrow \mathsf{red}\rangle)$. *L satisfies the constraint* $C_{\{\mathsf{NS},\mathsf{SH}\}}$ *if and only if* $(\langle \mathsf{NS} \leftarrow \mathsf{blue}\rangle, \langle \mathsf{SH} \leftarrow \mathsf{red}\rangle)$ *is a member of* $C_{\{\mathsf{NS},\mathsf{SH}\}}$.

2.1.5 Formal definition of the constraint satisfaction problem

With the terminology introduced in the previous sections we are now able to give a formal definition of the constraint satisfaction problem.

Definition 13 (constraint satisfaction problem) *A* constraint satisfaction problem *is a triple (V,δ,C) where*

V is a finite set of variables $\{X_1, X_2, \ldots, X_n\}$,

δ is a function which maps every variable in V to a set of objects (possible values). The set $\delta(X_i)$ *is denoted by* D_{X_i} *and called the* domain *of* X_i.

C is a finite set of constraints *on an arbitrary subset of variables in V. In other words, C is a set of sets of compound labels.*

In the literature, the constraint satisfaction problem is often defined in a stronger sense where the domains of all variables are required to be discrete finite sets [Tsang, 1993]. Although we will also focus on problems conforming to this stronger definition, we shall introduce this subclass of the general CSP separately as *finite constraint satisfaction problem* (FCSP). This allows to talk about CSPs and FCSPs separately, as CSPs not conforming to this stronger definition, e.g. by also containing numerical variables, require different techniques than can be applied to FCSPs.

Definition 14 (finite constraint satisfaction problem) *Let* $\mathcal{P} = (V, \delta, C)$ *be a CSP. If the domain* D_{X_i} *of each variable* $X_i \in V$ *is finite, then* $\mathcal{P}$ *is called a* finite constraint satisfaction problem *(FCSP).*

Example 6 (finite constraint satisfaction problem) *As there are a finite number of states (variables), a finite number of borders between them (constraints), and a finite number of colors (values in the domains) to be used, the German-map coloring problem is a finite constraint satisfaction problem.*

So far we have now defined how constraint satisfaction problems can be *represented*. The task in *solving* a CSP then is to assign a value to each variable such that all the constraints are satisfied simultaneously. To express that the resulting compound label is a solution for the given CSP, we can formally define the relationship $\mathsf{solution}$ as follows.

Definition 15 (solution of a CSP) *Let $\mathcal{P} = (V, \delta, C)$ be a CSP on the variables $V = \{X_1, X_2, \ldots, X_n\}$. A* solution *of $\mathcal{P}$ is an n-compound label for all variables in V which satisfies all the constraints in C:*

$$\begin{aligned}
&\forall V, \delta, C : \forall X_1, X_2, \ldots, X_n \in V : \forall v_1 \in D_{X_1}, v_2 \in D_{X_2}, \ldots, v_n \in D_{X_n} : \\
&\quad \mathsf{solution}((\langle X_1 \leftarrow v_1\rangle, \langle X_2 \leftarrow v_2\rangle, \ldots, \langle X_n \leftarrow v_n\rangle), (V, \delta, C)) \\
&\qquad \equiv ((V = \{X_1, X_2, \ldots, X_n\}) \quad \wedge \\
&\qquad\quad (\forall c \in C : \mathsf{satisfies}((\langle X_1 \leftarrow v_1\rangle, \langle X_2 \leftarrow v_2\rangle, \ldots, \langle X_n \leftarrow v_n\rangle), c)))
\end{aligned}$$

A CSP is *satisfiable* if a solution exists. Depending on the requirements of an application, there can be different meanings of how many and which solutions are required: In some CSPs one has to find *any* solution, in other CSPs one is interested in finding *all* solutions, and in a third class of CSPs one may have to find the *optimal* solution, where optimality is defined according to some domain knowledge, and from a more theoretical point of view one may only be interested in the *CSP decision problem*, i.e. in determining whether the given CSP is satisfiable at all.

2.1.6 Binary constraint satisfaction problems

Since a lot of research focuses on problems with unary and binary constraints only, we define the term *binary CSP* for proper reference.

Definition 16 (binary CSP, general CSP) *A* binary constraint satisfaction problem*, or* binary CSP*, is a CSP with unary and binary constraints only. A CSP with constraints not limited to unary and binary will be referred to as a* general CSP*.*

Example 7 (binary CSP) *The German-map coloring problem is a binary CSP as the only type of constraint in this problem, the inequality of the colors assigned to neighboring states, is a binary constraint.*

We can show that any general CSP can be transformed into a an equivalent binary CSP preserving the set of solutions of the general CSP.

Proposition 1 *Let $P_g = (V, \delta, C)$ be a general CSP. Then P_g can be transformed into a binary CSP P_b having the same set of solutions concerning the variables V of the general CSP, i.e. $\{s_g \mid \mathsf{solution}(s_g, P_g)\} = \{l_b \mid \mathsf{solution}(s_b, P_b) \wedge \mathsf{projection}(s_b, l_b) \wedge \mathsf{vars}(l_b) = V\}$.*

Proof. Each k-ary constraint $C_{\{X_1, X_2, \ldots, X_k\}} \in C$ can be replaced by a new variable W and k binary constraints as follows. The domain of W is the set of all compound labels in $C_{\{X_1, X_2, \ldots, X_k\}}$, i.e. $D_W = C_{\{X_1, X_2, \ldots, X_k\}}$. Each of the k newly created binary constraints connects W and one of the k variables $X_1, \ldots, X_k$; the constraint $C_{\{W, X_i\}}$ which connects W and a variable X_i requires X_i to take a value which is projected from some compound label in the domain of W, i.e. $C_{\{W,X_i\}} = \{(\langle W \leftarrow v_W\rangle, \langle X_i \leftarrow v_{X_i}\rangle) \mid \mathsf{projection}(v_W, (\langle X_i \leftarrow v_{X_i}\rangle))\}$. Thus, a $(k+1)$-compound label L for the variables W, X_1, X_2, …, and X_k satisfies all the constraints $C_{\{W,X_i\}}$, $1 \leq i \leq k$, if and only if its projection on the variables X_1 to X_k satisfies the constraint $C_{\{X_1,X_2,\ldots,X_k\}}$. □

This result will enable us to focus our theoretical considerations on binary CSPs, or binary representations of CSPs. However, it should be noted that by removing k-ary constraints for all $k > 2$, we introduce new variables which have large domains. Therefore, although from the theoretical point of view it makes no difference, for practical applications we will have to investigate and implement CSP solving techniques that are able to solve general CSPs directly without transforming them into binary ones.

2.2 A Logical View on Constraint Satisfaction

Since its early days, the formulation of the constraint satisfaction paradigm has yielded substantial theoretical and practical results [Dechter, 1992, Mackworth, 1992a]. However, it is important not to conceive of the constraint satisfaction paradigm in isolation but to see it in its proper context, namely, as a highly restricted logical calculus with associated properties and algorithms. In this section we shall place the constraint satisfaction paradigm in that context and will establish connections amongst the different logical views of constraint satisfaction problems.

We will present the FCSP framework as a restricted logical calculus within a space of logical representation and reasoning systems. The FCSP will be formulated in a variety of logical settings: theorem proving in first order predicate calculus, propositional theorem proving, the Prolog and Datalog approaches, constraint network algorithms, a logical interpreter for networks of constraints, and the constraint logic programming (CLP) paradigm.

2.2.1 Finite constraint satisfaction as theorem proving

For studying FCSPs from the theorem proving point of view, we will regard variable domains as unary constraints (cf. Section 2.1.3). Extending the given constraints in an FCSP by unary constraints for the variable domains is covered by the following definition:

Definition 17 (extended constraints set) *Let $\mathcal{P} = (V, \delta, C)$ be an FCSP. Then the extended set of constraints, denoted by* extension(C)*, is defined as*

$$C \cup \{C_{\{X_i\}} = \{(\langle X_i \leftarrow v_{i_1}\rangle), \ldots, (\langle X_i \leftarrow v_{i_k}\rangle)\} \mid 1 \leq i \leq n, v_{i_j} \in D_{X_i}\}$$

Solving a finite constraint satisfaction problem can now be formulated as theorem proving in a restricted form of first-order predicate calculus as follows [Mackworth, 1977, Bibel, 1988]. Consider a finite constraint satisfaction problem $\mathcal{P} = (V, \delta, C)$ with $V = \{X_1, X_2, \ldots, X_n\}$. The decision problem for $\mathcal{P}$ is equivalent to determining if $\mathsf{Constraints} \vdash \mathsf{Query}$ where Query has the form

$$\exists X_1 \, \exists X_2 \, \cdots \, \exists X_n \; \mathsf{QueryMatrix}(X_1, X_2, \ldots, X_n)$$

with QueryMatrix combining all formulas (identified with constraints) over subsets of $\{X_1, X_2, \ldots, X_n\}$

$$\begin{aligned}
\mathsf{QueryMatrix}(X_1, \ldots, X_n) &= \bigwedge_{\substack{\{X_{i_1}, \ldots, X_{i_k}\} \\ \subseteq \{X_1, \ldots, X_n\}}} \left\{ \mathsf{p}_{X_{i_1}, \ldots, X_{i_k}}(X_{i_1}, \ldots, X_{i_k}) \,\middle|\, \begin{array}{l} C_{\{X_{i_1}, \ldots, X_{i_k}\}} \\ \in \mathsf{extension}(C) \end{array} \right\} \\
&= \mathsf{p}_{X_1}(X_1) \wedge \mathsf{p}_{X_2}(X_2) \wedge \cdots \wedge \mathsf{p}_{X_n}(X_n) \wedge \\
&\quad\ \mathsf{p}_{X_1,X_2}(X_1, X_2) \wedge \mathsf{p}_{X_1,X_3}(X_1, X_3) \wedge \cdots \wedge \\
&\quad\ \mathsf{p}_{X_1,X_2,X_3}(X_1, X_2, X_3) \wedge \cdots \wedge \\
&\quad\ \vdots \\
&\quad\ \mathsf{p}_{X_1,X_2,X_3,\ldots,X_n}(X_1, X_2, X_3, \ldots, X_n)
\end{aligned}$$

and Constraints being a set of ground atoms specifying the extensions of the constraints

$$\mathsf{Constraints} = \left\{ \mathsf{p}_{X_{i_1} \ldots X_{i_m}}(c_{i_1}, \ldots, c_{i_m}) \,\middle|\, \begin{array}{l} 1 \le i_k \le n, 1 \le k \le n, \\ C_{\{X_{i_1}, \ldots, X_{i_m}\}} \in \mathsf{extension}(C), \\ \mathsf{satisfies}((\langle X_{i_1} \leftarrow c_{i_1} \rangle, \ldots, \langle X_{i_m} \leftarrow c_{i_m} \rangle), \\ \qquad\qquad C_{\{X_{i_1}, \ldots, X_{i_m}\}}) \end{array} \right\}$$

where the c_i are constants. Notice that as in the CSP formulation of Section 2.1, in this formulation we are only specifying the value combinations allowed by a relation, not the combinations forbidden, since Constraints consists of positive literals.

2.2.2 Finite constraint satisfaction decision problems

A finite constraint satisfaction problem can now be specified by a (Constraints,Query) pair. A common candidate formulation of the decision problem for FCSPs is to determine whether it can be shown that a solution exists (i.e., Constraints $\vdash$ Query) or does not exist (i.e., Constraints $\vdash \neg$Query). However, given the positive form specified for Constraints, it is never possible to establish that Constraints $\vdash \neg$Query, so this candidate formulation is not acceptable. Later, when we consider the completion of Constraints, we shall return to a variant of this formulation.

Definition 18 (finite constraint satisfaction decision problem) *Given an FCSP specified by the pair (*Constraints,Query*). The finite constraint satisfaction decision problem (FCSDP) then is to determine if it can be shown that a solution exists or if it cannot be shown that a solution exists:* Constraints $\vdash$ Query *or* Constraints $\not\vdash$ Query.

If the decision problem is posed in the form of an FCSDP and the constraints are supplied or discovered incrementally in the form of additional allowed tuples, extending the set Constraints, then the answers to FCSDP are monotonic: a "No" may change to "Yes" but not vice versa.

Proposition 2 (Decidability of the FCSDP) *Given a finite constraint satisfaction problem specified by the pair* (Constraints, Query)*, then the FCSDP is decidable, i.e. it can be determined whether* Constraints $\vdash$ Query *or* Constraints $\not\vdash$ Query.

Proof. The Herbrand universe H of the theory $\mathsf{Constraints} \cup \{\neg\mathsf{Query}\}$ is the set $H = \{c \mid \mathsf{p}_V(\ldots, c, \ldots) \in \mathsf{Constraints}\}$. H is finite. Now consider the following algorithm

```
algorithm FCSDA:
Success ← No
for each $(c_1, c_2, \ldots, c_n) \in H^n$ do
    if Constraints $\vdash$ QueryMatrix$(c_1, c_2, \ldots, c_n)$ then
        Success ← Yes
    endif
endfor
return Success
```

Figure 2.2: A naive decision algorithm for the FCSDP

where $\mathsf{Constraints} \vdash \mathsf{QueryMatrix}(c_1, c_2, \ldots, c_n)$ holds iff for each Atom mentioned in $\mathsf{QueryMatrix}(c_1, c_2, \ldots, c_n)$ it is the case that $\mathsf{Atom} \in \mathsf{Constraints}$. The algorithm FCSDA always terminates. It returns "Yes" iff $\mathsf{Constraints} \vdash \mathsf{Query}$ and returns "No" iff $\mathsf{Constraints} \not\vdash \mathsf{Query}$. □

The number of predicate evaluations made by FCSDA is equal to the product of the number of atoms in $\mathsf{QueryMatrix}(c_1, c_2, \ldots, c_n)$ and the cardinality of H^n.

Example 8 (German-map coloring problem in first-order logic) *By using the FCSP formalism presented in Section 2.2.1 we can formulate the map-coloring problem as follows: Introducing variables* SH, HH, NS, HB, NW, HS, RP, SL, BW, *and* BY *for the colors assigned to the different states as shown in Figure 2.1, we obtain the* Query

$$\begin{aligned}\exists \mathsf{SH}\,\exists \mathsf{HH} \cdots \exists \mathsf{BW}\,\exists \mathsf{BY} \quad & \mathsf{p}_{SH}(\mathsf{SH}) \wedge \mathsf{p}_{HH}(\mathsf{HH}) \wedge \cdots \wedge \mathsf{p}_{BW}(\mathsf{BW}) \wedge \mathsf{p}_{BY}(\mathsf{BY}) \wedge \\ & \neq(\mathsf{SH},\mathsf{HH}) \wedge \neq(\mathsf{SH},\mathsf{NS}) \wedge \neq(\mathsf{HH},\mathsf{NS}) \wedge \neq(\mathsf{NS},\mathsf{HB}) \wedge \\ & \neq(\mathsf{NS},\mathsf{NW}) \wedge \neq(\mathsf{NS},\mathsf{HS}) \wedge \neq(\mathsf{NW},\mathsf{HS}) \wedge \neq(\mathsf{NW},\mathsf{RP}) \wedge \\ & \neq(\mathsf{HS},\mathsf{RP}) \wedge \neq(\mathsf{HS},\mathsf{BW}) \wedge \neq(\mathsf{HS},\mathsf{BY}) \wedge \neq(\mathsf{RP},\mathsf{SL}) \wedge \\ & \neq(\mathsf{RP},\mathsf{BW}) \wedge \neq(\mathsf{BW},\mathsf{BY})\end{aligned}$$

and the following Constraints*:*

$$\begin{aligned}\{\ & \mathsf{p}_{SH}(\mathsf{red}), \mathsf{p}_{SH}(\mathsf{blue}), \mathsf{p}_{SH}(\mathsf{green}), && \mathsf{p}_{HH}(\mathsf{red}), \\ & \mathsf{p}_{NS}(\mathsf{red}), \mathsf{p}_{NS}(\mathsf{blue}), \mathsf{p}_{NS}(\mathsf{green}), && \mathsf{p}_{HB}(\mathsf{red}), \\ & \mathsf{p}_{NW}(\mathsf{red}), \mathsf{p}_{NW}(\mathsf{blue}), \mathsf{p}_{NW}(\mathsf{green}), && \mathsf{p}_{HS}(\mathsf{green}), \\ & \mathsf{p}_{RP}(\mathsf{red}), \mathsf{p}_{RP}(\mathsf{blue}), \mathsf{p}_{RP}(\mathsf{green}), && \mathsf{p}_{SL}(\mathsf{red}), \\ & \mathsf{p}_{BW}(\mathsf{red}), \mathsf{p}_{BW}(\mathsf{blue}), \mathsf{p}_{BW}(\mathsf{green}), && \mathsf{p}_{BY}(\mathsf{red}), \mathsf{p}_{BY}(\mathsf{blue}), \mathsf{p}_{BY}(\mathsf{green}), \\ & \neq(\mathsf{red},\mathsf{blue}), \neq(\mathsf{red},\mathsf{green}), \neq(\mathsf{blue},\mathsf{red}), \neq(\mathsf{blue},\mathsf{green}), \neq(\mathsf{green},\mathsf{red}), \neq(\mathsf{green},\mathsf{blue})\ \}\end{aligned}$$

The Herbrand universe is $H = \{\mathsf{red},\mathsf{blue},\mathsf{green}\}$. *On the FCSP (*Query,Constraints*), the algorithm FCSDA checks* $\mid H \mid^{|V|} = 3^{10} = 59.049$ *tuples and finally returns "Yes" succeeding on the tuple (*green, red, blue, red, red, green, blue, red, red, blue*) which represents the solution* SH = green, HH = red, NS = blue, HB = red, NW = red, HS = green, RP = blue, SL = red, BW = red, *and* BY = blue *(see Figure 2.3). From these*

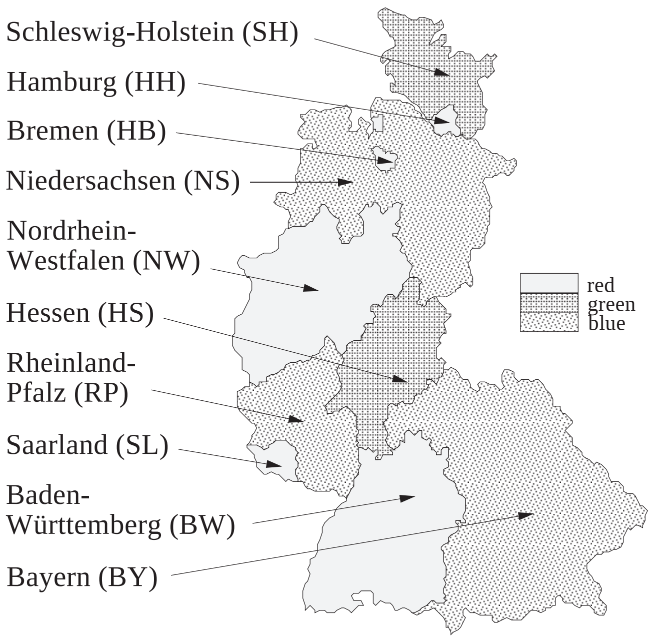


Figure 2.3: A solution to the map-coloring problem

values, of course, only the values for SH*,* NS*,* NW*,* RP*,* BW *and* BY *are of interest; the values for* HH*,* HB*,* HS *and* SL *were known in advance.*

We now consider the completion of Constraints with respect to the Query. Each predicate mentioned in Query can be completed, using Reiter's Closed World Assumption [Reiter, 1988], in the following sense:

$$\mathsf{completion(Constraints)} = \\ \mathsf{Constraints} \cup \{\neg \mathsf{p}_V(c_1, c_2, \ldots, c_k) \mid c_i \in H, \mathsf{p}_V(c_1, c_2, \ldots, c_k) \notin \mathsf{Constraints}\}$$

where $H = \{c \mid \mathsf{p}_V(\ldots, c, \ldots) \in \mathsf{Constraints}\}$ is the Herbrand universe of the theory Constraints. In other words, the complete extension of each k-ary predicate over H^k is specified, positively and negatively, in completion(Constraints). Note that now Constraints $\vdash$ Query iff completion(Constraints) $\vdash$ Query and on the other hand Constraints $\not\vdash$ Query iff completion(Constraints) $\vdash \neg$Query.

2.2.3 Logical representation and reasoning systems

When faced with a problem in representation and reasoning, a wide spectrum of logical representation systems is available to us. Some representation and reasoning systems are shown in Figure 2.4, organized as a directed, acyclic graph (DAG). If there is a downward arc from a system A to a system B, then the descriptive capabilities of A are a strict superset of those of B.

In Section 2.2.1, for example, finite constraint satisfaction (FCS) was shown to be equivalent to theorem proving in a very restricted form of first order predicate calculus

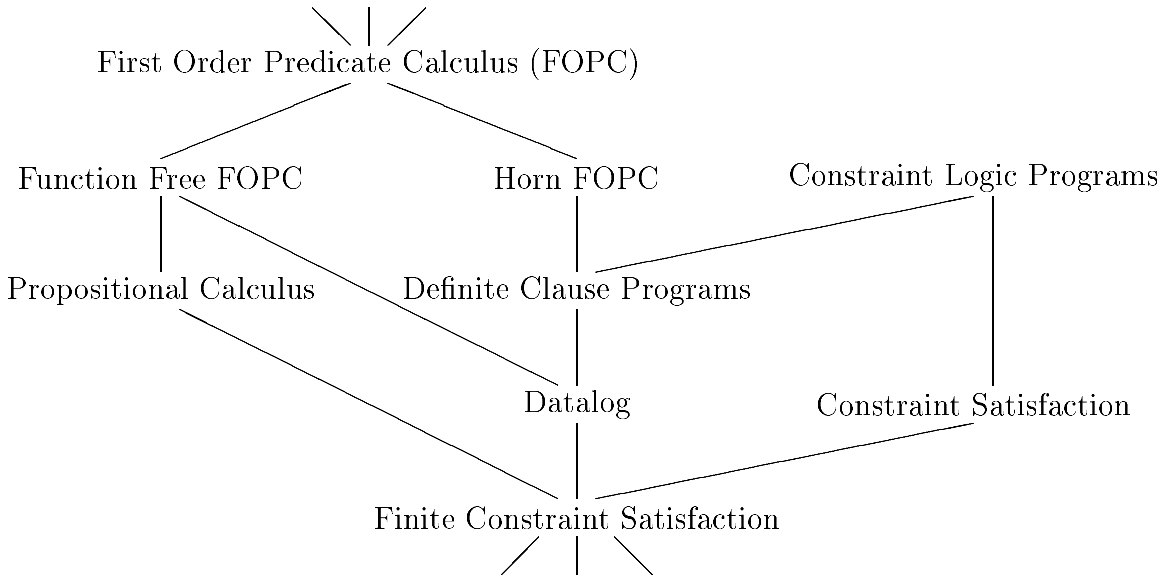


Figure 2.4: Some logical representation and reasoning systems

(FOPC). Horn FOPC restricts FOPC in only allowing Horn clauses[3]. Definite clause programs[4] (DCP), without predicate completion, restrict Horn FOPC by allowing only one negative clause which serves as the query. Datalog essentially restricts DCP by eliminating function symbols. And, finally, finite constraint satisfaction restricts Datalog by disallowing rules, i.e. mixed Horn clauses.

There are several further restrictions on FCS possible with corresponding gains in tractability and some generalizations of FCS with gains in expressive power. We shall examine various logical formulations of FCS and investigate some of their relationships in the following.

2.2.4 FCS as propositional theorem proving

The algorithm FCSDA can be interpreted as implementing a view of FCS as theorem proving in the propositional calculus. Query is a theorem to be proved. If a solution exists, the theory Constraints $\cup$ $\{\neg\mathsf{Query}\}$ leads to a contradiction. Taking the formalization introduced in Section 2.2.1, we have:

$$\begin{aligned}\neg\mathsf{Query} &= \neg\exists X_1\neg\exists X_2\cdots\neg\exists X_n\mathsf{QueryMatrix}(X_1,X_2,\ldots,X_n)\\ &= \forall X_1\forall X_2\cdots\forall X_n\neg\mathsf{QueryMatrix}(X_1,X_2,\ldots,X_n)\end{aligned}$$

[3]A clause, a disjunction of literals, is Horn if it has at most one positive literal. A Horn clause takes one of four forms: a positive unit clause consisting of a single positive literal, a negative clause consisting only of negative literals, a mixed Horn clause consisting of one positive literal and, trivially, the empty clause, $\Box$.

[4]A definite clause has exactly one positive literal; it is either a unit positive clause or a mixed Horn clause.

A solution exists iff $\mathsf{Constraints} \cup \{\neg\mathsf{Query}\}$ has no (Herbrand) models. There are no universal quantifiers in Query and so there are no existential quantifiers in the theory $\mathsf{Constraints} \cup \{\neg\mathsf{Query}\}$. Hence no Skolem functions are introduced when the theory is converted to clausal form. This important restriction guarantees that the Herbrand universe H is finite. This allows us to replace each of the universal quantifiers by the conjunction of the $\neg\mathsf{QueryMatrix}(X_1, X_2, \ldots, X_n)$ clauses instantiated over H^n. The theory rewritten in this way has the same set of Herbrand models as the original theory. It is now a propositional formula in conjunctive normal form (CNF) and has a particular restricted form for any FCSP. It consists only of a set of positive unit clauses, arising from the constraints, and a set of negative clauses, i.e. clauses with only negative literals, from the query. There are no mixed clauses. Note that it is also always Horn. It is unsatisfiable iff the FCSP has a solution. Since it is Horn, SAT is linear time in the size of the formula [Dowling and Gallier, 1984], but, of course, there are $\mid H \mid^n$ negative clauses in the formula. Also note that unit resolution alone is complete for this class of formulas.

Example 9 (German-map coloring problem as propositional theory)
For the map-coloring problem the rewritten theory $\mathsf{Constraints} \cup \{\neg\mathsf{Query}\}$ *results as follows:*

$$
\begin{array}{ll}
\{ & \mathsf{p}_{SH}(\mathsf{red}), \mathsf{p}_{SH}(\mathsf{blue}), \mathsf{p}_{SH}(\mathsf{green}), \quad \mathsf{p}_{HH}(\mathsf{red}), \\
& \mathsf{p}_{NS}(\mathsf{red}), \mathsf{p}_{NS}(\mathsf{blue}), \mathsf{p}_{NS}(\mathsf{green}), \quad \mathsf{p}_{HB}(\mathsf{red}), \\
& \mathsf{p}_{NW}(\mathsf{red}), \mathsf{p}_{NW}(\mathsf{blue}), \mathsf{p}_{NW}(\mathsf{green}), \quad \mathsf{p}_{HS}(\mathsf{green}), \\
& \mathsf{p}_{RP}(\mathsf{red}), \mathsf{p}_{RP}(\mathsf{blue}), \mathsf{p}_{RP}(\mathsf{green}), \quad \mathsf{p}_{SL}(\mathsf{red}), \\
& \mathsf{p}_{BW}(\mathsf{red}), \mathsf{p}_{BW}(\mathsf{blue}), \mathsf{p}_{BW}(\mathsf{green}), \quad \mathsf{p}_{BY}(\mathsf{red}), \mathsf{p}_{BY}(\mathsf{blue}), \mathsf{p}_{BY}(\mathsf{green}), \\
& \neq(\mathsf{red}, \mathsf{blue}), \neq(\mathsf{red}, \mathsf{green}), \neq(\mathsf{blue}, \mathsf{red}), \neq(\mathsf{blue}, \mathsf{green}), \neq(\mathsf{green}, \mathsf{red}), \neq(\mathsf{green}, \mathsf{blue}) \ \} \\
\cup & \\
\{ & \neg\mathsf{p}_{SH}(\mathsf{red}) \lor \neg\mathsf{p}_{HH}(\mathsf{red}) \lor \cdots \lor \neg\mathsf{p}_{BW}(\mathsf{red}) \lor \neg\mathsf{p}_{BY}(\mathsf{red}) \lor \neg{\neq}(\mathsf{red}, \mathsf{red}), \\
& \neg\mathsf{p}_{SH}(\mathsf{red}) \lor \neg\mathsf{p}_{HH}(\mathsf{red}) \lor \cdots \lor \neg\mathsf{p}_{BW}(\mathsf{red}) \lor \neg\mathsf{p}_{BY}(\mathsf{green}) \lor \neg{\neq}(\mathsf{red}, \mathsf{red}) \lor \\
& \neg{\neq}(\mathsf{red}, \mathsf{green}), \\
& \vdots \\
& \neg\mathsf{p}_{SH}(\mathsf{green}) \lor \neg\mathsf{p}_{HH}(\mathsf{red}) \lor \neg\mathsf{p}_{NS}(\mathsf{blue}) \lor \neg\mathsf{p}_{HB}(\mathsf{red}) \lor \neg\mathsf{p}_{NW}(\mathsf{red}) \lor \\
& \neg\mathsf{p}_{HS}(\mathsf{green}) \lor \neg\mathsf{p}_{RP}(\mathsf{blue}) \lor \neg\mathsf{p}_{SL}(\mathsf{red}) \lor \neg\mathsf{p}_{BW}(\mathsf{red}) \lor \neg\mathsf{p}_{BY}(\mathsf{blue}) \lor \qquad (*) \\
& \neg{\neq}(\mathsf{red}, \mathsf{green}) \lor \neg{\neq}(\mathsf{red}, \mathsf{blue}) \lor \neg{\neq}(\mathsf{green}, \mathsf{red}) \lor \\
& \neg{\neq}(\mathsf{green}, \mathsf{blue}) \lor \neg{\neq}(\mathsf{blue}, \mathsf{red}) \lor \neg{\neq}(\mathsf{blue}, \mathsf{green}), \\
& \vdots \\
& \neg\mathsf{p}_{SH}(\mathsf{blue}) \lor \neg\mathsf{p}_{HH}(\mathsf{blue}) \lor \cdots \lor \neg\mathsf{p}_{BW}(\mathsf{blue}) \lor \neg\mathsf{p}_{BY}(\mathsf{blue}) \lor \neg{\neq}(\mathsf{blue}, \mathsf{blue}) \ \}
\end{array}
$$

Repeated unit resolution on the clause marked $(*)$ *reduces it to the empty clause* $\square$*, corresponding to the solution* $\{\mathsf{SH} = \mathsf{green}, \mathsf{HH} = \mathsf{red}, \mathsf{NS} = \mathsf{blue}, \mathsf{HB} = \mathsf{red}, \mathsf{NW} = \mathsf{red}, \mathsf{HS} = \mathsf{green}, \mathsf{RP} = \mathsf{blue}, \mathsf{SL} = \mathsf{red}, \mathsf{BW} = \mathsf{red},$ *and* $\mathsf{BY} = \mathsf{blue}\}$.

For any FCSP, using subsumption[5] does not affect the completeness result. Iterating unit resolution followed by subsumption leaves invariant the special properties of

[5]If clauses of the generic form C and $C \lor D$ are both present, subsumption will delete $C \lor D$.

the formula (only negative and unit positive clauses) and, moreover, only decreases its size. Hence, it terminates (correctly). As, of course, does the linear time HornSAT algorithm. The HornSAT algorithm exactly mimics the algorithm FCSDA. The propositional variable in each unit positive literal is set to T and each negative clause is checked: if any clause has each (negative) literal required to be F then the formula is unsatisfiable otherwise it is satisfiable.

2.2.5 FCS as theorem proving in definite theories

The methods discussed so far are not serious candidates for actually solving an FCSP: they simply serve to clarify the semantics and methods of the serious candidates. One such candidate is a PROLOG interpreter, which is a theorem prover for theories consisting of definite clauses. Since an FCSP can be seen as a pure PROLOG program, SLD-resolution is a sound and complete solution method. Just as for the algorithm FCSDA, we may interpret PROLOG's failure to find a proof for this class of theories as meaning either $\mathsf{Constraints} \not\vdash \mathsf{Query}$ or $\mathsf{completion(Constraints)} \vdash \neg\mathsf{Query}$.

A depth-first SLD-resolution strategy may fail to find a proof for a query that is in fact a theorem for an arbitrary definite clause program but will always do so for an FCSP. It is also, generally speaking, more efficient than other resolution methods such as the one embodied in algorithm FCSDA.

Example 10 (German-map coloring problem as definite theory) *For solving the map-coloring problem in* PROLOG, *we assert the constraints as ground facts in a* PROLOG *database*

```
pSH(red).  pSH(blue).  pSH(green).  pHH(red).
pNS(red).  pNS(blue).  pNS(green).  pHB(red).
pNW(red).  pNW(blue).  pNW(green).  pHS(green).
pRP(red).  pRP(blue).  pRP(green).  pSL(red).
pBW(red).  pBW(blue).  pBW(green).
pBY(red).  pBY(blue).  pBY(green).
ne(red,blue).  ne(red,green).  ne(blue,green).
ne(blue,red).  ne(green,red).  ne(green,blue).
```

and then define and pose the conjunctive query to PROLOG*:*

```
?-  pSH(SH), pHH(HH), pNS(NS), pHB(HB), pNW(NW), pHS(HS),
    pRP(RP), pSL(SL), pBW(BW), pBY(BY),
    ne(SH,HH), ne(SH,NS), ne(HH,NS), ne(NS,HB), ne(NS,NW),
    ne(NS,HS), ne(NW,HS), ne(NW,RP), ne(HS,RP), ne(HS,BW),
    ne(HS,BY), ne(RP,SL), ne(RP,BW), ne(BW,BY).
```

The PROLOG *system will then compute the solution*

```
SH = green, HH = red, NS = blue, HB = red, NW = red,
HS = green, RP = blue, SL = red, BW = red, BY = blue
```

In finding this solution the interpreter essentially checks every possible set of bindings for the variables SH, HH, NS, HB, NW, HS, RP, SL, BW, *and* BY, *which gives a total number of* $3^6 + 1^4 = 729$ *combinations to check. For finding the solution, 262 checks of the* ne *constraint have to be performed; and a total number of 1960 checks to prove that it is the only solution.*

In order to decrease the number of constraint checks to be performed, the query can be reordered such that a partially completed set of bindings can be rejected by a single failure.

Example 11 (German-map coloring problem, reordered query) *By reordering the query for the map-coloring problem such that the* ne *constraints are checked as soon as their variables have been instantiated, we obtain the following query*

```
?-  pSH(SH), pHH(HH), ne(SH,HH), pNS(NS), ne(SH,NS), ne(HH,NS),
    pHB(HB), ne(NS,HB), pNW(NW), ne(NS,NW), pHS(HS), ne(NS,HS),
    ne(NW,HS), pRP(RP), ne(NW,RP), ne(HS,RP), pSL(SL), ne(RP,SL),
    pBW(BW), ne(HS,BW), ne(RP,BW), pBY(BY), ne(HS,BY), ne(BW,BY).
```

for which the Prolog *system has only 18 checks to perform on the* ne *constraints to find the solution, and 38 checks to find that it is the only one.*

However, in general no fixed variable ordering can avoid *thrashing* by repeatedly rediscovering incompatible variable bindings [Mackworth, 1977]. Therefore, heuristics such as instantiating the most constrained variable next, can be used to reorder the query dynamically and will be discussed in Chapter 6 when we extend SLD-resolution by various constraint-solving techniques.

2.2.6 FCS as Datalog

Since FCS is a restriction of Datalog, the techniques developed in the relational database community are available [Maier, 1983]. The solution relation is the natural join of the relations for the individual constraints. The consistency techniques discussed below can be interpreted similarly; for example, making an arc consistent in a constraint network can be interpreted as a semi-join. Results that exploit this interpretation can be found in [Bibel, 1988, Dechter, 1992, Mackworth, 1992a].

2.2.7 FCS in constraint networks

Consideration of the drawbacks of the SLD-resolution approach mentioned above leads to a view of FCS in *constraint networks*. A constraint network represents each variable in the query as a vertex. The unary constraint $\mathsf{p}_x(X)$ establishes the domain of X, and each binary constraint $\mathsf{p}_{XY}(X, Y)$ is represented as the edge (X, Y), composed of arc (X, Y) and arc (Y, X). This easily generalizes to k-ary predicates using hypergraphs.

Example 12 (German-map coloring constraint network) *The constraint network for the map-coloring problem is shown in Figure 2.5. The domains of the variables are also shown in the vertices where* r *stands for* red*,* b *stands for* blue*, and* g *for* green*.*

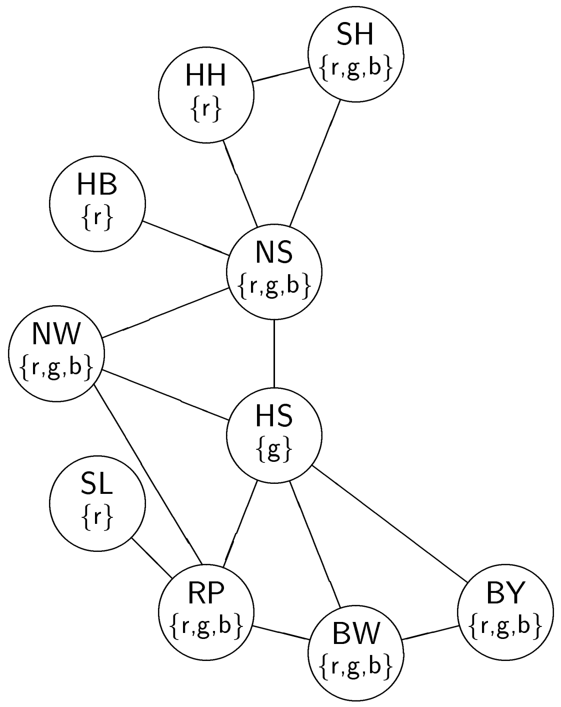


Figure 2.5: Constraint network for the map-coloring problem

Representing CSPs as constraint networks focuses on the connections between variables. It also allows to identify subgraphs as subproblems of the CSP and serves as a basis for a set of graph-based constraint satisfaction methods that will be discussed later.

We will restrict our discussion on binary FCSPs and define when the domains of two variables X and Y are consistent with the constraint that is represented by the arc (X, Y).

Definition 19 (arc-consistency) *An arc* (X, Y) *is* consistent *iff the following meta-language sentence holds for* Constraints*:*

$$\forall x[\mathsf{p}_X(x) \rightarrow \exists y[\mathsf{p}_Y(y) \land \mathsf{p}_{XY}(x, y)]] \tag{2.1}$$

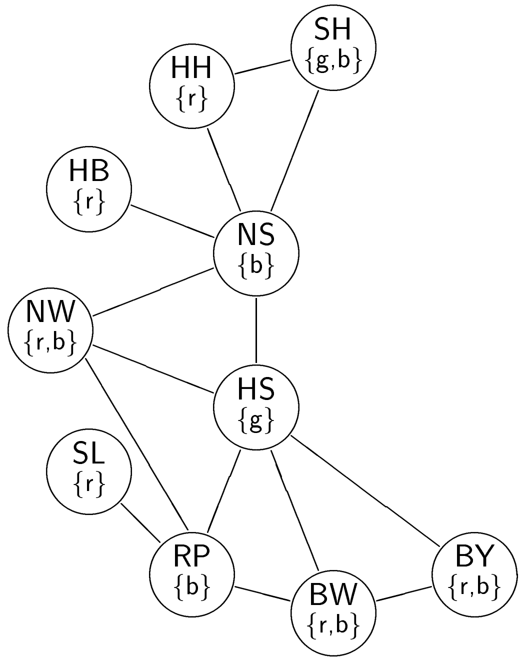
A constraint network is arc-consistent if all its arcs are consistent.

An arc (X, Y) can be made consistent without affecting the total set of solutions by deleting the values from the domain of X that are not compatible with some value in Y. By deleting such incompatible values, other arcs that have been consistent, may become inconsistent. Thus, restrictions of a variables domain can be *propagated* to other arcs, respectively constraints, that are connected to that variable. This process is also known as *constraint propagation* [Güsgen and Hertzberg, 1988].

Example 13 (arc-consistency in the German-map coloring problem) *The map-coloring network as shown in Figure 2.5 is not arc-consistent because, for example, the arc (*NS,HB*) is inconsistent, i.e. the following instance of (2.1) with X = NS, Y = HB, and $\mathsf{p}_{XY} = \neq$ does not hold, because for x =* red *there is no value for y satisfying the constraints $\mathsf{p}_{\mathsf{HB}}(y)$ and $\neq(\mathsf{red}, y)$:*

$$\forall x[\mathsf{p}_{\mathsf{NS}}(x) \rightarrow \exists y[\mathsf{p}_{\mathsf{HB}}(y) \wedge \neq(x,y)]]$$

Thus, deleting the value red *from the domain of* NS *makes arc (*NS,HB*) consistent. Enforcing consistency for the arcs (*NW,HS*), (*NS,HS*), (*RP,HS*), (*BW,HS*), and (*BY,HS*) will delete the value* green *from the domains of* NW, NS, RP, BW, *and* BY. *Further applying this operation to arcs (*SH,HH*) and (*RP,SL*) will then also delete the value* red *from the domains of* SH *and* RP. *Thus, we obtain the following intermediate constraint network:*



One can now easily recognize that the map-coloring constraint network is still not arc-consistent: Due to the removal of inconsistent values in the previous steps, three other previously consistent arcs have become inconsistent:

- *Consider the arc (*NW,RP*) or the arc (*NW,NS*). In any case the arc turns out to be inconsistent and thus the value* blue *has to be removed from the domain of* NW.
- *Applying the before-mentioned schema to the arc (*SH,NS*) then furthermore results in deleting the value* blue *from the domain of* SH.
- *The same happens for the domain of* BW *when considering the arc (*BW,RP*).*

Due to the removal of blue *from the domain of* BW *in the last step, now arc (*BY,BW*) also becomes inconsistent and causes the* red *to be removed from the domain of* BY. *This finally results in getting the whole map-coloring network consistent as shown in Figure 2.6. In*

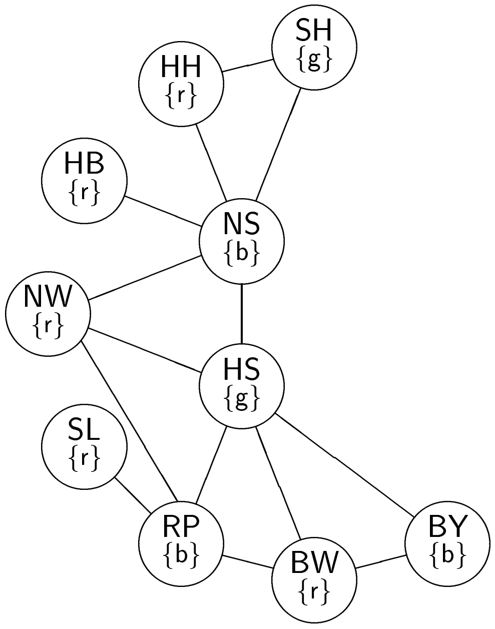


Figure 2.6: Arc-consistent network for the map-coloring problem

our example, now all variable domains hold exactly one value, such that the arc-consistent constraint network does also represent the global solution to the FCSP.

Arc-consistency can be enforced in time linear in the number of binary constraints; moreover, if the constraint graph is a tree, as it can be shown for the map-coloring problem[6], then arc-consistency alone suffices as a decision procedure for the FCSP [Mackworth and Freuder, 1985]. Various other graph-theoretic properties of the constraint network can be used to characterize and solve FCSPs [Freuder, 1990]. We will discuss these issues in more detail in Chapter 3.

2.2.8 An arc-consistency rewrite system for FCSPs

Using the ideas discussed so far we can implement a logical interpreter for FCSPs as an arc-consistency enforcing rewrite system [Van Hentenryck, 1989]. Given an FCSP formulation of the form (Constraints,Query) as introduced in Section 2.2.1, following [Clark, 1978] we complete each predicate in Constraints and represent it by its *definition*, its necessary and sufficient conditions.

Restricting the interpreter to enforcing arc-consistency, it non-deterministically rewrites Constraints using AC rewrite rules of the following form:

$$\mathsf{p}_X(X) \Leftarrow \mathsf{p}_X(X) \wedge \exists Y[\mathsf{p}_Y(Y) \wedge \mathsf{p}_{XY}(X,Y)].$$

[6]The map-coloring network as shown in Figure 2.5 can easily be transformed into a tree by substituting any n-ary vertex V that represents a single-valued variable (like HH, HB, SL, and HS) by n independent copies of V. Doing so for the network shown in Figure 2.5 will result in a very simple tree structure.

The set of AC rewrite rules, corresponding to the set of arcs in the constraint network, applied to $\mathsf{Constraints}$ constitutes a production system with the Church-Rosser property [Mackworth, 1992b]. Repeated application of the rules, rewriting the definitions of the unary predicates, reduces $\mathsf{Constraints}$ to a fixpoint.

Example 14 (rewriting the German-map coloring problem) *For the* Query

$$\begin{array}{ll} \exists \mathsf{SH}\, \exists \mathsf{HH} \cdots \exists \mathsf{BY} & \mathsf{p}_{SH}(\mathsf{SH}) \wedge \mathsf{p}_{HH}(\mathsf{HH}) \wedge \cdots \wedge \mathsf{p}_{BY}(\mathsf{BY}) \wedge \\ & \neq(\mathsf{SH},\mathsf{HH}) \wedge \neq(\mathsf{SH},\mathsf{NS}) \wedge \neq(\mathsf{HH},\mathsf{NS}) \wedge \\ & \neq(\mathsf{NS},\mathsf{HB}) \wedge \neq(\mathsf{NS},\mathsf{NW}) \wedge \neq(\mathsf{NS},\mathsf{HS}) \wedge \\ & \neq(\mathsf{NW},\mathsf{HS}) \wedge \neq(\mathsf{NW},\mathsf{RP}) \wedge \neq(\mathsf{HS},\mathsf{RP}) \wedge \\ & \neq(\mathsf{HS},\mathsf{BW}) \wedge \neq(\mathsf{HS},\mathsf{BY}) \wedge \neq(\mathsf{RP},\mathsf{SL}) \wedge \\ & \neq(\mathsf{RP},\mathsf{BW}) \wedge \neq(\mathsf{BW},\mathsf{BY}) \end{array}$$

following [Clark, 1978], we obtain the following completed set of $\mathsf{Constraints}$*:*

$$\begin{array}{rcl} \mathsf{p}_{SH}(X) & \leftrightarrow & ((X = \mathsf{red}) \vee (X = \mathsf{blue}) \vee (X = \mathsf{green})), \\ \mathsf{p}_{HH}(X) & \leftrightarrow & (X = \mathsf{red}), \\ \mathsf{p}_{NS}(X) & \leftrightarrow & ((X = \mathsf{red}) \vee (X = \mathsf{blue}) \vee (X = \mathsf{green})), \\ \mathsf{p}_{HB}(X) & \leftrightarrow & (X = \mathsf{red}), \\ \mathsf{p}_{NW}(X) & \leftrightarrow & ((X = \mathsf{red}) \vee (X = \mathsf{blue}) \vee (X = \mathsf{green})), \\ \mathsf{p}_{HS}(X) & \leftrightarrow & (X = \mathsf{green}), \\ \mathsf{p}_{RP}(X) & \leftrightarrow & ((X = \mathsf{red}) \vee (X = \mathsf{blue}) \vee (X = \mathsf{green})), \\ \mathsf{p}_{SL}(X) & \leftrightarrow & (X = \mathsf{red}), \\ \mathsf{p}_{BW}(X) & \leftrightarrow & ((X = \mathsf{red}) \vee (X = \mathsf{blue}) \vee (X = \mathsf{green})), \\ \mathsf{p}_{BY}(X) & \leftrightarrow & ((X = \mathsf{red}) \vee (X = \mathsf{blue}) \vee (X = \mathsf{green})), \\ \neq(X,Y) & \leftrightarrow & (((X = \mathsf{red}) \wedge (Y = \mathsf{blue})) \vee ((X = \mathsf{red}) \wedge (Y = \mathsf{green})) \vee \\ & & ((X = \mathsf{blue}) \wedge (Y = \mathsf{red})) \vee ((X = \mathsf{blue}) \wedge (Y = \mathsf{green})) \vee \\ & & ((X = \mathsf{green}) \wedge (Y = \mathsf{red})) \vee ((X = \mathsf{green}) \wedge (Y = \mathsf{blue}))). \end{array}$$

Applying the AC rewrite rule to the definition of p_{NS} *represents the first arc-consistency step mentioned in Section 2.2.7:*

$$\begin{array}{rcll} \mathsf{p}_{NS}(\mathsf{NS}) & \Leftarrow & \mathsf{p}_{NS}(\mathsf{NS}) \wedge \exists \mathsf{HB}[\mathsf{p}_{HB}(\mathsf{HB}) \wedge \neq(\mathsf{NS},\mathsf{HB})] & \\ & \Leftarrow & ((\mathsf{NS} = \mathsf{red}) \vee (\mathsf{NS} = \mathsf{blue}) \vee (\mathsf{NS} = \mathsf{green})) & \\ & & \wedge \exists \mathsf{HB}[\mathsf{HB} = \mathsf{red} \wedge & ((\mathsf{NS} = \mathsf{red}) \wedge (\mathsf{HB} = \mathsf{blue})) \vee \\ & & & ((\mathsf{NS} = \mathsf{red}) \wedge (\mathsf{HB} = \mathsf{green})) \vee \\ & & & ((\mathsf{NS} = \mathsf{blue}) \wedge (\mathsf{HB} = \mathsf{red})) \vee \\ & & & ((\mathsf{NS} = \mathsf{blue}) \wedge (\mathsf{HB} = \mathsf{green})) \vee \\ & & & ((\mathsf{NS} = \mathsf{green}) \wedge (\mathsf{HB} = \mathsf{red})) \vee \\ & & & ((\mathsf{NS} = \mathsf{green}) \wedge (\mathsf{HB} = \mathsf{blue}))] \\ & \Leftarrow & (\mathsf{NS} = \mathsf{blue}) \vee (\mathsf{NS} = \mathsf{green}). & \end{array}$$

Thus, the constraint $\mathsf{p}_{NS}(X) \leftrightarrow ((X = \mathsf{red}) \vee (X = \mathsf{blue}) \vee (X = \mathsf{green}))$ *gets rewritten to* $\mathsf{p}_{NS}(X) \leftrightarrow ((X = \mathsf{blue}) \vee (X = \mathsf{green}))$*. Repeated application of the AC rewrite rules* *reduces* Constraints *to the following fixpoint representing the solution for the map-coloring problem:*

$$\begin{array}{rcl}
\mathsf{p}_{SH}(X) & \leftrightarrow & (X = \mathsf{green}), \\
\mathsf{p}_{HH}(X) & \leftrightarrow & (X = \mathsf{red}), \\
\mathsf{p}_{NS}(X) & \leftrightarrow & (X = \mathsf{blue}), \\
\mathsf{p}_{HB}(X) & \leftrightarrow & (X = \mathsf{red}), \\
\mathsf{p}_{NW}(X) & \leftrightarrow & (X = \mathsf{red}), \\
\mathsf{p}_{HS}(X) & \leftrightarrow & (X = \mathsf{green}), \\
\mathsf{p}_{RP}(X) & \leftrightarrow & (X = \mathsf{blue}), \\
\mathsf{p}_{SL}(X) & \leftrightarrow & (X = \mathsf{red}), \\
\mathsf{p}_{BW}(X) & \leftrightarrow & (X = \mathsf{red}), \\
\mathsf{p}_{BY}(X) & \leftrightarrow & (X = \mathsf{blue}), \\
\neq(X,Y) & \leftrightarrow & (((X = \mathsf{red}) \wedge (Y = \mathsf{blue})) \vee ((X = \mathsf{red}) \wedge (Y = \mathsf{green})) \vee \\
& & ((X = \mathsf{blue}) \wedge (Y = \mathsf{red})) \vee ((X = \mathsf{blue}) \wedge (Y = \mathsf{green})) \vee \\
& & ((X = \mathsf{green}) \wedge (Y = \mathsf{red})) \vee ((X = \mathsf{green}) \wedge (Y = \mathsf{blue}))).
\end{array}$$

In general, a logical interpreter like the one presented here must interleave the AC rewriting with some non-deterministic case analysis or higher-order network consistency, as arc-consistency may not suffice to solve an FCSP.

2.2.9 Relations to constraint logic programming

Constraint logic programming (CLP) [Jaffar and Lassez, 1987] is a generalization of logic programming where the domain of computation is abstracted. The interpreter is still based on the resolution principle but the concept of syntactic unification in the Herbrand universe is replaced by the more general concept of constraint satisfaction in some computation domain. Thus, the CLP scheme defines a class of languages CLP($\mathcal{D}$) parameterized by the computation domain $\mathcal{D}$. For example, the CLP($\mathcal{R}$) language implements the CLP paradigm in the domain $\mathcal{R}$ of real arithmetic. It allows for constraints over real numbers and uses a modified Simplex method for solving them, where non linear constraints are delayed, i.e. temporarily suspended from consideration, until a sufficient number of variables are instantiated and the constraints become linear.

In general, a CLP program is defined by a collection of clauses of the form

$$H \leftarrow C_1 \wedge \ldots \wedge C_n \wedge B_1 \wedge \ldots \wedge B_m$$

where $H, B_1, \ldots, B_m$ are atoms and $C_1, \ldots, C_n$ are constraints. In Figure 2.4 we have drawn an arc from CLP to definite clause programs (DCP), i.e. Logic Programming (LP), because LP can be regarded as an instance of the CLP($\mathcal{D}$) scheme, where the computation domain $\mathcal{D}$ is equal to the Herbrand universe H and the constraints are equalities on terms.

Finally, it is also worth noting that the arc-consistency rewrite system discussed in the context of finite CSPs can also be applied to general CSPs and CLP programs as well. Consider, for example, the following CLP($\mathcal{R}$) program:

$$\begin{aligned} p(X) &\leftarrow (1 \leq X) \wedge (X \leq 3), \\ q(Y) &\leftarrow (0 \leq Y) \wedge (Y \leq 2), \\ r(X, Y) &\leftarrow X \leq Y \end{aligned}$$

Taking this set of Constraints together with a Query of the form

$$\exists X \exists Y [p(X) \wedge q(Y) \wedge r(X, Y)]$$

we obtain a CSP formulation (Constraints,Query) in the sense of Section 2.2.1. Using the same AC rewrite rule as used for finite CSPs in Section 2.2.8, the set of Constraints is refined to the fixpoint

$$\begin{aligned} p(X) &\leftarrow (1 \leq X) \wedge (X \leq 2), \\ q(Y) &\leftarrow (1 \leq Y) \wedge (Y \leq 2), \\ r(X, Y) &\leftarrow X \leq Y. \end{aligned}$$

In the previous sections, we have presented several standard, and some not-so-standard, logical methods that can be used to solve FCSPs. The syntactic treatment allows algorithms and results from these desperate areas to be imported, and specialized, to FCSP. It also allows export to the related areas. By casting CSP both as a generalization of FCSP and as a specialization of CLP it is observed that some FCSP solving techniques, e.g. arc-consistency rewrite rules, lift to CSP and thereby to CLP. This observation will be one of the main motivations for our work on the integration of constraint satisfaction techniques in logic programming that will be presented in Chapter 6.

2.3 Towards Efficient Constraint Satisfaction

In Section 2.1 we have discussed how to formulate a given problem *declaratively* as a CSP and in Section 2.2 we then studied how to perform constraint satisfaction in different representation and reasoning systems. Based on these considerations, we are now in a position to focus on algorithms for *efficiently* solving an FCSP.

In this section we will present a generic algorithm for solving FCSPs which basically implements a parameterizable backtracking search process in the space of possible value combinations. As one instance of this generic framework we obtain an implementation of the standard backtracking approach applied to FCSP solving. This approach, however, suffers from several drawbacks such as thrashing and repeated computation of the same partial solutions. The chapter will then be concluded by presenting three directions towards the development of more efficient FCSP solving techniques that avoid these drawbacks. All of them have been exploited in the implementation of the CONTAX and FIDO systems that will be presented in the next chapters.

2.3.1 From generate-and-test to backtracking

Any FCSP can be solved using the *generate-and-test* paradigm (GT). The algorithm FCSDA of Section 2.2.2 and the PROLOG query on page 36 basically implement this paradigm. As the assignment space $D = D_1 \times D_2 \times \cdots \times D_n$ is finite, it is possible to systematically generate and test each n-compound label for the Variables X_1 to X_n in the CSP to see if it satisfies all the constraints. The number of combinations considered by this method is given by the size of the Cartesian product of all the variable domains which results in the generate-and-test algorithm being an easy to implement but very inefficient algorithm for solving CSPs.

A more efficient method uses the *backtracking* (BT) paradigm. In this method variables are instantiated sequentially. As soon as all the variables relevant to a constraint are instantiated, it is checked whether the constraint is satisfied. If a partial instantiation violates any of the constraints, backtracking is performed to the most recently instantiated variable that still has alternatives available. Clearly, whenever a partial instantiation violates a constraint, backtracking is able to eliminate a subspace from the Cartesian product of all variable domains. The backtracking method essentially performs an uninformed systematic search [Brown and Purdom, 1981] in the space of potential solutions of the CSP.

2.3.2 A generic backtracking algorithm for solving FCSPs

In the following, we will present a generic algorithm for searching the space of possible value combinations for a solution of the FCSP. The algorithm is generic in the sense that in the essential parts it can be parameterized to obtain a concrete implementation.

For the presentation of the algorithm we need the following notions: At each time the already instantiated variables are called *past variables*, the non-instantiated variables are called *future variables*, and the variable which is being instantiated in the current step is called the *current variable*.

1. *Initialize:* Assign to each variable an indefinite value. S is empty.
2. *Select variable:* Select as current variable one of the variables that have not been assigned yet. If all variables have already been assigned go to step 5.
3. *Select value:* Select a non-used value as the current value for the current variable. Mark this value as used. If there is no such value, go to step 6.
4. *Test:* Check all the constraints in which the current variable and the past variables are involved. If they are satisfied push the current variable into S and go to step 2, otherwise go to step 3.
5. *Solution:* The current assignment of values to variables is a solution. If only one solution is needed, stop, otherwise go to step 3.

6. *Backtracking:* Assign an indefinite value to the current variable. Delete this variable from S and select the new current variable in S. If S is empty stop as no (more) solutions can be found, otherwise go to step 3.

This algorithm can be parameterized by using different orderings or ordering schemes (static or dynamic) for variable selection (step 2), for selecting the next value for each variable (step 3), and for past variable selection for testing (step 4). It uses a data structure S for recording past variables and backtrack variable selection (step 6). The implementation of this data structure as a stack, as a queue, or using some more sophisticated (dynamic) ordering of past variables, will be crucial for the efficiency of a concrete implementation. Using a stack as structure for recording past variables imposes a chronological ordering for backtrack variable selection. Together with a fixed static ordering of the variables and their domains we obtain the *standard backtracking paradigm* as an instance of this generic schema. Using more 'intelligent' methods for backtrack variable selection based on some dependency analysis results in implementing a *dependency-directed* [Stallman and Sussman, 1977] or *intelligent backtracking* [Bruynooghe and Pereira, 1984] approach.

The temporal complexity of this core algorithm is exponential $O(a^n)$ and its spatial complexity is logarithmic $O(log\ n)$, where a denotes the maximum cardinality of the variable domains. The algorithm finds one or all the problem solutions because it performs an exhaustive search. However, the algorithm is inefficient as it carries out more operations than are necessary to find a solution. As causes of this inefficiency can be pointed out:

1. Thrashing: One of the reasons for the inefficiency of the backtracking paradigm is that it suffers from *thrashing* [Gaschnig, 1979], i.e. search in different parts of the space keeps failing for the same reasons. The simplest cause of thrashing concerns unary constraints, and is referred to as *node-consistency* [Mackworth, 1977]. If the domain D_{X_i} of a variable X_i contains a value v that does not satisfy some unary constraint $C_{\{X_i\}}$, then the instantiation of X_i to v will always result in an immediate failure. Another possible source of thrashing is illustrated by the following example.

Example 15 (thrashing) *Suppose the variables in a CSP are instantiated in the order* $X_1, X_2, \ldots, X_i, \ldots, X_j, \ldots, X_n$. *Suppose further that the binary constraint* $C_{\{X_i,X_j\}}$ *is such that for* $X_i = v$, *it disallows any value of* X_j, *and assume that the variables* X_{i+1} *to* X_{j-1} *can take compatible values among them and with the variables* X_1 *to* X_i. *Whenever* X_i *is instantiated to* v, *the algorithm will force every variable in* $\{X_{i+1}, \ldots, X_{j-1}\}$ *to take all possible values before changing the assignment for* X_i.

The cause of this kind of thrashing is referred to as lack of *arc-consistency* [Mackworth, 1977]. It can be avoided by checking every time a value is assigned that some compatible values exist in the future variable domains.

2. Repeated computation of partial solutions: The algorithm does not remember previous actions. If the first value in the domain of X_i is incompatible with the first value in the domain of some variable X_j with $j > i + 1$, this will be detected the first time a value for X_j will be assigned. However, this incompatibility will be checked again each time that backtracking arrives to a variable previous to X_i. If the incompatibilities detected are recorded, it is possible to avoid actions which results are already known.

2.4 Discussion and Directions for Further Work

Once the inefficiency of the general backtracking search algorithm has been established, the goal is to develop more efficient algorithms for solving FCSPs. Since FCSP is $\mathcal{NP}$-complete [Mackworth, 1977], this does not mean to achieve a polynomial complexity, but to get a better run-time performance. This can be achieved by performing only the operations that are strictly needed to find a solution and to avoid all unnecessary work. In general, efficiency improvements tend to lose the initial simplicity of the algorithm. The spatial complexity increases, and sometimes complexity problems in the temporal scope are transferred to the spatial scope. However, for the topic of this thesis it is most important to preserve the declarative problem formulation property and the generality of the approach when studying efficiency improvements. The main consequence for us is, that we will not consider any specialized techniques that exploit specific features of a limited class of CSPs, although it may yield substantial improvements, but not in the general case.

In this thesis we will study two basic approaches to improve the efficiency of constraint satisfaction algorithms that aim at reducing the search space that has to be explored in order to find a solution to the CSP. The basic idea is to modify the search space by deleting some regions that do not contain any solutions. If substantial reductions are made in the search space, a better performance of the search process is expected. When all solutions have to be computed, the search space has to be explored exhaustively. In this case a reduction in the search space means a net reduction in the time required by the search process. There are two approaches in methods that reduce the search space:

- One approach consists of algorithms that reduce the search space prior to starting the search process. They are called *consistency-improving algorithms* because all of them are based on achieving a given level of local consistency.
- The other approach consists of algorithms that modify the search space during the search process. They are called *hybrid algorithms* because the process of search is interleaved with the reduction of the search space.

The Contax system that will be presented in Chapters 3, 4, and 5 basically implements a consistency-improving algorithm which is then extended towards a hybrid version by interleaving subsequent search with consistency checking when committing to a value assignment.

The FiDo system to be presented in Chapters 6 and 7 basically also realizes a hybrid algorithm. However, the focus is on improving backtracking search in logic programming, e.g. Prolog, by using consistency-improving algorithms locally when committing to a value assignment for a variable that is subject to a constraint rather than an ordinary Prolog variable.

3

Constraint Propagation in Contax

In the previous chapter, we have discussed several views on constraint satisfaction and have presented a generic backtracking algorithm for solving FCSPs. This algorithm, however, suffers from various sources of inefficiency. Thus, in order to improve the efficiency of FCSP solving, we have outlined three approaches, one of which is to reduce the search space prior to starting the entire search process. This is the underlying idea of *constraint propagation* techniques which we will discuss in Section 3.1. Based on these considerations, Section 3.2 will then present the Contax system, a solver for general FCSPs that we have developed at DFKI. As Contax supports constraints of arbitrary arity, the notion of arc-consistency for binary CSPs will have to be extended to hyperarc-consistency for general CSPs and algorithms will be presented to efficiently establish hyperarc-consistency in a constraint network such that the reduced search space can be explored afterwards in order to ultimately find the solutions to the FCSP. The constraint propagation techniques as well as the user-interface to Contax will again be illustrated using the German-map coloring problem.

3.1 Constraint Propagation Techniques

As we have discussed in Section 2.3.2, one of the main causes of inefficiency when trying to solve an FCSP using standard backtracking is the effect of thrashing.

Thrashing due to node inconsistency can be eliminated by simply removing those values from the domains D_{X_i} of each variable X_i that do not satisfy some unary constraint $C_{\{X_i\}}$. Thrashing due to arc-consistency can be avoided if, before search starts, each arc (X_i, X_j)[1] of the constraint graph is made consistent.

3.1.1 Arc consistency

As discussed in Section 2.2.7, an arc (X_i, X_j) is consistent if for every value v in the current domain D_{X_i} of X_i there is some value $v_j \in D_{X_j}$ such that the compound label $(\langle X_i \leftarrow v\rangle,\langle X_j \leftarrow v_j\rangle)$ is permitted by the constraint between X_i and X_j, in other words

[1]Almost all theoretical and practical work on constraint satisfaction algorithms is focused on binary CSPs. For this reason, only binary CSPs will be considered in this section, but without loss of generality as we have shown in Section 2.1.6.

$(\langle X_i \leftarrow v\rangle,\langle X_j \leftarrow v_j\rangle) \in C_{\{X_i,X_j\}}$. Note that the concept of arc-consistency is directional; i.e. if an arc (X_i, X_j) is consistent, then it does not necessarily mean that (X_j, X_i) is also consistent.

An arc (X_i, X_j) can be made consistent by simply deleting those values from the domain D_{X_i} for which the above condition is not true.[2] This is what the following procedure Revise does:

```
procedure Revise($X_i$,$X_j$):
Delete ← false
for each $v \in D_{X_i}$ do
    if there is no $v_j \in D_{X_j}$ such that $(\langle X_i \leftarrow v\rangle,\langle X_j \leftarrow v_j\rangle) \in C_{\{X_i,X_j\}}$ then
        delete $v$ from $D_{X_i}$
        Delete ← true
    endif
endfor
return Delete
```

Figure 3.1: Algorithm Revise

Note that to make every arc of the constraint graph consistent, it is not sufficient to execute Revise for each arc just once as we have seen in Section 2.2.7. Once Revise reduces the domain of some variable X_i, then each previously revised arc (X_i,X_j) has to be revised again, as some elements of D_{X_j} may no longer be compatible with any remaining elements of the revised domain D_{X_i}. The following algorithm performs a fixpoint iteration similar to that in the arc-consistency rewrite system discussed in Section 2.2.8. It thus obtains arc consistency for the whole constraint graph:

```
algorithm AC1:
Queue ← set of all arcs ($X_i$,$X_j$) in the CSP
repeat
    Change ← false
    for each ($X_i$,$X_j$) ∈ Queue do
        Change ← Revise($X_i$,$X_j$) ∨ Change
    endfor
until not(Change)
```

Figure 3.2: Algorithm AC1

The major problem with the above algorithm is that successful revision of even one arc in some iteration forces all the arcs to be revised in the next iteration even though only a small number of them are affected by this revision. In [Mackworth, 1977] a variation of

[2]Note that deletions of such values do not eliminate any solutions of the CSP.

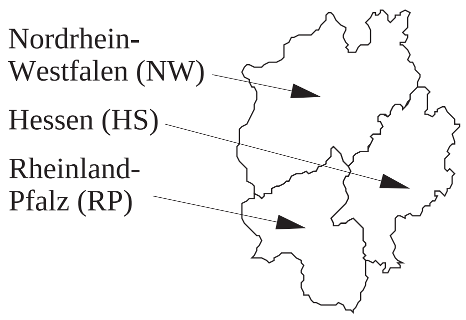
this algorithm (called AC3) is presented that eliminates this drawback. This algorithm[3] performs re-revision only for those arcs that are possibly affected by a previous revision.

```
algorithm AC3:
$Queue \leftarrow$ set of all arcs $(X_i,X_j)$ in the CSP
while $Queue$ not empty do
    select and delete some arc $(X_k,X_m)$ from $Queue$
    if REVISE$(X_k,X_m)$ then
        $Queue \leftarrow Queue \cup \{(X_i,X_k) | \, i \neq k, i \neq m\}$
    endif
endwhile
```

Figure 3.3: Algorithm AC3

Assume that the domain size for each variable is d, and the total number of binary constraints, i.e. the arcs in the constraint graph, is e. The complexity of the AC3 algorithm is $O(ed^3)$ [Mackworth and Freuder, 1985]. In [Mohr and Henderson, 1986], another arc-consistency algorithm, AC4, is presented which has a complexity of $O(ed^2)$. Since the lower bound on the time complexity is $O(ed^2)$, this is also an optimal complexity algorithm.[4] However, recently it was shown that this optimality is only of theoretical nature and that in most realistic cases "AC3 is almost always better than AC4" [Wallace, 1993].

In Section 2.2.7, we found a solution for the German-map coloring problem simply by enforcing arc-consistency for the corresponding constraint network. Thus, the question arises whether for an arc-consistent constraint graph any compound label instantiating all variables from their current domains is also a solution to the CSP, or in other words, whether achieving arc-consistency can completely eliminate the need for backtracking? If the domain size of each variable becomes 1 after obtaining arc-consistency as it was the case with the Map-coloring problem, then the CSP has exactly one solution. Otherwise, the answer is "no" in general. Consider, for example, the following sub-problem of the German-map coloring problem restricted to only coloring the states Nordrhein-Westfalen, Hessen, and Rheinland-Pfalz:



[3]The Waltz filtering algorithm [Waltz, 1972] introduced in Chapter 1 is a special case of this algorithm, and is equivalent to another algorithm AC2 discussed in [Mackworth, 1977].

[4]To verify arc-consistency, each arc must be inspected at least once which takes $O(d^2)$ steps, hence the lower bound of $O(ed^2)$.

If we restrict the domains to only allowing green and red as colors to be used, the corresponding constraint network shown in Figure 3.4 is arc-consistent, but none of the possible instantiations of the variables are a solution to the CSP.

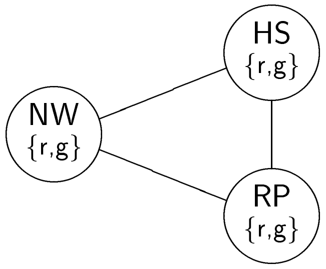


Figure 3.4: An arc-consistent but unsolvable constraint network

In general, even after achieving arc-consistency, a network may have no solutions (as shown in Figure 3.4), one solution (e.g., the original German-map coloring problem), or even more than one solution: if we do not require Hessen (HS) to be colored red in the German-map coloring problem, we obtain the following arc-consistent constraint network shown in Figure 3.5 from which two solutions can be derived.

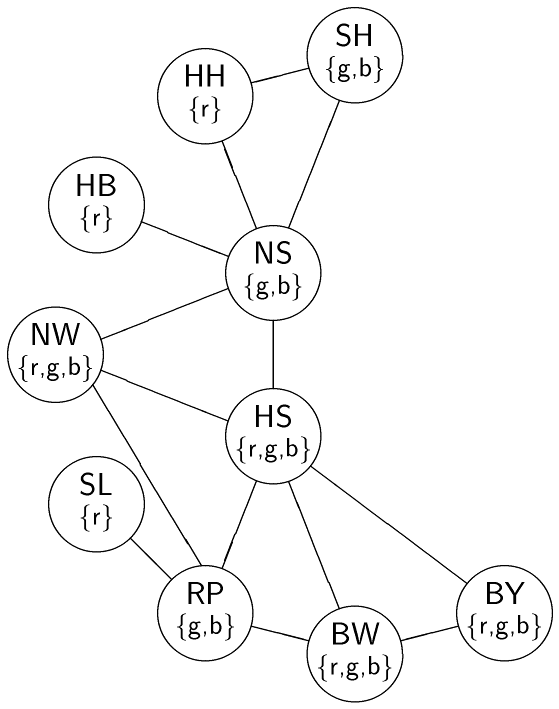


Figure 3.5: An arc-consistent constraint network with two solutions

Arc-consistency algorithms carry out a *local constraint propagation* [Voß and Voß, 1987, Güsgen and Hertzberg, 1988]. They eliminate some values from the variable domains that will never appear in a solution. To solve a problem whose corresponding graph is arc-consistent, a search algorithm with backtracking may still be needed to find the solution(s) or to discover that there is no solution because the constraints are not globally propagated and dead-end situations can arise. Nevertheless, by making the constraint graph arc-consistent, we can usually avoid a lot of unnecessary search in the backtracking procedure. The backtracking search algorithm of Section 2.3.2 will show a better performance on a locally consistent problem than on the original one because the search tree of the former has been pruned eliminating useless paths that are present in the search tree of the latter. Given the polynomial complexity of local consistency algorithms, they can be used as a preprocess to improve the problem structure before the search process starts. Figure 3.6 illustrates the gains in pruning the search space by using *local constraint propagation* before search:

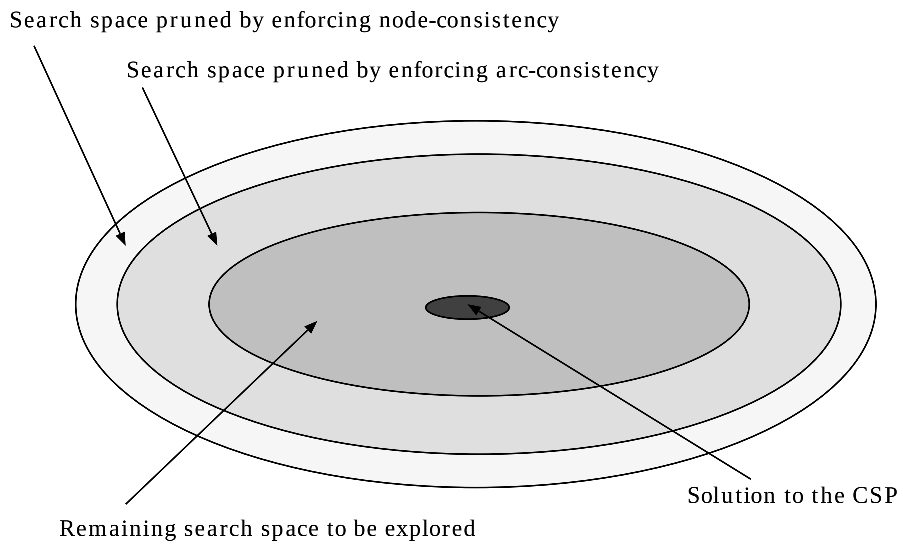

Search space pruned by enforcing node-consistency
Search space pruned by enforcing arc-consistency
Solution to the CSP
Remaining search space to be explored

Figure 3.6: Search space reduction by local constraint propagation

Waltz showed that for the problem of labeling polyhedral scenes, arc-consistency substantially reduced the search space [Waltz, 1972]. In some of the instances of this problem, the solution was found after no further search. However, if in the Map-coloring problem we also omit the other initial assignments for Hamburg (HH), Bremen (HB), and Saarland (SL), then the initial constraint graph is already arc-consistent and thus enforcing arc-consistency will not reduce the search space at all.

3.1.2 Higher-level consistency

Having seen that arc-consistency is not sufficient to eliminate the need for backtracking, we will now investigate whether there is a stronger notion of consistency that may eliminate the need for search. The notion of *k-consistency* defined below captures different degrees of consistency for different k.

Definition 20 (k-consistency) *The corresponding constraint graph for a CSP* (V, δ, C) *is* k-consistent *if for all subsets of* $k-1$ *variables* $\{X_{i_1}, \ldots, X_{i_{k-1}}\} \subseteq V$ *and for each compound label* $L_{k-1} = (\langle X_{i_1} \leftarrow v_{i_1}\rangle, \ldots, \langle X_{i_{k-1}} \leftarrow v_{i_{k-1}}\rangle)$ *that satisfies all constraints* c

among the $k-1$ variables, i.e. $\mathsf{satisfies}(L_{k-1}, c) \wedge c \in C \wedge \mathsf{vars}(c) \subseteq \{X_{i_1}, \ldots, X_{i_{k-1}}\}$*, then for each other variable* $X_{i_k} \in V - \{X_{i_1}, \ldots, X_{i_{k-1}}\}$ *there exists a value* $v_{i_k} \in D_{X_{i_k}}$ *such that the extended k-compound label* $L_k = (\langle X_{i_1} \leftarrow v_{i_1}\rangle, \ldots, \langle X_{i_{k-1}} \leftarrow v_{i_{k-1}}\rangle, \langle X_{i_k} \leftarrow v_{i_k}\rangle)$ *satisfies all constraints* $\hat{c}$ *on the k variables* $X_{i_1}, \ldots, X_{i_{k-1}}, X_{i_k}$*, i.e. the following holds:* $\mathsf{satisfies}(L_k, \hat{c}) \wedge \hat{c} \in C \wedge \mathsf{vars}(\hat{c}) \subseteq \{X_{i_1}, \ldots, X_{i_{k-1}}, X_{i_k}\}$

Definition 21 (strong k-consistency) *A constraint graph is* strongly k-consistent *if it is j-consistent for all* $j \leq k$*.*

Node-consistency discussed earlier is equivalent to strong 1-consistency, and arc-consistency is equivalent to strong 2-consistency. Strong 3-consistency is related[5] to the concept of path-consistency defined in [Montanari, 1974]. Algorithms exist to make a constraint graph strongly k-consistent for $k > 2$. Clearly, if a constraint graph containing n nodes is strongly n-consistent, then a solution to the corresponding CSP can be found without any search. But the complexity of the algorithm for obtaining n-consistency in an n-node constraint graph is also exponential. If the graph is only k-consistent for $k < n$, then in general, backtracking cannot be avoided.

3.1.3 Backtrack-free search

Although in the general case, k-consistency for $k < n$ does not suffice to eliminate backtracking at all, we can identify a certain class of FCSPs for which solutions can be found backtrack-free after achieving some level of k-consistency for $k < n$.

Definition 22 (ordered constraint graph) *An* ordered constraint graph *is a constraint graph whose nodes have been ordered linearly. A linear order on the variables in a CSP induces a corresponding ordered constraint graph.*

Note that in the backtracking paradigm, the variables of a CSP can be instantiated in many different orders (cf. step 2 in Section 2.3.2). Each ordered constraint graph of a CSP provides one such order of variable instantiations. It turns out that for some CSPs, some orderings of the constraint graph are better than other orderings in the following sense: if the backtracking algorithm uses these orderings to instantiate the variables, then it can find a solution to the CSP without search, i.e. the first path searched by backtracking leads to a solution.

Next, we define the concept of width of a constraint graph which will be used to identify such CSPs.

Definition 23 (width) *The* width *at a node in an ordered constraint graph is the number of arcs that lead from the node to the previous nodes in the linear order. The width of an ordered constraint graph is the maximum of the width of its nodes. The width of a CSP is the minimum width of all the corresponding ordered constraint graphs.*

Hence, the width of a CSP depends on the structure of the corresponding constraint graph.

[5]For complete constraint graphs, i.e. graphs in which there is a constraint between every pair of nodes, strong 3-consistency is identical to path-consistency.

Example 16 (width of the German-map coloring problem) *The width of the German-map coloring problem without taking into account the initial assignments of colors for Hamburg, Bremen, Hessen, and Saarland is 2. The variables (nodes) can be ordered as shown in Figure 3.7 (from left to right):*

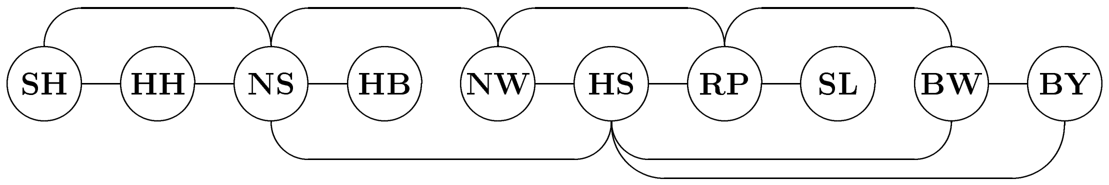


Figure 3.7: A constraint graph for the map-coloring problem with width 2

We can now prove the following proposition which relates the degree of k-consistency needed to achieve backtrack-free search to the width of a CSP:

Proposition 3 (backtrack-free search) *If a constraint graph is strongly k-consistent, and $k > w$ where w is the width of the CSP, then the search for solutions to the CSP can be performed backtrack-free.*

Proof. If w is the width of the graph, then there exists an ordering of the graph such that the number of arcs (constraints) that lead from one node of the graph to the previous nodes in the linear order is at most w. Now, if the variables are instantiated using this ordering in the backtracking paradigm, then whenever a new variable is instantiated, a value for this variable consistent with all previous assignments can be found because: (1) this value has to be consistent with the assignments of at most w past variables that are connected to the current variable, (2) the graph is k-consistent, and (3) $k > w$. □

It is relatively easy to determine the ordering of a given constraint graph that has minimum width w. It seems that all we need to do is to enforce strong $(w + 1)$-consistency using the algorithms in [Freuder, 1978]. Unfortunately, for $k > 2$, the algorithm for obtaining k-consistency adds extra arcs in the constraint graph, which can increase the width of the graph. This means that a higher degree of consistency has to be achieved before a solution can be found without any backtracking. In most cases, even the algorithm for obtaining strong 3-consistency adds so many arcs in the original n-variable constraint graph that the width of the resulting graph becomes n [Kumar, 1992]. But, however, we can use node-consistency or arc-consistency algorithms to make the graph strongly 2-consistent. None of these algorithms adds any new nodes or arcs to the constraint graph. Hence, if a constraint graph has width 1 , then after making it node- and arc-consistent, we can obtain a solution to the corresponding CSP without backtracking.

Interestingly, all tree-structured constraint graphs have width 1, i.e. at least one of the orderings of a tree-structured constraint graph has width equal to 1. Hence, if a given CSP has a tree-structured graph, it can be solved without any backtracking once it has been made node- and arc-consistent.

Example 17 (tree-structured constraint graph) *The constraint graph for the German-map coloring problem (cf. Figure 2.5) can be transformed into a tree-structured* *constraint graph by substituting each node that has already been assigned a definite value in the initial problem statement (*HH, HB, HS, *and* SL*) by one copy of the node for each arc connecting the node with any other nodes.*

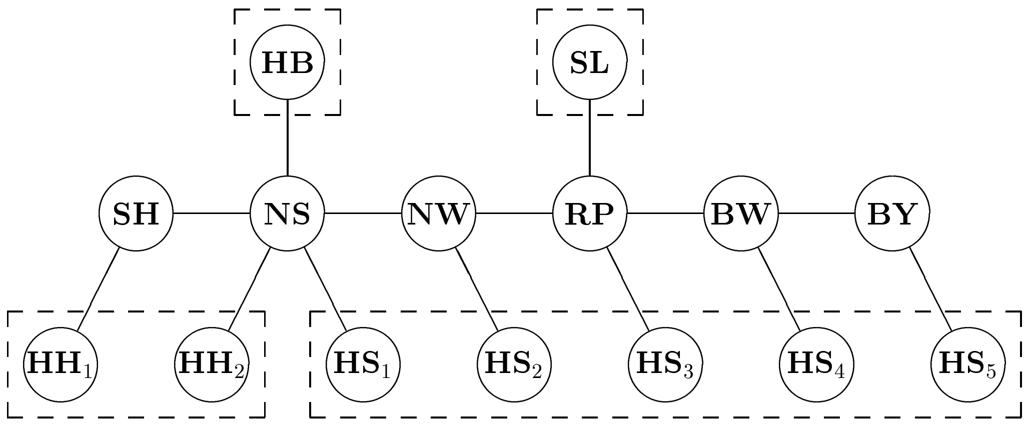


Figure 3.8: A tree-structured constraint graph for the map-coloring problem

A generalization for constraint graphs of any width is *adaptive consistency* which means to require a different consistency level for each node depending on the width of the node. Given a node ordering, the adaptive consistency algorithm visits the nodes in the inverse order. For each node X_i its width w_i is calculated, and a level of $(w_i + 1)$-consistency is achieved between the current node X_i and the set of nodes previous to X_i in the ordering and which X_i is connected with.

It can be shown that each ordered constraint graph processed with the adaptive consistency algorithm becomes backtrack-free [Dechter and Pearl, 1988a]. However, the temporal complexity of this algorithm is exponential in the graph width. Therefore, although an upper bound for this complexity can be calculated in advance in polynomial time for a given graph, for arbitrary CSPs it is unlikely that enforcing adaptive-consistency will really pay off. The same is true for several other techniques that take advantage of the structure of the constraint graph, e.g. *cycle-cutsets* [Dechter and Pearl, 1986], *tree-clustering* [Dechter and Pearl, 1988b], and *stable sets* [Freuder and Quinn, 1985].

Deciding the level of consistency that should be enforced on the network before starting backtracking search for a solution is not a clear-cut choice. Generally speaking, backtracking will benefit from representations that are as explicit as possible, having higher consistency level. However, we have seen that in general the complexity of enforcing k-consistency is exponential in k, and thus there is a trade-off between the effort spent on preprocessing and that spent on search (backtracking)[6].

As we are interested in preserving declarativity and efficiency for a large class of problems, we will focus on building a system that performs best on the problems that are most likely to appear in real-life applications.

[6]Experimental analyses of this trade-off have been published in [Dechter and Meiri, 1989] and [Haralick and Elliott, 1980].

3.2 The CONTAX System

Although specialized consistency algorithms allow for solving some CSPs very efficiently, e.g. those that can be represented by tree-structured constraint graphs, our main interest is to be able to solve those problems efficiently that are expected to occur in real-life applications.

One crucial point in this context comes with the fact that almost all consistency algorithms that can be found in the literature are limited to dealing with binary CSPs. As long as the application to be tackled fits into this restricted view, techniques like adaptive consistency may be well used on problems with small width. However, in most applications the constraints are **not** necessarily of binary nature. Also, transforming a general CSP into a binary one as discussed in Section 2.1.6 will not be of much help as the structure of the corresponding constraint graph for the resulting binary CSP will rarely exhibit the nice properties like small width that the specialized algorithms require to be efficiently applicable.

Therefore, for the design of the CONTAX constraint solver, we will focus on solving general CSPs directly without transforming them into binary CSPs, and on using a general arc-consistency algorithm for preprocessing the CSP before starting backtracking search.

3.2.1 The basic consistency algorithm in CONTAX

In order to be able to deal with general CSPs, we will first extend the concept of arc-consistency towards consistency of constraints over an arbitrary number of variables. What arc-consistency means for constraint graphs, can be lifted to hyperarc-consistency for constraint hypergraphs that represent general CSPs.

Definition 24 (hyperarc-consistency) *A given k-ary constraint C with* $\mathsf{vars}(C) = \{X_1, \ldots, X_k\}$ *is* hyperarc-consistent *if for all variables $X \in \mathsf{vars}(C)$ and all values from the current domain of X there are values in the domains of the remaining variables such that the simultaneous assignment of these k values satisfies the constraint C, i.e. the following holds:*

$$\begin{array}{l} \forall X_j \in \mathsf{vars}(C) : \\ \quad \forall v_j \in D_{X_j} : \\ \quad\quad \exists v_1 \in D_{X_1}, \ldots, v_{j-1} \in D_{X_{j-1}}, v_{j+1} \in D_{X_{j+1}}, v_k \in D_{X_k} : \\ \quad\quad\quad \mathsf{satisfies}((\langle X_1 \leftarrow v_1\rangle, \langle X_2 \leftarrow v_2\rangle, \ldots, \langle X_k \leftarrow v_k\rangle), C) \end{array}$$

A general CSP is hyperarc-consistent *if all its hyperarcs (constraints) are hyperarc-consistent.*

Note that this notion of hyperarc-consistency is undirected as opposed to arc-consistency as defined on page 38 (Definition 19) because each binary constraint on variables X_i and X_j was represented by two (directed) arcs (X_i, X_j) and (X_j, X_i) which are both checked for (directed) arc-consistency.

In order to implement an algorithm for enforcing hyperarc-consistency, a straightforward approach is to implement a procedure HyperRevise in the style of the Revise procedure (Figure 3.1 on page 48) for enforcing arc-consistency for a binary constraint.

```
procedure HyperRevise($c$):
$Delete \leftarrow$ false
for each $X_j \in \mathsf{vars}(c)$ do
    for each $v_j \in D_{X_j}$ do
        if there are no $v_1 \in D_{X_1}, \ldots, v_{j-1} \in D_{X_{j-1}}, v_{j+1} \in D_{X_{j+1}}, \ldots, v_k \in D_{X_k}$
        such that $\mathsf{satisfies}((\langle X_1 \leftarrow v_1\rangle, \langle X_2 \leftarrow v_2\rangle, \ldots, \langle X_k \leftarrow v_k\rangle), c)$
        then
            delete $v_j$ from $D_{X_j}$
            $Delete \leftarrow$ true
        endif
    endfor
endfor
return $Delete$
```

Figure 3.9: Preliminary procedure HyperRevise

For achieving hyperarc-consistency for the whole constraint network, all constraints in the CSP have to be checked using HyperRevise, and the removal of values from some variable domains must be propagated to connected constraints as it may affect their hyperarc-consistency. This can be done in a similar way as in the AC-3 algorithm (Figure 3.3 on page 49) by adding the possibly affected constraints again on the queue of constraints that have to be checked for hyperarc-consistency.

```
algorithm HyperAC3:
$Queue \leftarrow$ set of all constraints in the CSP
while $Queue$ not empty do
    select and delete some constraint $c$ from $Queue$
    if HyperRevise($c$) then
        $Queue \leftarrow Queue \cup \{\hat{c} \in C \mid \mathsf{vars}(\hat{c}) \cap \mathsf{vars}(c) \neq \emptyset\}$
    endif
endwhile
```

Figure 3.10: Preliminary algorithm HyperAC3

However, even if HyperRevise(c) decreases the domain of only one variable of c, then the HyperAC3 algorithm will put all constraints $\hat{c}$ on the *Queue* that share any variables with c, no matter whether the domain of at least one of these shared variables has been modified by HyperRevise and thus $\hat{c}$ needs to be checked again. This obviously results in a lot of unnecessary calls to HyperRevise for constraints which variable domains have not changed at all since the last time the constraint had been checked.

Therefore, the procedure HYPERREVISE has to be extended to keep track of which variable domains have been decreased and to return this information to the HYPERAC3 algorithm. By using a vector of booleans instead of a single boolean variable, *Delete*, the modified HYPERREVISE procedure can be modified as follows:

```
procedure HyperRevise($c$):
Delete[X] $\leftarrow$ false for all variables $X \in \mathsf{vars}(c)$
for each $X_j \in \mathsf{vars}(c)$ do
    for each $v_j \in D_{X_j}$ do
        if there are no $v_1 \in D_{X_1}, \ldots, v_{j-1} \in D_{X_{j-1}}, v_{j+1} \in D_{X_{j+1}}, \ldots, v_k \in D_{X_k}$
        such that $\mathsf{satisfies}((\langle X_1 \leftarrow v_1\rangle, \langle X_2 \leftarrow v_2\rangle, \ldots, \langle X_k \leftarrow v_k\rangle), c)$
        then
            delete $v_j$ from $D_{X_j}$
            Delete[$X_j$] $\leftarrow$ true
        endif
    endfor
endfor
return Delete[$X_1, \ldots, X_k$]
```

Figure 3.11: Procedure HYPERREVISE

This more detailed information about which variable domains have been decreased by HYPERREVISE can be used in the HYPERAC3 algorithm such that only those constraints will be put on the *Queue* that share at least one variable with the current constraint whose domain has been restricted:

```
algorithm HyperAC3:
Queue $\leftarrow$ set of all constraints in the CSP
while Queue not empty do
    select and delete some constraint $c$ from Queue
    Delete[...] $\leftarrow$ HyperRevise($c$)
    Queue $\leftarrow$ Queue $\cup \{\hat{c} \in C \mid \exists X \in \mathsf{vars}(\hat{c}) \cap \mathsf{vars}(c) : \textit{Delete}[X] = \text{true}\}$
endwhile
```

Figure 3.12: Algorithm HYPERAC3

Careful investigation of the above algorithm shows that in case the constraints c and $\hat{c}$ in the definition of the HYPERAC3 algorithm share a variable X whose domain has been restricted; then in a later step when taking $\hat{c}$ from the *Queue*, HYPERREVISE($\hat{c}$) will perform the outer loop also on the variable X. But as the domains of the other variables in $\hat{c}$ have not changed, this step is not necessary.

To remedy this, we can modify the algorithm such that the *Queue* holds constraint-variable pairs instead of constraints only. Then, the HYPERAC3 algorithm will in each step take one such pair (c, X_j) from the *Queue* and run only the inner loop of HYPERREVISE on c and X_j. This inner loop can also be regarded as enforcing *directed hyperarc-consistency* for a constraint C focusing on the variable X_j.

```
procedure DirectedHyperRevise($c, X_j$):
$Delete \leftarrow$ false
for each $v_j \in D_{X_j}$ do
    if there are no $v_1 \in D_{X_1}, \ldots, v_{j-1} \in D_{X_{j-1}}, v_{j+1} \in D_{X_{j+1}}, \ldots, v_k \in D_{X_k}$
    such that satisfies(($\langle X_1 \leftarrow v_1\rangle$,$\langle X_2 \leftarrow v_2\rangle$,...,$\langle X_k \leftarrow v_k\rangle$), $c$) holds
    then
        delete $v_j$ from $D_{X_j}$
        $Delete \leftarrow$ true
    endif
endfor
return $Delete$
```

Figure 3.13: Procedure DIRECTEDHYPERREVISE

It is also worth mentioning that in the case of binary CSPs DIRECTEDHYPERREVISE is equivalent to REVISE because each arc (X_i, X_j) representing a binary constraint c in one direction is represented here as the pair (c, X_i). The complementary arc (X_j, X_i) is represented as the pair (c, X_j) accordingly.

If DIRECTEDHYPERREVISE will remove a value from the domain of X_j, then for all constraints $\hat{c}$ that share the variable X_j with c, (including c itself) all pairs $(\hat{c}, X_k)$ with $X_k \in \mathsf{vars}(\hat{c})$ and $X_k \neq X_j$ will have to be put on the *Queue* to be checked again. Thus, we obtain the following algorithm for enforcing hyperarc-consistency that constitutes the kernel of the CONTAX constraint solver:

```
algorithm EnforceHyperArcConsistency:
$Queue \leftarrow \{(c, X) \mid c \in C, X \in \mathsf{vars}(c)\}$
$Consistent \leftarrow$ true
while $Queue$ not empty and $Consistent$ do
    select and delete some pair $(c, X)$ from $Queue$
    if DirectedHyperRevise($c, X$) then
        $Queue \leftarrow Queue \cup \{(\hat{c}, X_k) \mid \hat{c} \in C, X \in \mathsf{vars}(\hat{c}), X_k \in \mathsf{vars}(\hat{c}), X \neq X_k\}$
        if $D_X = \emptyset$ then $Consistent \leftarrow$ false endif
    endif
endwhile
return $Consistent$
```

Figure 3.14: Algorithm ENFORCEHYPERARCCONSISTENCY

Again, when considering binary constraints only, we obtain the AC3 algorithm as a special case of the ENFORCEHYPERARCCONSISTENCY algorithm.

3.2.2 Termination and complexity considerations

As we are dealing with finite constraint satisfaction problems, we can easily prove that the algorithm ENFORCEHYPERARCCONSISTENCY will always terminate.

Lemma 1 (Termination of DIRECTEDHYPERREVISE)
The procedure DIRECTEDHYPERREVISE *always terminates.*

Proof. As the domains of all variables are finite, there are only finitely many values $v_j \in D_{X_j}$ and only finitely many values in the domains of the other variables $X_1, \ldots, X_{j-1}, X_{j+1}, \ldots, X_k$. Thus, only a finite number of compound labels have to be checked for satisfying the given constraint c. □

Proposition 4 (Termination of ENFORCEHYPERARCCONSISTENCY)
The algorithm ENFORCEHYPERARCCONSISTENCY *always terminates.*

Proof. The algorithm terminates as soon as *Queue* becomes empty. As the number of constraints and the number of variables in an FCSP are finite, there are only finitely many possible values (pairs) on the *Queue*. In each step, the number of pairs on the *Queue* is first reduced by one, but in case that DIRECTEDHYPERREVISE returns true, a finite number of pairs is added to *Queue* and thus the number of pairs on the *Queue* may increase. However, DIRECTEDHYPERREVISE can only finitely often return true as in each such case one value is deleted from some variable's domain. Therefore, *Queue* can only finitely often be increased by a finite number of pairs, and hence after finitely many steps, it will become empty. Thus, the algorithm will always terminate. □

In the following, we will also prove that for any fixed maximum arity k of the constraints in a CSP, ENFORCEHYPERARCCONSISTENCY will establish hyperarc-consistency in polynomial time.

Lemma 2 (Complexity of DIRECTEDHYPERREVISE) *Let d be the maximum domain size and k be the maximum arity of the constraints, then the temporal complexity of* DIRECTEDHYPERREVISE *is $O(d^k)$.*

Proof. There are at most d values $v_j \in D_{X_j}$ for which at most d^{k-1} value combinations from the domains of the other variables $X_1, \ldots, X_{j-1}, X_{j+1}, \ldots, X_k$ can be found that have to be checked for satisfying the given constraint c. Checking a compound label L for satisfying a constraint c means checking whether $L \in c$. It can thus be regarded a constant-time operation. Hence, the overall complexity is $O(d^k)$. □

Proposition 5 (Complexity of ENFORCEHYPERARCCONSISTENCY) *Given a finite constraint satisfaction problem with e constraints of maximum arity k and n variables ranging over domains with maximum cardinality d. Then, for each fixed value for k the algorithm* ENFORCEHYPERARCCONSISTENCY *has a polynomial temporal complexity.*

Proof. The complexity considerations are based on the observation that new pairs can be added on the *Queue* only if DIRECTEDHYPERREVISE returns true which can happen at most $d \times n$ times, because each time one value is deleted from some variable's domain.

Whenever new constraint-variable pairs are put on the *Queue*, their number is limited by $e \times n$. Therefore, DIRECTEDHYPERREVISE will be called at most $dn \times en$ times, which results in an overall complexity for the algorithm ENFORCEHYPERARCCONSISTENCY of $O(n^2ed^{k+1})$. Hence, for any fixed k, the time that it needs to establish hyperarc-consistency is limited by a polynomial function in the number of variables, constraints, and the maximum domain size. □

We already recognized that the AC3 algorithm can be seen as a special instance of the ENFORCEHYPERARCCONSISTENCY algorithm. When considering only binary constraints ($k = 2$), hyperarc-consistency reduces to arc-consistency and can be established using the ENFORCEHYPERARCCONSISTENCY algorithm in time cubic in the domain size—just as with the AC3 algorithm presented in Section 3.1.1.

Thus, we have shown that the ENFORCEHYPERARCCONSISTENCY algorithm provides an efficient means to establish hyperarc-consistency for a given CSP, such that the resulting variable domains can be efficiently searched to finally obtain a solution to the CSP. This algorithm constitutes the kernel of the CONTAX system that will be presented in the following section.

3.2.3 The CONTAX system architecture

The ENFORCEHYPERARCCONSISTENCY algorithm presented in the previous section is used in CONTAX to establish hyperarc-consistency in the constraint network that corresponds to a given CSP. In Section 3.1.1 we have seen that, in general, arc-consistency in binary CSPs does not suffice to avoid backtracking search for ultimately solving the problem. The same is obviously true for hyperarc-consistency in the case of general CSPs. Therefore, in CONTAX we use the ENFORCEHYPERARCCONSISTENCY algorithm for preprocessing the constraint network and then explore the remaining search space by using a backtracking search algorithm, called GENERATESOLUTIONS, which instantiates the generic backtracking search schema presented in Section 2.3.2 as follows:

- The GENERATESOLUTIONS algorithm runs only on those variables that have not already been assigned a single value by ENFORCEHYPERARCCONSISTENCY. The other variables which have already been labeled are considered as past variables.

- For variable selection (step 2), the algorithm uses a *first-fail* heuristics which selects as current variable the one with the currently smallest domain. The idea is that by using this heuristics invalid assignments to past variables can be discovered earlier as a smaller number of values for the current variable have to be tried unsuccessfully.

- Most important, the test for consistency of the current assignment (step 4) is performed not only on the current and past variables but also involving the future variables: the domain of the current variable is temporarily restricted to hold only the one value that has been selected in step 3; then the algorithm again tries to establish hyperarc-consistency in the constraint network. If it succeeds, the value assignment passed the test, otherwise not. Thus, the algorithm performs search space reduction not only before search starts (preprocessing) but also dynamically each time a value assignment is committed to. As the domain of the current variable is set to hold the selected value when checking consistency, the algorithm maintains a data structure Δ_{X_i} that holds the remaining values that can be tried for X_i.

- Backtracking (step 6), if necessary, is organized using a stack-based data structure for recording past variable selections. Thus, the algorithm basically performs chronological backtracking. Because checking consistency may also reduce the domains of some future variables in contrast to the generic backtracking scheme, these restrictions have to be undone when backtracking to an earlier variable. Therefore, choice points on the stack S do not only contain the variable X_i itself to which the system can backtrack but also stores the remaining values Δ_{X_i} that can be tried for X_i and, most important, the current domains D_{X_j} for all variables X_j that have not yet been instantiated.

Figure 3.15 presents the algorithm GENERATESOLUTIONS as it is used in the current implementation of the CONTAX system.

```
algorithm GENERATESOLUTIONS:
INIT:        if ENFORCEHYPERARCCONSISTENCY = false then stop endif
SELVAR:      select a variable $X_i \in V$ with minimum $| D_{X_i} | > 1$
             if there is no such variable left then goto SUCCESS endif
             $\Delta_{X_i} \leftarrow D_{X_i}$
SELVAL:      if $\Delta_{X_i} = \emptyset$ then goto BACKTRACK endif
             select and delete a value $v \in \Delta_{X_i}$
             $D_{X_i} \leftarrow \{v\}$
CHECK:       $Inconsistent \leftarrow$ false
             $Queue \leftarrow \{(c, X) \mid c \in C, X_i \in \mathsf{vars}(c), X \in \mathsf{vars}(c), X_i \neq X\}$
             while $Queue$ not empty and not($Inconsistent$) do
                 select and delete some pair $(c, X)$ from $Queue$
                 if DIRECTEDHYPERREVISE$(c, X)$ then
                     $Queue \leftarrow Queue \cup \{(\hat{c}, X_k) \mid \hat{c} \in C, \{X, X_k\} \subseteq \mathsf{vars}(\hat{c}), X_k \neq X\}$
                     if $D_X = \emptyset$ then $Inconsistent \leftarrow$ true endif
                 endif
             endwhile
             if $Inconsistent$ then goto SELVAL endif
             push choice point $(X_i, \Delta_{X_i}, \{D_{X_j} \mid X_j$ not yet assigned$\})$ on $S$
             goto SELVAR
SUCCESS:     print solution: $D_{X_1}, D_{X_2}, \ldots, D_{X_n}$
             if more solutions required then goto SELVAL else stop endif
BACKTRACK:   if $S$ is empty then stop endif
             restore $X_i$, $\Delta_{X_i}$, and the domains $D_{X_j}$ from the top element in $S$
             delete the top element from $S$
             goto SELVAL
```

Figure 3.15: Algorithm GENERATESOLUTIONS

In the next section, we will now study how the Contax system appears to the user, that is, how FCSPs have to be formulated for being solved by Contax using the GenerateSolutions algorithm.

3.2.4 A user's view on Contax

An FCSP basically consists of a set of variables ranging over some finite domains and a set of constraints. Thus, in order to support the formulation of an FCSP in Contax, the system has to provide means for representing domains, variables, and constraints.

When using a constraint solver in practice, it often appears that many variables range over the same domain, or that an instance of the same constraint is required to hold on various sets of variables. Therefore, in designing the Contax system we decided to provide domain and constraint *definitions* separately from the concrete variables and constraints that constitute a constraint network for the given FCSP. In order to use Contax for representing and solving an FCSP, one has to perform the following tasks:

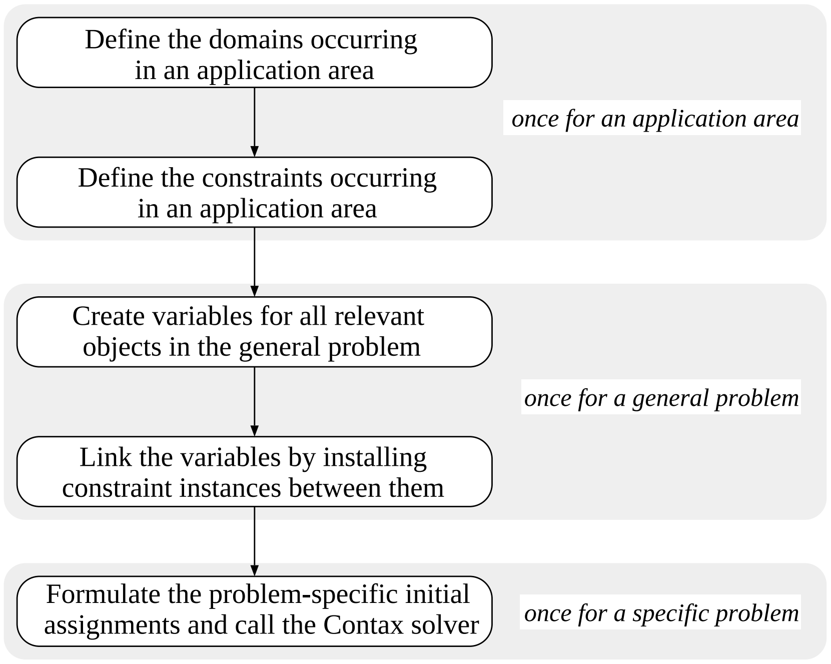

Define the domains occurring in an application area
Define the constraints occurring in an application area
once for an application area
Create variables for all relevant objects in the general problem
Link the variables by installing constraint instances between them
once for a general problem
Formulate the problem-specific initial assignments and call the Contax solver
once for a specific problem

Figure 3.16: Five steps in representing and solving a CSP with Contax

The main advantages of this stepwise domain and problem formulation are that Contax supports

- defining the domains and constraint types occurring in an application area independently from any concrete problem (*defining the problem area*),
- then creating a constraint network for a given problem independently from the concrete values for some parameters (*defining the problem in general*), and

- finally incorporating the concrete data for one specific problem instance to be solved.

Defining the problem area (domains and constraints)

Domains in CONTAX are finite discrete sets of values. They can be easily defined by enumerating all elements in a domain. For example, the statement

```
(def-domain colors (red green blue))
```

introduces a new domain named `colors` that holds the set of colors that can be used in the map-coloring example.

CONTAX provides different types of constraints: primitive (or extensional), predicative, and compound constraints. All constraint types may be defined over any number of variables.

Primitive constraints are defined by enumerating all the value combinations (compound labels) that satisfy the constraint. Since basically to check whether a compound label satisfies the constraint is the same as to lookup a database, this kind of constraint can also be regarded as a *database constraint.* Consider, for example, the following definition of a constraint requiring different colors to be assigned to its variables:

```
(def-primitive-constraint different-colors
    :interface (color1 color2)
    :domains (colors colors)
    :tuples ((red green)
             (red blue)
             (green red)
             (green blue)
             (blue red)
             (blue green)))
```

Some constraints occurring in a real-life application are difficult or even impossible to be explicitly enumerated as primitive constraints. This is true, for example, for equalities, inequalities or numerical comparisons. Therefore, CONTAX allows for defining a constraint by providing a LISP function or LISP expression (as argument to the `:predicate` keyword) which then will be evaluated each time to test a given compound label for satisfying the constraint.[7] Consider again the `different-colors` constraint in the map-coloring example. This constraint can also be defined as such a *predicative constraint* as follows:

```
(def-predicative-constraint different-colors
    :interface (color1 color2)
    :domains (colors colors)
    :predicate (not (equal color1 color2)))
```

[7]Note that this extension is compatible with our definitions so far if we extend definition 10 on page 27 accordingly.

Often it may happen that the same constraint subnet occurs many times between different variables of the entire CSP. Therefore, it becomes very useful to define this subnet as a *compound constraint* which itself represents an entire constraint net. *Local variables* of the constraint subnet that only serve to connect local constraints need not to occur in the `:interface` list.

For example, one may be interested in defining an inequality constraint over three variables in the map-coloring problem. In CONTAX this can be easily done by defining a compound constraint that combines three binary inequality constraints into one constraint of the following form:

```
(def-compound-constraint different3
    :interface (color1 color2 color3)
    :constraints ((different-colors color1 color2)
                  (different-colors color1 color3)
                  (different-colors color2 color3)))
```

Defining the problem in general

For software engineering reasons, the CONTAX system is implemented in an object-oriented fashion using the COMMON LISP OBJECT SYSTEM (CLOS) [Bobrow *et al.*, 1988] such that constraint networks are directly represented as networks of CLOS objects. Therefore, the constraint network for a given CSP can simply be built by first creating objects representing variables and then linking variables together by creating instances of the constraint classes that have been defined before.

For example, the variables `HH` and `SH` that represent the colors assigned to Hamburg and Schleswig-Holstein respectively are created as follows:

```
(make-variable :name HH
               :domain colors)
(make-variable :name SH
               :domain colors)
```

After having created variable objects for all problem variables, the problem constraints can then be posed on the variables by creating constraint instances and connecting them to the variables that they shall constrain.

For example, the constraint named `C1` requiring that Hamburg and Schleswig-Holstein should be colored differently can directly be represented by creating an instance of the `different-colors` constraint and linking this constraint instance to the variables `HH` and `SH`:

```
(make-constraint :name C1
                 :type different-colors
                 :color1 HH
                 :color2 SH)
```

By using this object-oriented implementation and representation schema, the CONTAX system provides a user interface for formulating constraint satisfaction problems very declaratively.

As an example, Figure 3.17 shows (one possible) CONTAX representation of the German-map coloring problem using a compound constraint for combining all neighboring constraints between the states. Of course, we could also explicitly create as many instances of the `different-colors` constraint as there are borders between states in Germany; this is exactly what the CONTAX system automatically does when creating an instance of the compound constraint `german-map-coloring`.

```
(def-domain colors (red green blue))

(def-primitive-constraint different-colors
   :interface (color1 color2)
   :domains (colors colors)
   :tuples ((red green) (red blue) (green red)
            (green blue) (blue red) (blue green)))

(def-compound-constraint german-map-coloring
   :interface (SH HH NS HB NW HS RP SL BW BY)
   :constraints ((different-colors SH HH) (different-colors SH NS)
                 (different-colors HH NS) (different-colors NS HB)
                 (different-colors NS NW) (different-colors NS HS)
                 (different-colors NW HS) (different-colors NW RP)
                 (different-colors HS RP) (different-colors HS BW)
                 (different-colors HS BY) (different-colors RP SL)
                 (different-colors RP BW) (different-colors BW BY)))

(make-variable :name SH :domain colors)
(make-variable :name HH :domain colors)
(make-variable :name NS :domain colors)
(make-variable :name HB :domain colors)
(make-variable :name NW :domain colors)
(make-variable :name HS :domain colors)
(make-variable :name RP :domain colors)
(make-variable :name SL :domain colors)
(make-variable :name BW :domain colors)
(make-variable :name BY :domain colors)

(make-constraint :name german-map-coloring-problem
                 :type german-map-coloring
                 :SH SH :HH HH :NS NS :HB HB :NW NW
                 :HS HS :RP RP :SL SL :BW BW :BY BY)
```

Figure 3.17: Representing the German-map coloring problem in CONTAX

Solving a specific problem instance with Contax

Once a CSP has been represented in Contax, the system is ready to start searching for a solution to the CSP. Many real-life CSPs are often defined once very generally, and then several concrete instances have to be solved that differ, e.g., only in some initial value assignments for some variables. Therefore, Contax allows to specify initial assignments for a subset of the variables when starting to solve a CSP.

For example, in the German-map coloring problem the assignments for Hamburg, Bremen, Hessen, and Saarland can be specified when finally submitting the problem to Contax for computing a consistent map-coloring. The following call to the function `solve` starts solving the map-coloring problem and returns the following list of all, here exactly one, solutions.

```
(solve german-map-coloring-problem
       :HH (red) :HB (red) :HS (green) :SL (red)
       :solutions all)

⟹ ((SH HH NS HB NW HS RP SL BW BY)
    (green red blue red red green blue red red blue))
```

Figure 3.18: Solving the German-map coloring problem with Contax

The only difference between creating an instance of the compound constraint `german-map-coloring` and creating 14 seperate instances of the `different-colors` constraint is the following: Although in both ways Contax generates exactly the same network of clos objects, in the latter case we cannot provide a name for our constraint problem to later refer to it. Thus, the only way then to restrict propagation to the constraints involved in the map-coloring problem would be to provide a list of them with the call to the function `solve` or to simply propagate `all`[8] constraints in case there are no other constraints currently present in the system. At the end of the next chapter, we will show how to represent and solve an extended version of the Map-coloring problem with Contax (Figure 4.9 on page 87). There we will need to explicitly create constraint instances for each border and will thus have to run Contax on all constraint instances presently known to the system.

3.3 Discussion and Related Work

So far, we have now presented the kernel of the Contax system which allows for declaratively representing finite constraint satisfaction problems. In Section 3.2.2, we have also

[8]Trying to solve `all` constraints can only become a problem if there are several independant constraint networks loaded into Contax. Even if all but the one we are interested in would be in a consistent state, using the '`all`' option would initially check all constraints, i.e. put them on the queue for EnforceHyperArcConsistency.

seen that the basic consistency algorithm used in CONTAX supports an efficient preprocessing of the constraint network such that the remaining search space can be explored afterwards using an instance of the generic backtracking search algorithm introduced in Section 2.3.2. Thus, the CONTAX system as presented in this chapter already helps declarativity meet efficiency.

The functionality of the CONTAX kernel is not very different from other CSP solvers. Indeed, it was initially inspired by Güsgen's CONSAT system [Güsgen, 1988] from where also some terminology and syntax has been taken. CONSAT also provides means for a purely declarative problem formulation as CONTAX does. However, as efficient problem solving is concerned, the algorithms implemented in CONTAX clearly outperform the results obtained with CONSAT[9]. Parts of the efficiency of the CONTAX system stems from its object-oriented implementation and from the compilation of constraint networks into networks of CLOS objects such that constraint propagation is mapped to message passing on the level of CLOS objects.

In contrast to CONTAX, CONSAT offers an indexing mechanism that uses dependancy structures to record the information about satisfying compound labels that have been generated when checking hyperarc-consistency. By using these index structures, some amount of backtracking search can be avoided. However, when running CONSAT in `:INDEXED` mode on some non-toy problems it appeared that indexing structures rapidly became very large and their maintenance soon required most of the time that was gained by avoiding some backtrack search. Therefore, no indexing mechanism has been implemented in CONTAX; instead, backtracking search was interleaved with consistency checking. This approach appears to be less sensitive to growing size (in the number of constraints) of a problem and therefore seems well suited for tackling real-life applications. It will therefore be used as the implementational basis for further extensions of our approach that are needed to make use of some special characteristics of and to meet the requirements posed by real-life applications.

In the next chapter, we will now investigate how to extend our approach in order to be able to deal with *overconstrained* problems which very often occur in various real-life applications and which cannot be dealt with using standard CSP techniques—also not with the CONTAX system as we have presented it so far. Hence, extending our approach towards dealing with overconstrained problems will also contribute to let theory meet application.

[9]We have installed and tested the Allegro CommonLisp implementation of CONSAT on a Macintosh IIfx and compared with CONTAX in Lucid CommonLisp on a Sun 4. Even when taking into account the different hardware environment by using a factor of 3 to get runtime results comparable, most of the benchmarks still ran 5 to 12 times faster using CONTAX.

4

Partial Constraint Satisfaction and Constraint Relaxation

In the previous chapter, we have discussed basic constraint propagation techniques and their extension to dealing with constraints of arbitrary arity, and have presented their realization in the CONTAX system. CONTAX as presented so far, provides an efficient tool for solving FCSPs—as long as they are solvable at all.

However, in real-life applications we are often faced with problems for which no solution exists satisfying all constraints and which therefore have to be relaxed in some way to become at least partially solvable. This class of *partial constraint satisfaction problems* (PCSPs) will be studied in Section 4.1 and will be illustrated by our running map-coloring example. In Section 4.2, we will then present techniques for solving such overconstrained problems by using *constraint relaxation* methods. The methods developed will enable us to solve an overconstrained problem as good as possible. This will then lead us to the problem of how to measure the quality of an approximate solution, or the other way around, how to identify different degrees of relaxation. These discussions will again be motivated by using an extension of the German-map coloring problem that will illustrate the basic issues for constraint relaxation. As constraint relaxation has been identified as a crucial topic for solving real-life application problems, Section 4.3 will finally present how CONTAX has been extended to support weighted constraints and how to perform constraint relaxation using these weights.

4.1 Partial Constraint Satisfaction Problems

Standard constraint satisfaction methods halt when no solution can be found satisfying all constraints simultaneously. In contrast, *partial constraint satisfaction* involves finding values for the variables that satisfy a subset of the constraints [Freuder, 1992]. This means, we are willing to *relax* some of the constraints to permit additional acceptable value combinations. Partial constraint satisfaction problems arise in several contexts:

- The problem is overconstrained and admits of no complete solution.
- The problem is too difficult to solve completely but we are willing to settle for a "good enough" solution.

- We are seeking the best solution obtainable within fixed resource bounds.
- Real time demands require an any-time algorithm, which can report *some* partial solution almost immediately, improving on it if and when time allows.

Let us again consider our running map-coloring example. In Chapter 2 we introduced the German-map coloring problem limited to coloring the political map of West Germany as of before the reunification in 1990 (Problem 1, page 23). We will now update this problem and extend it by including the new eastern states:

Problem 2 (Extended German-map coloring problem) *The extended German-map coloring problem is to decide how to color the political map of Germany as shown in Figure 4.1 using only three colors,* red, blue, *and* green, *such that neighbored states are colored differently.*

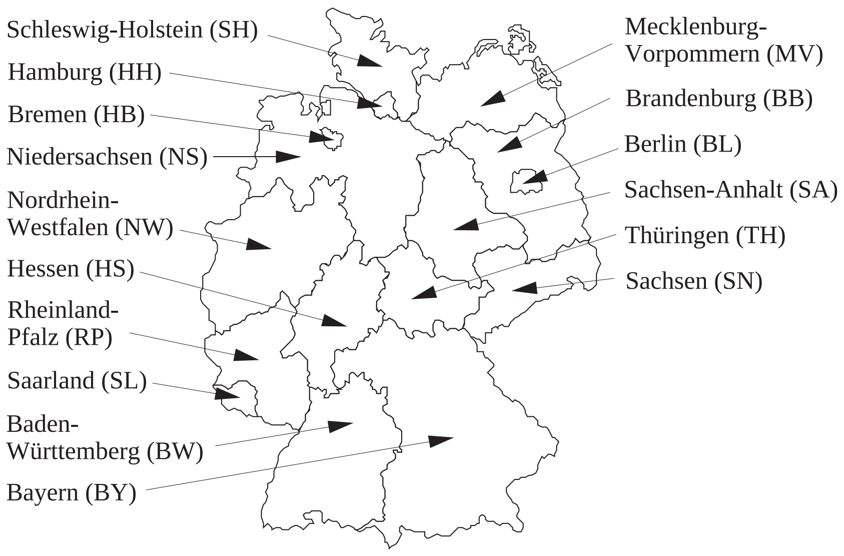


Figure 4.1: The political map of reunified Germany

It can be easily shown that this problem is overconstrained, i.e. it is impossible to color the political map of Germany with three colors such that all neighboring states are colored differently. Consider the state Thüringen which is surrounded by a ring of five neighboring states. One color is needed for Thüringen itself, thus two colors are left for its neighbors which obviously does not suffice as the ring contains an odd number of them. Figure 4.2 shows the constraint network corresponding to the extended map-coloring problem.

Assume that for some reasons we are **not** allowed to use an additional color, i.e. to extend the domains of the variables, but on the other side are forced to come up with at least a partial solution to the problem. Thus, we need to find a solution to a relaxed version of the problem that is as close as possible to the original formulation. The term "as close as possible" already shows that partial constraint satisfaction problems can also be regarded as a kind of optimization problem. In the following section, we will take

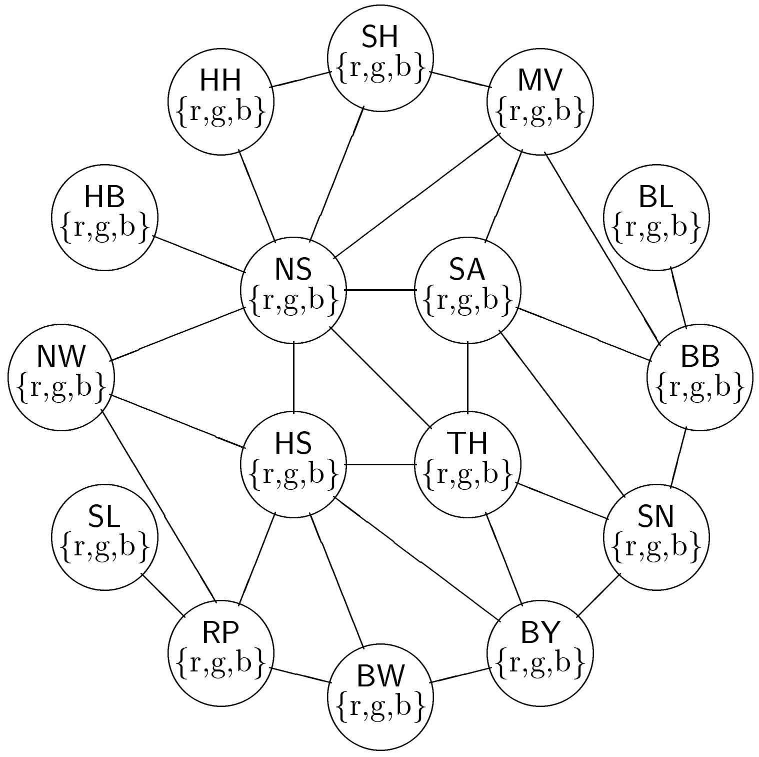


Figure 4.2: The constraint network for the extended map-coloring problem

this optimization view on constraint relaxation and will present techniques for finding an *optimal* solution to a PCSP which relaxes the problem only as much as necessary to obtain a solution.

4.2 Constraint Relaxation Techniques

For now we define a partial constraint satisfaction problem (PCSP) as a CSP where we are willing to accept a solution that violates some of the constraints. A more formal definition will be given in Section 4.3. An optimal solution in this setting is one that satisfies as many constraints as possible. Thus, we will refer to this setting as *maximal constraint satisfaction*. In Section 2.3, we have identified backtracking as the classic algorithm for solving CSPs. As we recognized maximal constraint satisfaction as an optimization problem, our attention is drawn to branch and bound [Lawler and Wood, 1966, Reingold *et al.*, 1977] which is a widely used optimization technique that may be viewed as a variation on backtracking. Thus, it is a natural choice to study how to use branch and bound for finding optimal, specifically maximal, partial solutions to a PCSP.

4.2.1 Maximal constraint satisfaction

Branch and bound for maximal constraint satisfaction is the natural analogue of backtracking for constraint satisfaction. Backtrack search tries all value combinations exhaustively if necessary, but can avoid considering some combinations by observing that a subset of values cannot be extended to a full solution, thus pruning a subtree of the search space. Branch and bound operates in a similar fashion to backtracking in a context where we are seeking a *maximal solution*, one which satisfies as many constraints as possible. Branch and bound basically keeps track of the best solution found so far and abandons a line of search when it becomes clear that it cannot lead to a better solution. A version of backtracking that searches for all solutions rather than the first solution, most naturally compares with the branch and bound extension to find a maximal solution.

The branching in the search tree corresponds to variable values, e.g. the choice of red for HH. The nodes in the search tree represent labels, i.e. assignments of values to variables during the search. The search path leading down to the most recently chosen label is the *current search path*. It holds as a compound label the current set of choices of values for variables. Thus, it represents a proposed *incomplete* solution, unless it includes values for all the variables.

Search concludes when we find a perfect solution, which is of course not available in overconstrained problems like the extended map-coloring problem, or run out of things to try. We could also quit when we reach a preset *sufficient bound* ε_0, which specifies that we will be satisfied if we find a partial solution that violates no more than ε_0 constraints. We may know, for example, that no perfect solution to the map-coloring problem is possible and thus be able to set ε_0 to 1. We may be willing to settle for a "close enough" or *sufficient solution*. Obviously the larger we set ε_0 the easier the problem is likely to be.

Circumstances may also impose resource bounds. In particular, real time processing may require immediate answers that can be refined later if time allows. The branch and bound process is well suited to providing resource-bounded solutions. We can simply report the best solution available, when, for example, a time bound is exceeded. The branch and bound process is also clearly well-suited to support an any-time algorithm, which can repeatedly provide a "best-so-far" answer when queried. It can quickly provide *some* answer with a better one perhaps to follow as time allows.

A branch-and-bound algorithm for finding a maximal solution to a PCSP

The function FindMaximalSolution shown in Figure 4.3 performs a branch and bound search for finding the maximal solution to a PCSP. It takes five arguments: $\mathcal{P}$ represents the current search path, ε measures the number of constraints violated by the labels in $\mathcal{P}$. X is the variable to be assigned at the current node in the search tree, $\mathcal{V}$ is the set of variables that remain to be labeled (including X), and Δ_X is the (remaining) set of values that can be tried for X.

The variables $\mathcal{P}_{\min}$, $\varepsilon_{\min}$, and ε_0 are global to the function FindMaximalSolution, all other variables are local. The variable $\mathcal{P}_{\min}$ represents the best solution found so far; $\varepsilon_{\min}$ is used during the search to store the number of inconsistencies in the best solution found up to that point in the search. $\varepsilon_{\min}$ is the *necessary bound* in the sense that to do better it is necessary to find a solution with fewer inconsistencies. The necessary bound $\varepsilon_{\min}$ can be set initially based on a priori knowledge that a solution is available

```
function FindMaximalSolution($\mathcal{P}$, $\varepsilon$ , $X$, $\mathcal{V}$, $\Delta_X$):
if $\mathcal{V} = \emptyset$ then /* $\mathcal{P}$ holds a compound label assigning values to all variables */
    $\mathcal{P}_{\min} \leftarrow \mathcal{P}$
    $\varepsilon_{\min} \leftarrow \varepsilon$
    if $\varepsilon_{\min} \leq \varepsilon_0$ then return true else return false endif
 else if $\Delta_X = \emptyset$ then /* all values in $D_X$ have been tried for extending $\mathcal{P}$ */
    return false
    endif
 else /* try to extend $\mathcal{P}$ using another value $v \in \Delta_X$ */
    select a value $v \in \Delta_X$
    $\hat{\varepsilon} \leftarrow \varepsilon$ /* check whether we can do better when committing to $\langle X \leftarrow v \rangle$ */
    $C_X \leftarrow \{c \in C \mid X \in \mathsf{vars}(c) \wedge \mathsf{vars}(c) \subseteq \mathsf{vars}(\mathcal{P} \cup \{\langle X \leftarrow v \rangle\})\}$
    while $C_X \neq \emptyset$ and $\hat{\varepsilon} < \varepsilon_{\min}$ do
        select and delete a constraint $\hat{c}$ from $C_X$
        if $\neg\mathsf{satisfies}(\mathcal{P} \cup \{\langle X \leftarrow v \rangle\}, \hat{c})$ then $\hat{\varepsilon} \leftarrow \hat{\varepsilon} + 1$ endif
    endwhile
    select a variable $\hat{X} \in \mathcal{V} - \{X\}$ unless $\mathcal{V} - \{X\} = \emptyset$
    if $\hat{\varepsilon} < \varepsilon_{\min}$ and FindMaximalSolution($\mathcal{P} \cup \{\langle X \leftarrow v \rangle\}$, $\hat{\varepsilon}$ , $\hat{X}$, $\mathcal{V} - \{X\}$, $D_{\hat{X}}$)
    then
        return true
    else
        return FindMaximalSolution($\mathcal{P}$, $\varepsilon$ , $X$, $\mathcal{V}$, $\Delta_X - \{v\}$)
    endif
endif
```

Figure 4.3: Function FindMaximalSolution

that violates fewer than $\varepsilon_{\min}$ constraints, or on an a priori requirement that we are not interested in any solutions that violate more than $\varepsilon_{\min} - 1$ constraints. As branch and bound proceeds, if a solution is found that violates $\hat{\varepsilon} < \varepsilon_{\min}$ constraints, $\varepsilon_{\min}$ is replaced by $\hat{\varepsilon}$.

In this algorithm as in all retrospective procedures, a value $v \in \Delta_X$ being considered for inclusion in the solution $\mathcal{P}$ is compared with values already chosen, to determine whether constraints between the instantiated variables $\mathsf{vars}(\mathcal{P} \cup \{\langle X \leftarrow v \rangle\})$ and the current one X are satisfied. Each such test is a constraint check. Since the total number of constraint checks is a standard measure of CSP algorithm efficiency, we wish to minimize this quantity. To this end, the new distance $\hat{\varepsilon}$ is compared with the bound $\varepsilon_{\min}$ after each constraint check, so that if the bound is reached, the current value v is not checked further. Note that, due to our procedure for minimizing constraint checks, some checks are avoided at many points of the search tree, including at some of the lowest-level nodes. In some cases subtrees are pruned; in some cases a value does not need to be checked against the entire preceding search path.

The general worst-case bound for this algorithm is of course exponential. However, it is no worse than that for a backtracking algorithm for finding a perfect solution. Both, in the worst case, will end up trying all possible combinations of values, and testing all the constraints among them. On the other hand, the exponential worst-case bound is bad enough. Therefore, we will in the following consider techniques that may help to avoid achieving this bound.

Finding the maximal solution to the extended map-coloring problem

The function FindMaximalSolution can be easily applied to the extended map-coloring problem where the necessary bound $\varepsilon_{\min}$ is set to ∞ as we do not know about any partial solution so far, and the sufficient bound ε_0 can be set to 1 as we already know that no perfect solution exists. Calling the function FindMaximalSolution on

- the empty compound label $\emptyset$ as the current search path $\mathcal{P}$,
- its distance from a perfect solution being $\varepsilon = 0$ (the empty search path does not violate any constraints),
- some arbitrarily chosen start variable $X \in V$, say SH,
- the set of remaining variables $\mathcal{V}= V$,
- and the domain $\Delta_X= D_X = \{$red, green, blue$\}$

will return 'true' which means that a maximal solution with distance 1 has been found, i.e. which violates only one constraint. The corresponding map-coloring is shown in Figure 4.4.

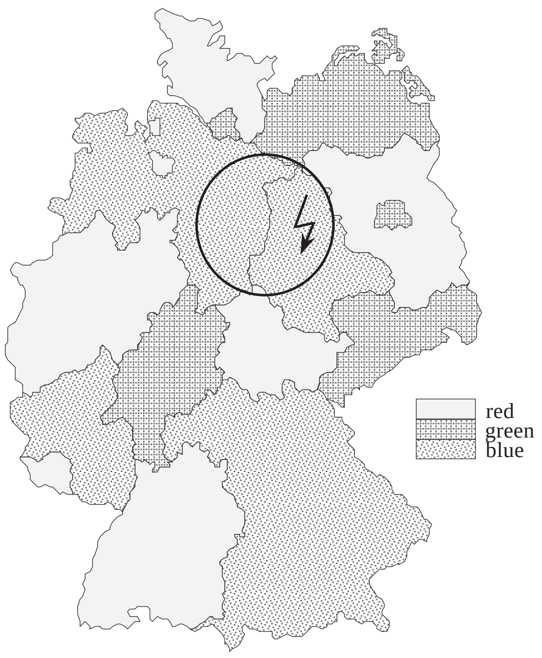


Figure 4.4: A maximal solution to the extended map-coloring problem

Although the solution shown in Figure 4.4 is optimal in the sense that it violates the smallest number of constraints, it appears that it may not be the best solution for the purpose of map-coloring: The idea behind the constraint that neighbored states should be colored differently is that the area of different states can easily be recognized when having a look at the colored map. The solution in Figure 4.4 violates only one constraint, namely the constraint which requires Niedersachsen and Sachsen-Anhalt to take different colors. However, the reader may agree that violating this constraint should be regarded much worse than violating, for example, the constraint between Mecklenburg-Vorpommern and Sachsen-Anhalt which share a much smaller common border. Thus, it appears that the problem with the maximal constraint satisfaction approach as discussed so far is that all constraints are treated equally and that the approach offers no means for expressing priorities or preferences etc.

4.2.2 Optimal constraint satisfaction

The above mentioned drawback can be remedied when we allow to express with each constraint some kind of weight expressing, for example, the importance of the constraint being satisfied for obtaining a *useful* solution.

Most classical optimization approaches in Operations Research [Budnick *et al.*, 1977] use real numbers as weights and perform optimization using a weighted sum as a *cost function* in order to evaluate the quality of a solution and the effect on it when changing value assignments.

We can easily adapt this view to the maximal constraint satisfaction approach by specifying a *weight* for each constraint which expresses its importance to be satisfied in a partial solution. Thus, we obtain the following definition where we assume the set of weights W to be the set of real numbers $\mathbb{R}$:

Definition 25 (weighted constraint) *A* weighted constraint *C is a constraint that has been assigned a weight w from a given set of weights W. The expression* $\mathsf{weight}(C)$ *denotes the weight of a constraint C.*

For incorporating weighted constraints in the branch-and-bound algorithm of Figure 4.3, the only necessary change is to increase the distance $\hat{\varepsilon}$ of a partial solution by the weight $\mathsf{weight}(\hat{c})$ of a constraint $\hat{c}$ if assigning the current value v to X violates the constraint $\hat{c}$.

Finding the optimal solution to the extended map-coloring problem

The introduction of weighted constraints will now enable us to find an optimal solution to the extended map-coloring problem in contrast to the maximal solution (with minimum number of constraint violations) shown in Figure 4.4. By using the length of the common border between two states represented by the variables X and Y as the weight of the constraint $C_{\{X,Y\}}$ on them, the function FindMaximalSolution finds an optimal solution which is shown in Figure 4.5.

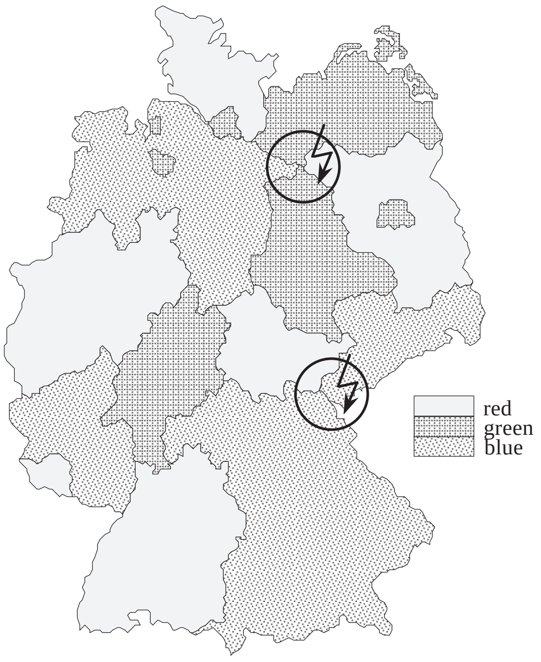


Figure 4.5: An optimal solution to the extended map-coloring problem

This solution is different from the one in Figure 4.4 and is not maximal in the number of constraints satisfied. However, it seems obvious that for the purpose of easily recognizing the areas of the states, the *optimal* solution should be preferred to the *minimal* one.

From real-weighted constraints to discrete weights

In the map-coloring problem it appears that real-weighted constraints, i.e. having real numbers as weights, are well suited for representing the importance of the constraints which directly correlates with the length of the border between the corresponding states. Thus, in this example, one can exactly measure the importance of a constraint and express it as a real number. This, however, needs not to be the case in many other applications where the importance of a constraint can in the best case be expressed qualitatively. For example, in the lathe-tool selection application to be presented in Chapter 8 we have economical constraints of the form

"In the roughing process, a square insert should be preferred to a triangular one."

as well as some geometrical constraints like the following:

"The sum of the angles α, β, and γ has to be less or equal $180°$*."*

Obviously, the latter has to be satisfied in any case while the first can be relaxed if necessary. However, it would be quite unnatural to assign some real number to the constraints in order to represent their importance or strength w.r.t. relaxation. Where should the numbers come from? What should be the difference between importance 5.67

and 5.68? Instead, it seems more natural to express their strength in qualitative terms, e.g. by using attributes like *hard*, *strong*, *weak* or *soft* for the constraints.

Moreover, when using real numbers as weights for the constraints the strictly declarative nature of the CSP formulation is lost because it would then be very easy to include *control* information in the problem representation which from the declarative point of view should only contain the descriptive, *logic*al part of the problem statement.

Therefore, as we are essentially interested in preserving the declarative aspect of the CSP approach, we will focus on the qualitative approach which means assigning values from a small discrete set of weights to express the importance of a constraint. This is the approach which is also supported by the CONTAX system and will be presented in the following.

4.3 Constraint Relaxation in CONTAX

For the above mentioned reasons, in CONTAX we use a small set of discrete qualitative weights to express the importance of constraints or priorities between them. The weights are represented symbolically as `hard`, `strong`, `medium`, `weak`, and `soft` where there is a linear order '<' defined over them such that

$$\texttt{soft} < \texttt{weak} < \texttt{medium} < \texttt{strong} < \texttt{hard}.$$

Thus, the CONTAX user only has access to these qualitative expressions. The key point is not that we allow for only five weights[1] but the fact that we require weights to be expressed qualitatively. However, using a set of ten weights as in [Descotte and Latombe, 1985] seems not to be desirable because then again the question arises what the semantic difference is between two weights, say 7 and 8. Offering too many discrete weights will result in the same problems concerning declarativity as we have discussed in the context of real-weighted constraints.

However, it is a completely different issue how these quantitative weights are *processed*. Indeed, we can simply use the branch-and-bound algorithm of Figure 4.3 to find the optimal solution to a PCSP that uses qualitative weights in its problem representation; the only thing we will need is a mapping φ from the set of qualitative weights W onto a numerical domain over which the branch-and-bound algorithm will operate.

4.3.1 A formal definition of partial constraint satisfaction

Thus, we are now in a position to give a more formal definition of the *partial constraint satisfaction problem* as follows.

Definition 26 (partial constraint satisfaction problem) *A* partial constraint satisfaction problem *(PCSP) is a tuple* (V, δ, C, φ) *such that* (V, δ, C) *is a finite constraint*

[1]In the current CONTAX implementation the set of weights can be defined by the user and can therefore easily be extended.

satisfaction problem, C is a set of weighted constraints using some set of weights W, and $\varphi : W \mapsto \mathbb{R}$ is a mapping that assigns a positive real number to each weight in W. The PCSP is a discrete partial constraint satisfaction problem *if W is a discrete finite set of weights.*

We can now also define a solution to a PCSP as the optimal solution where optimality is based on the weights $w \in W$ and their numerical correspondences given by $\varphi(w)$. First, we define the *badness of a compound label* as follows:

Definition 27 (badness of a compound label) *Let $\mathcal{P} = (V, \delta, C, \varphi)$ be a PCSP on the variables $V = \{X_1, ..., X_n\}$. The badness of a compound label L is defined as the sum of the weights of all the constraints that are violated by L and is denoted by* $\mathsf{badness}(L)$*:*

$$\mathsf{badness}(L) = \sum \{ \varphi(c) \mid c \in C \wedge \mathsf{vars}(c) \subseteq \mathsf{vars}(L) \wedge \neg \mathsf{satisfies}(L, c)\}$$

Thus, the solution to a PCSP is required to be optimal in the sense of minimal badness:

Definition 28 (solution to a PCSP) *Let $\mathcal{P} = (V, \delta, C, \varphi)$ be a PCSP on the variables $V = \{X_1, X_2, \ldots, X_n\}$. A* solution to the PCSP *$\mathcal{P}$ is an n-compound label L assigning values to all variables in V such that it has minimal badness, i.e. there does not exist any n-compound label $\hat{L} \neq L$ with* $\mathsf{vars}(\hat{L}) = V$ *and* $\mathsf{badness}(\hat{L}) < \mathsf{badness}(L)$*.*

In Section 4.2 we have presented an algorithm that uses branch and bound for finding the maximal solution to a PCSP (Figure 4.3). The branch-and-bound approach to solving PCSPs has been introduced as a natural analogue to backtracking for solving CSPs. As a consequence, the FindMaximalSolution function suffers from similar drawbacks as the backtracking algorithm introduced in Section 2.3.2. The main reason is that both approaches are limited to retrospective constraint checking, i.e. the current value assignment is checked only for compatibility with past variables.

As a remedy for the backtracking approach, in Chapter 3 we have studied arc-consistency and hyperarc-consistency techniques which perform a prospective constraint checking in the sense that checking the current value assignment for compatibility with previous ones can also affect the domains of future variables and thus reduces the search space ahead. Moreover, by applying these consistency enforcing techniques once before starting backtrack search, often results in a very efficient search space pruning and sometimes even completely eliminates the need for backtracking search afterwards. The algorithm GenerateSolutions shown in Figure 3.15 (page 61) which is used in Contax for solving FCSPs efficiently integrates the prospective hyperarc-consistency checking within the generic backtracking scheme to ensure that finally all solutions to the FCSP will be found.

Unfortunately, for solving PCSPs we cannot make use of these consistency-enforcing techniques, as CSP and PCSP solving are based on different definitions of local *failure* during CSP and PCSP search. A CSP search path fails as soon as a single inconsistency is encountered. A PCSP search path will not fail until enough inconsistencies accumulate to reach a cutoff bound. Therefore, even if arc-consistency or hyperarc-consistency checking will recognize that some value v for some future variable X has no support in some constraint c and thus cannot take part in any solution to a CSP, from the viewpoint of partial constraint satisfaction we are not allowed to remove this value v from the domain of X because it may still be part of a solution to the PCSP which then, of course, violates the constraint c. Thus, it seems as we cannot take advantage of the pruning power of hyperarc-consistency when solving a PCSP. However, this is not completely true as we will see in the following section.

4.3.2 Enforcing Hyperarc-Consistency in PCSP Solving

The negative conjecture drawn from the above observations is only true as long as all constraints can be subject to constraint violations by a PCSP solution. However, if there is a subset $C_\infty \subseteq C$ of the constraints that cannot be violated by any solution to the PCSP, then we can use e.g. a variant of the ENFORCEHYPERARCCONSISTENCY algorithm presented in Figure 3.14 (page 58) to enforce hyperarc-consistency among the constraints in C_∞.

The constraint class C_∞ can be identified as those problem constraints that have to be satisfied in any case, therefore these constraints are regarded as *hard constraints*. In CONTAX this class is identical to the set of constraints that are weighted `hard`. From the operational point of view, the weight of all hard constraints is defined as ∞ such that no hard constraint can be violated by any solution to the PCSP.

Figure 4.6 shows the ENFORCEHYPERARCCONSISTENCYONHARDCONSTRAINTS function that establishes hyperarc-consistency among the hard constraints. It takes an argument $\mathcal{Q}$ which initializes the queue of constraint-variable pairs that have to be checked.

```
function EnforceHyperArcConsistencyOnHardConstraints($\mathcal{Q}$):
$Queue \leftarrow \mathcal{Q}$
$Consistent \leftarrow$ true
while $Queue$ not empty and $Consistent$ do
    select and delete some pair $(c, X)$ from $Queue$
    if DirectedHyperRevise$(c, X)$ then
        $Queue \leftarrow Queue \cup \{(\hat{c}, X_k) \mid \hat{c} \in C, \mathsf{weight}(\hat{c}) = \infty, \{X, X_k\} \subseteq \mathsf{vars}(\hat{c}), X \neq X_k\}$
        if $D_X = \emptyset$ then $Consistent \leftarrow$ false endif
    endif
endwhile
return $Consistent$
```

Figure 4.6: Function ENFORCEHYPERARCCONSISTENCYONHARDCONSTRAINTS

The function ENFORCEHYPERARCCONSISTENCYONHARDCONSTRAINTS can be integrated in a variant of the FINDMAXIMALSOLUTION function shown in Figure 4.3 (page 73) that performs the branch-and-bound search for the solution to a PCSP. The resulting function FINDOPTIMALSOLUTION is shown in Figure 4.7.

Besides adding the weight $\mathsf{weight}(\hat{c})$ of a violated constraint $\hat{c}$ to the new distance ε instead of simply incrementing ε with each constraint violation, it mainly differs from

```
function FindOptimalSolution($\mathcal{P}$, $\varepsilon$ , $X$, $\mathcal{V}$, $\Delta_X$):
if $\mathcal{V}=\emptyset$ then /* $\mathcal{P}$ holds a compound label assigning values to all variables */
    $\mathcal{P}_{\min} \leftarrow \mathcal{P}$
    $\varepsilon_{\min} \leftarrow \varepsilon$
    if $\varepsilon_{\min} \leq \varepsilon_0$ then return true else return false endif
 else if $\Delta_X = \emptyset$ then /* all values in $D_X$ have been tried for extending $\mathcal{P}$ */
    return false
    endif
 else /* try to extend $\mathcal{P}$ using another value $v \in \Delta_X$ */
    select a value $v \in \Delta_X$
    $D_X \leftarrow \{v\}$
    $\hat{\varepsilon} \leftarrow \varepsilon$ /* check whether we can do better when committing to $\langle X \leftarrow v\rangle$ */
    $C_X \leftarrow \{c \in C \mid X \in \mathsf{vars}(c) \wedge \mathsf{vars}(c) \subseteq \mathsf{vars}(\mathcal{P} \cup \{\langle X \leftarrow v\rangle\})\}$
    while $C_X \neq \emptyset$ and $\hat{\varepsilon} < \varepsilon_{\min}$ do
        select and delete a constraint $\hat{c}$ from $C_X$
        if $\neg\mathsf{satisfies}(\mathcal{P} \cup \{\langle X \leftarrow v\rangle\}, \hat{c})$ then $\hat{\varepsilon} \leftarrow \hat{\varepsilon} + \varphi(\mathsf{weight}(\hat{c}))$ endif
    endwhile
    select a variable $\hat{X} \in \mathcal{V} - \{X\}$ unless $\mathcal{V} - \{X\} = \emptyset$
    $\hat{\mathcal{Q}} \leftarrow \{(c, X_k) \mid c \in C, \mathsf{weight}(c) = \infty, \{X, X_k\} \subseteq \mathsf{vars}(c), X \neq X_k\}$
    push $\{D_{X_i} \mid X_i \in V\}$ on stack $S$
    if $\hat{\varepsilon} < \varepsilon_{\min}$
    and EnforceHyperArcConsistencyOnHardConstraints($\hat{\mathcal{Q}}$)
    and FindOptimalSolution($\mathcal{P} \cup \{\langle X \leftarrow v\rangle\}$, $\hat{\varepsilon}$ , $\hat{X}$, $\mathcal{V} - \{X\}$, $D_{\hat{X}}$)
    then
        return true
    else
        restore $\{D_{X_i} \mid X_i \in V\}$ from stack $S$ and pop $S$
        return FindOptimalSolution($\mathcal{P}$, $\varepsilon$ , $X$, $\mathcal{V}$, $\Delta_X - \{v\}$)
    endif
endif
```

Figure 4.7: Function FindOptimalSolution

the FindMaximalSolution function in integrating the hyperarc-consistency checking as another condition for committing to a new value assignment $\langle X \leftarrow v\rangle$. The argument to EnforceHyperArcConsistencyOnHardConstraints is the set $\hat{\mathcal{Q}}$ of all hard-constraint-variable pairs that have to be checked when extending the search path $\mathcal{P}$ by $\langle X \leftarrow v\rangle$.

Establishing hyperarc-consistency on the hard constraints may now also result in some values being removed from some variables domain. On the one hand side this is exactly the prospective checking that we wanted to include in the branch-and-bound framework. But as the embedded call to FindOptimalSolution may fail to extend the current search path including $\langle X \leftarrow v\rangle$ to an optimal solution, the effect of enforcing hyperarc-consistency due to committing to the assignment $\langle X \leftarrow v\rangle$ has to be undone on the way back. Thus, we have to include a stack S that holds sets of all variable domains for each recursive call to FINDOPTIMALSOLUTION on an extended search path.

In Section 3.2.3, we have presented the final algorithm GENERATESOLUTIONS (Figure 3.15 on page 61) for finding the solutions to an FCSP by using hyperarc-consistency for preprocessing and pruning the search space before starting an interleaved combination of backtracking and hyperarc-consistency constraint propagation to ultimately find or generate all solutions. The direct analogue can also be built for solving a PCSP as shown in Figure 4.8. Of course, in the case of PCSPs the initial hyperarc-consistency step for preprocessing is restricted to checking hard constraints only. But nevertheless, as we can assume a substantial body of constraints in all real-life applications to be fixed and hence not relaxable, i.e. hard constraints, a considerable pruning of the search space can be expected by preprocessing the hard constraints and enforcing hyperarc-consistency among them.

```
algorithm GENERATEBESTSOLUTION:
INIT:    $\varepsilon_0 \leftarrow 0$
         $\varepsilon_{\min} \leftarrow \infty$
         $\mathcal{P}_{\min} \leftarrow \emptyset$
PRUNE:   $\mathcal{Q} \leftarrow \{(c, X) \mid c \in C, \mathsf{weight}(c) = \infty, X \in \mathsf{vars}(c)\}$
         if ENFORCEHYPERARCCONSISTENCYONHARDCONSTRAINTS($\mathcal{Q}$) then
             goto SEARCH
         else /* hard constraints violated */
             return false
         endif
SEARCH:  select a variable $X \in V$
         if FINDOPTIMALSOLUTION($\emptyset$, $\infty$, $X$, $V$, $D_X$) then
             return $\mathcal{P}_{\min}$ /* all constraints satisfied */
         else if $\varepsilon_{\min} = \infty$ then
             return false /* hard constraints cannot be satisfied */
         else
             return $\mathcal{P}_{\min}$ /* solution to minimally relaxed problem */
         endif
```

Figure 4.8: Algorithm GENERATEBESTSOLUTION

Thus, what GENERATESOLUTIONS is for FCSPs, GENERATEBESTSOLUTION is for PCSPs. Also note with the names for the algorithms that GENERATESOLUTIONS reports all (perfect) solutions to an FCSP while GENERATEBESTSOLUTION will return only one solution: the best (imperfect) one. In the following section, we will show that GENERATEBESTSOLUTION will always terminate and find the best solution to the PCSP.

4.3.3 Termination of the GENERATEBESTSOLUTION algorithm

The GENERATEBESTSOLUTION algorithm basically builds on calling the two functions ENFORCEHYPERARCCONSISTENCYONHARDCONSTRAINTS and FINDOPTIMALSOLU-

TION. Thus, we basically have to prove the termination of these functions to obtain a termination result for the GENERATEBESTSOLUTION algorithm.

Lemma 3 (termination ENFORCEHYPERARCCONSISTENCYONHARDCONSTRAINTS**)** *The function* ENFORCEHYPERARCCONSISTENCYONHARDCONSTRAINTS *always terminates.*

Proof. As the function ENFORCEHYPERARCCONSISTENCYONHARDCONSTRAINTS is a variant of the ENFORCEHYPERARCCONSISTENCY algorithm in Figure 3.14 restricted to checking hard constraints, it is clear that it will always terminate due to proposition 4 on page 59. □

Proposition 6 (termination of FINDOPTIMALSOLUTION**)** *The function* FINDOPTIMALSOLUTION *always terminates.*

Proof. The while loop for estimating the new distance $\hat{\varepsilon}$ when committing to the assignment $\langle X \leftarrow v\rangle$ trivially terminates because with each loop a constraint $\hat{c}$ is deleted from the set C_X and $C_X = \emptyset$ will terminate the while loop. Also, the call to ENFORCEHYPERARCCONSISTENCYONHARDCONSTRAINTS will cause no termination problems due to lemma 3. Thus, it remains to prove that the recursive calls to FINDOPTIMALSOLUTION will always terminate. Therefore, we define the *size* of the parameters in a call to the function FINDOPTIMALSOLUTION($\mathcal{P}$, ε , X, $\mathcal{V}$, Δ_X) as

$$\psi(\mathcal{P}, \varepsilon\ , X, \mathcal{V}, \Delta_X) = (|\mathcal{V}| \cdot d) + |\Delta_X|$$

where d denotes the maximum cardinality of the domains D_{X_i} for all variables $X_i \in V$. Obviously, $\psi(\mathcal{P}, \varepsilon\ , X, \mathcal{V}, \Delta_X) \geq 0$ for all calls to FINDOPTIMALSOLUTION. We can now show that the size of the parameters passed to the function FINDOPTIMALSOLUTION decreases strong monotonically with each recursive call. Let ν be the cardinality of $\mathcal{V}$ and μ be the cardinality of Δ_X in the call to FINDOPTIMALSOLUTION. If $\nu = 0$ or $\mu = 0$ then FINDOPTIMALSOLUTION will terminate immediately; so we can assume $\nu > 0$ and $\mu > 0$ when studying the recursive calls to FINDOPTIMALSOLUTION. As there are two of them in the definition of FINDOPTIMALSOLUTION we have to distinguish two cases.

Case 1: Consider the call FINDOPTIMALSOLUTION($\mathcal{P}\cup\{\langle X \leftarrow v\rangle\}$,$\hat{\varepsilon}$,$\hat{X}$,$\mathcal{V}-\{X\}$,$D_{\hat{X}}$). The size $\psi(\mathcal{P}\cup\{\langle X \leftarrow v\rangle\}, \hat{\varepsilon}\ ,\hat{X},\mathcal{V}-\{X\},D_{\hat{X}})$ of the parameters in the call is given as $(|\mathcal{V}-\{X\}| \cdot d) + |D_{\hat{X}}| = (\nu - 1) \cdot d + |D_{\hat{X}}| \leq \nu \cdot d < \nu \cdot d + \mu$ and hence the size of the call is decreasing.

Case 2: Consider the second call FINDOPTIMALSOLUTION($\mathcal{P}$,ε ,X,$\mathcal{V}$,$\Delta_X-\{v\}$). The size of its parameters $\psi(\mathcal{P}, \varepsilon\ ,X,\mathcal{V},\Delta_X-\{v\})$ is $\nu \cdot d + |\Delta_X-\{v\}| = \nu \cdot d + (\mu - 1) < \nu \cdot d + \mu$ and hence again the size of the call is decreasing.

Therefore, only a finite number of recursive calls to FINDOPTIMALSOLUTION can take place and thus the function FINDOPTIMALSOLUTION will always terminate. □

From lemma 3 and proposition 6 we can then derive the termination of the GENERATEBESTSOLUTION algorithm.

It can also be easily proved that the GENERATEBESTSOLUTION algorithm always finds the optimal solution w.r.t. the weights assigned to the constraints. The key observation is that weights are mapped to positive numbers and that the function FINDOPTIMALSOLUTION closes a search path and prunes the corresponding search tree if and only if it cannot lead to a solution with a smaller weight than that of the best solution so far. However, this result requires the *necessary bound* $\varepsilon_{\min}$ to be set to ∞ and the *sufficient bound* ε_0 to be set to 0 when starting search. Otherwise, optimality cannot be guaranteed in general. But, as the GENERATEBESTSOLUTION algorithm initializes $\varepsilon_{\min}$ and ε_0 with ∞ and 0 respectively, FINDOPTIMALSOLUTION finds the optimal solution to the PCSP.

4.3.4 Weighted constraints and weighted value combinations

All discussions so far assumed that weights are assigned to entire constraints in a PCSP. However, in some applications it is useful to be able to express preferences on the level of compound labels that constitute the definition of a constraint. Therefore, CONTAX supports weighted constraints as well as weighted value combinations, i.e. assigning weights to individual compound labels within a constraint definition. As these weights express preferences on the level of value combinations, they are referred to as combination weights.

Definition 29 (weighted value combination) *Let L be a compound label and ω be a weight from a given set Ω of combination weights. Then the pair (L, ω) is a weighted value combination and ω is the weight of (L, ω), denoted by* $\mathsf{weight}((L, \omega)) = \omega$.

Note that the set W of weights used for entire constraints is different from the set Ω of combination weights used for value combinations. A small weight (from the set W) assigned to a constraint C expresses a low importance of satisfying the constraint C. In contrast to that, a small combination weight (from the set Ω) assigned to a compound label $L \in C$ means to prefer this particular value combination to others with a higher weight when trying to satisfy the constraint C. Thus, the notion of weighted value combinations allows to represent sets of additional compound labels in a constraint definition that can be added to the set of legal value combinations and taken into account if necessary. It hence offers an elegant way to specify possible relaxations of a constraint together with its definition. Such a constraint containing weighted value combinations is called a *fine-weighted constraint* and is defined as follows:

Definition 30 (fine-weighted constraint) *A* fine-weighted constraint *C over a set of variables $V = \{X_1, \dots, X_n\}$ is a set of weighted value combinations (L, ω) where L is an n-compound label with* $\mathsf{vars}(L) = V$.

As the GENERATEBESTSOLUTION algorithm deals with weighted constraints only, fine-weighted constraints have to be transformed into weighted constraints. The basic idea of this transformation is to create weighted constraints of increasing importance (i.e., weights) by successively including compound labels of increasing preference (i.e., combination weights). Thus, first a weighted constraint with minimal weight (`soft`) will be created containing only the compound labels of those weighted value combinations in the fine-weighted constraint that have minimal combination weights. Then, with each step more and more compound labels are selected from the fine-weighted constraint making up weighted constraints of increasing weights until finally a constraint with maximum weight (`hard`) is created containing all value combinations occurring in the fine-weighted constraint. Assume a discrete finite set of weights $W = \{w_1, \ldots, w_m\}$ with $\varphi(w_i) < \varphi(w_j)$ for all $i < j$ and a discrete finite set of combination weights[2] $\Omega = \{\omega_1, \ldots, \omega_m\}$ such that $\varphi(\omega_i) = \varphi(w_i)$ for all $1 < i < m$. Then each fine-weighted constraint C can be transformed into m weighted constraints C_{w_i} as follows.

The constraint C_{w_i} contains all compound labels L that occur in C and have a combination weight less or equal to w_i:

$$C_{w_i} = \{L \mid (L, \omega_j) \in C \wedge \varphi(\omega_j) \leq \varphi(w_i)\}$$

Each of the m weighted constraints C_{w_i} is assigned the weight w_i.

Contax uses the set of constraint weights $W = \{$`soft`, `weak`, `medium`, `strong`, `hard`$\}$ with $\varphi($`soft`$) < \varphi($`weak`$) < \varphi($`medium`$) < \varphi($`strong`$) < \varphi($`hard`$)$. For assigning weights or preferences to value combinations, Contax provides the following set of combination weights $\Omega = \{$`prefer`, `relax1`, `relax2`, `relax3`, `relax4`$\}$. Thus, transforming a fine-weighted constraint C to a set of weighted constraints results in all `prefer`red value combinations in C being allowed by a new `hard` constraint. All value combinations in C weighted `prefer` or `relax1` are covered by a `strong` constraint, all combinations weighted `prefer`, `relax1`, or `relax2` form a `medium` weighted constraint, and so on. Finally, all value combinations in C, no matter what their weights are, constitute the body of a new `soft` constraint.

Having transformed all fine-weighted constraints into weighted ones then allows using the GenerateBestSolution algorithm to find an optimal solution to the PCSP.

However, the real weights that Contax actually assigns to the generated constraints differ from the real weights that it assigns for other weighted constraints. The reason is that the generated constraints C_{w_i} are overlapping, i.e. $C_{w_1} \subseteq C_{w_2} \subseteq \cdots \subseteq C_{w_m}$. Hence, violating the generated constraint C_{w_i} also violates the corresponding constraints C_{w_j} with $j < i$. For the branch-and-bound search process in the FindOptimalSolution this puts a disadvantage on those constraints that have been generated as the result of incorporating fine-weighted constraints. Therefore, the real weight assigned to these constraints C_{w_i} is computed as

$$\varphi_{w_i} = \varphi(w_i) - \sum_{j<i} \varphi(w_j)$$

such that violating a constraint C_{w_i} after having violated a constraint C_{w_j}, $j < i$, earlier in the search path will have the same result as if C_{w_i} were an ordinary constraint which is now violated.

[2]We assume equal cardinality of W and Ω as it helps defining the transformation of fine-weighted constraints into weighted constraints. However, the transformation can also be defined dealing with different numbers of weights.

4.3.5 A user's view on weighted constraints in CONTAX

Besides the functionality discussed in Section 3.2.4, the current implementation of the CONTAX system also contains a solver for PCSPs that is based on the GENERATEBEST-SOLUTION algorithm presented in Figure 4.8 on page 81. As mentioned earlier, CONTAX restricts the weights to be symbols that are mapped to numerical values internally. Although from the viewpoint of declarativity a small number of weights should be preferred, it is open to the user to include new (symbolic) weights as necessary. In the standard version, CONTAX offers a set of five weights ranging from `hard` to `soft`. These weights have been sufficient even for the applications that will be studied in Chapter 8.

The assignment of weights to constraints could in principle take place at two different steps when representing a PCSP in CONTAX. In principle, whenever defining a new constraint, e.g. by `def-primitive-constraint`, a weight could be assigned to the constraint. However, as different instances of one constraint may be of different importance for the solution to the problem, in CONTAX weights are assigned not to constraint definitions but to constraint instances. Thus the only (visible) extension is that `make-constraint` accepts another keyword parameter holding the weight of the constraint instance. The `different-colors` constraint between the variables HH and SH can thus be declared as a `hard` constraint by the following:

```
(make-constraint :name C1
                 :type different-colors
                 :weight hard
                 :color1 HH
                 :color2 SH)
```

For primitive constraints, CONTAX also allows to assign weights to value combinations. Thus, the constraint type `different-colors` (cf. Figure 3.17 on page 65) can now also be defined in a way that it expresses a strong preference for different colors but also allows for equal colors as an ultimate relaxation, hence the name `preferably-different-colors`:

```
(def-weighted-constraint preferably-different-colors
    :interface (color1 color2)
    :domains (colors colors)
    :prefer ((red green) (red blue)
             (green red) (green blue)
             (blue red) (blue green))
    :relax4 ((green green)
             (blue blue)
             (red red)))
```

When creating an instance of a fine-weighted constraint, CONTAX automatically performs the above mentioned transformation and installs a set of constraint instances, one for each constraint weight. For the above example two constraints are generated: one with weight (nearly[3]) `hard` containing all 9 value combinations and another `soft`

[3]The exact weight results as φ(`hard`)$-\varphi$(`strong`)$-\varphi$(`medium`)$-\varphi$(`weak`)$-\varphi$(`soft`).

constraint containing only the first 6 value combinations. Thus, creating an instance `C1` of the fine-weighted constraint `preferably-different-colors` linked to the variables HH and SH would be (nearly) equivalent to defining two intermediate constraints `preferably-different-colors-1`

```
(def-primitive-constraint preferably-different-colors-1
    :interface (color1 color2)
    :domains (colors colors)
    :tuples ((red green) (red blue)
             (green red) (green blue)
             (blue red) (blue green)))
```

and `preferably-different-colors-2`

```
(def-primitive-constraint preferably-different-colors-2
    :interface (color1 color2)
    :domains (colors colors)
    :tuples ((red green) (red blue)
             (green red) (green blue)
             (blue red) (blue green)
             (green green)
             (blue blue)
             (red red)))
```

before creating one instance of each of these constraints:

```
(make-constraint :name C1-1
                 :type preferably-different-colors-1
                 :weight soft
                 :color1 HH
                 :color2 SH)

(make-constraint :name C1-2
                 :type preferably-different-colors-2
                 :weight hard
                 :color1 HH
                 :color2 SH)
```

Solving the extended German-map coloring problem with CONTAX

Using weighted constraints, we are now also able to represent and solve the extended map-coloring problem. The CONTAX representation for this PCSP is very similar to that of the German-map coloring problem in Figure 3.17 on page 65. The main difference besides the increased number of variables is the assignment of weights to the various instances of the `different-colors` constraints. In our representation, the weight assigned to a constraint instance qualitatively correlates to the length of the corresponding border. Thus, the constraint between the variables MV and SA is the only one weighted `soft` as the corresponding border between Mecklenburg-Vorpommern and Sachsen-Anhalt is by far the shortest in the map. Figure 4.9 shows the CONTAX representation for the extended map-coloring problem.

```
(def-domain colors (red green blue))

(def-primitive-constraint different-colors
   :interface (c1 c2)
   :domains (colors colors)
   :tuples ((red green) (red blue) (green red)
            (green blue) (blue red) (blue green)))

(make-variable :name SH :domain colors)
   ... 14 more variables ...
(make-variable :name SN :domain colors)

(make-constraint :name different-colors :c1 SH :c2 HH :weight medium)
(make-constraint :name different-colors :c1 SH :c2 NS :weight medium)
(make-constraint :name different-colors :c1 HH :c2 NS :weight medium)
(make-constraint :name different-colors :c1 NS :c2 HB :weight medium)
(make-constraint :name different-colors :c1 NS :c2 NW :weight hard)
(make-constraint :name different-colors :c1 NS :c2 HS :weight medium)
(make-constraint :name different-colors :c1 NW :c2 HS :weight strong)
(make-constraint :name different-colors :c1 NW :c2 RP :weight strong)
(make-constraint :name different-colors :c1 HS :c2 RP :weight strong)
(make-constraint :name different-colors :c1 HS :c2 BW :weight medium)
(make-constraint :name different-colors :c1 HS :c2 BY :weight strong)
(make-constraint :name different-colors :c1 RP :c2 SL :weight medium)
(make-constraint :name different-colors :c1 RP :c2 BW :weight medium)
(make-constraint :name different-colors :c1 BW :c2 BY :weight hard)
(make-constraint :name different-colors :c1 SH :c2 MV :weight medium)
(make-constraint :name different-colors :c1 NS :c2 MV :weight medium)
(make-constraint :name different-colors :c1 NS :c2 SA :weight strong)
(make-constraint :name different-colors :c1 MV :c2 SA :weight soft)
(make-constraint :name different-colors :c1 NS :c2 TH :weight medium)
(make-constraint :name different-colors :c1 HS :c2 TH :weight medium)
(make-constraint :name different-colors :c1 BY :c2 TH :weight strong)
(make-constraint :name different-colors :c1 MV :c2 BB :weight strong)
(make-constraint :name different-colors :c1 SA :c2 BB :weight strong)
(make-constraint :name different-colors :c1 BB :c2 BL :weight medium)
(make-constraint :name different-colors :c1 SA :c2 SN :weight medium)
(make-constraint :name different-colors :c1 BB :c2 SN :weight strong)
(make-constraint :name different-colors :c1 SA :c2 TH :weight strong)
(make-constraint :name different-colors :c1 TH :c2 SN :weight medium)
(make-constraint :name different-colors :c1 BY :c2 SN :weight weak)
```

Figure 4.9: Representing the extended German-map coloring problem in CONTAX

If we now run the GenerateBestSolution algorithm on the extended German-map coloring problem, we obtain the optimal solution which corresponds to the map-coloring shown in Figure 4.5 on page 76.

```
(bestsolve :all)

⟹ ((SH HH NS HB NW HS RP SL BW BY MV BB BL SA TH SN)
    (red green blue green red green blue red
     red blue green red green green red blue))
```

Figure 4.10: Solving the extended German-map coloring problem with Contax

Note that the Contax system offers different interfaces to the GenerateBestSolution algorithm for solving a PCSP (function `bestsolve`) and for the GenerateSolutions algorithm intended for solving an FCSP (function `solve`). The reason is that even if the given CSP contains weighted constraints, it needs not necessarily be solved using the GenerateBestSolution algorithm but can also be dealt with using the 'standard' GenerateSolutions algorithm. In this case the weights are used to incrementally strengthen the CSP as long as it remains solvable. This incremental strengthening approach will be discussed in the following section.

4.4 Incremental strengthening of a discrete PCSP

The GenerateBestSolution algorithm can be seen as performing a constraint relaxation approach where the best solution is defined as requiring the smallest amount of relaxation of the given constraints. In this section, we will study another approach which works the opposite way: instead of relaxing a constraint, i.e. violating the constraint in a partial solution, the problem gets incrementally strengthened until no more (sets of) constraints can be satisfied. The main advantage of this approach is that we can make use of the pruning power and efficiency of CSP solving techniques as realized in the GenerateSolutions algorithm (Figure 3.15 on page 61). Of course, in contrast to the GenerateBestSolution algorithm that is based on branch and bound, using incremental strengthening we cannot guarantee finding the optimal solution to a PCSP.

The basic idea of incremental strengthening is to separate the constraints into pairwise disjoint sets according to their weights. As constraints that are not explicitly weighted are regarded as `hard` constraints, we obtain as many constraint sets as there are weights supported by the system. So, in the standard version of Contax, we obtain five constraint sets, $\mathcal{C}_{\texttt{hard}}$, $\mathcal{C}_{\texttt{strong}}$, $\mathcal{C}_{\texttt{medium}}$, $\mathcal{C}_{\texttt{weak}}$, and $\mathcal{C}_{\texttt{soft}}$ where

$$\mathcal{C}_w = \{c \in C \mid \mathsf{weight}(c) = w\}$$

Note that the weights of constraints which are generated by the transformation of fine-weighted constraints as described in Section 4.3.4 need not be modified as for use with the

GenerateBestSolutions algorithm. Moreover, no mapping φ from symbolic weights to numbers is required at all.

Having built such constraint sets, the principle of the IncrementalStrengthening algorithm is very simple and will only be outlined here: The system starts with the core CSP that contains as its set of constraints the set $\mathcal{C}_{\texttt{hard}}$. If the GenerateSolutions algorithms succeeds, the set of solutions is stored in a variable, say $\mathcal{S}$; if it fails, then the given CSP is unsolvable and the algorithm stops. Otherwise it incrementally adds the constraints from the sets $\mathcal{C}_{\texttt{strong}}$, $\mathcal{C}_{\texttt{medium}}$, $\mathcal{C}_{\texttt{weak}}$, and finally $\mathcal{C}_{\texttt{soft}}$ to the CSP which thereby gets incrementally strengthened. After each such strengthening step, the algorithm GenerateSolutions is called again. If it succeeds, $\mathcal{S}$ is substituted by the new set of solutions and strengthening can proceed as long as there are constraint sets (corresponding to decreasing weights) to be added; otherwise the maximal solvable strengthening of the CSP has been reached in the last step and the value of $\mathcal{S}$ will be returned as the 'solution' to the CSP.

As constraint sets are added to the CSP as a whole, this approach runs a high risk of returning solutions that are far from being optimal: If at some level all but one from a large number of constraints with the same weight can be satisfied, the 'solution' that will be returned by the algorithm (if any) will not necessarily satisfy most of these constraints. Even if all solutions are computed and stored at each level, a lot of 'solutions' will be returned that may violate many of the constraints mentioned.

Therefore, the IncrementalStrengthening algorithm offers only a very rough relaxation behavior. However, it can be implemented very efficiently and proved to be sufficient for a class of applications where only the optimal level of satisfiable constraint sets is required in contrast to an optimal solution that can be provided by the GenerateBestSolution algorithm—at higher costs, of course.

4.5 Discussion and Related Work

The utility of some form of partial constraint satisfaction has been repeatedly recognized. A variety of applications has motivated a variety of approaches. Conflicting constraints have arisen in a variety of domains. Descotte and Latombe made "compromises" among antagonist constraints in a planner for machining problems [Descotte and Latombe, 1985]. Borning used constraint "hierarchies" to deal with situations in which a set of requirements and preferences for the graphical display of a physical simulation cannot all be satisfied [Borning *et al.*, 1987]; these hierarchies have been embedded in a constraint logic programming language [Borning *et al.*, 1989].

Scheduling problems are a natural source of constraint satisfaction problems, and schedule conflicts a natural source of PCSPs. Fox added the concepts of constraint relaxation as the selection of constraint alternatives, "preferences" among relaxations and constraint "importance" to constraint representations to cope with conflicting constraints in job-shop scheduling [Fox, 1987]. Feldman and Golumbic used "priorities" in looking for optimal student schedules [Feldman and Golumbic, 1990].

Machine vision has also strong relations to work on partial constraint satisfaction.

Shapiro and Haralick treated inexact matching of structural descriptions using an extension of constraint satisfaction that they called the *inexact consistent labeling problem*, which sought a solution within a given error bound [Shapiro and Haralick, 1981]. Mohr and Masini suggested a modification of local consistency processing to deal with errors, permitting values to fail to satisfy some constraints, in order to cope with noise in domains such as computer vision [Mohr and Masini, 1988].

The concept of weighted value combinations is motivated by machine vision as well [Rosenfeld *et al.*, 1976]. The expression of preferences in database queries is also a related problem [Lacroix and Lavency, 1987]. Also note that the concept of optimization can play a role in constraint satisfaction even when all constraints are satisfied; there may be an additional criterion to optimize among alternative solutions [Dechter *et al.*, 1990]. Recently, Mantha and co-workers also started investigating the problem of constraint relaxation and optimization within the framework of Horn clause programs using preference logics [Brown *et al.*, 1993].

This shows that partial constraint satisfaction and constraint relaxation have already been recognized as an important issue in CSP research for a long time. However, most approaches deal with this issue from a pragmatic point of view and develop techniques to solve PCSPs specialized to certain application problems. In contrast to them, the research presented here provides a general framework for solving PCSPs that on the one hand side preserves the declarative representation of a CSP or PCSP respectively and on the other hand allows for an efficient problem solving by incorporating consistency enforcing techniques like hyperarc-consistency within the optimization process.

5

Constraint Satisfaction over Hierarchically Structured Domains

The basic constraint satisfaction approach as discussed in Chapter 3 as well as the extension towards constraint relaxation presented in Chapter 4 assume that the domains are supplied extensionally as unstructured sets, listing the finite number of members. However, for many real world problems the domain elements often cluster into sets with common properties and relations. Those sets, in turn, group to form higher level sets. This clustering or categorization into "natural kinds" [Havens and Mackworth, 1983] can be represented as a specialization/generalization hierarchy.

The main topic of this chapter is the exploitation of the structure provided when the domains can be naturally represented as specialization hierarchies. In Section 5.1, we will first introduce hierarchically structured domains and how they can be represented declaratively. Section 5.2 will then present an extended notion to hyperarc-consistency that will also respect the hierarchical structure of the domains. The resulting algorithm for enforcing hierarchical hyperarc-consistency that is presented in Section 5.3 allows to efficiently deal with hierarchically structured domains. Hence, the extension of CONTAX towards dealing with hierarchically structured domains that will be presented in Section 5.4 ideally contributes to both of our goals: increasing the declarative representation capabilities as well as the problem-solving efficiency.

5.1 Hierarchically Structured Domains

In order to exploit the structure of a domain, it can be interpreted as a domain graph with each node corresponding to a set of elements and each arc representing the subset relation between sets. Domain graphs are, of course, quite distinct from the constraint graphs introduced earlier.

In general, a domain graph is not a strict tree since a set may be a direct subset of more than one superset; the general characterization is a directed acyclic graph representing the lattice induced by the partial ordering of the subset relation. For the purposes of this chapter we shall assume that the domain hierarchies are singly rooted strict trees. Each subset has only one direct superset and the subsets of a set are mutually exclusive and exhaustive.

For the illustration of our approach, we will again use a variant of the German-map coloring problem; its definition is now extended towards hierarchically structured domains as follows:

Problem 3 (hierarchically extended German-map coloring problem) *The hierarchically extended German-map coloring problem also is to decide how to color the political map of Germany as shown in Figure 4.1 (page 70) such that neighbored states are colored differently. However, as the extended map-coloring problem previously defined in Chapter 3 using only three different colors (*red, blue, *and* green*) has been shown to be unsolvable, we now introduce bright and dark variants of the colors* blue *and* green*. For* red *we make no difference, that is, we regard* red *as a bright color. We still require neighbored states to be assigned different colors but for the states that share only a small border (Mecklenburg-Vorpommern and Sachsen-Anhalt, Bayern and Sachsen as well as Hessen and Baden-Württemberg) we are willing to allow the same color to be used but with different brightness, e.g. to use* dark blue *for Mecklenburg-Vorpommern and* bright blue *for Sachsen-Anhalt. Additionally, the smallest states Berlin, Hamburg, Bremen, and Saarland shall be assigned exactly the same color; Hessen should take some* green *color, and finally, no neighbored states are allowed to be both assigned a dark color.*

The domain of individual colors in the hierarchically extended map-coloring problem is the set $D = \{$red, bright green, dark green, bright blue, dark blue$\}$. This 'flat' color domain can now be structured as shown in Figure 5.1.

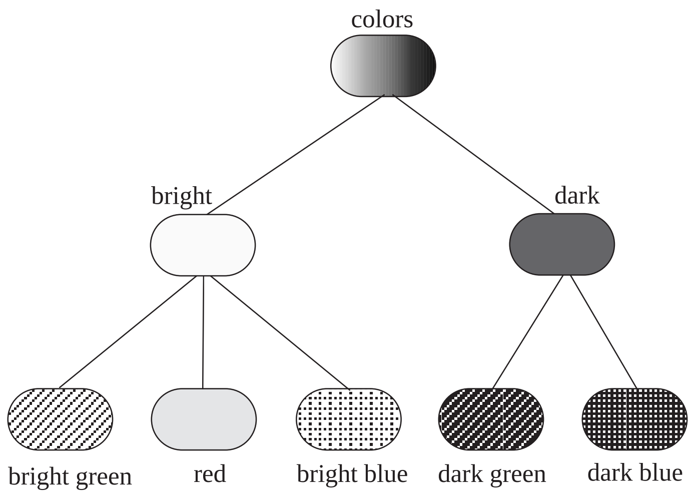


Figure 5.1: Hierarchical structure of the color domain

The aim of exploiting the structure of the domains is to make the DirectedHyperRevise procedure used by the EnforceHyperArcConsistency algorithm (Figures 3.13 and 3.14) considerably more efficient by reducing the number of predicate evaluations (constraint checks) it must perform. The currently active elements of a domain can be represented by a set of tree nodes that dominate those members. DirectedHyperRevise can then retain, eliminate, or further examine entire subsets of the domain with only one or two predicate evaluations.

To achieve this aim, two new families of predicates will be introduced. These new predicates apply not to combinations of individual domain elements for the related variables but to combinations of subsets of the domains of the variables. These new predicates are easily computed from the compound labels supplied with the CSP relating individual domain elements. The expectation is that the domains are structured so that the elements of a subset frequently share consistency properties that permit them to be retained or eliminated as a unit.

We shall use D_X to represent the original domain for variable X and Δ_X to represent the active subset of D_X at any point of the constraint propagation process. Δ_X may be implemented as a set of the active subdomains of D_X. The subdomains of D_X are $\{D_X^{i_1 i_2 \ldots i_k}\}$ which can be arranged as a tree as shown in Figure 5.2, where an arc indicates that the subdomain at the bottom of the arc is a direct subset of the subdomain at the top of the arc.[1] The notation for $D_X^{i_1 i_2 \ldots i_k}$ indicates that the subdomain is on the kth level of the domain tree for D_X and it is the i_kth subdomain of its superset $D_X^{i_1 i_2 \ldots i_{k-1}}$ at level $k-1$ which is the i_{k-1}th subdomain of its superset, and so on.

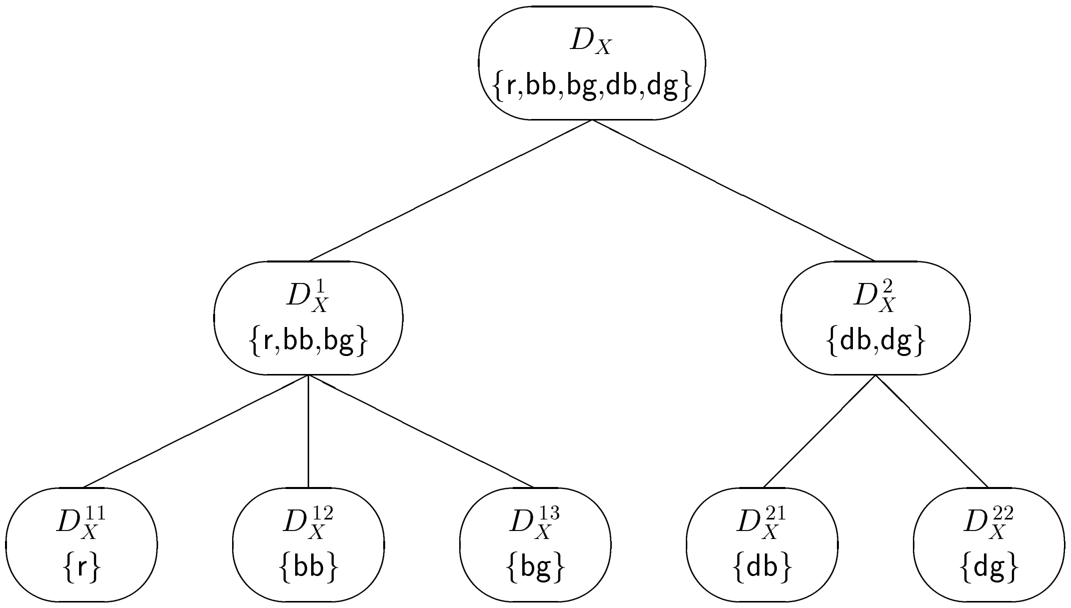


Figure 5.2: The domain tree for the color domain

The set $D_X^{i_1 i_2 \ldots i_k}$ is partitioned into a number of $b_X^{i_1 i_2 \ldots i_k}$ mutually exclusive subsets. In other words, the following two conditions hold on all subdomains $D_X^{i_1 i_2 \ldots i_k} \subseteq D_X$:

$$D_X^{i_1 i_2 \ldots i_k} = \bigcup_{1 \le j \le b_X^{i_1 i_2 \cdots i_k}} D_X^{i_1 i_2 \ldots i_k j}$$

$$D_X^{i_1 i_2 \ldots i_k j_1} \cap D_X^{i_1 i_2 \ldots i_k j_2} = \emptyset \quad \forall 1 \le j_1 < j_2 \le b_X^{i_1 i_2 \ldots i_k}$$

[1]We shall from now on use the following abbreviations for the colors: r, b, and g for red, blue, and green as before, and additionally bg for bright green bb for bright blue, dg for dark green, and db for dark blue.

The number $b_X^{i_1 i_2 \ldots i_k}$ denotes the *branching rate* at the node $D_X^{i_1 i_2 \ldots i_k}$ in the domain tree for variable X, i.e. it denotes into how many subdomains the domain $D_X^{i_1 i_2 \ldots i_k}$ is split. Obviously, if $D_X^{i_1 i_2 \ldots i_k}$ represents a leaf in the domain tree, i.e. $\mid D_X^{i_1 i_2 \ldots i_k} \mid = 1$, then $b_X^{i_1 i_2 \ldots i_k} = 0$.

Hierarchically structured domains can be represented very declaratively by allowing domain elements to also represent entire subdomains instead of individual values only. That means, we can simply define the color domain as the set holding the two values bright and dark. These subdomains can then be further refined, e.g. the domain bright can be defined as the set {bright green, red, bright blue}:

```
(def-domain colors (bright dark))
(def-domain bright (bg r bb))
```

Note, that from this viewpoint all subdomains represent sets of values; thus the leaves representing the individual members in the domain, are handled as singleton sets.

In the algorithm we shall develop, Δ_X, the set of still active elements of the original D_{X_i}, is the union of a number of mutually exclusive sets $D_X^{i_1 i_2 \ldots i_k}$. At all times, $\Delta_X \subseteq D_X$. Now suppose the active subset Δ_X to be the union of some subdomains in the domain tree:

$$\Delta_X = \bigcup_{\bar{q}} \Delta_X^{\bar{q}}$$

We call each $\Delta_X^{\bar{q}}$ an *abstract label* of Δ_X. Each abstract label $\Delta_X^{\bar{q}}$ is identical to a subset $D_X^{i_1 i_2 \ldots i_k} \subseteq D_X$ for some $\bar{q} = i_1 i_2 \ldots i_k$. Δ_X will be represented by the set $\{\Delta_X^{\bar{q}}\}$.

5.2 Hierarchical hyperarc-consistency

We will now show how to efficiently implement the domain restriction step of the DirectedHyperRevise(c,X_j) that has been presented in Figure 3.13 (page 58).

Informally, what we have to do is test each abstract label $\Delta_{X_j}^{\bar{q}}$ of Δ_{X_j} with respect to a constraint c on the variables vars(c) = $\{X_1, \ldots, X_j, \ldots, X_m\}$, using a generalized version of DirectedHyperRevise. If there are $\Delta_{X_1}^{\bar{q}_1}, \ldots, \Delta_{X_{j-1}}^{\bar{q}_{j-1}}, \Delta_{X_{j+1}}^{\bar{q}_{j+1}}, \ldots, \Delta_{X_m}^{\bar{q}_m}$ such that *every* domain element in $\Delta_{X_j}^{\bar{q}}$ is compatible with *some* elements in the other domains $\Delta_{X_i}^{\bar{q}_i}$, $1 \leq i \leq m$ and $i \neq j$, then $\Delta_{X_j}^{\bar{q}}$ survives unchanged in Δ_X. If not, then check to see if there are $\Delta_{X_1}^{\bar{q}_1}, \ldots, \Delta_{X_{j-1}}^{\bar{q}_{j-1}}, \Delta_{X_{j+1}}^{\bar{q}_{j+1}}, \ldots, \Delta_{X_m}^{\bar{q}_m}$ such that *some* domain element in $\Delta_{X_j}^{\bar{q}}$ is compatible with *some* elements in the other domains $\Delta_{X_i}^{\bar{q}_i}$, $1 \leq i \leq m$ and $i \neq j$. If not, then $\Delta_{X_j}^{\bar{q}}$ is simply removed from Δ_{X_j}. If there are such $\Delta_{X_1}^{\bar{q}_1}, \ldots, \Delta_{X_{j-1}}^{\bar{q}_{j-1}}, \Delta_{X_{j+1}}^{\bar{q}_{j+1}}, \ldots, \Delta_{X_m}^{\bar{q}_m}$, then the abstract label $\Delta_{X_j}^{\bar{q}}$ is replaced in Δ_{X_j} by its subdomains $\Delta_{X_j}^{\bar{q}1}$ up to $\Delta_{X_j}^{\bar{q}b}$ where b is the branching rate $b_{X_j}^{\bar{q}}$.

When all subsets $\Delta_{X_j}^{\bar{q}}$ of Δ_{X_j} have been processed this way (including the new ones generated in the course of processing) then the constraint c is hyperarc-consistent as concerns the variable X_j. Note that for ensuring hyperarc-consistency of an entire constraint c it must be hyperarc-consistent concerning each variable $X_j \in$ vars(c), i.e. for each of the variables in $\mathsf{vars}(c)$ all values in the respective domain have to be supported by combinations of values in the domains of the other variables of c.

We are now in a position to generalize the definition of hyperarc-consistency (Definition 24 on page 55) as follows.

Definition 31 (hierarchical hyperarc-consistency) *A given k-ary constraint c with $\mathsf{vars}(c) = \{X_1, \ldots, X_m\}$ is* hierarchically hyperarc-consistent *if it is hierarchically hyperarc-consistent for all variables $X_j \in \mathsf{vars}(c)$. A constraint c is* hierarchically hyperarc-consistent for a variable $X_j \in \mathsf{vars}(c)$ *if each abstract label of Δ_{X_j} is hierarchically hyperarc-consistent with a combination of some abstract labels of Δ_{X_1}, …, $\Delta_{X_{j-1}}$, $\Delta_{X_{j-1}}$, …, Δ_{X_m}. An abstract label $\Delta^{\bar{q}}_{X_j}$ is hierarchically hyperarc-consistent with a combination of abstract labels $\Delta^{\bar{q}_1}_{X_1}$, …, $\Delta^{\bar{q}_{j-1}}_{X_{j-1}}$, $\Delta^{\bar{q}_{j+1}}_{X_{j+1}}$, …, $\Delta^{\bar{q}_m}_{X_m}$ iff the set of leaf labels below $\Delta^{\bar{q}}_{X_j}$ in the domain tree is hyperarc-consistent with the sets of leaf labels below the $\Delta^{\bar{q}_1}_{X_1}$, …, $\Delta^{\bar{q}_{j-1}}_{X_{j-1}}$, $\Delta^{\bar{q}_{j+1}}_{X_{j+1}}$, …, $\Delta^{\bar{q}_m}_{X_m}$.*

5.2.1 The hierarchical domain predicates $\mathcal{A}_c$ and $\mathcal{S}_c$

In order to implement a generalized version of DirectedHyperRevise we need two new sets of predicates derived from the set of compound labels that constitute a constraint c. These are predicates on the abstract labels $D^{i_1 i_2 \ldots i_k}_X$ needed to carry out the operations described above. We define

$$\mathcal{A}_c(D^{\bar{q}}_{X_j}, D^{\bar{q}_1}_{X_1}, \ldots, D^{\bar{q}_{j-1}}_{X_{j-1}}, D^{\bar{q}_{j+1}}_{X_{j+1}}, \ldots, D^{\bar{q}_m}_{X_m})$$

to be true iff for *all* elements belonging to $D^{\bar{q}}_{X_j}$ there are elements in $D^{\bar{q}_1}_{X_1}$, …, $D^{\bar{q}_{j-1}}_{X_{j-1}}$, $D^{\bar{q}_{j+1}}_{X_{j+1}}$, …, $D^{\bar{q}_m}_{X_m}$ that are compatible with it. Analogously, we define

$$\mathcal{S}_c(D^{\bar{q}}_{X_j}, D^{\bar{q}_1}_{X_1}, \ldots, D^{\bar{q}_{j-1}}_{X_{j-1}}, D^{\bar{q}_{j+1}}_{X_{j+1}}, \ldots, D^{\bar{q}_m}_{X_m})$$

to be true iff for *some* element belonging to $D^{\bar{q}}_{X_j}$ there are elements in $D^{\bar{q}_1}_{X_1}$, …, $D^{\bar{q}_{j-1}}_{X_{j-1}}$, $D^{\bar{q}_{j+1}}_{X_{j+1}}$, …, $D^{\bar{q}_m}_{X_m}$ that are compatible with it.

For each constraint $c = C_{\{X_1,\ldots,X_m\}}$ we have to compute the hierarchical domain predicates $\mathcal{A}_c$ and $\mathcal{S}_c$. This can be done inductively by starting from the compound labels in the constraint definition as we well see in the next section.

5.2.2 Inductive computation of the hierarchical predicates

The hierarchical predicate $\mathcal{A}_c$ is true iff all values in the abstract label $D^{\bar{q}}_{X_j}$ are supported by some values in the abstract labels for the remaining variables of the constraint c.

$$\begin{aligned}&\mathcal{A}_c(D^{\bar{q}}_{X_j}, D^{\bar{q}_1}_{X_1}, \ldots, D^{\bar{q}_{j-1}}_{X_{j-1}}, D^{\bar{q}_{j+1}}_{X_{j+1}}, \ldots, D^{\bar{q}_m}_{X_m})\\ &\quad \equiv \forall v_j \in D^{\bar{q}}_{X_j} : \exists v_1 \in D^{\bar{q}_1}_{X_1} \cdots \exists v_{j-1} \in D^{\bar{q}_{j-1}}_{X_{j-1}} \exists v_{j+1} \in D^{\bar{q}_{j+1}}_{X_{j+1}} \cdots \exists v_m \in D^{\bar{q}_m}_{X_m} :\\ &\qquad \mathsf{satisfies}((\langle X_1 \leftarrow v_1\rangle, \ldots, \langle X_j \leftarrow v_j\rangle, \ldots, \langle X_m \leftarrow v_m\rangle), c)\end{aligned}$$

As we assume the constraint c to be given as the set of all compound labels representing exactly those value combinations that are allowed by the constraint, we can represent each m-ary constraint as an m-dimensional Boolean array $\mathcal{B}_c$, in each dimension i ranging over the domain size of the ith variable X_i.

The notion of hierarchical hyperarc-consistency is undirected as hyperarc-consistency is. Thus, we have to compute the following base instances of the predicate $\mathcal{A}_c$ for each variable $X_j = X_1, \ldots, X_m$.

$$\mathcal{A}_c(D_{X_j}^{\bar{q}_j}, D_{X_1}^{\bar{q}_1}, \ldots, D_{X_{j-1}}^{\bar{q}_{j-1}}, D_{X_{j+1}}^{\bar{q}_{j+1}}, \ldots, D_{X_m}^{\bar{q}_m}) \equiv \mathcal{B}_c[v_1, \ldots, v_m]$$

where the $D_{X_1}^{\bar{q}_1}$ to $D_{X_m}^{\bar{q}_m}$ are leaf elements with the singleton values v_1 to v_m, i.e. $D_{X_i}^{\bar{q}_i} = \{v_i\}$ for all $1 \leq i \leq m$, which also means that $b_{X_i}^{\bar{q}_i} = 0$ for all $1 \leq i \leq m$. Starting from these base instances, we can then inductively compute the 'higher' instances as follows, that is, we move from the leafs to the top of the domain trees.

$$\mathcal{A}_c(D_{X_j}^{\bar{q}_j}, D_{X_1}^{\bar{q}_1}, ..., D_{X_l}^{l_1 \ldots l_{k_l-1}}, ..., D_{X_m}^{\bar{q}_m}) = \bigvee_{1 \leq l_{k_l} \leq b} \mathcal{A}_c(D_{X_j}^{\bar{q}_j}, D_{X_1}^{\bar{q}_1}, ..., D_{X_l}^{l_1 \ldots l_{k_l-1} l_{k_l}}, ..., D_{X_m}^{\bar{q}_m})$$

with b being the branching rate at node $D_{X_l}^{l_1 \ldots l_{k_l-1}}$. This means that on the existentially quantified (non-first) argument positions of the predicate $\mathcal{A}_c$ we can move up in the domain tree by performing a bitwise OR on the predicate instances for all the subdomains of that node. Clearly, for finally computing the predicate $\mathcal{A}_c$ with moving up in the first argument position, a bitwise AND on the predicate instances for all the b subdomains has to be performed where b denotes the branching rate at node $D_{X_j}^{j_1 \ldots j_{k_j-1}}$:

$$\mathcal{A}_c(D_{X_j}^{j_1 \ldots j_{k_j-1}}, D_{X_1}^{\bar{q}_1}, ..., D_{X_m}^{\bar{q}_m}) = \bigwedge_{1 \leq j_{k_j} \leq b} \mathcal{A}_c(D_{X_j}^{j_1 \ldots j_{k_j-1} j_{k_j}}, D_{X_1}^{\bar{q}_1}, ..., D_{X_m}^{\bar{q}_m})$$

We shall now illustrate the inductive computation of the hierarchical domain predicate $\mathcal{A}_c$ for c being the `different-colors` constraint in the definition of the hierarchical extended map-coloring problem requiring two neighbored states to be assigned different colors.[2] As mentioned above, the constraint definition can be represented as a relation matrix that forms an m-ary Boolean array—in this case a two-dimensional array.

$$\mathcal{B}_c = \left[\begin{array}{c|ccccc} & \mathsf{r} & \mathsf{bb} & \mathsf{bg} & \mathsf{db} & \mathsf{dg} \\ \hline \mathsf{r} & 0 & 1 & 1 & 1 & 1 \\ \mathsf{bb} & 1 & 0 & 1 & 0 & 1 \\ \mathsf{bg} & 1 & 1 & 0 & 1 & 0 \\ \mathsf{db} & 1 & 0 & 1 & 0 & 0 \\ \mathsf{dg} & 1 & 1 & 0 & 0 & 0 \end{array}\right]$$

Although the formalism presented can be applied to constraints of arbitrary arity, for illustration purposes a binary constraint appears most convenient.

[2]For this example we do not consider the 'relaxed' version where for some pairs of states same colors are allowed to get the problem solvable.

Thus, we directly obtain the base instances for the hierarchical domain predicate $\mathcal{A}_c$.

$$\mathcal{A}_c(D_{X_i}^{i_1i_2}, D_{X_j}^{j_1j_2}) = \left[\begin{array}{c|ccccc} & D_{X_j}^{11} & D_{X_j}^{12} & D_{X_j}^{13} & D_{X_j}^{21} & D_{X_j}^{22} \\ \hline D_{X_i}^{11} & 0 & 1 & 1 & 1 & 1 \\ D_{X_i}^{12} & 1 & 0 & 1 & 0 & 1 \\ D_{X_i}^{13} & 1 & 1 & 0 & 1 & 0 \\ D_{X_i}^{21} & 1 & 0 & 1 & 0 & 0 \\ D_{X_i}^{22} & 1 & 1 & 0 & 0 & 0 \end{array}\right]$$

From these base predicates we can then move up in the domain tree of the second argument position, i.e. move from $D_{X_j}^{1j_2}$ to $D_{X_j}^{1}$ and from $D_{X_j}^{2j_2}$ to $D_{X_j}^{2}$, respectively, and further from $D_{X_j}^{1}$ and $D_{X_j}^{2}$ to D_{X_j} by ORing together corresponding columns:

$$\mathcal{A}_c(D_{X_i}^{i_1i_2}, D_{X_j}^{j_1}) = \left[\begin{array}{c|cc} & D_{X_j}^{1} & D_{X_j}^{2} \\ \hline D_{X_i}^{11} & 1 & 1 \\ D_{X_i}^{12} & 1 & 1 \\ D_{X_i}^{13} & 1 & 1 \\ D_{X_i}^{21} & 1 & 0 \\ D_{X_i}^{22} & 1 & 0 \end{array}\right] \qquad \mathcal{A}_c(D_{X_i}^{i_1i_2}, D_{X_j}) = \left[\begin{array}{c|c} & D_{X_j} \\ \hline D_{X_i}^{11} & 1 \\ D_{X_i}^{12} & 1 \\ D_{X_i}^{13} & 1 \\ D_{X_i}^{21} & 1 \\ D_{X_i}^{22} & 1 \end{array}\right]$$

The first column of $\mathcal{A}_c(D_{X_i}^{i_1i_2}, D_{X_j}^{j_1})$ is obtained by bitwise ORing the first, second, and third columns of $\mathcal{A}_c(D_{X_i}^{i_1i_2}, D_{X_j}^{j_1j_2})$; the second column of $\mathcal{A}_c(D_{X_i}^{i_1i_2}, D_{X_j}^{j_1})$ is obtained by bitwise ORing the fourth and fifth columns of $\mathcal{A}_c(D_{X_i}^{i_1i_2}, D_{X_j}^{j_1j_2})$. From $\mathcal{A}_c(D_{X_i}^{i_1i_2}, D_{X_j}^{j_1j_2})$ we can also move up in the domain tree of the first argument position, i.e. move from $D_{X_i}^{1i_2}$ to $D_{X_i}^{1}$ and from $D_{X_i}^{2i_2}$ to $D_{X_i}^{2}$, respectively, and then further from $D_{X_i}^{1}$ and $D_{X_i}^{2}$ to D_{X_i}:

$$\mathcal{A}_c(D_{X_i}^{i_1}, D_{X_j}^{j_1j_2}) = \left[\begin{array}{c|ccccc} & D_{X_j}^{11} & D_{X_j}^{12} & D_{X_j}^{13} & D_{X_j}^{21} & D_{X_j}^{22} \\ \hline D_{X_i}^{1} & 0 & 0 & 0 & 0 & 0 \\ D_{X_i}^{2} & 1 & 0 & 0 & 0 & 0 \end{array}\right]$$

$$\mathcal{A}_c(D_{X_i}, D_{X_j}^{j_1j_2}) = \left[\begin{array}{c|ccccc} & D_{X_j}^{11} & D_{X_j}^{12} & D_{X_j}^{13} & D_{X_j}^{21} & D_{X_j}^{22} \\ \hline D_{X_i} & 0 & 0 & 0 & 0 & 0 \end{array}\right]$$

Furthermore, from $\mathcal{A}_c(D_{X_i}^{i_1i_2}, D_{X_j}^{j_1})$ we can also induce $\mathcal{A}_c(D_{X_i}^{i_1}, D_{X_j}^{j_1})$ and further obtain $\mathcal{A}_c(D_{X_i}, D_{X_j}^{j_1})$ as follows:

$$\mathcal{A}_c(D_{X_i}^{i_1}, D_{X_j}^{j_1}) = \left[\begin{array}{c|cc} & D_{X_j}^{1} & D_{X_j}^{2} \\ \hline D_{X_i}^{1} & 1 & 1 \\ D_{X_i}^{2} & 1 & 0 \end{array}\right] \qquad \mathcal{A}_c(D_{X_i}, D_{X_j}^{j_1}) = \left[\begin{array}{c|cc} & D_{X_j}^{1} & D_{X_j}^{2} \\ \hline D_{X_i} & 1 & 0 \end{array}\right]$$

To illustrate the meaning of these numbers, consider the table for $\mathcal{A}_c(D_{X_i}, D_{X_j}^{j_1})$: The entry in the first column is a 1, which indicates that it is true that for all elements in $D_{X_i} = \{\mathsf{r}, \mathsf{bb}, \mathsf{bg}, \mathsf{db}, \mathsf{dg}\}$ there is an element in $D_{X_j}^1 = \{\mathsf{r}, \mathsf{bb}, \mathsf{bg}\}$ (the bright colors) such that the `different-colors` constraint is satisfied. However, as the entry in the second column is a 0, it is not true that for all elements in D_{X_i} there is also an element in $D_{X_j}^2 = \{\mathsf{db}, \mathsf{dg}\}$ (the dark colors) such that the constraint is satisfied.

Finally, we can then take $\mathcal{A}_c(D_{X_i}^{i_1 i_2}, D_{X_j})$ and obtain first $\mathcal{A}_c(D_{X_i}^{i_1}, D_{X_j})$ and in the very last step $\mathcal{A}_c(D_{X_i}, D_{X_j})$:

$$\mathcal{A}_c(D_{X_i}^{i_1}, D_{X_j}) = \left[\begin{array}{c|c} & D_{X_j} \\ \hline D_{X_i}^1 & 1 \\ D_{X_i}^2 & 1 \end{array}\right] \qquad \mathcal{A}_c(D_{X_i}, D_{X_j}) = \left[\begin{array}{c|c} & D_{X_j} \\ \hline D_{X_i} & 1 \end{array}\right]$$

As introduced in Section 5.2.1, we also define a set of hierarchical domain predicates $\mathcal{S}_c$ such that the predicate $\mathcal{S}_c$ is true iff some value in the abstract label $D_{X_j}^{\bar{q}}$ is supported by some values in the abstract labels for the remaining variables of the constraint c.

$$\begin{aligned} &\mathcal{S}_c(D_{X_j}^{\bar{q}}, D_{X_1}^{\bar{q}_1}, \ldots, D_{X_{j-1}}^{\bar{q}_{j-1}}, D_{X_{j+1}}^{\bar{q}_{j+1}}, \ldots, D_{X_m}^{\bar{q}_m}) \\ &\quad \equiv \exists v_j \in D_{X_j}^{\bar{q}} : \exists v_1 \in D_{X_1}^{\bar{q}_1} \cdots \exists v_{j-1} \in D_{X_{j-1}}^{\bar{q}_{j-1}} \exists v_{j+1} \in D_{X_{j+1}}^{\bar{q}_{j+1}} \cdots \exists v_m \in D_{X_m}^{\bar{q}_m} : \\ &\quad\quad \mathsf{satisfies}((\langle X_1 \leftarrow v_1\rangle, \ldots, \langle X_j \leftarrow v_j\rangle, \ldots, \langle X_m \leftarrow v_m\rangle), c) \end{aligned}$$

The predicate $\mathcal{S}$ can also be computed inductively starting from the relation matrix $\mathcal{B}_c$. As for the $\mathcal{A}$ predicate, the relation matrix $\mathcal{B}_c$ also constitutes the base instances of $\mathcal{S}$. Similarly to the computation of $\mathcal{A}$, we can compute the 'higher' instances as follows:

$$\mathcal{S}_c(D_{X_j}^{\bar{q}_j}, D_{X_1}^{\bar{q}_1}, ..., D_{X_l}^{l_1 .. l_{k_l - 1}}, ..., D_{X_m}^{\bar{q}_m}) = \bigvee_{1 \le l_{k_l} \le b_{k_l}} \mathcal{S}_c(D_{X_j}^{\bar{q}_j}, D_{X_1}^{\bar{q}_1}, ..., D_{X_l}^{l_1 .. l_{k_l - 1} l_{k_l}}, ..., D_{X_m}^{\bar{q}_m})$$

$$\mathcal{S}_c(D_{X_j}^{j_1 .. j_{k_j - 1}}, D_{X_1}^{\bar{q}_1}, ..., D_{X_m}^{\bar{q}_m}) = \bigvee_{1 \le j_{k_j} \le b_{k_j}} \mathcal{S}_c(D_{X_j}^{j_1 .. j_{k_j - 1} j_{k_j}}, D_{X_1}^{\bar{q}_1}, ..., D_{X_m}^{\bar{q}_m})$$

with b_{k_l} and b_{k_j} being the branching rates at nodes $D_{X_l}^{l_1 .. l_{k_l - 1}}$ and $D_{X_j}^{j_1 .. j_{k_j - 1}}$, respectively. In other words the hierarchy of $\mathcal{S}_c$ predicates is computed by collapsing the $\mathcal{B}_c$ table: ORing together subsets of rows or subsets of columns. Hence, we need not make any difference between existentially quantified and universally quantified argument positions—as all abstract domains are existentially quantified in the definition of $\mathcal{S}_c$.

5.3 A hierarchical hyperarc-consistency algorithm

We are now in a position to define a generalized ENFORCEHIERARCHICALHYPERARC-CONSISTENCY algorithm. The algorithm uses the general constraint propagation structure of ENFORCEHYPERARCCONSISTENCY (see Figure 3.14 on page 58) but uses DIRECTEDHIERARCHICALHYPERREVISE as an extension to DIRECTEDHYPERREVISE (see

```
procedure DirectedHierarchicalHyperRevise($c, X_j$):
$Delete \leftarrow$ false
$Q_1 \leftarrow \Delta_{X_j}$
$\Delta_{X_j} \leftarrow \emptyset$
while $Q_1 \neq \emptyset$ do
    select and delete an element $\Delta^{\bar{q}}_{X_j}$ from $Q_1$
    $Q_2 \leftarrow \{(d_1, \ldots, d_{j-1}, d_{j+1}, \ldots, d_m) \mid d_i \in \Delta_{X_i}\}$
    $Found \leftarrow$ false
    while $Q_2 \neq \emptyset$ and not $Found$ do
        select and delete a combination $(d_1, \ldots, d_{j-1}, d_{j+1}, \ldots, d_m)$ from $Q_2$
        if $\mathcal{A}_c(\Delta^{\bar{q}}_{X_j}, d_1, \ldots, d_{j-1}, d_{j+1}, \ldots, d_m)$ then
            $\Delta_{X_j} \leftarrow \Delta_{X_j} \cup \Delta^{\bar{q}}_{X_j}$
            $Found \leftarrow$ true
        endif
    endif
    if not $Found$ then
        $Delete \leftarrow$ true
        if $b^{\bar{q}}_{X_j} > 0$ then
            $Q_2 \leftarrow \{(d_1, \ldots, d_{j-1}, d_{j+1}, \ldots, d_m) \mid d_i \in \Delta_{X_i}\}$
            while $Q_2 \neq \emptyset$ and not $Found$ do
                select and delete a combination $(d_1, \ldots, d_{j-1}, d_{j+1}, \ldots, d_m)$ from $Q_2$
                if $\mathcal{S}_c(\Delta^{\bar{q}}_{X_j}, d_1, \ldots, d_{j-1}, d_{j+1}, \ldots, d_m)$ then
                    for $r$ from 1 to $b^{\bar{q}}_{X_j}$ do
                        $Q_1 \leftarrow Q_1 \cup \{D^{\bar{q}r}_{X_j}\}$
                    endfor
                    $Found \leftarrow$ true
                endif
            endwhile
        endif
    endif
endwhile
return $Delete$
```

Figure 5.3: Procedure DirectedHierarchicalHyperRevise

Figure 3.13 on page 58) in order to enforce (directed) hierarchical hyperarc-consistency on a constraint c concerning a variable $X_j \subseteq \mathsf{vars}(c) = \{X_1, \ldots, X_m\}$.

The procedure DirectedHierarchicalHyperRevise implements the generalized hyperarc-consistency algorithm introduced in Section 5.2. In particular, after the application of DirectedHierarchicalHyperRevise(c,X_j) the constraint c will be *hierarchically hyperarc-consistent for the variable* X_j in the sense of Definition 31.

The main while loop tests each abstract label in Δ_{X_j} to see if it is hierarchically hyperarc-consistent. It does that by testing each label $\Delta^{\bar{q}}_{X_j}$ in Δ_{X_j} against the abstract labels in Δ_{X_i} for all other variables X_1 to X_{j-1} and X_{j+1} to X_m.

The inner while loop looks for a combination[3] $(d_1, \dots, d_{j-1}, d_{j+1}, \dots, d_m)$ of abstract labels in the respective current domains $\Delta_{X_i}^{\bar{q}_i}$ of the other variables such that $\mathcal{A}_c$ is true. In that case, $\Delta_{X_j}^{\bar{q}}$ survives unchanged in Δ_{X_j}.

If not and $\Delta_{X_j}^{\bar{q}}$ is not a leaf ($b_{X_j}^{\bar{q}} > 0$) then the third while loop looks for a combination $(d_1, \dots, d_{j-1}, d_{j+1}, \dots, d_m)$ of abstract labels such that at least some elements in $\Delta_{X_j}^{\bar{q}}$ are compatible with values in the domains d_1 to d_{j-1} and d_{j+1} to d_m in which case $\mathcal{S}_c$ is true. In that case, $\Delta_{X_j}^{\bar{q}}$ is replaced in Q_1 by its subdomains. They must be tested similarly on $\mathcal{A}_c$ and maybe also $\mathcal{S}_c$ before this invocation of DirectedHierarchicalHyperRevise returns. The return value finally indicates to the caller whether the domain of the variable X_j has been restricted or not.

5.3.1 Correctness and Termination

It should be shown that DirectedHierarchicalHyperRevise is correct and always terminates.

Proposition 7 (correctness of DirectedHierarchicalHyperRevise) *Let c be a constraint and X_j be a variable with $X_j \in \mathsf{vars}(c) = \{X_1, \dots, X_m\}$. Then, after applying* DirectedHierarchicalHyperRevise *the constraint c is hierarchically hyperarc-consistent for variable X_j.*

Proof. The current domain Δ_{X_j} starts as the empty set and adds members $\Delta_{X_j}^{\bar{q}}$ only within the second while loop when $\mathcal{A}_c(\Delta_{X_j}^{\bar{q}}, d_1, \dots, d_{j-1}, d_{j+1}, \dots, d_m)$ is true. Thus, when DirectedHierarchicalHyperRevise returns, all members of Δ_{X_j} are hierarchically hyperarc-consistent with a combination of some abstract labels in the domains of the other variables X_1 to X_{j-1} and X_{j+1} to X_m. Hence, the constraint c is hierarchically hyperarc-consistent for variable X_j and thus the algorithm is correct. □

Proposition 8 (termination of DirectedHierarchicalHyperRevise) *The procedure* DirectedHierarchicalHyperRevise *always terminates.*

Proof. All while loops in the procedure are controlled by the queues Q_1 and Q_2. The number of elements on the queues is strong monotonically decreasing, except for the case when the $\mathcal{S}_c$ predicate has been tested successfully and a finite number of subdomains are added to Q_1. However, as the domain trees are noncyclic, this can happen only finitely many times, and hence the procedure DirectedHierarchicalHyperRevise will always terminate. □

5.3.2 Solving CSPs over hierarchically structured domains

As mentioned before, the extensions for dealing with hierarchically structured domains only affect the domain revision procedure and require no change at all to the Enforce-

[3]Note that it is not necessary to compute the set of all possible combinations and assign it to the queue Q_2 as the algorithm suggests. Instead, it suffices to generate combinations until one is found such that $\mathcal{A}_c$ or $\mathcal{S}_c$ is true. This is also the way how Contax checks for hierarchical hyperarc-consistency. However, the presentation as in Figure 5.3 is used to have the algorithm easier to understand.

HyperArcConsistency procedure. Thus, we can define a EnforceHierarchicalHyperArcConsistency procedure the same as EnforceHyperArcConsistency, except that it uses the procedure DirectedHierarchicalHyperRevise instead of DirectedHyperRevise.

Using the EnforceHierarchicalHyperArcConsistency procedure, we can then also easily adapt the GenerateSolutions algorithm presented in Figure 3.15 to dealing with hierarchically structured domains. The only necessary change (besides replacing EnforceHyperArcConsistency and DirectedHyperRevise by EnforceHierarchicalHyperArcConsistency and DirectedHierarchicalHyperRevise, respectively) is that in step SelVal the algorithm does not select individual values $v \in \Delta_{X_i}$ but selects abstract domains $\Delta^{\bar{q}}_{X_i} \in \Delta_{X_i}$. The remaining part of the GenerateSolutions algorithm can be reused unchanged.

Moreover, as the current domain Δ_{X_i} of a variable X_i may finally contain non-leaf abstract labels, the algorithm can also return *intensional solutions* which are more compact than enumerating all the individual elements belonging to an abstract label.

In order to illustrate how the DirectedHierarchicalHyperRevise procedure works, we consider a small example.

Example 18 (Intensional solutions using hierarchical hyperarc-consistency) *Assume the variable* HH *to be already assigned the color* red*, i.e.* $\Delta_{\mathsf{HH}} = \{\mathsf{r}\}$*, and* c *being the* `different-color` *constraint on* HH *and* SH *where the current domain of* SH *still contains all colors, i.e.* $\Delta_{\mathsf{SH}} = \{\mathsf{r}, \mathsf{bg}, \mathsf{dg}, \mathsf{bb}, \mathsf{db}\}$*. Now consider the procedure call*

DirectedHierarchicalHyperRevise*(c,*SH*).*

As $\mathcal{A}_c(\Delta_{\mathsf{SH}}, \Delta_{\mathsf{HH}}) = 0$ *but* $\mathcal{S}_c(\Delta_{\mathsf{SH}}, \Delta_{\mathsf{HH}}) = 1$*, the subdomains* Δ^1_{SH} *(representing the bright colors) and* Δ^2_{SH} *(representing the dark colors) are added to the queue* Q_1*.*

In the next loop, Δ^1_{SH} *is checked:* $\mathcal{A}_c(\Delta^1_{\mathsf{SH}}, \Delta_{\mathsf{HH}}) = 0$ *and* $\mathcal{S}_c(\Delta^1_{\mathsf{SH}}, \Delta_{\mathsf{HH}}) = 1$*. Thus,* Δ^1_{SH} *is replaced in* Q_1 *by* $\Delta^{11}_{\mathsf{SH}}$ *representing* r*,* $\Delta^{12}_{\mathsf{SH}}$ *representing* bg*, and* $\Delta^{13}_{\mathsf{SH}}$ *representing* bb*.*

The next main loop checks Δ^2_{SH} *(representing the dark colors). As* $\mathcal{A}_c(\Delta^2_{\mathsf{SH}}, \Delta_{\mathsf{HH}}) = 1$*, all dark colors for* SH *are compatible with* r *for* HH*. Thus, the abstract label* Δ^2_{SH} *is added to* Δ_{SH}*.*

The final three main loops check the abstract labels $\Delta^{11}_{\mathsf{SH}}$*,* $\Delta^{12}_{\mathsf{SH}}$*, and* $\Delta^{13}_{\mathsf{SH}}$ *for* SH *from which* $\Delta^{12}_{\mathsf{SH}}$ *and* $\Delta^{13}_{\mathsf{SH}}$ *survive because* $\mathcal{A}_c(\Delta^{11}_{\mathsf{SH}}, \Delta_{\mathsf{HH}}) = 0$*, but* $\mathcal{A}_c(\Delta^{12}_{\mathsf{SH}}, \Delta_{\mathsf{HH}}) = 1$ *and* $\mathcal{A}_c(\Delta^{13}_{\mathsf{SH}}, \Delta_{\mathsf{HH}}) = 1$*.*

Hence, the resulting domain Δ_{SH} *is the set of abstract labels* $\{\Delta^2_{\mathsf{SH}}, \Delta^{12}_{\mathsf{SH}}, \Delta^{13}_{\mathsf{SH}}\}$ *that represents the set of all dark colors* (Δ^2_{SH}) *together with the individual colors* bright green $(\Delta^{12}_{\mathsf{SH}})$ *and* bright blue $(\Delta^{13}_{\mathsf{SH}})$*.*

Although the intensional solution Δ^2_{SH} does only represent two individual domain elements, this small example may already illustrate the advantages of dealing with abstract labels (intensional solutions) rather than with individual elements as in the procedure DirectedHyperRevise before. In the next section, we will present how the Contax user can benefit of the extensions discussed in this chapter.

5.4 Hierarchically Structured Domains in CONTAX

In this section, we will now present how hierarchically structured domains can be represented in CONTAX and how constraints over them can be efficiently processed. Finally, we will also show how the hierarchical map-coloring problem can be represented and solved in CONTAX.

5.4.1 Representing hierarchical domains

In Section 5.1, we have already seen how hierarchical domains can be represented in CONTAX using a generalized version of the `def-domain` macro introduced in Section 3.2.4. The generalization simply is that any already defined domain can be further refined hierarchically by defining its subdomains in just the same way using the `def-domain` macro. This means that in a statement of the form

$$\texttt{(def-domain } \mathit{dom} \texttt{ (} \mathit{dom}_1, \ldots, \mathit{dom}_k \texttt{))},$$

the dom_i do not necessarily represent individual objects. Individual objects are represented by leaf nodes in the domain hierarchy, i.e. as singleton domains that are not further refined. For example, the domains for the hierarchical map-coloring problem can in CONTAX be defined as follows:

```
(def-domain colors (bright dark))
(def-domain bright (r bb bg))
(def-domain dark (db dg))
```

This extension to hierarchically define the domains in CONTAX can already well be used without the extensions for processing constraints over them that have been discussed in the previous sections: The hierarchical domain definitions above can simply be transformed into the following 'flat' domain definitions:

```
(def-domain colors (r bb bg db dg))
(def-domain bright (r bb bg))
(def-domain dark (db dg))
```

As it can be seen more declarative to express the hierarchical structure of the domain rather than to enumerate even redundantly all individuals in each domain, this extension directly improves the declarativity of our approach. Therefore, the CONTAX kernel presented in Section 3.2.4 already supports hierarchical domain definitions which are then compiled into 'flat' definitions for each non-leaf hierarchical domain.

5.4.2 Using TAXON taxonomies as hierarchical domains

Defining domains in the way described in Section 5.4.1 results in a *constitutive* approach to structuring the domain. However, in some applications it seems more appropriate to declaratively *define* what the common properties of all individuals are that belong to a particular domain. This can be done very declaratively using terminological knowledge representation systems in the tradition of KL-ONE [Brachman and Schmolze, 1985] that allow for defining *concepts* using terminological logics and provide mechanisms to automatically compute the hierarchical[4] structure of the concepts.

As terminological knowledge representation systems offer a purely declarative means to define the domain of variables in a CSP, CONTAX also allows to incorporate the hierarchical domain structure represented and computed by TAXON, a terminological knowledge representation system also developed at DFKI [Hanschke *et al.*, 1991, Hanschke, 1993].

Having loaded and *classified*[5] the relevant concepts in TAXON, the resulting domain structure can also be used in CONTAX. For example, in the CIM application from which the lathe-tool selection problem in Chapter 8 is taken, the domain of workpieces was defined in terms of TAXON concepts. It could be imported to and used by CONTAX simply by loading TAXON and CONTAX together in the same COMMONLISP environment and then start a concept-to-domain transformation that will not be discussed in this thesis:

```
(import-taxon-domain workpieces)
```

Given this call, CONTAX will define the domain `workpieces` together with domains corresponding to all TAXON concepts that are subsumed by the concept `workpieces`. Thus, the TAXON interface provided in CONTAX also contributes to a more declarative representation of CSPs, respectively their domains, and enables sharing and reusing of domain taxonomies such that they are defined once and can be used in many different applications.

5.4.3 Constraints over hierarchical domains

As presented so far, hierarchical domains are only a means of structured and more declarative representation of the domains without influencing the processing of constraints over them. Depending on the domain structure, using the hierarchical hyperarc-consistency technique presented in this chapter may not pay off for all constraints, especially not for constraints over variables that range over 'flat' domains. Therefore, CONTAX allows for explicitly request the hierarchical hyperarc-consistency technique to be used for a particular constraint. This is done when creating an instance of a constraint type. Consider, for example, creating the constraint instance that has been discussed in Example 18:

```
(make-constraint :name C7
                 :type different-colors
                 :propagation hierarchical
                 :color1 HH
                 :color2 SH)
```

[4]The resulting subsumption graph does not necessarily build a hierarchy, but may in general simply form a directed acyclic graph structure.

[5]Classification in terminological knowledge representation systems means computing the subsumption relations between all concepts defined in the system.

When creating a constraint instance with `hierarchical` propagation mode[6], CONTAX automatically computes the hierarchical domain predicates $\mathcal{A}$ and $\mathcal{S}$ such that they are available when needed for propagation, i.e. for enforcing hierarchical hyperarc-consistency.

As checking for hierarchical hyperarc-consistency is always local to a particular constraint, it is also possible to use different hierarchical structures of the same domain in different constraints. This becomes especially important when different constraints linked to the same variable exploit different properties of the individuals in the domain. Thus, one hierarchical structure of the domain may be better suited for one constraint than another.

For example, for expressing and processing the constraint that no neighbored states should be both assigned dark colors, the domain structure shown in Figure 5.1 is well suited as it supports the notion of bright and dark colors. However, for representing the basic constraint that neighbored states should be assigned different colors, the domain structure shown in Figure 5.4 seems more adequate.

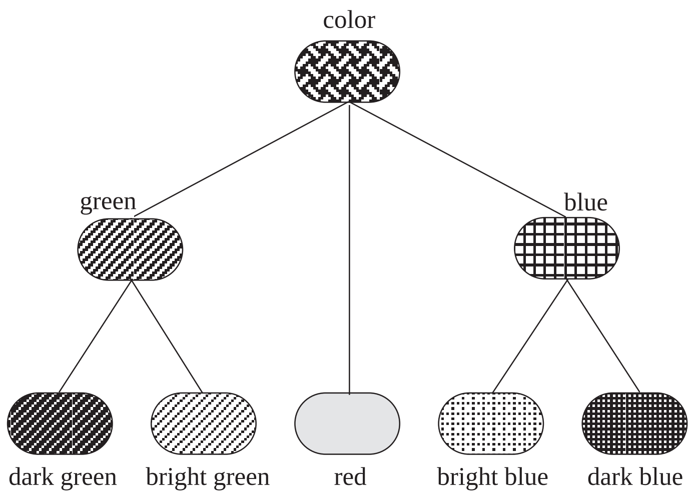


Figure 5.4: Alternative hierarchical structure of the color domain

Therefore, CONTAX allows to specify the domain structure to be used for a variable separately with each constraint type. This exactly is the reason for the presence of the `:domains` keyword in the definition of the `def-primitive-constraint` macro (see Section 3.2.4) which otherwise would be redundant as a domain is also specified with the creation of a variable.

Another benefit of using multiple domain structures is, that CONTAX also allows to use names of entire (sub)domains when enumerating all compound labels that constitute a constraint. If a value in such a compound label is the name of a domain, then CONTAX will successively substitute the domain by all individuals belonging to it. Thus, using domain names in constraint definitions provides a compact constraint representation.

[6]Alternatively, the user can specify `:propagation extensional` if 'normal' hyperarc-consistency shall be enforced on the constraint extension, i.e. constraint propagation shall work as described in Chapter 3. If no propagation mode is specified, the `extensional` mode will be assumed.

In particular, it also allows to reuse the definition of the `different-colors` constraint from Figure 3.17 even when some subdomains in the color domain have been refined to separate bright and dark variants of the colors.

5.4.4 The hierarchical map-coloring problem in CONTAX

In the previous sections, we have seen how CONTAX has been extended to support hierarchically structured domains. Thus, we can now discuss how to represent and finally solve the hierarchical map-coloring problem (Problem 3 on page 92) using CONTAX.

```
(def-domain colors-rgb (red green blue))
(def-domain red (r))
(def-domain green (bg dg))
(def-domain blue (bb db))

(def-domain colors-bd (bright dark))
(def-domain bright (r bb bg))
(def-domain dark (db dg))

(def-primitive-constraint different-colors
   :interface (c1 c2)
   :domains (colors-rgb colors-rgb)
   :tuples ((red green) (red blue) (green red)
            (green blue) (blue red) (blue green)))

(def-primitive-constraint not-both-dark
   :interface (c1 c2)
   :domains (colors-bd colors-bd)
   :tuples ((bright bright) (bright dark) (dark bright)))

(def-compound-constraint neighbored
   :interface (c1 c2)
   :constraints ((different-colors c1 c2) (not-both-dark c1 c2)))

(def-primitive-constraint smallborder
   :interface (c1 c2)
   :domains (colors-bd colors-bd)
   :tuples ((bright dark) (dark bright)
            (r bg) (r bb) (bg bb) (bg r) (bb r) (bb bg)))

(def-predicative-constraint 4equal
   :interface (c1 c2 c3 c4)
   :domains (colors-bd colors-bd colors-bd colors-bd)
   :predicate (= c1 c2 c3 c4))
```

Figure 5.5: Representing the hierarchical German-map coloring problem in CONTAX

Figure 5.5 shows the representation of domains and constraint types for the hierarchical extended map-coloring problem. The overall constraint `neighbored` between two neighboring states can ideally be represented as a compound constraint that combines two primitive constraints over the same variables but using different hierarchical structures for the color domain: `color-rgb` for the inequality `different-colors` on the level of the colors `red`, `green`, and `blue`; `color-bd` for the `not-both-dark` constraint which can be defined most naturally when grouping bright and dark colors to subdomains respectively.

Figure 5.6 finally shows the variables and constraint instances that complete the representation of the hierarchical map-coloring problem in Contax.

```
(make-variable :name SH :domain colors)
   ...  14 more variables  ...
(make-variable :name SN :domain colors)

(make-constraint :type neighbored :c1 SH :c2 HH :propagation hierarchical)
(make-constraint :type neighbored :c1 SH :c2 NS :propagation hierarchical)
(make-constraint :type neighbored :c1 HH :c2 NS :propagation hierarchical)
(make-constraint :type neighbored :c1 NS :c2 HB :propagation hierarchical)
(make-constraint :type neighbored :c1 NS :c2 NW :propagation hierarchical)
(make-constraint :type neighbored :c1 NS :c2 HS :propagation hierarchical)
(make-constraint :type neighbored :c1 NW :c2 HS :propagation hierarchical)
(make-constraint :type neighbored :c1 NW :c2 RP :propagation hierarchical)
(make-constraint :type neighbored :c1 HS :c2 RP :propagation hierarchical)
(make-constraint :type smallborder :c1 HS :c2 BW :propagation extensional)
(make-constraint :type neighbored :c1 HS :c2 BY :propagation hierarchical)
(make-constraint :type neighbored :c1 RP :c2 SL :propagation hierarchical)
(make-constraint :type neighbored :c1 RP :c2 BW :propagation hierarchical)
(make-constraint :type neighbored :c1 BW :c2 BY :propagation hierarchical)
(make-constraint :type neighbored :c1 SH :c2 MV :propagation hierarchical)
(make-constraint :type neighbored :c1 NS :c2 MV :propagation hierarchical)
(make-constraint :type neighbored :c1 NS :c2 SA :propagation hierarchical)
(make-constraint :type smallborder :c1 MV :c2 SA :propagation extensional)
(make-constraint :type neighbored :c1 NS :c2 TH :propagation hierarchical)
(make-constraint :type neighbored :c1 HS :c2 TH :propagation hierarchical)
(make-constraint :type neighbored :c1 BY :c2 TH :propagation hierarchical)
(make-constraint :type neighbored :c1 MV :c2 BB :propagation hierarchical)
(make-constraint :type neighbored :c1 SA :c2 BB :propagation hierarchical)
(make-constraint :type neighbored :c1 BB :c2 BL :propagation hierarchical)
(make-constraint :type neighbored :c1 SA :c2 SN :propagation hierarchical)
(make-constraint :type neighbored :c1 BB :c2 SN :propagation hierarchical)
(make-constraint :type neighbored :c1 SA :c2 TH :propagation hierarchical)
(make-constraint :type neighbored :c1 TH :c2 SN :propagation hierarchical)
(make-constraint :type smallborder :c1 BY :c2 SN :propagation extensional)
(make-constraint :type 4equal :c1 SL :c2 BL :c3 HB :c4 HH)
```

Figure 5.6: Representing the hierarchical German-map coloring problem in Contax

If we now start solving the hierarchical map-coloring problem with the initial assignment of the subdomain `green` for Hessen, we obtain a set of four solutions from which two of them contain a subdomain as value. As these subdomain contains two individuals, we have a total number of six solutions:

```
(solve :all
       :HS (green)
       :solutions all)

⟹ ((TH BB BW SL NW HB SH BL SN SA MV BY RP HS NS HH)
     (r r r dg r dg r dg db bg dg bb bb dg bb dg)
     (r r green dg r dg r dg db bg dg bb bb dg bb dg)
     (r r r bg r bg r bg db bg dg bb bb bg bb bg)
     (r r green bg r bg r bg db bg dg bb bb bg bb bg)
```

Figure 5.7: Solving the hierarchical German-map coloring problem with Contax

The first of these solutions is also shown in Figure 5.8. The other solutions can be obtained by (1) using any `green` color for Baden-Württemberg (`BW`) or (2) by using bright green (`bg`) instead of dark green (`dg`) for the small states. These modifications can all be made without affecting the colors assigned to the remaining states, and thus we obtain a total number of 6 solutions.

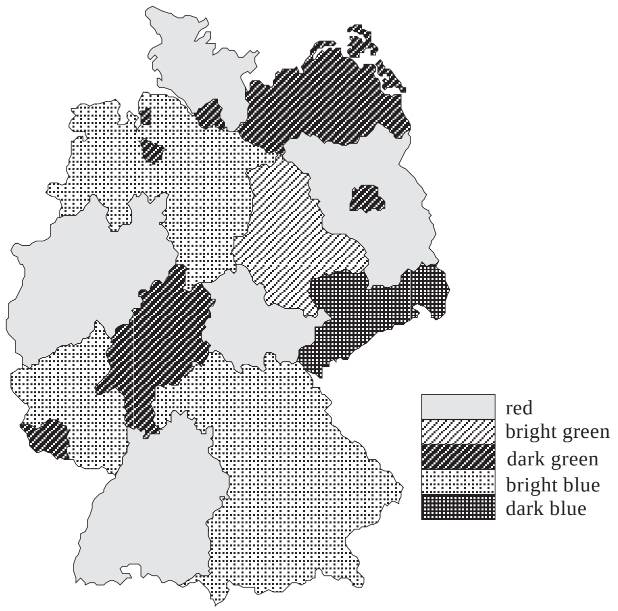


Figure 5.8: A solution to the hierarchical German-map coloring problem

5.5 Discussion and Related Work

In this chapter, we have presented the concept of hierarchical hyperarc-consistency that exploits the internal structuring of domain values into a hierarchy of subdomains. The procedure DirectedHierarchicalHyperRevise that we have developed uses the domain hierarchy in order to process values with common properties as one unit rather than individually as classical constraint solvers do.

Therefore, the algorithm is based on precompiled information about the relations between subdomains. These hierarchical domain predicates $\mathcal{A}_c$ and $\mathcal{S}_c$ have to be computed once for each constraint c. This can be done prior to solving a concrete application problem. Using a bit vector representation for the $\mathcal{A}_c$ and $\mathcal{S}_c$ matrices, these tables can be computed very efficiently even for large domains. However, it seems obvious that the effect of using hierarchical hyperarc-consistency processing instead of 'flat' hyperarc-consistency checking heavily depends on the structure of the domain under consideration.

Also, one should not expect hierarchical hyperarc-consistency to improve on the worst case performance of simple hyperarc-consistency. Indeed, since it relies on a hierarchical organization of the domain, intuition suggests that one could perversely structure the domains in the worst possible way to ensure worst case behavior worse than standard hyperarc-consistency. Empirical studies have shown that for small hierarchies or hierarchies with a high branching rate in relation to the height of the hierarchy, using hierarchical hyperarc-consistency may not pay off. Therefore, the Contax system allows to select those constraints for which hierarchically hyperarc-consistency processing is required. As this decision somehow depends on characteristics of the domain structures, it is worth studying how the decision can be made automatically based on these characteristics. However, this is still an open problem that also requires more empirical studies on large problems to be carried out. Nevertheless, it shows an interesting and promising direction for further research that will also contribute to further improve the declarativity of our approach.

In [Mackworth *et al.*, 1985], an algorithm called HAC (hierarchical arc-consistency) is presented that also builds on hierarchically structuring the variable domains. However, this approach is limited to dealing with complete strict binary trees as domain structures and can also only be applied to binary CSPs. Recently, Sidebottom and Havens have applied hierarchical arc-consistency techniques to numerical processing [Sidebottom and Havens, 1991]. The interesting relation to our work is that numerical domains, e.g. $\mathbb{R}$, are hierarchically refined into subdomains representing intervals. However, in this approach the hierarchical domain structure is generated dynamically whereas our approach of enforcing hierarchical hyperarc-consistency is based on a precompiled domain structure represented by the bit matrices $\mathcal{A}$ and $\mathcal{S}$.

Finite Domain Constraints in Logic Programming

In the previous chapters, we have presented techniques to deal with weighted constraints and overconstrained problems as well as with constraints over hierarchically structured domains. The intention for this research was to make constraint processing techniques available for a broader range of application problems: Weighted or overconstrained problems could not be tackled with classical CSP techniques and hierarchically structured domains could only be dealt with quite inefficiently.

However, to solve an application problem it is not sufficient to have a constraint solver at hand. In nearly all applications there are tasks to be performed that CSP techniques cannot deal with, e.g. user interaction and overall control organization. Therefore, constraint solving techniques have to be made available within a universal programming language.

One approach is to embed a constraint solver in some programming language such that the main program delegates appropriate subtasks to the constraint solver. As most academic constraint solvers are implemented in LISP, many of them provide interfaces to be called from within any LISP application: CONSTRAINTLISP [Liu and Ku, 1992] offers a LISP-based object-oriented constraint solver, CONSAT has been embedded in the knowledge representation workbench BABYLON [Güsgen *et al.*, 1987], and there is also a programmer's interface to the CONTAX system which has been used in the sports scheduling application to be presented in Chapter 8.

All these approaches can only offer a loose coupling of the programming language, e.g. LISP, with the constraint solver such that as a result constraint processing cannot be interleaved with processing of procedures or functions in the main program. However, if we choose a logic programming language like PROLOG as the host language for our constraint solver, we can achieve a much tighter integration as the relational form of knowledge representation and programming in PROLOG is very close to the notion of constraints. Therefore, in this chapter we will investigate how to integrate constraint solving techniques within logic programming. As the topic of this thesis are finite domain constraints, we will focus on consistency techniques over *Fi*nite *Do*mains. In Section 6.1 we will first introduce the basic idea of integrating consistency techniques in logic programming and will present the two most prominent consistency techniques, *forward-checking* and *looking-ahead*. Together with other approaches that have been presented in the previous chapters, these techniques will then in Section 6.2 be reviewed from the viewpoint of finding a compromise between problem reduction and backtrack search in order to minimize the total problem-solving costs. These considerations motivate the development the *weak looking-ahead* (WLA) technique, that will be presented more formally in Section 6.3. However, a detailed illustration again using a variant of our running example, the German-map coloring problem, will be given in Section 6.4. The chapter will then be concluded with Section 6.5 by giving an overview on the FiDo system, in which all consistency techniques discussed in this chapter have been implemented and for which various implementational approaches have been studied.

6.1 Consistency Techniques in Logic Programming

What makes logic programming especially well-suited for stating constraint problems, are the relational form it provides, and its nondeterminism: Since constraints are nothing but relations between objects represented as variables, they can be formulated naturally and conveniently in logic programs; and the nondeterminism liberates the programmer from doing explicit tree search and allows declarative formulation of problems.

Therefore, logic programming seems appropriate for *stating* constraints, and so for stating CSPs, e.g. discrete combinatorial problems. Unfortunately, standard logic programming as realized e.g. in Prolog does not support efficient methods for *solving* CSPs. The drawbacks of logic programming have led to intensive research efforts aimed at improving the control facilities of logic programming languages. One approach, to which increasing attention has been paid over the past years, is constraint logic programming. The main idea is to combine the strong points of logic programming, which are its *declarativity*, its simple and clear semantics based on a well-understood mathematical model and its nondeterminism, with the *efficiency* of constraint solving techniques as they have been presented in the previous chapters.

6.1.1 Constraint logic programming

The general structure of a constraint logic programming system reflects its purpose: the combination of logic programming and constraint solving. Figure 6.1 shows the organization of a constraint logic programming system.

Basically, it consists of two components, an inference machine doing the logic part, and an incremental constraint solver, which can be considered as a decision procedure for a class of constraints. The two modules communicate by an interface. The inference machine recognizes constraints and passes them to the constraint solver. The latter one incrementally creates a constraint network and tries to solve the corresponding CSP, reporting the results back to the inference machine[1].

[1]In real systems, this way of job-sharing can be somehow varying, e.g. some simple constraints can be directly solved by the inference engine.

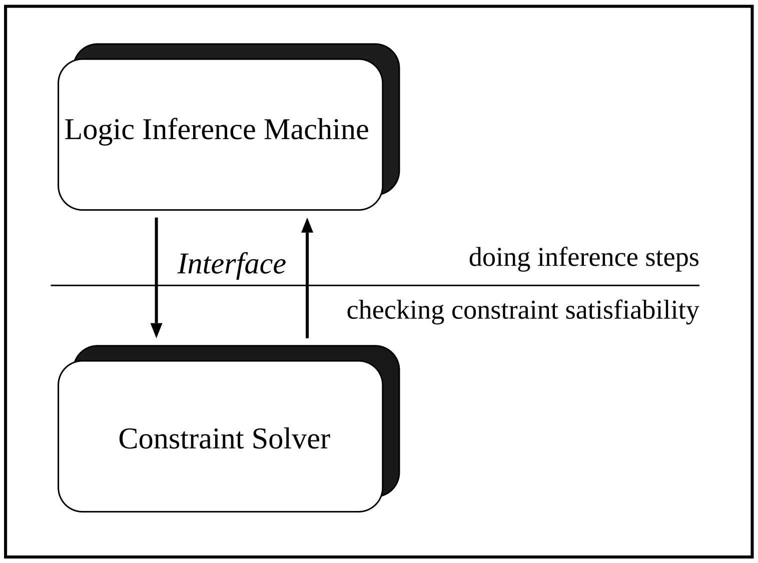


Figure 6.1: The general architecture of a constraint logic programming system

6.1.2 Requirements for finite domain consistency techniques

In the present thesis, techniques for solving constraints over finite domains are to be examined. Therefore, in the following some considerations are made about what is needed to incorporate constraint satisfaction techniques in the framework of logic programming.

- **Finite Domains:** In logic programming, all variables range over the *Herbrand universe* whereas in the FCSP framework, each variable ranges over a finite set of possible values, referred to as the *domain* of the variable. Thus, we need a *domain concept* for logic variables which enables us to restrict the domain of a variable in an active way.

- **Consistency Techniques:** Here, we have to define the techniques to be used in order to achieve an advanced control mechanism by enforcing some level of consistency in the constraint network and propagating domain refinements along the constraints. The crucial point is that the use of consistency techniques provides the ability of pruning the search space in an active, *a priori* manner. Values that are known to be inconsistent with the current variable states can be excluded from further consideration.

In the following sections we will discuss these issues in some more detail. Although most of our considerations will we presented quite informally, we shall also provide a formal definition of the consistency techniques that we discuss.[2]

[2]As a thorough introduction into logic programming is outside the scope of this thesis, the reader is expected to be familiar with the basic concepts of logic programming, the declarative and operational semantics of PROLOG, and also with some basic notations that are used in a standard meaning as can be found e.g. in [Lloyd, 1987] or [Van Hentenryck, 1989].

6.1.3 Finite domains in logic programming

In order to use FCSP solving techniques within logic programming, the variables to be handled by these techniques have to be assigned a finite domain of possible values. Domains in logic programming can be defined as non-empty sets of constants. Thus, the domain of usual *logical variables* is the Herbrand universe of the theory that is represented by the given logic program (cf. Section 2.2). Additionally, some variables are assigned their own finite domains which in turn are subsets of the Herbrand universe. These variables are called *domain-variables* [Van Hentenryck, 1989].

During SLD resolution logical variables can be bound to any term, including constants and arbitrary variables. A domain-variable X ranging over a finite domain D_X, however, can only be bound to (1) values $v \in D_X$ of their domain, or (2) to a domain-variable ranging over some domain $\hat{D_X} \subset D_X$. As soon as the domain of a domain-variable becomes a singleton set, the variable is bound to the single value in its domain and thus becomes a logical variable.

Domain-variables can also take part in unification with logical variables or other domain-variables. Hence, the following three additional cases have to be taken into consideration:

- unification of a logical variable with a domain-variable. In that case, the logical variable is bound to the domain-variable.
- unification of a domain variable with a constant. If the constant is member of the domain of the variable, unification succeeds and the domain-variable is bound to the constant. Otherwise, unification fails.
- unification of two domain variables. If the intersection of the two domains is not empty, unification succeeds and binds both variables to a new one ranging over that intersection. If the intersection is a singleton set, both variables are bound to the remaining constant. If the intersection is the empty set, unification fails.

Thus, domain variables can be embedded into logic programming, enforcing some extensions but preserving the main results such as the declarative and procedural semantics of the language. By extending unification, an active handling of the equality constraint is provided. Yet, in order to maintain other constraints that way, additional inference rules have to be defined, embodying the consistency techniques presented in this thesis.

6.1.4 Constraints in logic programming

A constraint in the framework of logic programming is characterized by the fact that the inference rules defined in the next section can be applied to it. Therefore, it must be decidable, i.e. fulfill the following conditions:

Definition 32 (constraint in logic programming) *An n-ary predicate* `p` *is a* constraint *iff for any ground terms t_1, …, t_n one of the following is true:* `p`*(t_1,…,t_n) has a successful refutation, or* `p`*(t_1,…,t_n) has only finitely failed derivations.*

In other words, a predicate `p` is a constraint, if all its ground instances either succeed or finitely fail.

Consistency techniques can be embedded in logic programming, e.g. SLD resolution, by defining additional inference rules to be applied. In the following sections, we will review the forward-checking and looking-ahead consistency techniques and also present the corresponding inference rules.

6.1.5 Forward-checking

The idea of forward-checking is formally expressed by the forward-checking inference rule (FCIR). Informally, a constraint $\mathcal{C}$ can be used in a forward-checking manner as soon as all except one, say X, of its domain-variable arguments are instantiated to a ground value. Then, $\mathcal{C}$ is called *forward-checkable*. $\mathcal{C}$ can be considered a unary predicate $\mathcal{C}'(X)$, and the set of possible values that can be given to X can be restricted to those elements a satisfying $\mathcal{C}'(a)$.

For example, let $\mathcal{C}(X,Y,Z)$ be X `\=` $Y+Z$, with $X=4$, $Z=1$ and Y ranging over $\{1,2,3\}$. Then $\mathcal{C}'(V) \equiv V$ `\=` 3. Thus, the domain of Y can be restricted to the set $\{1,2\}$. Furthermore, if the domain of a variable becomes singleton, the variable is instantiated to the singleton value. Thus, other constraints can become forward-checkable, keeping constraint propagation going on.

Before defining the FCIR, we shall first formally define when the FCIR can be applied to a constraint.

Definition 33 (forward-checkable) *Be* `p`$(t_1,\ldots,t_n)$ *an atom. Then* `p`$(t_1,\ldots,t_n)$ *is* forward-checkable, *if* `p` *is a constraint and there exists only one* t_i, $1 \leq i \leq n$, *that is a domain variable, all others being ground.*

t_i is often referred to as the forward variable of `p`. Now, we can define the forward-checking inference rule as follows:

Definition 34 (forward-checking inference rule, FCIR) *Let P be a program, and let $G_i =$?$-\, A_1,\ldots,A_k,\ldots,A_m$ be a goal and σ_{i+1} a substitution. Then the next goal G_{i+1} (resolvent) is derived by the FCIR from G_i, P, and σ_{i+1}, if the following holds:*

1. *A_k is forward-checkable, X be the forward variable inside A_k.*
2. *The new domain of X, $\hat{\Delta}_X$, is defined as $\hat{\Delta}_X = \{v \in \Delta_X \mid P \models A_k\{X \leftarrow v\}\}$.*
3. *The new substitution σ_{i+1} is defined as $\sigma_{i+1} = \{X \leftarrow c\}$, if $\Delta_{\hat{X}} = \{c\}$, or $\sigma_{i+1} = \{X \leftarrow \bar{X}\}$, where $\bar{X}$ is a new domain-variable with $D_{\bar{X}} = \hat{\Delta}_X$, otherwise.*
4. *Finally, the new goal is $G_{i+1} = ? - \sigma_{i+1}(A_1,\ldots,A_{k-1},A_{k+1},\ldots,A_m)$.*

Forward-checking has turned out to be one of the most popular consistency techniques [Van Hentenryck and Dincbas, 1988] for several reasons:

- Forward-checking is a technique which can be easily implemented. For example, [De Schreye *et al.*, 1990, Müller, 1991] use PROLOG systems with coroutining facilities to implement forward-checking, whereas [Hein, 1992, Stein, 1993] show how forward-checking can be integrated into PROLOG by adding a set of new WAM instructions.
- It yields reasonable pruning results for many applications, keeping the computational costs fairly low.
- There exist sound and complete proof procedures based on normal SLD resolution combined with forward-checking (see [Van Hentenryck, 1989]).

The main drawback of forward-checking is its strong applicability precondition: A predicate can be executed by the FCIR only if all except one of its variables are instantiated to a ground value. Thus:

- For predicates with many arguments and/or many variables, at a given point of computation, there is only a relatively small probability that forward-checking can be applied to them.
- Especially when computation starts, it is very often the case that no constraint is forward-checkable. That means that choices have to be made, i.e. variables are instantiated in a more or less random manner. Thus, no pruning is achieved and further processing will mainly be done as backtracking search.
- Some constraints, such as $=, >, <$ should not be executed by forward-checking at all, because they embody a great deal of structural information about the relation between their arguments. For example, the information that two variables X and Y are equal should not only be used if X or Y are ground. Rather, the equality constraint should be maintained from the moment it has been stated (see [Müller, 1991]).

However, although not really sufficient but as it can easily be implemented, nearly all finite domain logic programming extensions provide forward-checking as a basic consistency technique which is then often supplemented by some variant of the looking-ahead technique that is presented in the next section.

6.1.6 Looking-ahead

Looking-Ahead [Van Hentenryck, 1987] is a variant of arc-consistency checking [Mackworth, 1977] as discussed in Chapters 2 and 3 (see Definition 19 on page 38) and offers a powerful possibility to reduce the number of values that can be assigned to the variables of a constraint, even if this constraint is not yet forward-checkable. A constraint $\mathcal{C}$ is said to be *lookahead-checkable* if at least one of its variables is a domain variable, while all other variables are either ground or domain variables. Note that this applicability condition is much weaker than the one for forward-checking.

Informally, the looking-ahead procedure can be described as follows: for every domain variable X appearing as an argument of an n-ary constraint $\mathcal{C}$, and for every value within the domain of X, it must be checked whether there exists at least one admissible value from the domain of each domain variable Y appearing in $\mathcal{C}$ so that the constraint $\mathcal{C}$ is satisfied. As mentioned above, the arguments of $\mathcal{C}$ which are no domain variables must be ground.

For example, let $\mathcal{C}(X,Y,Z)$ be $X > Y + Z$, where X, Y, and Z range over $\{1,2,3,4\}$. Using looking-ahead, the domains can be immediately restricted to $X = \{3,4\}, Y = Z = \{1,2\}$. Note that if we used forward-checking instead, no pruning at all would be achieved since no forward-condition would be fulfilled. From now on, each time a value is removed from one of the domains, looking-ahead has to be repeated.

Again, we will give a more formal definition of the looking-ahead inference rule (LAIR). By that rule, constraints can be used, even if more than one variable is left uninstantiated. Thus, the LAIR generally leads to an earlier pruning of the search space than the FCIR.

Definition 35 (lookahead-checkable) *An atom* $\mathtt{p}(t_1,\ldots,t_n)$ *is lookahead-checkable if* $\mathtt{p}$ *is a constraint and there exists at least one* t_i *that is a domain-variable. All other* t_j *are either ground or domain variables.*

The domain variables of a lookahead-checkable constraint are referred to as *lookahead-variables.*

Definition 36 (lookahead inference rule, LAIR) *Let* P *be a program,* $G_i = ?- A_1, \ldots, A_k, \ldots, A_m$ *be a goal and* σ_{i+1} *a substitution.* G_{i+1} *is derived by the LAIR from* G_i*,* P*, and* σ_{i+1} *if the following holds:*

1. A_k *is lookahead-checkable,* $X_1,\ldots,X_n$ *are the lookahead variables of* A_k.
2. *For each* X_j*, the new domain* $\hat{\Delta}_{X_j} = \{v_j{\in}\Delta_{X_j} \mid \exists v_1{\in}\Delta_{X_1}, \ldots, v_{j-1}{\in}\Delta_{X_{j-1}}, v_{j+1}{\in}\Delta_{X_{j+1}}, \ldots, v_n{\in}\Delta_{X_n}\text{: } P \models \sigma(A_k) \text{ with } \sigma{=}\{X_1{\leftarrow}v_1, \ldots, X_n{\leftarrow}v_n\}\ \}$.
3. *For each* X_j*,* $\hat{X}_j$ *is bound to the constant* c *if* $\hat{\Delta}_{X_j} = \{c\}$ *or a new domain variable which ranges over* $\hat{\Delta}_{X_j}$*, otherwise.*
4. $\sigma_{i+1} = \{X_1{\leftarrow}y_1,\ldots,X_n{\leftarrow}y_n\}$.
5. G_{i+1} *is either* $? - \sigma_{i+1}(A_1,\ldots,A_{k-1},A_{k+1},\ldots,A_m)$ *if at most one* $\hat{X}_j$ *is a domain variable, or* $? - \sigma_{i+1}(A_1,\ldots,A_m)$*, otherwise.*

By using looking-ahead, the search space can be pruned at an early stage of computation. However, standard looking-ahead is a very expensive method of ensuring arc-consistency. Therefore, for most applications it is considered inappropriate [De Schreye *et al.*, 1990, Dechter, 1989].

This leads to the question on how much effort should be put on consistency enforcing, i.e. constraint propagation, and what should be left for standard inference, i.e. backtracking search. This issue will be discussed in the next section.

6.2 How much Constraint Propagation is useful?

The effort to be spent on solving a constraint problem always splits into one part spent on problem reduction and another part spent on searching for a solution in the reduced problem space. This is trivially true for the two extremes:

- **Generate-and-Test** (GT) as discussed in Section 2.3.1 does not perform any problem reduction at all. Thus, the total costs for solving a constraint problem using generate-and-test are made up by searching for a solution.
- For **Backtrack-free Search** (BFS) the problem is as much reduced as needed to avoid backtracking at all. In general, this requires establishing n-consistency for a constraint problem with n variables, unless special characteristics of the particular problem can be explored as we have discussed in Section 3.1.3. Thus, backtrack-free search puts maximum effort on enforcing an appropriate level of consistency such that no search will be needed at all.

Between these two extremes a broad range opens for various combinations of problem-reduction and search. Figure 6.2 shows how some of the techniques presented in this thesis fit into this range.

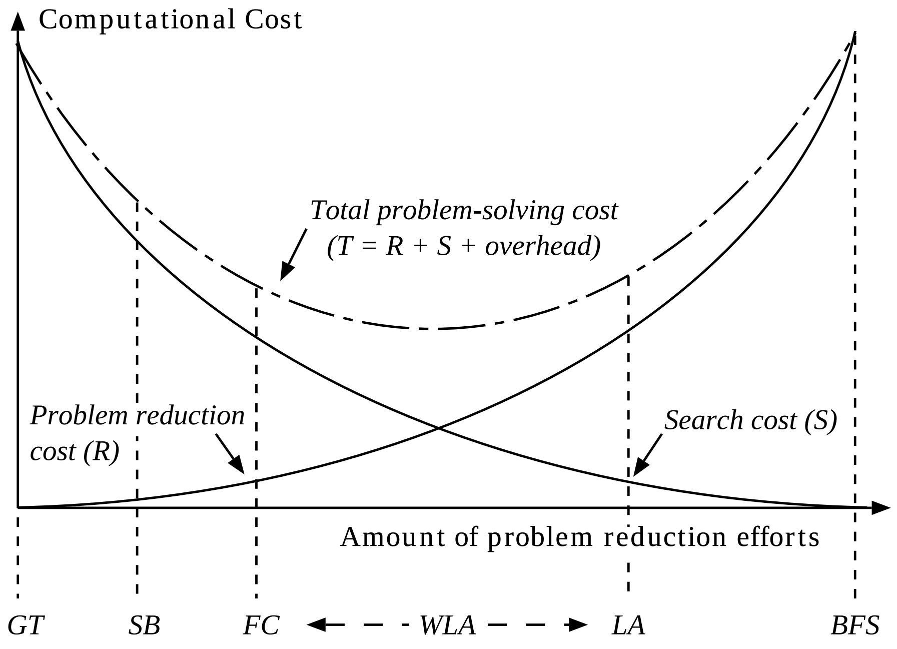


Figure 6.2: Problem-Reduction Costs vs Search Costs

The figure also shows a third curve representing the total problem-solving costs which are normally higher than the sum of the costs for problem-reduction and search. The reason is that except for the extremes some effort has to be made for organizing the interplay of problem-reduction, e.g. consistency enforcement, and backtracking search.

- The **Standard Backtracking** (SB) approach has been discussed in Section 2.3.1. It already performs some amount of problem reduction because partial solutions that violate some constraints among the past variables are discarded and thus the corresponding subtree in the search space is pruned. As a result, the cost for search is reduced if compared with the generate-and-test approach.

- **Forward-Checking** (FC) does also check *forward* in the sense that consistency enforcement is extended to future variables as soon as the constraint becomes forward-checkable. In that case, domains can be refined which results in a reduced problem with smaller search space left. Note also that the repeated test of the forward-checking condition amounts to the overhead cost mentioned in Figure 6.2.

- Finally, the **Looking-Ahead** (LA) technique performs much more consistency enforcement at each time a lookahead-checkable goal is selected from the current resolvent. It basically enforces arc-consistency on the corresponding constraint which can result in a significant pruning of the search space, thus reducing the overall search cost. However, checking arc-consistency at each time a constraint becomes lookahead-checkable is much more expensive than forward-checking the constraint and thus drastically increases the problem-reduction costs.

As we are interested in minimizing the overall cost of solving a constraint problem, Figure 6.2 suggests to search for some compromise between forward-checking and looking-ahead. In Section 6.3, we will present *weak looking-ahead*, which can be regarded as such a compromise between forward-checking and looking-ahead, and thus helps to obtain an early search-space pruning while avoiding the high costs for arc-consistency checking involved by standard looking-ahead: Let us assume that, in our above looking-ahead example, we would perform the first looking-ahead step as shown, but after that, we would *not* do any more looking-ahead, but instead solve the (now simplified) problem by normal resolution or by forward-checking. This procedure expresses the main idea of the weak looking-ahead strategy which we will point out in more detail in the following.

6.3 Weak Looking-Ahead

The weak looking-ahead (WLA) strategy combines the use of LAIR and FCIR. A similar technique has been informally proposed in [De Schreye *et al.*, 1990] as "first-order looking-ahead". We present a generalized technique that we call weak looking-ahead. This name seems more appropriate for expressing what the underlying algorithm really does. The basic idea of WLA is that each constraint can be selected by the looking-ahead part *not more than once*, and that this should happen at an appropriate time. After this, only the FCIR (or normal inference) can be applied to it. This idea is covered by the following definitions.

Definition 37 (WLA-checkable) *An atom $p(t_1, ..., t_n)$ is called* WLA-checkable *if p is a constraint and*

- *$p(t_1, ..., t_n)$ is lookahead-checkable and has not yet been selected by the WLA, or*
- *$p(t_1, ..., t_n)$ is forward-checkable and has already been selected by the WLA.*

Definition 38 (weak looking-ahead) *Let P be a program, $G_i = \ ?{-}A_1, ..., A_k, ..., A_m$ a goal and σ_{i+1} a substitution. G_{i+1} is derived by the WLA along with σ_{i+1} from G_i and P if A_k is WLA-checkable with $X_1, ..., X_n$ being the WLA variables in A_k and the following holds:*

- *If A_k is lookahead-checkable and the WLA has not been applied to A_k in the actual proof, then:*

  For each X_j, the new domain is $\hat{\Delta}_{X_j} = \{v_j{\in}\Delta_j \mid \exists v_1{\in}\Delta_{X_1}, \ ..., \ v_{j-1}{\in}\Delta_{X_{j-1}}, v_{j+1}{\in}\Delta_{X_{j+1}}, \ ..., \ v_n{\in}\Delta_{X_n}\colon P \models \sigma(A_k)$ with $\sigma = \{X_1{\leftarrow}v_1, \ ..., \ X_n{\leftarrow}v_n\}\}$.

  For each x_j, if $\hat{\Delta}_{X_j}$ has become a singleton set, i.e. $\hat{\Delta}_{X_j} = \{c\}$, then the new value y_j is the constant c, otherwise a new domain variable ranging over $\hat{\Delta}_{X_j}$. σ_{i+1} then is defined as $\sigma_{i+1} = \{X_1 \leftarrow y_1, ..., X_n \leftarrow y_n\}$.

  The new goal G_{i+1} is either ?- $\sigma_{i+1}(A_1, ..., A_{k-1}, A_{k+1}, ..., A_m)$, if at most one y_j is a domain variable, or G_{i+1} is ?-$\sigma_{i+1}(A_1, ..., A_m)$, otherwise.

- *If A_k is forward-checkable, then:*

  Let X be the forward variable inside A_k. Then, the new domain $\hat{\Delta}_X$ is defined as $\hat{\Delta}_X = \{a \in \Delta_X \mid P \models A_k\{X \leftarrow a\}\}$

  σ_{i+1} is defined as $\sigma_{i+1} = \{X \leftarrow c\}$, if $\hat{\Delta}_X = \{c\}$, i.e. $\hat{\Delta}_X$ has become a singleton. Otherwise, $\sigma_{i+1} = \{X \leftarrow X_d\}$, where X_d is a new domain variable ranging over $\hat{\Delta}_X$.

  The new goal G_{i+1} is ?- $\sigma_{i+1}(A_1, ..., A_{k-1}, A_{k+1}, ..., A_m)$.

The main point of the above definition is the definition of G_{i+1} in case of A_k being lookahead-checkable, which uses SLDFC resolution (SLD resolution with forward-checking, cf. [Van Hentenryck, 1989]) in order to complete the proof after some prepruning has been done by using the LAIR in a definite way.

Since the LAIR part gets involved not more than once for each goal, and since this happens as early as possible, the disadvantages of the LAIR, i.e. the high computational overhead, can be avoided. Therefore, weak looking-ahead offers an alternative to standard looking-ahead where an initial pruning of the search space is desired in order to early achieve instantiations such that for further constraint propagation, forward-checking will suffice. However, in the FiDo system that will be presented in Section 6.5 all three consistency techniques presented in this chapter have been implemented.

So far, we have now more formally presented the weak looking-ahead technique that combines the pruning power of looking-ahead (once at the first time a constraint is activated) with the computational efficiency of forward-checking (for further checking at each time a variable gets instantiated). In the next section, we will illustrate how weak looking-ahead really works by using a variant of our running example.

6.4 Using Weak Looking-Ahead for Map-Coloring

For illustrating the WLA technique, we take the following variant of the German-map coloring problem that builds on the initial German-map coloring problem but also includes the eastern states:

Problem 4 (Weak german-map coloring problem) *The weak german-map coloring problem is to color the political map of Germany as shown in Figure 4.1 on page 70. As in the initial problem statement, the following colors have already been fixed: Hessen should be colored* green *while Hamburg, Bremen and Saarland should be* red. *Besides* red *and* green *which are allowed for all states, the (old) western states may be colored* blue *while the (new) eastern states take* yellow *as their third possible color.*

According to the representation proposed in Chapter 2, the weak map-coloring problem can be represented and solved by the following logic program enhanced with declarations for domain variables and consistency techniques to be applied:

```
weak_coloring(TH,BB,BW,SL,NW,HB,SH,BL,SN,SA,MV,BY,RP,HS,NS,HH) :-
   domvars([BW,SL,NW,HB,SH,BY,RP,HS,NS,HH],[r,g,b]),
   domvars([TH,BB,BL,SN,SA,MV],[r,g,y]),
   wla(ne(SH,HH)), wla(ne(SH,MV)), wla(ne(NS,HH)), wla(ne(NS,SH)),
   wla(ne(NS,MV)), wla(ne(NS,SA)), wla(ne(SA,MV)), wla(ne(NS,HB)),
   wla(ne(NS,NW)), wla(ne(SA,BB)), wla(ne(BB,MV)), wla(ne(BB,BL)),
   wla(ne(TH,SA)), wla(ne(TH,NS)), wla(ne(HS,TH)), wla(ne(HS,NW)),
   wla(ne(HS,NS)), wla(ne(SN,BB)), wla(ne(SN,SA)), wla(ne(SN,TH)),
   wla(ne(BY,SN)), wla(ne(BY,TH)), wla(ne(BW,BY)), wla(ne(RP,BW)),
   wla(ne(RP,HS)), wla(ne(BW,HS)), wla(ne(BY,HS)), wla(ne(RP,NW)),
   wla(ne(RP,SL)),
   instantiate([TH,BB,BW,SL,NW,HB,SH,BL,SN,SA,MV,BY,RP,HS,NS,HH]).
```

In the following, we will discuss in detail how weak looking-ahead processes the corresponding query

```
?- weak_coloring(TH,BB,BW,SL,NW,HB,SH,BL,SN,SA,MV,BY,RP,HS,NS,HH).
```

As we have mentioned before, the main difference between solving the map-coloring problem using CSP techniques like the GenerateSolutions algorithm and solving it using constraint logic programming techniques like weak looking-ahead is that in the first approach the complete finite constraint network is present when starting processing whereas in the latter constraints are generated and processed dynamically as they are selected from the resolvent in the SLD resolution.[3]

The first constraint to be activated is `ne(SH,HH)` where the domain of `HH`, $\Delta_{\texttt{HH}}$, is already restricted to the singleton set {`r`}. Performing the initial looking-ahead step on `ne(SH,HH)` results in deleting the value `r` from $\Delta_{\texttt{SH}}$. Now, adding the next constraint `ne(SH,MV)` does not affect any domains of other constraints. Thus, after activating the

[3]Therefore, the latter approach can also be used for dealing with potentially infinte constraint problems.

first two constraints `ne(SH,HH)` and `ne(SH,MV)` we obtain the following constraint network as shown in Figure 6.3.

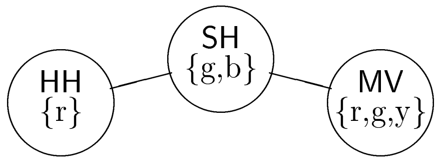


Figure 6.3: The map-coloring network after activating the first two constraints

Next, the constraint `ne(NS,HH)` is added which causes the deletion of the value {`r`} from the domain of `NS` analogously to `SH` in the first step. The following activation of the constraints `ne(NS,SH)`, `ne(NS,MV)`, `ne(NS,SA)`, and `ne(SA,MV)` again have no effect on any other domains, and thus we obtain the following constraint network (Figure 6.4).

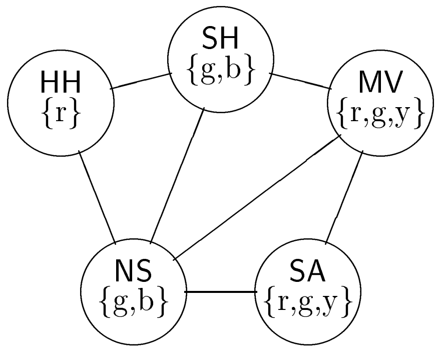


Figure 6.4: The map-coloring network after activating the first 7 constraints

Activating the next five constraints `ne(NS,HB)`, `ne(NS,NW)`, `ne(SA,BB)`, `ne(BB,MV)`, and `ne(BB,BL)` also do not affect any domains; the value `r` was already deleted from the domain of `NS`. Hence, we have the following constraint network after activating the first 12 constraints (Figure 6.5).

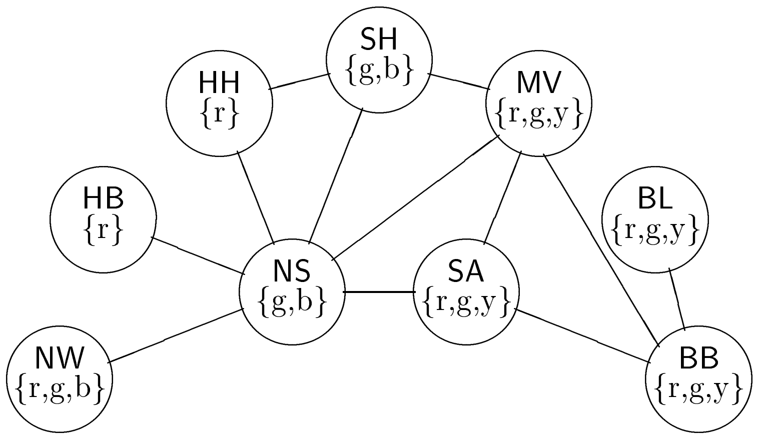


Figure 6.5: The map-coloring network after activating the first 12 constraints

Now, the constraints `ne(TH,SA)` and `ne(TH,NS)` are activated but cause no further restrictions. Then, the constraint `ne(HS,TH)` is added. As the domain of the variable `HS` is already restricted to {`g`}, the value `g` will be deleted from $\Delta_{\texttt{TH}}$. Next, the constraint `ne(HS,NW)` is added to the constraint network which analogously causes the value `g` to be deleted from domain of `NW` which is thus refined to $\Delta_{\texttt{NW}} = \{\texttt{r,b}\}$. At this stage we obtain the following constraint network (Figure 6.6).

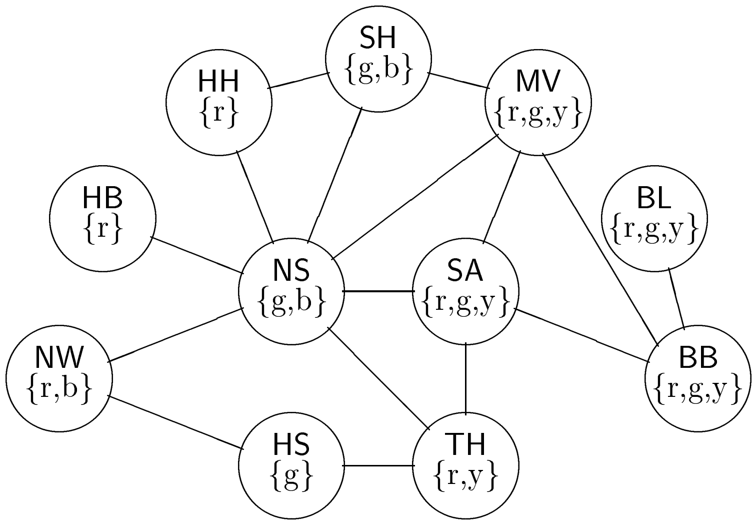


Figure 6.6: The map-coloring network after activating the first 16 constraints

When adding the constraint `ne(HS,NS)`, the looking-ahead step performed for this constraint deletes the value `g` from the domain of `NS`. Thereby, $\Delta_{\texttt{NS}}$ becomes the singleton set {`b`}, thus `NS` gets instantiated to the definite value `b`. This 'wakes up' all constraints connected to `NS` to be forward-checked: Checking `ne(NS,SA)`, `ne(NS,MV)`, and `ne(NS,TH)` is unspectacular while checking `ne(NS,SH)` and `ne(NS,NW)` results in deleting the value `b` from $\Delta_{\texttt{SH}}$ and $\Delta_{\texttt{NW}}$, respectively. Thereby, the domains of `NW` and `SH` both become singleton sets causing forward-checking to be performed on all constraints connected to `NW` or `SH`. Thus, checking `ne(SH,MV)` results in deleting `g` from $\Delta_{\texttt{MV}}$. Therefore, we now have the following constraint network (Figure 6.7).

As processing continues, more constraints are added to the constraint network: The constraints `ne(SN,BB)`, `ne(SN,SA)`, `ne(SN,TH)`, `ne(BY,SN)`, `ne(BY,TH)`, `ne(BW,BY)` and `ne(RP,BW)` are added without affecting any other constraints; then the constraints connecting `HS` with `RP`, `BW`, and `BY`, respectively, are added. As $\Delta_{\texttt{HS}} = \{\texttt{g}\}$, the value `g` is deleted from the domains of `RP`, `BW`, and `BY`, but without any further consequences to other constraints. Thus, we obtain the following, nearly complete, constraint network (Figure 6.8).

When finally adding the remaining two constraints `ne(RP,NW)` and `ne(RP,SL)`, the value `r` is deleted from the domain of `RP` due to $\Delta_{\texttt{NW}}$ respectively $\Delta_{\texttt{SL}}$ being the singleton set {`r`}. Thus, $\Delta_{\texttt{RP}}$ becomes {`b`}. This domain instantiation, however, forces the constraints `ne(RP,BW)` to be checked again which in turn results in deleting the value `b` from $\Delta_{\texttt{BW}}$, again reducing the domain of `BW` to a singleton set and thus instantiating `BW` to `r`. Thereby, the constraint `ne(BW,BY)` is 'waked up' resulting in reducing $\Delta_{\texttt{BY}}$ to {`b`}. Now,

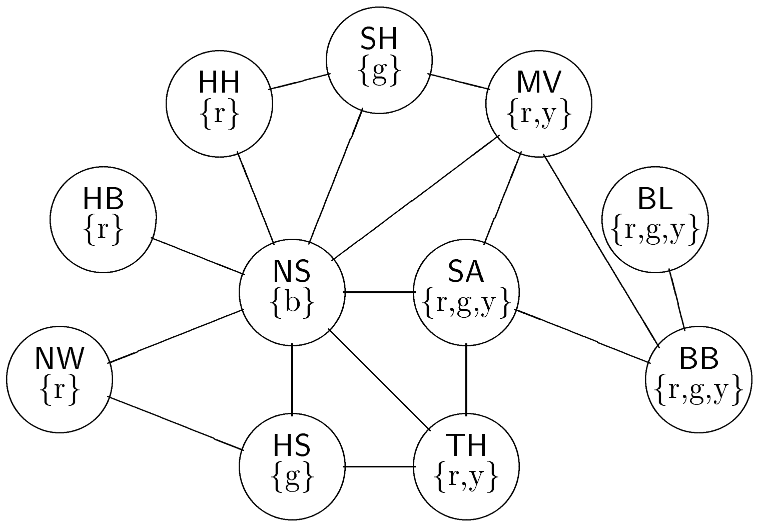


Figure 6.7: The map-coloring network after activating the first 17 constraints

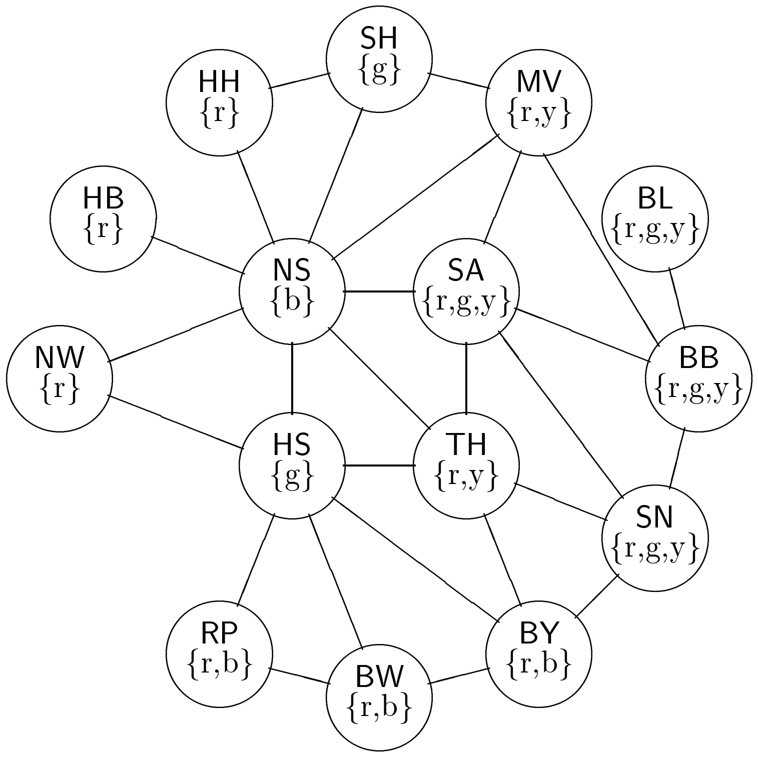


Figure 6.8: The map-coloring network after activating the first 27 constraints

this instantiation finally 'wakes up' the constraints `ne(BY,TH)` and `ne(BY,SN)` which are checked successfully without any further domain restrictions. Thus, all constraints of the map-coloring problem are now generated and we obtain the following constraint network (Figure 6.9).

One can easily recognize that all variables representing the colors for western states have already been instantiated whereas the domains of the remaining six variables for the

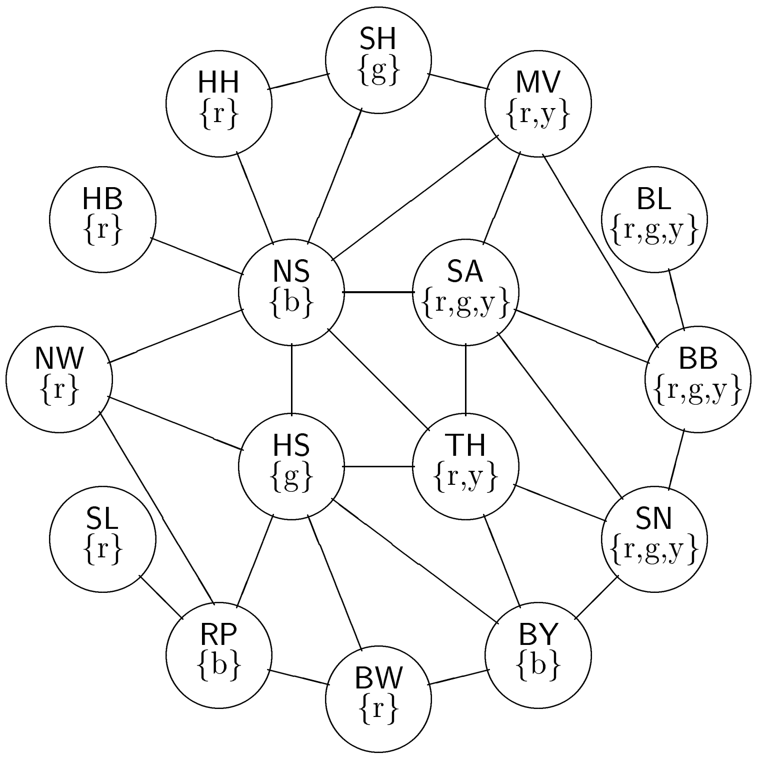


Figure 6.9: The complete map-coloring network after initialization

eastern states are nearly the same as in the beginning. This again illustrates that local consistency techniques like arc-consistency or looking-ahead may not suffice to completely solve a constraint problem. To finally obtain a solution, choices have to be made and further propagated such that all variables get instantiated. However, if only one or a limited number of solutions is required, the efficiency of finding them heavily depends on the order in which choices are made, i.e., which variable to select next and which value to try for it.

Therefore, constraint logic programming systems like the FiDo system that will be presented in the next section, provide `instantiate` primitives that make choices in order to achieve instantiations to all variables. As mentioned earlier, several heuristics can be applied for selecting the variable which will be instantiated next. Best results have been obtained using the remaining domain size or the number of related constraints as a criterion for variable selection. The basic idea here is to obtain failures as early as possible as then most parts of unsuccesfull search can be avoided. Hence, these heuristics are also referred to as *first-fail* heuristics.

In our concrete example the variable `TH` is selected to be instantiated next as its domain size is 2 and it is related to 5 other constraints. Thus, `TH` gets instantiated to a value from its domain, say `r`. Thereby, the constraints `ne(SA,TH)` and `ne(SN,TH)` become forward-checkable and the value `r` gets deleted from $\Delta_{\texttt{SN}}$ and $\Delta_{\texttt{SA}}$ and propagation stops still not obtaining a complete instantiation. Thus, another choice has to be made, say `y` for `SA` which results in a failure when the constraint `ne(MV,BB)` is waked up. Therefore, the system has to backtrack for the first time and tries another alternative value for `SA` which is to instantiate SA to g. As a consequence, g is deleted from Δ_{SN} which then becomes instantiated to y. Hence, g and y are deleted from Δ_{BB} which then is also instantiated to a single value, r. Finally, MV will also be instantiated to y and propagation stops. AS there is still one domain variable left (BL) a final choice has to be made and both choices, g and b for BL appear to be consistent; thus we obtain the first two solutions from the final constraint network (Figure 6.10).

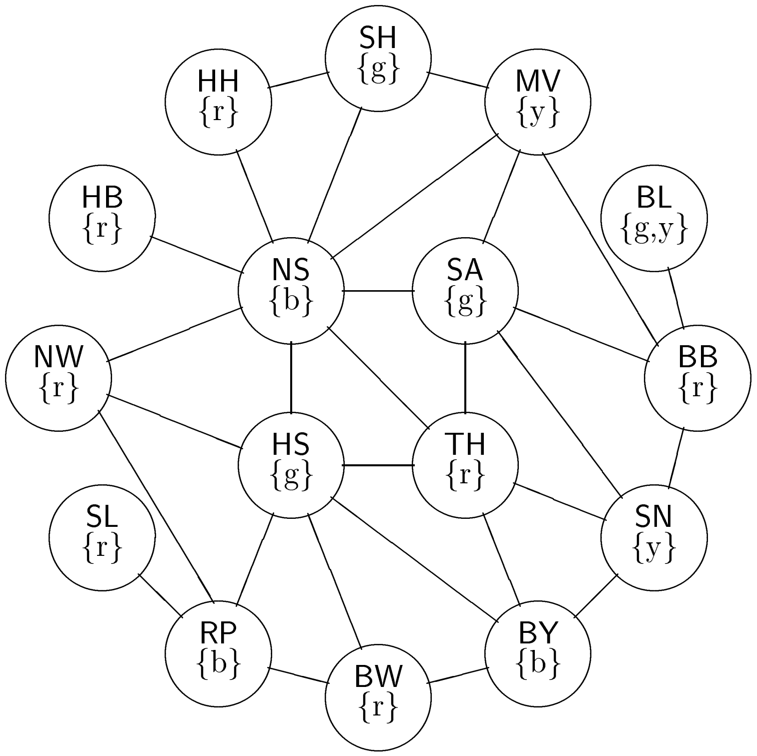


Figure 6.10: The final map-coloring network after propagating choices

If more than two solutions are required, backtracking will generate other choices and finally obtain two more solutions to the weak map-coloring problem.

This exhaustive trace of weak looking-ahead processing should have made clear the principle of how WLA processes a constraint problem within the framework of logic programming. In the next section, we will now discuss various approaches towards implementing consistency techniques like forward-checking and weak looking-ahead in a logic programming language in order to obtain a **Fi**nite **Do**main extension to Prolog.

6.5 Finite Domain Consistency Techniques in FiDo

In the FiDo framework, different approaches towards an integration of **FI**nite **DO**main Consistency Techniques in Logic Programming are investigated. The FiDo logic programming language extends Prolog by supporting special mechanisms for efficiently dealing with constraints over finite domains.

Starting with the implementation of a meta-interpreter for FiDo written in Prolog ([Schrödl, 1991]), we subsequently investigated the more sophisticated approach of horizontally compiling FiDo programs into SEPIA, a Prolog system developed at ECRC [Meier *et al.*, 1989], by making use of the built-in coroutining mechanism ([Müller, 1991]). When working on this horizontal compilation approach, it became clear that the integration and handling of domain variables is the major issue concerning runtime efficiency and implementation complexity. Therefore, we then studied a vertical compilation approach ([Hein, 1992]) compiling finite domain constraints vertically into a WAM architecture by extending the basic WAM data structures [Warren, 1983] and using a *freeze*-like control scheme [Carlsson, 1987].

6.5.1 The Meta-Interpretation Approach

When building finite domain extensions for Prolog by constructing a meta-interpreter, one has to implement his own meta-unification routine working on an explicit representation of meta-variables. Domain variables can be implemented by simply extending the terms representing meta-variables by some additional arguments holding the domain, the goals depending on this variable, and pointers to variables and constants which have to be checked for inequality with this domain variable.

In this approach, which is similar to that presented in [Holzbaur, 1990], we have studied two ways of representing the domains: bit vectors and ordered linked-lists. For the actual implementation, we only use the bit vector domain representation.

As in general with meta-interpretation, the main disadvantage of this approach is its lack of efficiency resulting from the necessity to do a lot of work on the meta-level which could be done far more efficiently by the underlying Prolog system.

6.5.2 The Horizontal Compilation Approach

In the horizontal compilation approach we study the compilation of FiDo programs which are basically Prolog programs together with domain and control declarations (e.g., forward-checking, looking-ahead, and weak looking-ahead) into a standard Prolog program using `delay` clauses. Thus, this method can be considered as a source-to-source transformation which has been proposed in [De Schreye *et al.*, 1990].

An important feature is the explicit handling of finite domains. A domain variable X is internally represented as a 6-tuple $(\&, X_{id}, X_{length}, X_{\#Constraints}, X_{value}, X_{domain})$. The first argument is a flag used to identify domain variables. X_{id} denotes a unique domain identifier, X_{value} contains the singleton value of X, X_{domain} unifies with an explicit copy of the domain, and X_{length} is the actual length of the domain. This is needed for an efficient implementation of the singleton test and for the usage of first-fail heuristics. The number of constraints, denoted by $X_{\#Constraints}$ is needed for first-fail as well. If these heuristics are not used, X_{length} and $X_{\#Constraints}$ are ignored.

A domain $\{el_1, \ldots, el_n\}$ is internally represented as a term `dom((`el_1`,_),...,(`el_n`,_))`, where the anonymous variables serve as flags. When a domain element is discarded from the domain, its flag is instantiated to "`0`". The advantage of this representation then is a direct access to the domain elements. The calls to predicates declared to be handled by some consistency technique are replaced by calls to redefined versions of the predicates. These redefinitions take into account the various instantiation possibilities, e.g. the $\neq$ constraint under forward-checking is compiled into `for_ne_dd`, `for_ne_nd`, `for_ne_dn` and `for_ne_nn` respectively, handling domain (d) and non-domain-variable (n) arguments. The effective calls to the forward-checking or looking-ahead algorithms are then performed within the specialized constraints, e.g. in `for_ne_dd`. For some frequently used built-in constraints, such as $\neq$, $=$, $<$ and $>$, efficient specialized versions of forward-checking algorithms are available. For other user-defined constraints, general forward-checking and looking-ahead algorithms are used.

At the first glance, this approach has some similarities with the one described in [De Schreye *et al.*, 1990]. However, there are some differences concerning the internal domain representations which give better performance. Moreover, the necessary declarations are created automatically in FiDo while in [De Schreye *et al.*, 1990] these statements have to be given by the programmer. The design used for domain declarations in FiDo therefore provides more declarativity and easier readability of programs.

In the horizontal compilation approach, FiDo supports local `forward`, `lookahead`, and `wla` declarations. This is opposed to systems like CHIP [Dincbas and Van Hentenryck, 1988] that use global declarations. We have chosen local declarations, because in some cases it might be useful to execute one constraint in different environments differently, say in one procedure call by using *forward-checking*, in another call by using a *looking-ahead* algorithm. Using global declarations this leads to serious problems, while using local declarations it can be achieved very easily, since not predicates are subject to declarations but predicate calls are.

6.5.3 The Vertical Compilation Approach

Meta-interpretation is a straightforward implementational technique — but rather slow. The more elaborated horizontal compilation technique was based on a delay mechanism and on a domain variable representation using structures in which an unbound variable holds successive bindings of the domains. Thus, some kind of dereferencing operation had to be implemented on top of SEPIA resulting in the problem that the more often the domain is rebound to a smaller domain, the more dereferencing steps have to be done. This basically is the reason why the runtime efficiency gained by the horizontal compilation approach decreases with the problem complexity.

Therefore, in order to be able to represent and solve the real-world applications we have in mind, we investigated how to compile FiDo programs directly (*vertically*) into code for an abstract machine. As the Warren Abstract Machine (WAM) is the model commonly used for Prolog compilation, we study how to extend this model for dealing with finite domain constraints. The two major ingredients for finite domain constraints in the WAM are a mechanism for the control strategies (forward-checking, looking-ahead, and weak looking-ahead) and an extension of the WAM unification algorithm coping with domain unification and a set of domain-variable specific WAM instructions.

Compiling FiDo programs

Currently, the WAM emulator we use for FiDo is implemented in LISP, thus hiding a lot of implementation details, e.g. the efficient representation of domains. FiDo programs are compiled into a "normal form" [Hein and Meyer, 1992] where additional information obtained by static analysis can be used. The normal form consists of very few constructs and can be easily transformed into a WAM-based language. The control instructions of this machine are similar to the original WAM. However, information from static analysis mainly affects the unification represented in WAM by the `get/put/unify`-instruction sets. We are compiling into LISP code which is put into arrays and compiled by the LISP compiler. Thus, the "dirty" task of compiler optimizing techniques is left to the underlying LISP system. However, we are able to do the unification optimizations on the abstract "normal form" level which are normally done on machine level.

Using results from static analysis

Looking at the runtime behavior of usual WAM code, we can see that the ordinary WAM instructions often do unnecessary work. Dereferencing is a good example, e.g. a `switch` instruction dereferences its argument and jumps to the corresponding code — normally a `get` instruction — which again dereferences its argument. On the WAM level, there is no way to avoid this redundant operation.

However, results from static analysis [Tucker and Zucker, 1992] can help to avoid generating unnecessary code. One part of useless computation originates from applying a general unification algorithm. The bird's-eye view of the general unification procedure reveals a kind of a "case-statement" branching on the type of the two arguments. Often we do not need to call the general unification but better a specialized matching or assignment routine when the types are already known. Thus, type inferences [Debray and Warren, 1986, Mellish, 1985] avoid useless tests and memory usage by such dead code, e.g. unifying a variable known to be unbound with a constant results in a simple binding (in general, with trailing).

For each optimization feature (dereferencing, trailing) we could "duplicate" parts of the WAM instruction set. Taking several optimizations into account, an inflationary number of new instructions would result. This can be avoided by doing the optimizations on the abstract (normal form) level and then applying the WAM compilation scheme [Hein and Meyer, 1992].

The static analyzer also has to cope with special problems arising from integrating finite domain constraints. We must take into account not only the handling of finite domains but also the difficult delay and wake-up mechanism which has not been examined before.

The extended WAM – from freeze to FiDo WAM

The first WAM extension supporting the delay of parts of the computation (`freeze`) was presented by [Carlsson, 1987] where the representation of unbound variables is enhanced by the notion of variable suspension. In FiDo we also need representations for domain variables and suspended domain variables.

The proposed implementation of `freeze` indicates an exception by a flag in the WAM status as soon as a suspended variable is touched by some binding. The woken procedure (Prolog code) is responsible for freezing any new variables; furthermore every freeze operation builds a new data structure in the runtime area holding the state of the frozen goal. For FiDo this would be a problem since testing whether the procedure may fire can happen very often and would result in creating this data structure over and over again.

The code for building the constraint net and maintaining the delayed goals fits perfectly into the philosophy of WAM compilation, e.g. argument places can be compiled left to right. However, constraints are checked too often due to the simple waking mechanism. Therefore, in FiDo we split each constraint procedure into building code for the constraint net, the firing code and structures waking up the constraint *only* if the firing conditions are fulfilled (after a binding operation). Doing so, we finally get a much better runtime performance.

6.6 Discussion and Related Work

In this chapter, we have shown how consistency-improving techniques can be integrated into a universal logic programming language and how constraint processing can be interleaved with SLD resolution. The standard consistency techniques that are available in most other constraint logic programming systems like CHIP [Dincbas and Van Hentenryck, 1988] or clp(FD) [Diaz and Codognet, 1993] have been identified as either being too weak (forward-checking) or inducing too much computation overhead (looking-ahead). Therefore, we have presented the weak-looking-ahead technique which combines the pruning power of looking-ahead with the simplicity and efficiency of forward-checking. Using weak looking-ahead for solving combinatorial problems thus improves the efficiency of constraint solving within logic programming while preserving the declarativity of the purely relational representation formalism.

However, dealing with overconstrained problems which most real-life applications demand, is still an open research issue in the context of logic programming. Only recently, work on preferential Horn theories for integrating preferences into logic programs has appeared [Brown *et al.*, 1993] which opens an interesting direction for further research.

7

Parallel Constraint Satisfaction in Logic Programming

In the previous chapters, we have seen that consistency techniques embedded in a logic programming framework allow for solving discrete combinatorial problems efficiently. Nevertheless most of these problems still remain computationally expensive and the emergence of parallel computers opens a new area for research: the combination of constraint satisfaction and parallel execution in the framework of logic programming.

In this chapter[1], we present a first step in that direction. After a brief introduction to parallel logic programming issues in Section 7.1, we will then study in Section 7.2 how *or*-parallelism and consistency techniques on finite domains can be combined to yield an efficient parallel language for representing and solving finite domain constraint problems. In Section 7.3, some problems raised by an *or*-parallel version of FiDo will be discussed in detail and a solution to them will be described as well. The chapter then concludes with a comparison of this approach to the SRI Model for *or*-parallel Prolog.

7.1 Parallel Logic Programming

Much research has been devoted during the last years to improving the efficiency of logic programming for combinatorial problems. This has led to the definition of several constraint programming languages (e.g., [Colmerauer, 1987, Dincbas and Van Hentenryck, 1988, Jaffar and Lassez, 1987, Voda, 1988]) combining the declarative aspects of logic programming with the efficiency of constraint-solving techniques.

Concurrently to this research, much attention has been devoted to parallelizing Prolog. The motivation behind this research stems from the emergence of commercial multiprocessor computers and the declarative aspects of logic programming which, in principle, liberates programmers from managing parallelism. Different computation models have been proposed to exploit either *or*-parallelism (e.g., [Clocksin and Alshawi, 1988, Lusk et al., 1988, Shapiro, 1987, Warren, 1987]) either *and*-parallelism (e.g., [De Groot, 1984, Hermenegildo, 1986]) or a combination of both (e.g., [Westphal and Robert, 1987]).

[1]This chapter is a revised version of a paper that has been published in the proceedings of the *Parallel Computing Technologies* (PaCT-91) conference [Meyer, 1991].

Several of these models have been successfully implemented and show promising results [Chassin *et al.*, 1988, Lusk et al., 1988].

The results in these two different directions, together with the fact that most constrained search problems are computationally expensive, were a major impetus to investigate parallel implementations of constraint logic programming languages. In this context, finite domains and consistency techniques for them such as forward-checking and looking-ahead are of particular interest. They have a large variety of applications in discrete combinatorial problems and offer large potentiality for parallelization. Currently solving a discrete combinatorial problem in parallel requires either the writing of a specialized program in a procedural language (e.g., [Kumar *et al.*, 1988, Wah and Ma, 1984]) or the use of a special-purpose package [Finkel and Manber, 1987]. The first approach is programming-intensive; solving a discrete combinatorial problem in sequential requires much programming effort and parallelism further increases the programming complexity. Special-purpose packages relieve programmers from managing parallelism but they are, at the moment, rather inefficient. Compared to the above approaches, a parallel version of FiDo is very appealing. Not only it reduces the programming effort to the sequential part itself simplified by the use of FiDo but it should also result in a good efficiency provided that the overhead of parallelism is kept small. It is therefore of much interest to study if the sequential efficiency of FiDo can be preserved in a parallel implementation.

In this chapter we will discuss a first step in that direction: the design of an *or*-parallel version of FiDo. We will show that the principles behind the sequential implementation can be generalized to support *or*-parallelism while keeping the overhead small. In particular, basic operations on domains remain fast, constant-time operations.

In Section 7.2 we will identify different opportunities for parallelism in FiDo before in Section 7.3 we will then discuss how to exploit *or*-parallelism together with finite domains and consistency techniques.

7.2 What Parallelism to Exploit?

In this section, we review the main opportunities for parallelism arising in FiDo. There are mainly two kinds of parallelism inside logic programming: *and*- and *or*-parallelism. However, since the computational model of FiDo differs from the one of ordinary Prolog, it is appropriate to reconsider that issue. As discussed in Chapter 6, the computational model of FiDo for solving finite domain constraint problems is best viewed as the iteration of two steps: propagation and choices. Both steps provide opportunities for parallelism, three of them being considered now.

First, a constraint can be solved in parallel. This opportunity for parallelism arises inside the forward and (weak) lookahead declarations. Indeed, these control mechanisms solve a constraint by trying out different combinations of values for the variables appearing inside the constraint and these combinations could possibly been tried in parallel. This kind of parallelism is implicit in the bitwise forward checking version described in [Haralick and Elliott, 1980] and is expected to be of small granularity in most cases.

Second, the constraints can be propagated in parallel. During the propagation step, several constraints have to be considered and we might think of propagating them in parallel. This leads to a kind of *and*-parallelism. Once again this parallelism is expected to be of small granularity in most cases and to introduce some synchronization problems since several constraints might share the same variable. It has attracted several researchers in the AI community (e.g., [Swain and Cooper, 1988]).

Third, choices can be made in parallel. This is the usual *or*-parallelism of Prolog except that FiDo has much more to offer in that context than usual logic languages. Indeed, choices can take various forms from instantiation (i.e., giving a value to a variable), to domain-splitting (i.e., dividing the domain of a variable in several parts), and to case analysis (i.e., using constraints for making choices). In all cases, a choice means choosing among different alternatives and the basic idea here is to explore several of them simultaneously. This parallelism has attracted many researchers outside the logic programming community (e.g., [Finkel and Manber, 1987, Kumar *et al.*, 1988, Wah and Ma, 1984]) and can be of large granularity. Parallelism has also an interesting side-effect on branch and bound algorithms. It introduces either a depth-first component in a best-first branch and bound or a breadth-first component in a depth-first algorithm. Therefore anomalies (e.g., superlinear speedups) can be observed [Lai and Sahni, 1987]. For depth-first branch and bound algorithms like the FindOptimalSolution algorithm presented in Chapter 4, parallelism might allow to find early a good (or even optimal) solution that would otherwise require a long time. This solution can then be used to prune the search space, avoiding spending time in parts of the tree which are not worth exploring.

Among the above sources of parallelism, *or*-parallelism seems to be the most promising. It applies to the whole computation and not a specific part of it (e.g., the constraint propagation). It can also be of large granularity and thus more amenable to implementation on a multiprocessor computer. Finally, it amounts to adding a breadth-first component which can be valuable for the class of problems considered. Therefore we decided to focus our investigations on the combination of *or*-parallelism and constraint-solving.

7.3 Supporting *Or*-Parallelism in FiDo

In order to support *or*-parallelism, it is necessary to adapt the current implementation schemes of FiDo. This section reviews the problems raised by *or*-parallel FiDo and proposes several solutions to it.

7.3.1 The Problem

The computation of a logic language can be seen as a tree whose root node is the initial goal and other nodes are obtained by reducing a goal using SLD-resolution or an extension of it. Different branches of the tree come from the different clauses used to reduce a goal. Prolog explores the computation tree using a depth-first search with chronological backtracking. In other words, when PROLOG can apply several clauses to reduce a goal, one of them is selected. The remaining clauses will be tried on backtracking. In *or*-parallel PROLOG, different processors are used to explore several branches of the tree in parallel, i.e., they try simultaneously several clauses for reducing the same goal. The main problem raised by *or*-parallel PROLOG is how to represent different bindings of the same variable on different branches of the computation tree. Different schemes have been proposed to deal with that problem (e.g., [Ciepielewski *et al.*, 1987, Warren, 1987, Westphal and Robert, 1987]). In addition to the binding problem, FIDO introduces a further difficulty due to domain-variables. Let us illustrate the problem through an example. Suppose X is ranging over {1,...,10} and the constraint `X` $\neq$ `3` has to be solved. In theory, this constraint is solved by binding `X` to a new variable `Y` ranging over {1,2,4,...,10}. However, for efficiency and memory reasons, the implementation scheme of FIDO removes the value 3 from the domain of `X` and trails the modification to undo it on backtracking. Therefore the new problem to be solved by *or*-parallel FIDO is how to represent different states of the same domain on different branches of the tree[2]. Although this problem looks similar to the binding problem of PROLOG, it has a main difference. In PROLOG, a variable is free or bound. When the variable is bound, it never changes its value. This is not the case for domain-variables. The domain fields (i.e., the fields representing the domain of a variable) can be updated several times. It follows that existing binding schemes do not apply directly although they provide useful insights on the present issue.

7.3.2 A Copy Solution

A straightforward solution to the above problem consists of requiring that *or*-parallel FIDO follows precisely the theory, i.e., each time the domain of a variable is updated, a new variable ranging over the reduced domain is created and the variable is bound to this newly created variable. This solution enables existing binding schemes of *or*-parallel PROLOG to be used for solving our problem. Its basic advantage is its simplicity. Its drawback is the unacceptable overhead it induces. In sequential FIDO, basic operations on domain-variables require constant time and do not depend on the size of the domains. This property is obviously violated by the copy solution. Since we aim at an efficiency on a single processor comparable to a good sequential implementation, a better scheme has to be devised.

7.3.3 Avoiding Copying

In order to avoid copying the domains, a new data structure local to a process, the domain frame, is introduced. It contains the state of domains for the branch, the process is working on. When a domain-variable is created, its domain is stored in the domain frame instead of in the global stack for the sequential implementation. It follows that each process has its own view of the domains and modifications of the domain fields only

[2]A similar problem arises with the constraints. Since the solution is the same in both cases, we only discuss domains in this thesis.

occur in the domain frame without affecting the view of other processes. The use of a domain frame makes sure that all basic operations on domains (e.g., testing if a value is in the domain) remain constant-time operations.

The main overhead of this approach arises when switching tasks. In that case, the process has to update the contents of the domain frame to reflect the state of the domains at the branch point it is grabbing work from. The update of the domain frames is achieved by using the usual value-trail which has to be generalized for that purpose to include three fields:

- the entry in the domain frame;
- the old value of the entry;
- the new value of the entry.

The *old value* field is necessary to restore the previous value of the entry on backtracking while the *new value* field is needed to update domain frames at switching tasks.

7.3.4 Avoiding Trailing

Since the value-trail is used not only on backtracking but also to update the domain frames, it is necessary to make sure that it contains all necessary information to achieve these two activities. An obvious way to enforce this requirement consists of trailing all modifications to the domain frames. However, this simple solution would induce a severe overhead and requires much more memory for the parallel version than for the sequential one. Moreover the size of the value-trail directly defines the cost of switching tasks and thus reducing the amount of trailing decreases the cost of switching tasks. For these reasons, special care should be devoted to avoid trailing. The sequential optimizations have to be generalized to make sure that the amount of trailing of *or*-parallel FiDo is proportional to the one of sequential FiDo.

7.4 Discussion and Related Work

In this chapter, we have considered the issue of exploiting parallelism and constraint-solving together in the logic programming framework. Starting with the computational model of FiDo for finite domain constraint problems, different opportunities for parallelism have been identified. The most promising alternative, the combination of *or*-parallelism and consistency techniques on finite domains, has been considered in some detail. The problems raised by an *or*-parallel FiDo have been defined and several possible solutions have been proposed. Among them was an efficient scheme which makes sure that all operations on domains remain constant-time operations and preserves most of the efficiency of the sequential implementation.

The above scheme has some similarities with the SRI model presented in [Warren, 1987]. Each process (resp. worker in the SRI model) has a local data-structure for storing the domains (resp. bindings) which make sure that the basic operations on domains (resp. variables) remain constant-time operations. The basic overhead arises at task switching due to the need to update the domain frame (resp. the binding array). However there are also some important differences between them.

The SRI model distinguishes between conditional and unconditional bindings. Only conditional bindings are recorded into the binding array and trailed. Conditional bindings are directly stored into the variable cell. This distinction comes from the single assignment property of the logical variable. Once bound, a variable never changes its value. This is not true for domain fields which can be updated several times and differently on different branches. It follows that all the domain fields have to store in the domain frame.

The scheme does not assume a global address space as does the SRI model. During dereferencing, a worker in the SRI model can access other workers' stacks. Since access to a non-local stack is transparent in that model, it is necessary to have a global address space. Conversely, the domain frame does not contain references but only pure data (i.e., the value of the domain fields) so that no global address space is necessary. Finally, it is worth noting that the scheme presented here is independent of the building scheme used for variables.

Thus, we have seen that exploiting parallelism for finite domain constraint solving provides another direction towards our overall goal of improving both declarativity and efficiency of finite domain constraint satisfaction. in this context the contribution of the research presented in this chapter clearly is on improving the problem-solving efficiency while preserving the declarative aspects of the FiDo system. Hence, the investigation of parallel constraint satisfaction techniques also contributes to let declarativity meet efficiency.

8

Applications

In this chapter, we will present in more detail the application of Contax to the automatic selection of lathe tools in the computer-aided production planning (CAPP) application that we have already briefly introduced in Chapter 1.

In Section 8.1, we will first introduce application scenario in which the lathe-tool selection problem appears. Section 8.2 then presents the problem setting in more detail and how the problem can be formulated as a CSP (Section 8.3). As this problem exhibits some characteristics that can be found in many, especially technical applications, in Section 8.4 we will put special consideration on these issues and will discuss to how far the functionality of the Contax system matches the requirements of the application.

In Section 8.6, the chapter will then be concluded by briefly presenting an application of Contax to University Sports Planning which is still under development but already shows impressive results.

8.1 The CAPP Application Scenario

The ARC-TEC project at DFKI constitutes an AI approach to implement the idea of computer-integrated manufacturing (CIM). Along with conceptual solutions, it provides a continuous sequence of software tools for the **A**cquisition, **R**epresentation, and **C**ompilation of **TEC**hnical knowledge (cf. [Bernardi *et al.*, 1991]). For its evaluation, an expert system for production planning had to be developed.

The input to the production planning system is a very low-level description of a rotational-symmetric workpiece as it comes from a CAD system. Geometrical description of the workpiece's surfaces and topological neighborhood relations are the central parts of this representation. If possible at all, production planning with these data starting from (nearly) first principles would require very complex algorithms. Thus, planning strategies on such a detailed level are neither available nor do they make sense. Instead human planners [Schmalhofer *et al.*, 1991] have a library of skeletal plans in their minds. Each of these plans is associated with a more or less abstract description of a (part of a) workpiece, which are called workpiece features [Klauck *et al.*, 1991]. Such a feature is defined by its association to a corresponding manufacturing method.

The generation of an abstract feature description of the workpiece is the first step of the production planning process. The obtained features characterize the workpiece with respect to its production. In a second step the skeletal plans (associated to the features) are retrieved and merged resulting in an abstract NC program, which is then transformed into code for the concrete CNC machine.

This schema shows a strong similarity to the *heuristic classification* approach presented in [Clancey, 1985]: feature recognition performs an abstraction from concrete details to abstract features which are then matched to skeletal plans. These are in turn then merged and refined in order to obtain the final plan for the CNC machine. Figure 8.1 illustrates the 'arc' of heuristic classification as it is applied to our CAPP application.

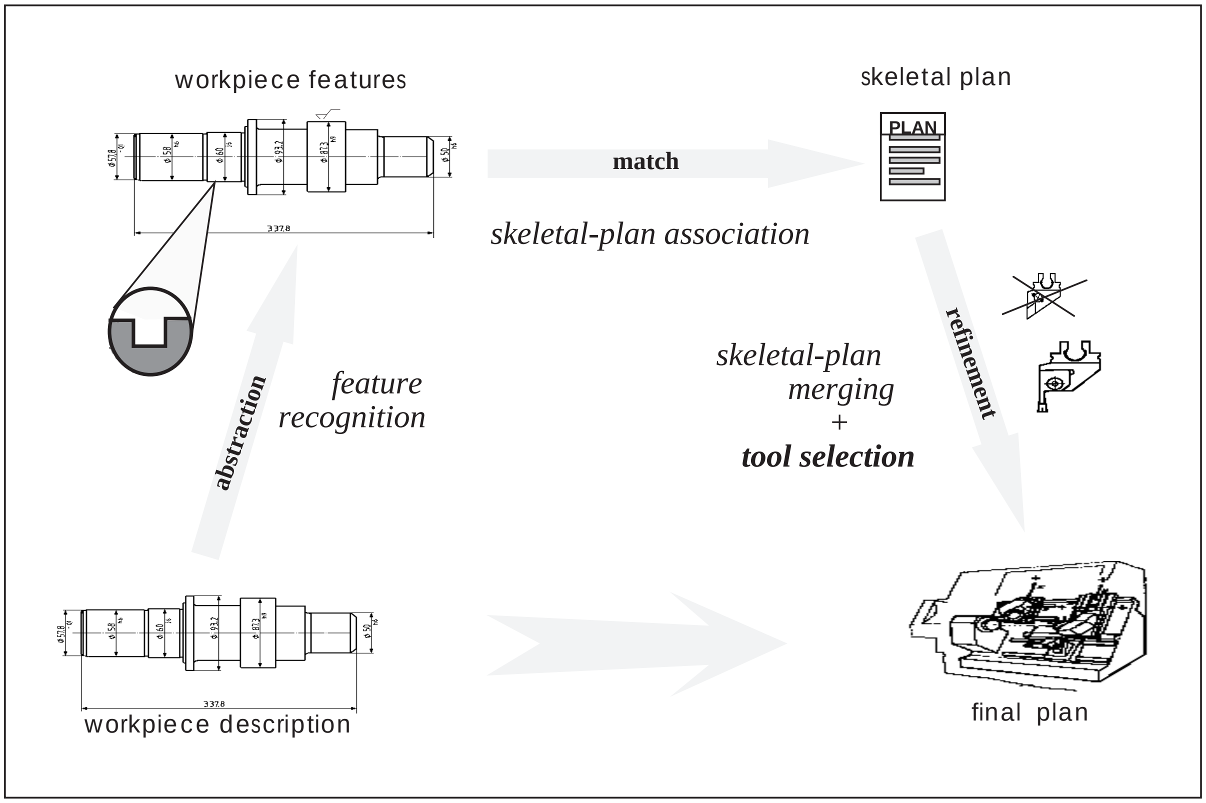


Figure 8.1: Heuristic classification applied to the CAPP scenario

The planning system has been developed using CoLab [Boley *et al.*, 1993], a hybrid knowledge representation and compilation laboratory, which integrates the power of forward and backward reasoning, constraint propagation, and taxonomic classification. The focus is not on an integrated smooth system, but on exemplifying methodologies for the use of hybrid formalisms at certain subtasks. These various subtasks require a number of specialized reasoning mechanisms integrated in CoLab: Feature aggregation is performed by the forward reasoning component Forward[1] together with the terminological

[1]Forward [Hinkelmann, 1992] is a declarative rule-based system with Horn clauses as its basic representation scheme, which is tightly coupled with RelFun to achieve bidirectional reasoning. It

component TAXON.[2] The derived features are finally collected by a program written in RELFUN[3], the backward reasoning component of COLAB. The abstract NC program is then generated from the classified workpiece by parameterized retrieval of skeletal plans using RELFUN and selecting the appropriate lathe tools via CONTAX which is also integrated in COLAB.

As a hybrid system, COLAB combines backward (RELFUN) and forward rules (FORWARD) with terminologies (TAXON) and constraints (CONTAX). It provides knowledge items and processing methods for each of these formalisms as well as interfaces between them.

8.2 The Lathe-Tool Selection Problem

The application problem we have solved with CONTAX appears in the refinement phase of the heuristic classification scenario shown in Figure 8.1. Once one or more appropriate skeletal plans have been associated to the workpiece and merged into one abstract plan, the problem comes up to find appropriate lathe tools for manufacturing the workpiece. This problem is now referred to as the lathe-tool selection problem.

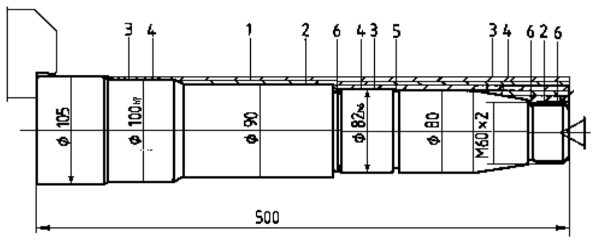


Figure 8.2: An example workpiece to be manufactured

According to the shape, the material and other attributes of the lathe part to be manufactured, the work-plan consists of a number of different steps. A typical work-plan

offers two different implementations: The first interprets bottom-up rules directly using a magic set transformation for goal-directed reasoning. The second transforms bottom-up and bidirectional rules to Horn clauses which are finally compiled into code for an extended WAM with a special forward code area. The cooperation of FORWARD and TAXON combines the general purpose reasoning power of rule-based systems with the inheritance abstraction provided by terminological systems together with their ability to check (concept) definitions for plausibility [Hanschke and Hinkelmann, 1992].

[2]TAXON [Baader and Hanschke, 1991, Hanschke, 1993] is a KL-ONE-like knowledge representation system. It provides two subformalisms: one to define and reason about terminologies, called Tbox, and another (called Abox) to reason about assertional knowledge. A terminology consists of a set of intensional concept definitions, which are arranged in a *subsumption hierarchy* (actually a directed acyclic graph) by the *classification service*. In the Abox the concepts can be instantiated by individuals. The individuals have attributes and belong to concepts. This assertional knowledge is used to determine the most specific concepts in the subsumption hierarchy to which the individuals belong (*realization service*).

[3]RELFUN [Boley, 1993] is a logic-programming language extended by call-by-value, non-deterministic, non-ground functions, and higher-order operations. Its integrating concept is *valued clauses* encompassing both Horn clauses for defining relations and directed conditional equations for defining functions.

may provide one step for roughing, another step for finishing and a third (facultative) step for doing the fine finishing of the lathe part. However, a work-plan can be much more complicated. For each processing step, appropriate tools have to be chosen.

This tool selection depends heavily on a lot of geometrical (e.g. the edge-angle) as well as technological parameters (e.g. material, process etc.). Moreover, the tool system itself consists of subparts that have to be combined, e.g. the tool holder, the material of the plate and its geometry. In practice, there are a lot of restrictions, 'which holder to use for which plate', 'which kind of plate geometry to use for which workpiece contour' and so on. Figure 8.2 shows a typical lathe workpiece where one can already identify the different processing steps for which lathe tools have to be selected. The lathe tools for the different manufacturing features and lathe-turning steps are finally selected by CONTAX and are shown in Figure 8.3.

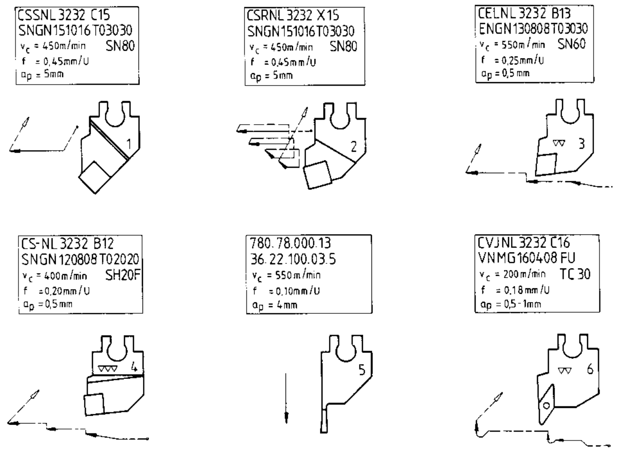


Figure 8.3: The selected lathe tools for the example workpiece

The lathe-tool selection problem can naturally be formulated as a CSP as will be shown here for a small part of the real problem. To keep things simple, we may assume that a lathe tool consists of two basic parts:

- the *cutting plate*, which actually cuts the material, and
- the *tool holder*, which serves to hold the cutting plates.

We can exchange either the cutting plate only or both plate and holder. There is a functional relation between holders and tools, i.e. for one holder, only a few tools are suited. A crucial point in this context is to minimize the number of tool exchanges, i.e. doing as many processing steps as possible with the same tool.

In our application, we are now concerned with finding a well-suited tool—or rather: a number of well-suited tools—starting from a set of constraints which describe the actual problem, i.e. information about the process to be performed, about the lathe part to be processed, and internal information about the compatibility of holders and cutting-plates as well as about holder and plate geometries. Lathe-tool selection will then result in a set of possible holder/tool combinations for each skeletal plan or manufacturing feature. Using this information, the planning layer formalized in RelFun will finally perform some global optimizations necessary to obtain the best workplan.

8.3 The Lathe-Tool Selection Problem as a CSP

When formalizing the tool selection problem as a CSP, one of the first steps to do is to determine the objects in the problem statement that will be represented by variables in a CSP. For the lathe-tool application we obtain, for example, the following variables:

- **Holder**: This variable denotes the tool holder. In the beginning, it ranges over the domain of all holders. During constraint propagation, it will be restricted to the set of holders which can currently be chosen.
- **Plate**: This variable denotes the cutting plate to be chosen. Analogously to the holder variable, it ranges over the set of all cutting plates and will be restricted subsequently.
- **Process**: This variable corresponds to the actual kind of processing.
- **WP-material**: This variable contains the material of the lathe workpiece.
- **Beta-max**: This variable denotes the maximal angle β appearing within the range of one feature of the workpiece. In general, each of these features corresponds to a single working process.
- **Edge-Angle**: This variable embodies the most important geometrical attribute of a cutting-plate, its edge-angle ε.
- **TC-Edge-Angle**: The tool cutting edge-angle χ is a geometrical characteristic of the tool holder. It denotes the angle between the horizontal cutting direction and the marginal cutting axis of the holder.

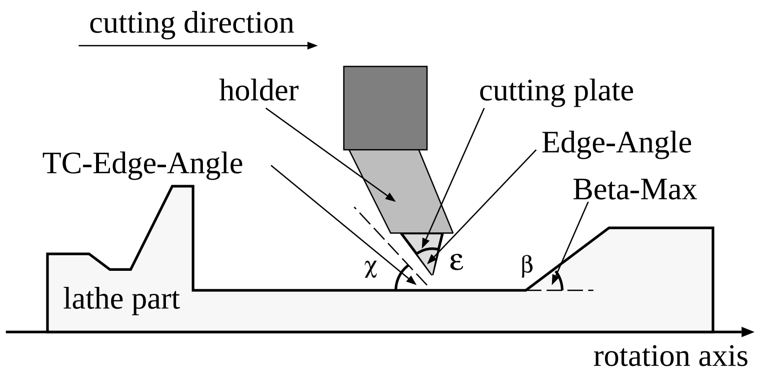


Figure 8.4: The Angle Constraint

Figure 8.4 gives a better understanding of the geometrical items introduced above.

Having identified the problem variables, the constraints can be put on the variables. In the following, we will consider only the most important constraints:

- **holder_tcea(Holder, TC-Edge-Angle)**: This constraint describes the functional relation between a holder and its tool-cutting edge-angle. It is represented as a primitive or database constraint by enumerating all the possible combinations.

- **plate_ea(Plate, Edge-Angle)**: This constraint is a database-constraint, too. It denotes the fact that each plate has its own edge-angle. Of course, we could have implemented the plate as a more complex data structure containing its edge-angle as an attribute. For the sake of uniformity, we implemented it as a constraint, just as we did with the **holder_tcea** constraint.

- **compatible(Holder, Plate)**: This constraint expresses the compatibility condition between tool holders and cutting plates.

- **hard_enough(Plate, WP-Material)**: For materials with different degrees of hardness, different cutting-plates have to be used. Processing hardened steel, e.g., may require ceramic or even diamond cutting plates, whereas aluminum can be cut with other, cheaper plates. Note, however, that hardness is just one of many attributes of a material which are important in order to choose the right cutting plate.

- **process_holder(Process, Holder)**: For the different steps of processing, different types of holders are appropriate. Well-suited for the purpose of roughing, e.g., are holders of the CSSNL class.

- **process_edge_angle(Process, Edge-Angle)**: This constraint expresses a rule of thumb which says that for roughing, plates with big edge-angles should be chosen, whereas for finishing, smaller edge-angles are appropriate.

- **TC-Edge-Angle + Edge-Angle + Beta-Max < 180°**: This numerical constraint expresses the condition that the sum of the tool-cutting edge-angle and the edge-angle must be less than the difference between 180° and the maximal ascending feature-angle. This constraint becomes evident when looking at Figure 8.4, where the angles are denoted by χ, ε, β, respectively.

In our example, all but one constraint are of binary nature. We have already discussed that this is not a necessary condition for applying the CONTAX system on this problem. However, many other constraint systems can process binary constraints only such that the ternary constraint would have to be transformed into 3 binary ones according to the scheme presented in Section 2.1.6. Figure 8.5 shows a part of the resulting constraint network as concerns the variables and constraints that have been discussed.

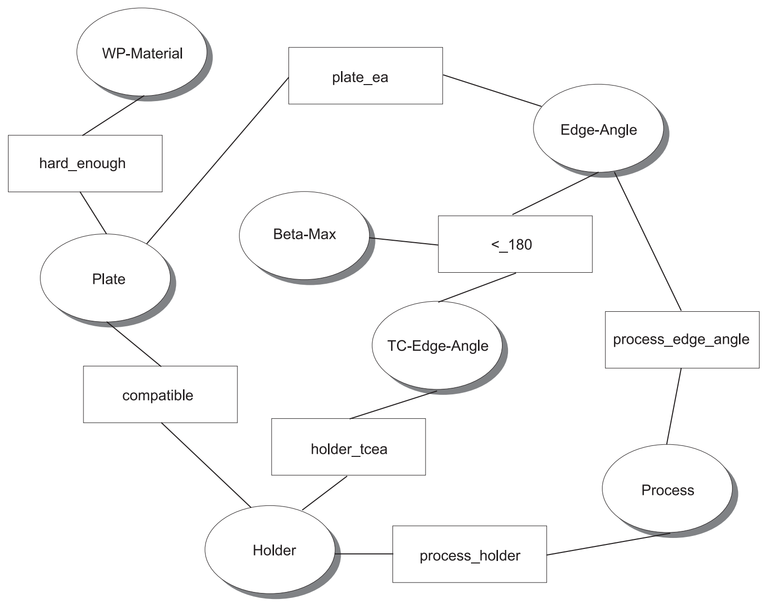


Figure 8.5: The Exemplary Constraint Net

8.4 Using CONTAX for Lathe-Tool Selection

The constraint system CONTAX presented in Chapter 3 has been used to formalize and solve the lathe-tool selection problem as it occurs in our CAPP application. The principal approach how to formalize the lathe-tool selection problem as a CSP has been given in Section 8.3 using a small example. However, in our real-life application we have to deal with a considerably larger set of constraints on some more variables ranging over much larger domains.

8.4.1 Hierarchically structured domains

The current implementation of the lathe-tool selection module [Tolzmann, 1992] covers 59 different cutting plates, 62 holders, and 22 different materials. As nearly all domains in the application exhibit some kind of hierarchical structure, we could make extensive use of CONTAX' capability to efficiently deal with hierarchically structured domains.

Indeed, we took this more complex application to evaluate the efficiency gain by exploiting hierarchical domain structures in constraint propagation. The results obtained are twofold:

- As already mentioned in Chapter 5, hierarchical hyperarc-consistency does not pay off for dealing with small domain hierarchies or hierarchies with a high branching rate in relation to the height. For example, the domain of workpiece materials contains 22 individual elements which have been arranged on a three-level hierarchy with an average branching rate of more than 5. For constraints over variables that range over this domain, the hierarchical hyperarc-consistency algorithm was about two times slower than the 'standard' hyperarc-consistency algorithm presented in Section 3.2.1.

- However, on the domain of holders which shows a maximum branching rate of 4 and a height of 8, hierarchical hyperarc-consistency clearly outperforms the 'standard' hyperarc-consistency algorithm by a factor of up to 4.

This again shows that it has to be carefully decided when to use the hierarchical version of the hyperarc-consistency algorithm and when to use the 'standard' (non-hierarchical) variant even if the domains exhibit some kind of hierarchical structure.

8.4.2 Weighted constraints

As already mentioned in Chapter 1, the problem constraints determining the tool selection include requirements of different strength. Besides the 'hard' requirements as defined e.g. by geometrical relations, there are a lot of more or less relaxable requirements in the problem specification. These include, e.g., technological 'rules' like

> **If** the workpiece is stable (a criterion depending on the length/diameter ratio) and the `edge-angle` is of class `large` or `medium`,
>
> **then** prefer a medium tool-cutting edge-angle (`tc-edge-angle`)

as well as economical ones like

> **If** the current process is rough-turning the workpiece,
>
> **then** a quadratic plate should be preferred to a triangular one wherever possible. Quadratic plates can be used more often in a process since they have one more tip.

Thus, it has been a crucial point for the success of applying CONTAX techniques to the lathe-tool selection problem that priorities could be expressed together with the requirements. Moreover, CONTAX provides means to express them most declaratively in the form of discrete symbolic weights which keeps the application knowledge base easily maintainable.

8.5 Discussion

The lathe-tool selection problem has been taken as a real-life application in order to evaluate in how far the features that have been discussed in Chapters 4 and 5 and added to the CONTAX system match the specific requirements that are present in most real application problems.

After all, it appeared that without being able to attach weights to constraints, it would have been much harder if not impossible to formalize the problem in CONTAX, at least requiring the LISP program from which CONTAX is called to perform the necessary optimization. Thus, this application has proved the need for weigted constraints to broaden the scope for CSP applications.

Also, even if not strictly necessary for formalizing the problem in CONTAX, the exploitation of the hierarchical domain structure made the resulting CONTAX program easier to understand as constraints could be formulated referring to subdomains instead of individual values. Thus, the declarativity of the representation was increased—as was the efficiency for most (unfortunately, not all) constraints over hierarchical domains.

Finally, since variables and constraints are compiled into CLOS objects, the most time-consuming task has been the generation of the constraint network. Thus, as in our concrete application all constraints and variables that have to be generated are known in advance, the constraint net can be built up when loading the whole planning system; the constraint-solving process itself can then be performed very efficiently.

8.6 Using Finite Domain Constraints for University Sports Scheduling

The University Sports Scheduling problem appears within a small project carried out at the University of Kaiserslautern that aims at using modern data processing technology to enhance productivity in the area of the local University Sports Organization (HSSP[4]).

One particular application problem in this area is the scheduling of tennis courts that has to be done every summer. Although it may sound as a problem quite easy to solve, this is surely not quite true which may be best proven by the fact that once a year the main HSSP group consisting of about 6 people spends about one afternoon to solve this problem which is stated as follows:

- In Spring each year, students, guests, and researchers apply for getting assigned one of four tennis courts for the whole season for one hour the week.
- Only time slots on week days between 8am and 8pm are assigned. On Saturdays, the latest date is from 1pm until 2pm. However, as one of the HSSP people will have to be present in order to open the sports arena in the morning and close it afterwards, they are interested in not assigning dates on Saturdays.
- People apply as couples for to be assigned one court for one hour every week. In order to have some criteria available for possible conflict resolution, applicants are asked to let the HSSP people know their preferred dates. Moreover, it is possible and required to give a set of three differently preferred choices, e.g. Monday 9am or 10am at priority 1, Wednesday after 4pm at priority 2, and Friday at 11am at priority 3.
- To let the problem statement become even more complicated, there are also preferences on the status of applicants, meaning that guests have a lower priority to

[4]in German: *Hochschulsport*, hence the abbreviation HSSP

get their preferred date assigned than e.g. students have. The same is true for university employees who get a higher priority than students.

- And finally, applicants can also pose very special requirements such as that two couples like to get assigned the same court for consecutive dates. The reason for this requirement is that they may want to part avoid having to clean the court afterwards.

The above mentioned requirements can all be represented as weighted finite domain constraints and can thus be solved using the Contax system. However, standard constraint solvers will only be able to solve this problem after performing a complete reformulation which in turn decreases the declarativity of the approach. Moreover, whether a reformulation for use with any other constraint system will result in better problem-solving performance respectively efficiency, has in the meantime become nearly irrelevant, as the current Contax implementation solves the 1994 Tennis Courts Scheduling Problem in less than 3 minutes.

As this result alone is already very impressive, we would nevertheless like to add that the Contax approach appears to be superior not only as concerns problem-solving speed but also, most important, as concerns problem-solving quality. Figure 8.6 shows the amount of first choices being satisfied by the solutions (a) manually obtained by the HSSP group and (b) obtained by the Contax application.

	HSSP group	CONTAX
Training	100.0 %	100.0 %
Special Arrangements	93.3 %	100.0 %
Priority 1 (1st choice)	31.25 %	50.0 %
Priority 2 (2nd choice)	59.2 %	60.2 %
Priority 3 (3rd choice)	76.9 %	72.7 %
Special Requirements	100.0 %	66.7 %

Figure 8.6: Problem-solving quality of Tennis Court Scheduling approaches

The quality of the solution can also easily be estimated by having a look at the resulting schedule generated with Contax which is shown in Figure 8.7. The very dark fields represent first choice assignments, i.e. the applicants received exactly that assignment they applied for.

The success of applying Contax to solving this problem heavily relies on its ability to handle priorities between constraints. Thus, once again it appears that many real-

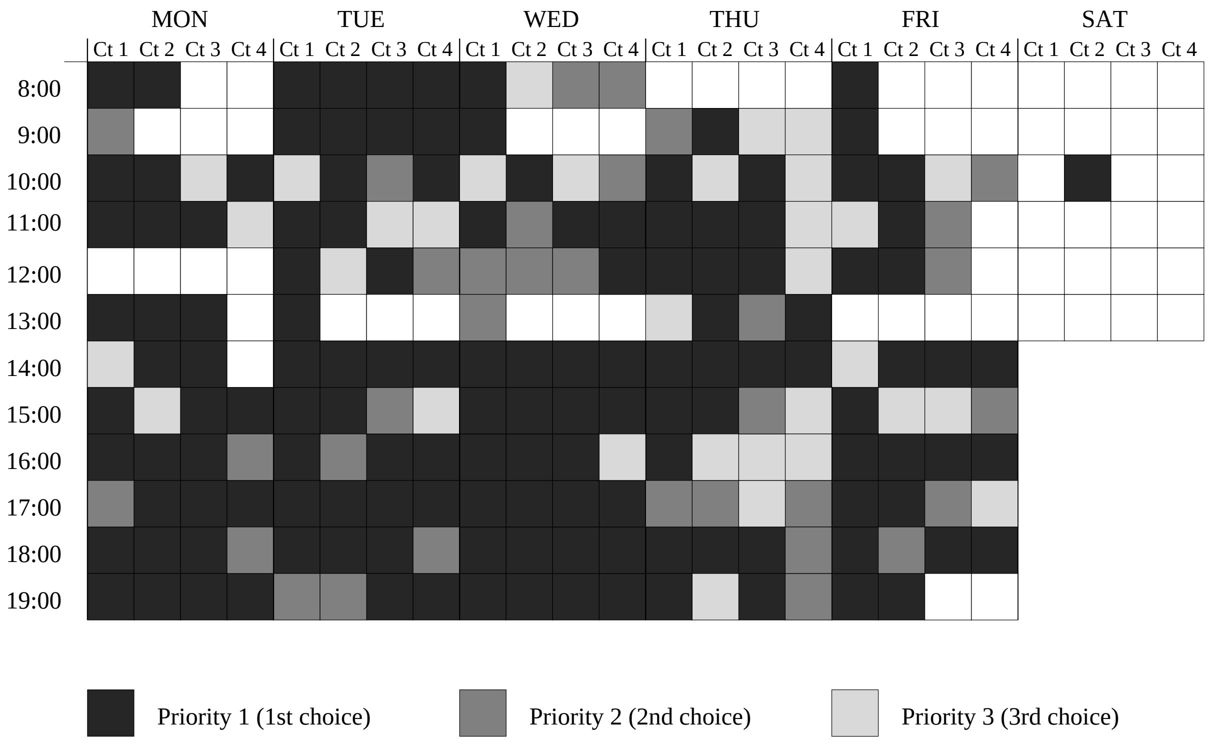


Figure 8.7: Resulting Tennis Court Schedule generated with Contax

life applications demand for constraint solving techniques that able to handle weighted constraints. Therefore, the Contax system again proves its applicability for tackling real-life problems.

9

Conclusions

The present thesis contributes to a number of research directions in the area of finite domain constraints. State-of-the-art constraint satisfaction techniques have been implemented and extended towards the needs brought in by various practical applications:

- The representation of real-life applications cannot be restricted to using binary constraints only. Thus, the first step was to extend the notion of arc-consistency in constraint graphs towards hyperarc-consistency in constraint-hypergraphs for arbitrary constraints (Chapter 3).
- Overconstrained problems demand for different techniques for solving them partially. Therefore, a partial constraint satisfaction approach has been presented that allows to find the maximal or optimal solution to a constraint problem (Chapter 4).
- Hierarchically structured domains can often be found in technical applications. In order to efficiently deal with the structure of such domains, an extension of hyperarc-consistency towards hierarchical hyperarc-consistency has been developed (Chapter 5).
- For nearly all non-toy applications, constraint solving alone does not suffice to solve the application task. Thus, we presented an integration of constraint solving techniques within the framework of logic programming and introduced a new consistency technique, weak looking-ahead (Chapter 6).
- Exploiting parallelism in logic programming is a current research issue. Hence, we investigated how finite domain constraint extensions can also benefit from the speedup that can be achieved by using or-parallelism (Chapter 7).
- Finally, two non-toy applications have been taken to study how some of the techniques presented in this thesis contribute to solving real problems in computer-aided manufacturing and in sports scheduling (Chapter 8).

Taking all techniques together, the research presented in this thesis enables us to solve a wider class of problems, allows for a declarative problem representation, and can be implemented efficiently. Thus, finally it contributes to let **Declarativity meet Efficiency** and also to let **Theory meet Application**.

Bibliography

Part A: References

[Allen and Koomen, 1983] J. F. Allen and J. A. Koomen. Planning using a temporal world model. In *Proceedings International Joint Conference on Artificial Intelligence (IJCAI-83)*, pages 741–747, 1983.

[Allen, 1983] J. Allen. Maintaining Knowledge about Temporal Intervals. *Communications of the ACM*, 26(11):832–843, 1983.

[Amarel, 1970] S. Amarel. On the Representation of Problems and Goal-Directed Procedures for Computers. In R. Banerji and M. Mesarovic, editors, *Theoretical Approaches to Non-Numerical Problem Solving*, pages 179–244. Springer-Verlag, Heidelberg, 1970.

[Baader and Hanschke, 1991] F. Baader and P. Hanschke. A Scheme for Integrating Concrete Domains into Concept Languages. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence*, 1991.

[Bajcsy, 1993] R. Bajcsy, editor. *Proc. of* 13th *Intl. Joint Conf. on AI*, volume 1, San Mateo, California, USA, 1993. Morgan Kaufmann Publishers, Inc.

[Bibel, 1988] W. Bibel. Constraint Satisfaction from a Deductive Viewpoint. *Artificial Intelligence*, 36:401–413, 1988.

[Bobrow *et al.*, 1988] D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. E. Keene, G. Kiczales, and D. A. Moon. Common Lisp Object System Specification. X3J13 Document 88-002R, June 1988.

[Boehm, 1988] B. Boehm. A Spiral Model of Software Development and Enhancement. *IEEE Computer*, May 1988.

[Boizumault *et al.*, 1993] P. Boizumault, Y. Delon, and L. Péridy. Solving a Real-Life Planning Exams Problem using Constraint Logic Programming. In M. Meyer, editor, *Constraint Processing: Proceedings of the International Workshop at CSAM'93*, DFKI Research Report RR-93-39, pages 107–112, German Research Center for Artificial Intelligence (DFKI), P.O. Box 2080, D-67608 Kaiserslautern, Germany, August 1993.

[Boley and Richter, 1991] H. Boley and M. M. Richter, editors. *Proceedings of the International Workshop on Processing Declarative Knowledge (PDK'91)*, number 567 in

Lecture Notes in Artificial Intelligence (LNAI). Springer-Verlag, Berlin, Heidelberg, 1991.

[Boley, 1993] H. Boley. A Direct Semantic Characterization of RELFUN. In E. Lamma and P. Mello, editors, *Proceedings of the 3rd International Workshop on ELP '92*, volume 660 of *LNAI*. Springer, 1993. Also available as Research Report RR-92-54, Nov. 1992, DFKI GmbH, P. O. Box 2080, D-6750 Kaiserslautern.

[Borning *et al.*, 1987] A. Borning, R. Duisberg, B. Freeman-Benson, A. Kramer, and M. Woolf. Constraint Hierarchies. In *Proceedings of Object-Oriented Programming, Systems and Applications (OOPSLA) 1987*, number ACM 0-89791-247-0/87/0010-0048, pages 48–60. Association for Computing Machinery, October 1987.

[Borning *et al.*, 1989] A. Borning, M. Maher, A. Martindale, and M. Wilson. Constraint Hierarchies and Logic Programming. In *Proceeding of the International Conference on Logic Programming (ICLP) 1989*, pages 149–164, 1989.

[Brachman and Schmolze, 1985] R. J. Brachman and J. G. Schmolze. An Overview of the KL-ONE Knowledge Representation System. *Cognitive Science*, 9(2):171–216, 1985.

[Brown and Purdom, 1981] C. A. Brown and P. W. Purdom. How to Search Efficiently. In *Proceedings IJCAI*, pages 588–594, 1981.

[Brown *et al.*, 1993] A. L. Brown, S. Mantha, and T. Wakayama. Relaxable Horn Clauses: Constraint Optimization using Preference Logics. In *Seventh International Symposium on Methodologies f or Intelligent Systems (ISMIS'93), Trondheim, Norway*, number 689 in Lecture Notes in Artificial Intelligence (LNAI). Springer-Verlag, Berlin, Heidelberg, June 1993.

[Bruynooghe and Pereira, 1984] M. Bruynooghe and L. M. Pereira. Deduction revision by intelligent backtracking. In *Implementations of Prolog*, pages 194–215. Ellis Horwood, 1984.

[Budnick *et al.*, 1977] F. S. Budnick, R. Morena, and T. Vollmann. *Principles of Operations Research for Management.* Richard D. Irwin Inc., U.S.A., 1977.

[Carlsson, 1987] M. Carlsson. Freeze, Indexing, and Other Implementation Issues in the WAM. In J.-L. Lassez, editor, *Proceeding of the International Conference on Logic Programming (ICLP) 1987*, Melbourne, Australia, May 1987. MIT Press.

[Charman, 1993] P. Charman. Solving Space Planning using Constraint Technology. In M. Meyer, editor, *Constraint Processing: Proceedings of the International Workshop at CSAM'93*, DFKI Research Report RR-93-39, pages 159–172, German Research Center for Artificial Intelligence (DFKI), P.O. Box 2080, D-67608 Kaiserslautern, Germany, August 1993.

[Chassin *et al.*, 1988] J. Chassin, J. Syre, and H. Westphal. Implementation of a Parallel PROLOG System on a Commercial Multiprocessor. In *Proceedings of the European*

Conference on Artificial Intelligence (ECAI) 1988, pages 278–283, Munich, Germany, August 1988.

[Ciepielewski *et al.*, 1987] A. Ciepielewski, B. Hausman, and S. Haridi. Or-parallel Prolog Made Efficient on Shared Memory Multiprocessors. In *Symposium on Logic Programming*, August 1987.

[Clancey, 1985] W. J. Clancey. Heuristic Classification. *Artificial Intelligence*, 27:289–350, 1985.

[Clark, 1978] K. L. Clark. Negation as Failure. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 293–322. Plenum Press, New York, 1978.

[Clocksin and Alshawi, 1988] W. Clocksin and H. Alshawi. A Method for Efficiently Executing Horn Clause Programs Using Multiple Processors. *New Generation Computing*, 5:361–376, 1988.

[Clowes, 1971] M. B. Clowes. On Seeing Things. *Artificial Intelligence*, 2:79–116, 1971.

[Colmerauer, 1987] A. Colmerauer. Opening the Prolog III Universe. *BYTE*, pages 177–182, August 1987.

[Dauboin, 1993] P. Dauboin. A Constraint-Based Vehicle Configurer. In *Sixth International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems (IEA/AIE-93), Edinburgh, Scotland*, pages 1–5. Gordon and Breach Science Publishers, June 1993.

[De Groot, 1984] D. De Groot. Restricted And-Parallelism. In *Proceedings of Fifth Generation Computer Systems (FGCS) 1984*, pages 471–478, Tokyo, Japan, 1984.

[De Schreye *et al.*, 1990] D. De Schreye, D. Pollet, J. Ronsyn, and M. Bruynooghe. Implementing finite-domain Constraint Logic Programming on Top of a Prolog-System with Delay-mechanism. In N. Jones, editor, *(ESOP) 1990*, pages 106–117, 1990.

[Debray and Warren, 1986] S. K. Debray and D. S. Warren. Automatic Mode Inference for Prolog Programs. In IEEE, editor, *IEEE Conference on Logic Programming*, pages 78–88, 1986.

[Dechter and Meiri, 1989] R. Dechter and I. Meiri. Experimental Evaluation of Preprocessing Techniques in Constraint Satisfaction Problems. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI) 1989*, pages 271–277. Morgan Kaufmann Publishers, San Mateo, CA, 1989.

[Dechter and Pearl, 1986] R. Dechter and J. Pearl. The cycle-cutset method for improving search performance in AI applications. Technical Report R-16, Cognitive Systems Lab, Computer Science Department, UCLA, Los Angeles, California, 1986.

[Dechter and Pearl, 1988a] R. Dechter and J. Pearl. Network-Based Heuristics For Constraint-Satisfaction Problems. *Artificial Intelligence*, (34):1–38, 1988.

[Dechter and Pearl, 1988b] R. Dechter and J. Pearl. Tree-clustering schemes for constraint processing. In *Proceedings AAAI-88*, 1988.

[Dechter *et al.*, 1989] R. Dechter, I. Meiri, and J. Pearl. Temporal Constraint Networks. In R. Brachman, H. Levesque, and R. Reiter, editors, *First Int. Conference on Principles of Knowledge Representation and Reasoning, Toronto, Canada*, pages 83–93. Morgan Kaufmann Publishers, San Mateo, CA, 1989.

[Dechter *et al.*, 1990] R. Dechter, A. Dechter, and J. Pearl. Optimization in Constraint Networks. In R. M. Oliver and J. Q. Smith, editors, *Influence Diagrams, Belief Nets, and Decision Analysis*, pages 411–425. Wiley, 1990.

[Dechter *et al.*, 1991] R. Dechter, I. Meiri, and J. Pearl. Temporal Constraint Networks. *Artificial Intelligence*, 49:61–95, 1991.

[Dechter, 1989] R. Dechter. Enhancement Schemes for Constraint Processing: Backjumping, Learning and Cutset Decomposition. *Artificial Intelligence*, 41:273–312, 1989.

[Dechter, 1992] R. Dechter. Constraint Networks. In S. C. Shapiro, editor, *Encyclopedia of AI*, pages 276–285. Wiley, New York, 1992.

[Descotte and Latombe, 1985] Y. Descotte and J.-C. Latombe. Making Compromises among Antagonist Constraints in a Planner. *Artificial Intelligence*, 27:183–217, 1985.

[Deßloch, 1993] S. Deßloch. *Semantic Integrity in advanced Database Management Systems*. PhD thesis, Computer Science Department, University of Kaiserslautern, Germany, 1993.

[Diaz and Codognet, 1993] D. Diaz and P. Codognet. A minimal Extension of the WAM for clp(FD). In *Proceeding of the International Conference on Logic Programming (ICLP) 1993*, 1993.

[Dincbas and Van Hentenryck, 1988] M. Dincbas and P. Van Hentenryck. The Constraint Logic Programming Language CHIP. In *Proceedings of Fifth Generation Computer Systems (FGCS) 1988*, Tokyo, Japan, December 1988.

[Dincbas *et al.*, 1989] M. Dincbas, H. Simonis, and P. Van Hentenryck. Solving the Car–Sequencing Problem in Constraint Logic Programming. In *Proceedings of the European Conference on Artificial Intelligence (ECAI) 1988*, pages 290–295, 1989.

[Dowling and Gallier, 1984] W. F. Dowling and J. H. Gallier. Linear-time algorithms for testing the satisfiability of propositional Horn formulae. *Journal of Logic Programming*, 3:267–284, 1984.

[Duncan, 1990] T. Duncan. Scheduling Problems and Constraint Logic Programming: A simple Example and its Solution. Technical Report AIAI-TR-120, AI Applications Institute, University of Edinburgh, October 1990.

[Evans, 1992] O. Evans. Factory Scheduling using finite Domains. In G. Comyn, N. Fuchs, and M. J. Ratcliff, editors, *Logic Programming Summer School 1992*, pages 45–53, September 1992.

[Feldman and Golumbic, 1990] R. Feldman and R. Golumbic. Optimization algorithms for student scheduling via constraint satisfiability. *Computer Journal*, 33:356–364, 1990.

[Fikes and Nilsson, 1971] R. E. Fikes and N. Nilsson. STRIPS: a new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

[Finkel and Manber, 1987] R. Finkel and U. Manber. DIB – A Distributed Implementation of Backtracking. *ACM Transactions on Programming Languages and Systems*, 9(2):235–256, April 1987.

[Fox and Sadeh, 1990] M. Fox and N. Sadeh. Why is Scheduling difficult? A CSP Perspective. In *Proceedings of the European Conference on Artificial Intelligence (ECAI) 1990*, 1990.

[Fox *et al.*, 1983] M. Fox, B. Allen, S. Smith, and G. Strohm. Isis: A Constraint–Directed Reasoning Approach to Job Shop Scheduling. In *Proceedings of the IEEE Conference on Trends and Applications*, Gaithersburg, Maryland, May 1983. National Bureau of Standards.

[Fox, 1986] M. Fox. Observations on the Role of Constraints in Problem Solving. In *Procedings of the 6th Canadian Conference on Artificial Intelligence, Montreal, 1986*, pages 172–187. Presses de l'Université de Québec, May 1986.

[Fox, 1987] M. S. Fox. *Constraint Directed Search: A Case Study of Job-Shop Scheduling.* Morgan Kaufmann, Los Altos, California, 1987.

[Frayman and Mittal, 1987] F. Frayman and S. Mittal. COSSACK: A Constraints–Based Expert System for Configuration Tasks. In D. Sriram and R. Adey, editors, *Knowledge Based Expert Systems in Engineering: Planning & Design.* Computational Mechanics Computation, 1987.

[Freuder and Quinn, 1985] E. C. Freuder and M. J. Quinn. Taking advantage of stable sets of variables in Constraint Satisfaction Problems. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1076–1078, 1985.

[Freuder, 1978] E. Freuder. Synthesizing Constraint Expressions. *Communications of the ACM*, 21(11):958–966, 1978.

[Freuder, 1990] E. Freuder. Complexity of k-Tree Structured Constraint Satisfaction Problems. In *Proceedings of the American Assiciation of Artificial Intelligence (AAAI) 1990*, pages 4–9, 1990.

[Freuder, 1992] E. C. Freuder. The Logic of Constraint Satisfaction. *Artificial Intelligence, Special Volume: Constraint-based Reasoning*, 58(1–3):21–70, 1992.

[Frühwirth, 1993] T. Frühwirth. Temporal Reasoning with Constraint Handling Rules. Working Paper ECRC-CORE-93-2, European Research Center (ECRC), ECRC, Munich, Germany, February 1993.

[Gaschnig, 1979] J. Gaschnig. *Performance measurement and analysis of certain search algorithms.* PhD thesis, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, 1979.

[Güsgen and Hertzberg, 1988] H.-W. Güsgen and J. Hertzberg. Some Fundamental Properties Of Local Constraint Propagation. *Artificial Intelligence*, (36):237–247, 1988.

[Güsgen *et al.*, 1987] H. W. Güsgen, U. Junker, and A. Voß. Constraints in a Hybrid Knowledge Representation System. In *Proceedings IJCAI-87*, 1987.

[Güsgen, 1988] H.-W. Güsgen. CONSAT — *A System for Constraint Satisfaction.* PhD thesis, University of Kaiserslautern, January 1988.

[Güsgen, 1989] H.-W. Güsgen. Spatial Reasoning Based on Allen's Temporal Logic. Technical Report TR–89–049, International Computer Science Institute, Berkley, CA, August 1989.

[Hanschke and Hinkelmann, 1992] P. Hanschke and K. Hinkelmann. Combining Terminological and Rule-based Reasoning for Abstraction Processes. In *Proceedings of the 16th German Workshop on Artificial Intelligence (GWAI-92)*, number 671 in Lecture Notes on Artificial Intelligence. Springer-Verlag, September 1992. Also available as DFKI Research Report RR-92-40.

[Hanschke *et al.*, 1991] P. Hanschke, A. Abecker, and D. Drollinger. TAXON: A Concept Language with Concrete Domains. In *[Boley and Richter, 1991]*, pages 411–413, 1991.

[Hanschke, 1993] P. Hanschke. *A Declarative Integration of Terminological, Constraint-based, Data-driven, and Goal-directed Reasoning.* PhD thesis, Universität Kaiserslautern, Fachbereich Informatik, 1993.

[Haralick and Elliott, 1980] R. M. Haralick and G. L. Elliott. Increasing Tree Search Efficiency for Constraint Satisfaction Problems. *Artificial Intelligence*, 14:263–313, 1980.

[Havens and Mackworth, 1983] W. S. Havens and A. K. Mackworth. Representing knowledge of the visual world. *IEEE Computer*, 16(10):90–96, 1983.

[Havens and Rehfuss, 1989] W. Havens and P. Rehfuss. PLATYPUS: A Constraint–Based Reasoning System. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI) 1989*, pages 48–53, 1989.

[Hermenegildo, 1986] M. Hermenegildo. An Abstract Machine for Restricted AND-parallel execution of logic programs. In *Proceeding of the International Conference on Logic Programming (ICLP) 1988*, pages 25–39, London, July 1986.

[Hinkelmann, 1992] K. Hinkelmann. Forward Logic Evaluation: Compiling a Partially Evaluated Meta-interpreter into the WAM. In *Proceedings of the 16th German Workshop on Artificial Intelligence (GWAI-92)*, number 671 in Lecture Notes on Artificial Intelligence. Springer-Verlag, September 1992.

[Holzbaur, 1990] C. Holzbaur. Realization of Forward Checking in Logic Programming through Extended Unification. Technical Report TR–90–11, Austrian Research Institute for Artifical Intelligence, June 1990.

[Hrycej, 1993] T. Hrycej. A temporal Extension of Prolog. *Journal of Logic Programming*, (15):113–145, 1993.

[Huffman, 1971] D. A. Huffman. Impossible Objects as Nonsense Sentences. *Machine Intelligence*, 6:295–323, 1971.

[Jaakola, 1990] J. Jaakola. Modifying The Simplex Algorithm To A Constraint Solver. In P. Deransart and J. Maluszynski, editors, *Programming Language Implementation And Logic Programming*, pages 89–105, August 1990.

[Jaffar and Lassez, 1987] J. Jaffar and J.-L. Lassez. Constraint Logic Programming. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages, Munich, Germany*, pages 111–119, Munich, Germany, January 1987. ACM.

[Klauck *et al.*, 1991] C. Klauck, A. Bernardi, and R. Legleitner. FEAT-REP: Representing Features in CAD/CAM. In 4th *International Symposium on Artificial Intelligence: Applications in Informatics*, 1991. An extended version is also available as DFKI Research Report RR-91-20.

[Kopisch, 1993] M. Kopisch. Spatial Relations for Configuration Tasks in General and for the Cabin Layout of the AIRBUS A340 in Particular. In *Sixth International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems (IEA/AIE-93), Edinburgh, Scotland*, pages 450–457. Gordon and Breach Science Publishers, June 1993.

[Kowalski, 1979] R. Kowalski. Algorithm = Logic + Control. *Communications of the ACM*, 22(7):424–436, July 1979.

[Kumar *et al.*, 1988] V. Kumar, K. Ramesh, and V. Nageshewara Rao. Parallel Best-First Search of State-Space Graphs: A Summary of Results. In *AAAI-88*, pages 122–127, Saint Paul, Minnesota, 1988.

[Kumar, 1992] V. Kumar. Algorithms for Constraint-Satisfaction Problems: A Survey. *AI Magazine*, 13(1):32–44, 1992.

[Lacroix and Lavency, 1987] M. Lacroix and P. Lavency. Preferences: Putting more knowledge into queries. In *Proceedings 15th International Conference on Very Large Data Bases (VLDB)*, pages 217–225, 1987.

[Lai and Sahni, 1987] T. Lai and S. Sahni. Anomalies in Parallel Branch and Bound Algorithms. *Communications of the ACM*, 27(6):594–602, March 1987.

[Lawler and Wood, 1966] E. L. Lawler and D. E. Wood. Branch-and-bound methods: a Survey. *Operations Research*, 14:699–719, 1966.

[Lee *et al.*, 1993] H. Lee, R. Lee, and G. Yu. Constraints Logic Programming and Mixed Integer Programming. In *Proc. of* 26*th* *Hawaian Intl. Conf. on System Sciences*, pages 543–552, 1993.

[Liu and Ku, 1992] B. Liu and Y.-W. Ku. ConstraintLisp: An Object-Oriented Constraint Programming Language. *ACM SIGPLAN Notices*, 27(11):17–26, November 1992.

[Liu and Popplestone, 1990] Y. Liu and R. Popplestone. Symmetry Constraint Inference in Assembly Planning. In *Proc. of AAAI 90*, pages 1038–1044, 1990.

[Lloyd, 1987] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, Berlin, Heidelberg, New York, 1987.

[Lusk et al., 1988] E. Lusk et al. The Aurora Or-Parallel Prolog System. In *Proceedings of Fifth Generation Computer Systems (FGCS) 1988*, Tokyo, Japan, December 1988.

[Mackworth and Freuder, 1985] A. Mackworth and E. Freuder. The Complexity of some Polynomial Network Consistency Algorithms for Constraint Satisfaction Problems. *Artificial Intelligence*, 25:65–73, 1985.

[Mackworth *et al.*, 1985] A. Mackworth, J. Mulder, and W. Havens. Hierarchical Arc Consistency: Exploiting Structured Domains in Constraint Satisfaction Problems. *Computational Intelligence*, 1:118–126, 1985.

[Mackworth, 1977] A. Mackworth. Consistency in Networks of Relations. *Artificial Intelligence*, 8:99–118, 1977.

[Mackworth, 1992a] A. K. Mackworth. Constraint Satisfaction. In S. C. Shapiro, editor, *Encyclopedia of AI*, pages 285–293. Wiley, New York, 1992.

[Mackworth, 1992b] A. Mackworth. The Logic of Constraint Satisfaction. *Artificial Intelligence, Special Volume: Constraint-based Reasoning*, 58(1–3):3–20, 1992.

[Maier, 1983] D. Maier. *The Theory of Relational Databases.* Computer Science Press, Rockville, Maryland, 1983.

[Meier *et al.*, 1989] M. Meier, A. Aggoun, D. Chan, P. Dufresne, R. Enders, D. de Villeneuve, A. Herold, P. Kay, B. Perez, E. van Rossum, and J. Schimpf. SEPIA – An Extendible Prolog System. In G. Ritter, editor, *Proceedings of the IFIP 11th World Computer Congress*, pages 1127–1132, August 1989.

[Mellish, 1985] C. S. Mellish. Some Global Optimizations for a Prolog Compiler. *Journal Logic Programming*, 1:43–66, 1985.

[Meseguer, 1989] P. Meseguer. Constraint Satisfaction Problems: An Overview. *AI Communications*, 2(1):3–17, March 1989.

[Meyer, 1993] M. Meyer, editor. *Proceedings of the Workshop on Constraint Processing at CSAM'93*, DFKI Research Report RR-93-39, DFKI, P.O. Box 2080, 67608 Kaiserslautern, Germany, August 1993.

[Meyer, 1994] M. Meyer, editor. *Constraint Processing: Workshop Notes.* ECAI'94, Amsterdam, August 1994.

[Mittal and Falkenhainer, 1990] S. Mittal and B. Falkenhainer. Dynamic Constraint Satisfaction Problems. In *Proc. of AAAI 90*, pages 25–32, 1990.

[Mohr and Henderson, 1986] R. Mohr and T. C. Henderson. Arc and Path Consistency Revisited. *Artificial Intelligence*, 28:225–233, 1986.

[Mohr and Masini, 1988] R. Mohr and G. Masini. Good old discrete relaxation. In *Proceedings European Conference on Artificial Intelligence*, pages 651–656, 1988.

[Montanari, 1974] U. Montanari. Networks of Constraints: Fundamental Properties and Applications to Picture Processing. *Information Science*, 7:95–132, 1974.

[Morgenstern, 1985] M. Morgenstern. The Role of Constraints in Databases, Expert Systems, and Knowledge Representation. In L. Kerschberg, editor, *Expert Database Systems*, pages 351–368. 1985.

[Müller, 1990] B. Müller. Planning with Constraint Satisfaction Algorithms. In H.-J. Appelrath, A. Cremers, and O. Herzog, editors, *The EUREKA Project PROTOS*, pages 59–66. April 1990.

[Parrello *et al.*, 1986] B. D. Parrello, W. C. Kabat, and L. Wos. Job-shop scheduling using automated reasoning: A case study of the car sequencing problem. *Journal of Automated Reasoning*, 2(1):1–42, 1986.

[Parrello, 1988] B. D. Parrello. CAR WARS. *AI Expert*, 3(1):60–64, 1988.

[Reingold *et al.*, 1977] E. M. Reingold, J. Nievergelt, and N. Deo. *Combinatorial Algorithms: Theory and Practice.* Prentice-Hall, Englewood Cliffs, New Jersey, 1977.

[Reiter, 1988] R. Reiter. Nonmonotonic Reasoning. In H. E. Shrobe, editor, *Exploring Artificial Intelligence*, pages 439–482. Morgan Kaufmann, San Mateo, California, 1988.

[Rosenfeld *et al.*, 1976] A. Rosenfeld, R. Hummel, and S. Zucker. Scene labeling by relaxation operations. *IEEE Transactions on Syst. Man. Cybern.*, 6:420–433, 1976.

[Rossi, 1988] F. Rossi. Constraint Satisfaction Problems In Logic Programming. *SIGART Newsletter*, (106):24–28, October 1988.

[Sacerdoti, 1974] E. D. Sacerdoti. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 5(2):115–135, 1974.

[Schmalhofer *et al.*, 1991] F. Schmalhofer, O. Kühn, and G. Schmidt. Integrated Knowledge Acquisition from Text, Previously Solved Cases, and Expert Memories. *Applied Artificial Intelligence*, 5:311–337, 1991.

[Shapiro and Haralick, 1981] L. Shapiro and R. Haralick. Structural descriptions and inexact matching. *IEEE Transactions on Pattern Anal.ysis and Machine Intelligence*, 3:504–519, 1981.

[Shapiro, 1987] E. Shapiro. An OR-Parallel Execution Algorithm for Prolog and its FCP Implementation. In *Fourth International Conference on Logic Programming*, pages 311–337, Melbourne, Australia, May 1987.

[Sidebottom and Havens, 1991] G. Sidebottom and W. Havens. Hierarchical Arc Consistency applied to Numeric Processing in Constraint Logic Programming. Technical Report CSS-IS TR 91-06, Expert Systems Lab, Centre for Systems Science and School of Computing Science, Simon Fraser University, Burnaby, British Columbia, V5A IS6, Canada, 1991.

[Stallman and Sussman, 1977] R. Stallman and G. Sussman. Forward Reasoning and Dependency–Directed Backtracking in a System for Computer–Aided Circuit Analysis. *Artificial Intelligence*, 9:135–196, 1977.

[Swain and Cooper, 1988] M. Swain and P. Cooper. Parallel Hardware for Constraint Satisfaction. In *AAAI-88*, pages 682–686, Saint Paul, Minnesota, 1988.

[Tate, 1977] A. Tate. Generating project networks. In *Proceedings Fifth International Joint Conference on Artificial Intelligence (IJCAI-77)*, pages 888–893, 1977.

[Tolba, 1993] H. Tolba. TemPro: A Temporal Constraint-Based Reasoning Tool. In M. Meyer, editor, *Constraint Processing: Proceedings of the International Workshop at CSAM'93*, DFKI Research Report RR-93-39, pages 133–142, German Research Center for Artificial Intelligence (DFKI), P.O. Box 2080, D-67608 Kaiserslautern, Germany, August 1993.

[Tolzmann, 1992] E. Tolzmann. Realisierung eines Werkzeugauswahlmoduls mit Hilfe des Constraint-Systems CONTAX, June 1992.

[Tsang, 1993] E. Tsang. *Foundations of Constraint Satisfaction.* Academic Press, 1993.

[Tucker and Zucker, 1992] J. V. Tucker and J. I. Zucker. Deterministic and Nondeterministic Computation, and Horn Programs, on Abstract Data Types. *Journal Logic Programming*, 13:23–55, 1992.

[Van Hentenryck and Carillon, 1988] P. Van Hentenryck and J.-P. Carillon. Generality versus Specificity: an Experience with AI and OR techniques. In *AAAI-88*, pages 660–664, Saint Paul, Minnesota, 1988.

[Van Hentenryck and Dincbas, 1988] P. Van Hentenryck and M. Dincbas. Forward Checking in Logic Programming. In *Proc. of ICLP 88*, pages 229–256, 1988.

[Van Hentenryck, 1987] P. Van Hentenryck. *Consistency Techniques in Logic Programming.* PhD thesis, University of Namur, Belgium, July 1987.

[Van Hentenryck, 1989] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, Cambridge, Ma., 1989.

[Voda, 1988] P. Voda. The Constraint Language Trilogy: Semantics and Computations. Technical Report, Complete Logic Systems, North Vancouver, British Columbia, Canada, 1988.

[Voß and Voß, 1987] A. Voß and H. Voß. Formalizing Local Constraint Propagation Methods. Arbeitspapiere der GMD 248, Gesellschaft für Mathematik und Datenverarbeitung mbH, May 1987.

[Wah and Ma, 1984] B. Wah and Y. Ma. MANIP: A Multicomputer Architecture for Solving Combinatorial Extremum-Search Problems. *IEEE Transactions on Computers*, 33(5):377–390, May 1984.

[Wallace, 1993] R. Wallace. Why AC-3 is almost always better than AC-4 for Establishing Arc Consistency in CSPs. In *[Bajcsy, 1993]*, pages 239–245, 1993.

[Waltz, 1972] D. L. Waltz. Generating Semantic Descriptions from Drawings of Scenes with Shadows. Technical Report AI-TR-271, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1972.

[Warren, 1983] D. H. D. Warren. *An Abstract Prolog Instruction Set.* Technical Note 309, SRI International, Menlo Park, California, 1983.

[Warren, 1987] D. Warren. The SRI Model for Or-Parallel Execution of Prolog. Abstract Design and Implementation Issues. In *International Symposium on Logic Programming*, pages 46–53, September 1987.

[Wegner, 1990] P. Wegner. Object-Oriented Megaprogramming. *ACM Member Supplement to the Communications of the ACM*, October 1990.

[Westphal and Robert, 1987] Westphal and Robert. The PEPSys Model: Combining Backtracking, AND- and OR-parallelism. In *International Symposium on Logic Programming*, pages 436–448, September 1987.

[Wilkins, 1988] D. E. Wilkins. *Practical Planning: Extending the classical AI planning paradigm.* Morgan Kaufmann, San Mateo, California, 1988.

Part B: Related Publications of the Author

Parts of the content of this thesis have been published in the following conference papers and journal articles:

[Bachmann and Meyer, 1991] B. Bachmann and M. Meyer. Konfiguration als dynamisches CSP über strukturierten Domänen. In A. Günter and R. Cunis, editors, *Beiträge zum 5. Workshop Planen und Konfigurieren*, pages 148–151. Labor für Künstliche Intelligenz (LKI), Universität Hamburg, Bodenstedtstraße 16, D-2000 Hamburg 50, April 1991.

[Bernardi *et al.*, 1991] A. Bernardi, H. Boley, K. Hinkelmann, P. Hanschke, C. Klauck, O. Kühn, R. Legleitner, M. Meyer, M. M. Richter, G. Schmidt, F. Schmalhofer, and W. Sommer. ARC-TEC: Acquisition, Representation and Compilation of Technical Knowledge. In *Expert Systems and their Applications: Tools, Techniques and Methods*, Avignon, France, 1991. Also available as DFKI Research Report RR-91-27, DFKI, P.O. Box 2080, 67608 Kaiserslautern, Germany.

[Boley *et al.*, 1993] H. Boley, P. Hanschke, K. Hinkelmann, and M. Meyer. COLAB: A hybrid knowledge compilation laboratory. DFKI Research Report RR-93-08, DFKI, P.O. Box 2080, 67608 Kaiserslautern, Germany, Kaiserslautern, Germany, January 1993. Also to appear in *Annals of Operations Research.*

[Hein and Meyer, 1992] H.-G. Hein and M. Meyer. A WAM Compilation Scheme. In A. Voronkov, editor, *Logic Programming: Proceedings of the* 1^{st} *and* 2^{nd} *Russian Conferences*, volume 592 of *Lecture Notes in Artificial Intelligence (LNAI)*, pages 201–214. Springer-Verlag, Berlin, Heidelberg, 1992.

[Meyer and Müller, 1993a] M. Meyer and J. Müller. Combining Forward-Checking and Looking-Ahead in Constraint Logic Programming. In J. Wu, J. Yang, Y. Li, and W. Gao, editors, *Towards the Future: Proceedings of the Third International Conference for Young Computer Scientists (ICYCS'93), Beijing, China*, pages 460–463. Tsinghua University Press, Beijing, China, July 1993.

[Meyer and Müller, 1993b] M. Meyer and J. Müller. Finite Domain Consistency Techniques: Their Combination and Application in Computer-Aided Process Planning. In *Seventh International Symposium on Methodologies for Intelligent Systems (ISMIS'93), Trondheim, Norway*, number 689 in Lecture Notes in Artificial Intelligence (LNAI), pages 385–394. Springer-Verlag, Berlin, Heidelberg, June 1993.

[Meyer and Müller, 1993c] M. Meyer and J. Müller. Using Weak Looking-Ahead for Lathe-Tool Selection in a CIM Environment. In *Sixth International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems (IEA/AIE-93), Edinburgh, Scotland*, pages 26–35. Gordon and Breach Science Publishers, June 1993. This paper received the IEA/AIE-93 Best Paper Award. Also available as DFKI Research Report RR-93-22, DFKI, P.O. Box 2080, 67608 Kaiserslautern, Germany.

[Meyer and Müller, 1994] M. Meyer and J. Müller. Solving Configuration Tasks Efficiently Using Finite Domain Consistency Techniques. *International Journal of Applied Intelligence*, 1994. To appear.

[Meyer and Stein, 1993] M. Meyer and W. Stein. Finite Abstract Emulation of Logic Programs. In *10. Workshop Alternative Konzepte für Sprachen und Rechner der GI-Fachgruppe 2.1.4, Bad Honnef*, April 1993.

[Meyer *et al.*, 1991] M. Meyer, J. Müller, and S. Schrödl. FIDO: Exploring Finite Domain Consistency Techniques in Logic Programming. In H. Boley and M. M. Richter, editors, *Proceedings of the International Workshop on Processing Declarative Knowledge (PDK'91)*, number 567 in Lecture Notes in Artificial Intelligence (LNAI), pages 425–427. Springer-Verlag, Berlin, Heidelberg, 1991.

[Meyer *et al.*, 1992a] M. Meyer, H.-G. Hein, and J. Müller. FIDO: Finite Domain Consistency Techniques in Logic Programming. In A. Voronkov, editor, *Logic Programming: Proceedings of the* 1^{st} *and* 2^{nd} *Russian Conferences*, volume 592 of *Lecture Notes in Artificial Intelligence (LNAI)*, pages 294–301. Springer-Verlag, Berlin, Heidelberg, 1992.

[Meyer *et al.*, 1992b] M. Meyer, H.-G. Hein, and W. Stein. FIDO: Finite Domain Consistency Techniques in a Compiler-Based Environment. In S. Hölldobler, editor, *Logische Programmierung*, pages 54–57. AIDA-Report AIDA-92-10, TH Darmstadt, Germany, October 1992.

[Meyer, 1991] M. Meyer. Parallel Constraint Satisfaction in a Logic Programming Framework. In *Proceedings of the International Conference on Parallel Computing Technologies (PaCT-91)*, pages 148–157. World Scientific Publishing, Singapore, September 1991.

[Meyer, 1992a] M. Meyer. Hierarchical Constraint Satisfaction and its Application in Computer-Aided Production Planning. In *Proceedings of the* 13^{th} *National Computer Conference, Riyadh, Saudi-Arabia*, pages 54–72. King Abdulaziz City for Science & Technology, Riyadh, Kingdom of Saudi-Arabia, November 1992.

[Meyer, 1992b] M. Meyer. Hierarchical Constraint Satisfaction and its Application in Computer-Aided Production Planning. In *Expert Systems 92, Cambridge, U.K.*, December 1992.

[Meyer, 1992c] M. Meyer. Using Hierarchical Constraint Satisfaction for Lathe-Tool Selection in a CIM Environment. In *Fifth International Symposium on Artificial Intelligence (ISAI), Cancun, Mexico*, pages 167–177. AAAI Press, December 1992. Also available as DFKI Research Report RR-92-35, DFKI, P.O. Box 2080, 67608 Kaiserslautern, Germany.

[Meyer, 1993] M. Meyer. Finite Domain Constraints: Eine deklarative Wissensrepräsentationsform mit effizienten Verarbeitungsverfahren. In H. Boley, F. Bry, and U. Geske, editors, *Neuere Entwicklungen der deklarativen KI-Programmierung — Proceedings, Workshop auf der 17. Fachtagung für Künstliche Intelligenz, Berlin*, pages

105–111. Research Report RR-93-35, DFKI, P.O. Box 2080, D-67608 Kaiserslautern, Germany, September 1993.

[Meyer, 1994] M. Meyer. Using Hierarchical Constraint Satisfaction for Lathe-Tool Selection in a Computer-Integrated Manufacturing Environment. *Integrated Computer-Aided Engineering*, 1(3):241–251, 1994.

Part C: Related Work by the Author's Students

Parts of the content of this thesis are based on work students did under the author's supervision for their master's theses or *Projektarbeit* at the University of Kaiserslautern:

[Bachmann, 1991] B. Bachmann. *HieraCon — A Knowledge Representation System with Typed Hierarchies and Constraints.* Master's thesis, University of Kaiserslautern, Computer Science Department, August 1991. Also available as DFKI Document D-91-12, DFKI, P.O. Box 2080, 67608 Kaiserslautern, Germany.

[Hein, 1992] H.-G. Hein. *Propagation Techniques in WAM based Architectures — The FIDO-III Approach.* Master's thesis, University of Kaiserslautern, Computer Science Department, August 1992. Also available as DFKI Technical Memo TM-93-04, DFKI, P.O. Box 2080, 67608 Kaiserslautern, Germany, August 1993.

[Kamp, 1991] G. Kamp. *Entwurf, vergleichende Bewertung und Integration eines Arbeitsplanerstellungssystems für Drehteile.* Master's thesis, University of Kaiserslautern, Computer Science Department, March 1991. Also available as DFKI Document D-91-06, DFKI, P.O. Box 2080, 67608 Kaiserslautern, Germany.

[Kreinbihl, 1994] M. Kreinbihl. *COSYPLAN: Constraintbasiertes System zur Terminplanung.* Master's thesis, University of Kaiserslautern, Computer Science Department, P.O. Box 3049, 67608 Kaiserslautern, Germany, 1994.

[Müller, 1991] J. Müller. *Design and Implementation of a Finite Domain Constraint Logic Programming System based on PROLOG with Coroutining.* Master's thesis, University of Kaiserslautern, Computer Science Department, October 1991. Also available as DFKI Document D-91-12, DFKI, P.O. Box 2080, 67608 Kaiserslautern, Germany, November 1991.

[Schrödl, 1991] S. Schrödl. *FIDO: Implementation eines Constraint Logic Programming Systems mit Finite Domains.* Projektarbeit, University of Kaiserslautern, Computer Science Department, P.O. Box 3049, 67608 Kaiserslautern, Germany, July 1991.

[Stein, 1993] W. Stein. *Nutzung globaler Analysetechniken in einem optimierenden Compiler für die Constraint-Logic-Programming-Sprache FIDO-III.* Master's thesis, University of Kaiserslautern, Computer Science Department, P.O. Box 3049, 67608 Kaiserslautern, Germany, July 1993.

[Steinle, 1993] F. Steinle. *HAMLET: Erweiterung eines Constraint-Systems um Negation und Disjunktion und dessen Anbindung an eine Konzeptbeschreibungssprache.* Projektarbeit, University of Kaiserslautern, Computer Science Department, P.O. Box 3049, 67608 Kaiserslautern, Germany, August 1993.

[Steinle, 1994] F. Steinle. *Entwurf und Implementierung eines Frameworks für Constrainttechniken.* Master's thesis, University of Kaiserslautern, Computer Science Department, P.O. Box 3049, 67608 Kaiserslautern, Germany, 1994.

[Tolzmann, 1992] E. Tolzmann. *Realisierung eines Werkzeugauswahlmoduls mit Hilfe des Constraint-Systems CONTAX.* Projektarbeit, University of Kaiserslautern, Computer Science Department, Juli 1992. Also available as DFKI Document D-92-26, DFKI, P.O. Box 2080, 67608 Kaiserslautern, Germany, September 1992.

List of Figures