Over-Constrained Systems in CLP and CSP

Michael Benjamin Jampel

Ph.D. Thesis

Department of Computer Science
City University

19 September 1996

Contents

List of Figures

Acknowledgements

Many thanks to my main supervisor David Gilbert and to my second supervisor Bernie Cohen. Thanks also to Sebastian Hunt and Jean-Marie Jacquet, who were unofficial additional supervisors, and to Pierre-Yves Schobbens for detailed comments on an earlier draft of this thesis.

Thanks to Sara Jones, Peter Osmon, Julie Porteous, Rob Scott, and many others for moral support.

Thanks to Alan Borning, Michael Maher, Micha Meier, Thomas Schiex, Peter Stuckey, and Roland Yap for help with HCLP, CLP($\mathcal{R}$), and other technical issues.

Thanks to the Vice-Chancellor of City University for funding the first two years of my PhD, to the School of Informatics for funding the third year, and to Peter Osmon, David Gilbert, Geoff Dowling, and Peter Smith for arranging it all. Thanks to the European Community for awarding me a TMR grant for pursuing research in Belgium which greatly helped me to complete my thesis. In addition to fees being paid on my behalf, I have received more than £25,000 in grants and conference expenses, not including the European money; this works out as almost £200 per page of this thesis. I hope everyone thinks it is worth it.

Movie-making: Two per cent is actually making the movie. Ninety-eight per cent is hustling. That's an awful way to spend your life.

Orson Welles

Declaration

Abstract

This thesis is concerned with constraint-based logical and computational frameworks for resolving and relaxing over-constrained systems. The context is provided by two such frameworks, Hierarchical Constraint Logic Programming (HCLP) and Partial Constraint Satisfaction Problem techniques (PCSP), both of which have been extensively discussed in the literature. Our work is driven by the reasons why over-constrained systems arise, the characteristics of the 'ideal' paradigm for resolving them, and the issue of compositionality which is very important in general if we wish to examine large systems by examining and then combining smaller parts. We abstract away from programming language issues in order to focus on constraint solving.

The main original work of this thesis is divided into three parts. Firstly we present a complete method for transforming between the HCLP and PCSP representations of a problem, thus showing that theoretically they have equivalent expressive power. Secondly, having discussed compositionality in general, we present a two-stage variant of HCLP; the first stage is compositional but calculates a superset of the solutions we expect from HCLP. The second stage removes precisely those solutions which are not acceptable to HCLP, but at the cost of re-introducing HCLP's non-compositional behaviour. We also discuss the compositional aspects of PCSP.

The third part of this thesis presents GOCS, our system which allows the use of both the HCLP and PCSP approaches to problem relaxation and ordering. The GOCS integrated framework has HCLP and PCSP as special cases and also subsumes all of their separate advantages, when considering the characteristics of the ideal system for relaxing and resolving over-constrained problems. We present examples throughout the thesis, some of which are comparative and so may be used to substantiate our claims. Finally, we present conclusions and discuss further work.

Part I

Overview of the Thesis

Chapter 1

Transformation, Composition, and Integration of HCLP and PCSP

1.1 Motivation

Constraint programming is becoming more successful all the time. It provides an elegant, abstract, and declarative way to specify problems, backed-up by efficient and flexible implementations. Constraint-based systems have a strong theoretical basis, and yet also appeal to industry: there are many successful practical applications solving difficult and commercially important real-world problems. Examples include scheduling for factories and for computer instruction sets; financial applications such as options and portfolio analysis; and modelling water usage, DNA, and electrical circuits [3, 10, 23, 24, 38, 39, 64, 72, 78, 79, 86]. Many problems which previously appeared local to a particular application domain, and which were therefore solved in an ad-hoc manner, can now be seen as instances of constraint problems. For example, AI applications as disparate as machine vision and belief revision can now be considered in terms of varieties of constraints [12, 62]. This has led to re-use of ideas and techniques across different domains, and consequently to improved solutions for everybody.

The two main paradigms based on computation over constraints are Constraint Logic Programming (CLP) and Constraint Satisfaction Problem techniques (CSP). CLP can be thought of as a generalisation of logic programming where unification is replaced by constraint solving, often implemented by CSP algorithms, giving a 'pure' logical reading to what were previously considered 'impurities' of particular languages such as Prolog.

CLP and CSP have both been very successful in solving real-world problems. There are a number of differences between them, such as the fact that CLP defines a class of programming languages whereas CSP is a set set of techniques and algorithms which can be implemented in any language. But there are also important similarities, and in addition to the relationships suggested in the first paragraph, CLP can be thought of as providing a declarative general-purpose programming language framework in which CSP and other techniques can be embedded. This has benefited CLP, in giving it access to a large body of previous work, and it has also benefited the CSP community, who gain all the standard advantages of declarative programming (such as focusing on representation and high-level issues and being freed from lower-level tasks).

Over-constrained systems arise when a set of constraints does not have any solutions. Such situations occur for a number of reasons; in our opinion the two main reasons are to do with composition of problems and lack of expressivity of languages. We discuss this in more detail in Section 2.5. We consider composition to be very important as it is linked to the fundamental scientific methodology of reductionism, i.e. solving a complex problem by solving its parts separately and then combining the solutions. Composition is discussed in more detail in Part IV. We mention these two reasons here in order to have them in mind during the rest of this discussion.

The CLP and CSP communities, as well as those in AI and OR, have addressed the issue of over-constrained systems in various disparate ways, and the links which have been mutually beneficial in the standard case have not been explored when considering the various 'preference systems' which have been developed in the separate paradigms. Throughout this thesis we use the term **preference system** to mean any paradigm which allows the expression of preferences for some constraints over others, or for some variables over others, or any other method of selecting a partial problem from the original one. We focus on two such paradigms, HCLP (Hierarchical Constraint Logic Programming) and PCSP (Partial Constraint Satisfaction Problems). We will concentrate on the application of preference systems to the resolution of over-constrained systems.

Current approaches to dealing with over-constrained systems tend to two extremes. The first extreme is exemplified by producing all the maximally consistent subsets of the inconsistent set. This is computationally expensive and not very useful: the user cannot tell which of the flood of maximally consistent subsets is the one he wants. The second extreme is for the user to have to do everything, i.e. for each constraint decide whether it is essential or not. This is only possible if the user has a great deal of time available, and if he has explicit knowledge of the problem down to the level of every constraint. We free the user from this necessity, while still giving him the freedom to express preference information about individual constraints whenever it is desirable to do so.

Numerous different approaches are described in the literature, many of them tending towards the second extreme, having the same basic idea of putting all the constraints in some kind of partial order, or using strength labels on each constraint to induce an order. Then various techniques are used to choose the 'best' solution, with various slightly different definitions of 'best'. HCLP is usually not cited in these papers, despite often subsuming the approach being described. Unfortunately, HCLP has slightly unusual semantics, and is non-monotonic. We develop a two-stage variant of HCLP, each stage having relatively simple semantics and being easy to implement efficiently.

It is possible that there is some duplication in dealing with preference information in the two fields CLP and CSP (as well as in AI in and Operations Research in general), and also some re-inventing of the wheel, but markedly different formalisms and approaches inhibit cross-fertilisation and mask genuine differences. The distinction between strength levels and comparators in HCLP is blurred in most other paradigms; this is not too important at the implementation level, and may perhaps even be beneficial, but it is important semantically: there is a difference between choosing a more preferred constraint rather than a less preferred one, and choosing one of two equally preferred but mutually contradictory constraints. The former choice is easy to make deterministically, and should always be made; it is guaranteed in HCLP by the property of 'respecting the strength-label hierarchy'. The latter choice may introduce non-determinism, or negotiation, or averaging, and can be embodied in the choice of comparator. For example it is interesting to note that when we transform between HCLP and PCSP in Part III, we must merge the HCLP strength labels and comparator together to create a distance function in the PCSP representation. When transforming in the other direction, we only need to use one level of the hierarchy. But this difference is hard to evaluate given the disparate natures of the two formalisms. We clarify this issue and provide a general model for transforming between HCLP and PCSP in Part III.

There have been two recent works which provide general frameworks encompassing various preference systems including PCSP and (indirectly) HCLP (Bistarelli, Montanari and Rossi, and Schiex, Fargier and Verfaillie [4, 5, 71]), but neither discuss the links and differences *between* HCLP and PCSP, only treating them as instantiations of the general scheme. One genuine difference between HCLP and PCSP concerns how to relax problems, rather than how to chose which relaxation is the best. HCLP reorganises the structure of the problem, by specifying relationships *between* constraints; PCSP keeps a flat structure to the problem, but changes the *meaning* of the individual constraints by adding elements to its domain. This sounds absurd, but there is a strong motivating example from the PCSP literature [31]: if none of your shirts match your tie, you could say that this constraint is not too important (HCLP), or you could buy a *new* tie (PCSP domain augmentation).

Clarifying the semantics of HCLP by developing a compositional variant, providing a common framework for discussing HCLP and PCSP, showing if their methods for selecting relaxations are equivalent, and providing an integrated framework which allows both their different approaches to the creation of relaxed problems to be used, are all significant advances for both paradigms, for the field of preference systems and over-constrained problems, and for the study of constraints in general.

1.2 The thesis

The claim that we make in this thesis is as follows: while current preference systems have various advantages, they do not approach the ideal. Small extensions to current systems can be made, and bring some benefits, but to get really close to the ideal it is necessary to create a new, more general, framework, which we present in this thesis.

1.3 Our approach

We can improve on the current situation regarding preference systems in a number of different ways. The simplest approach is to try and modify either HCLP or PCSP in order to improve them and make them closer to the ideal preference system, thus avoiding discarding the general body of work done on these paradigms. We take this approach in Parts III and IV: in Part III we provide a transformation between HCLP and PCSP. This allows the user to have the specification advantages of one of them and the implementation advantages of the other, where this is appropriate. In Part IV we consider compositionality, and improve the semantics of HCLP in this and other respects. Following this approach leads to a certain amount of success, but still leaves us some distance away from the ideal preference system.

In Part V we investigate another approach, namely defining our own preference system called GOCS; it is a new general compositional constraint framework with similarities to HCLP and PCSP, and indeed has each of these systems as a special case. However, it also permits their two different styles of problem relaxation to be used at the same time! GOCS defines a new class of solvers, which would in general require a certain amount of implementation effort, but some of the work involved can be avoided by re-use of the results of Parts III and IV. GOCS has significantly more of the characteristics of the ideal preference system than either HCLP or PCSP, and we believe it is sufficiently general to include as a special case any other preference system which may be developed.

1.4 Chapter guide

This thesis is structured as a series of parts. Part I, the current part, contains the claim of the thesis, with brief motivation and benefits. Part II begins with Chapter 2 which contains a discussion of how over-constrained systems arise as well as brief overviews of CLP and CSP, and some definitions and clarifications. This is followed by background material on HCLP (Chapter 3) and PCSP (Chapter 4). Then Chapter 5 specifies the characteristics of the ideal system for resolving over-constrained systems, and analyses to what extent HCLP and PCSP conform to the ideal. The part ends with a brief overview of certain mathematical tools which will be useful subsequently (Chapter 6).

In Part III, Chapter 7, we present transformations from HCLP into PCSP, and in Section 7.3 we also discuss the transformation in the opposite direction, from PCSP into HCLP. We describe both transformations in logic, using a single equivalence relation.

Part IV begins in Chapter 8 with a general discussion of composition, what it is and why it is important. In Chapter 9 there is a discussion of the lack of compositionality in HCLP, followed by a presentation of our variant of HCLP which *is* compositional, in as much as any preference system can be. Then we discuss the compositionality of PCSP (Chapter 10), against a background of the mathematics of lattices and order.

Part V presents GOCS, a preference system we have developed. It is compositional,

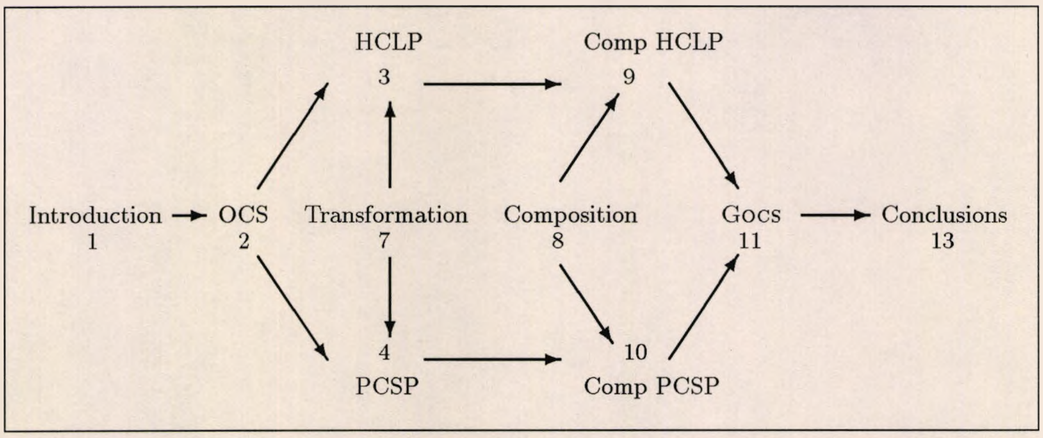

Figure 1.1: Thesis flow

general, includes HCLP and PCSP as special cases, and has all their advantages as well as some benefits which they do not have. We also discuss implementations.

Part VI contains summary and conclusions, discussion and benefits, and pointers to further work.

The thesis has some Appendices, and also References.

1.5 Benefits

Benefits of our work include:

for the theoretician:

- a framework in which to discuss, compare, and contrast HCLP and PCSP simultaneously
- transformations between HCLP and PCSP
- a compositional variant of HCLP, showing at what stage non-monotonic (disorderly) behaviour is introduced
- a proof of compositionality for PCSP distance functions with subset and subset-closed derived orders

for HCLP implementors and users:

- a two-stage implementation of (a variant of) HCLP, the first stage being compositional and incremental. Therefore possibility of more efficient implementations
- ability to delay choice of HCLP comparator until after the first stage, when an idea of the approximate number of solutions will be available

for implementors of preference systems generally:

- an analysis of the reasons why over-constrained systems arise, leading to a list of desirable characteristics for preference systems to have
- transformations between HCLP and PCSP, thus allowing an implementor of one of them to re-use an implementation of the other
- clarification of the two different approaches to relaxation of problems, combined with unification of two important methods for choosing which relaxation is best

for the user:

- the combination of HCLP and PCSP approaches. No need to label every constraint, no need to construct a sophisticated, imperative, distance function
- hence easier and more expressive specification of constraint systems, and quicker and easier debugging and maintenance.

Part II

Background

In this part we provide background information on constraint languages and systems, on HCLP and PCSP, on bags and on lattices. We also discuss why over-constrained systems arise and what are the desirable characteristics of the preference systems we use to resolve them.

Chapter 2

Constraint Programming and Over-Constrained Systems

2.1 Constraint languages, constraint solving, and constraint logic programming

Constraint programming systems are interesting because they have a sound theoretical basis and yet they also have practical applications. Constraint languages are defined with respect to alphabets of symbols, rules for constructing formulae, and so on, with the goal of *expressing* constraints. This is separate from an implementation of an algorithm for *solving* constraints over that language, i.e. for deciding if a set of constraints is consistent, or for reducing constraints to a solved form. There is no point defining a language of constraints over a certain domain if no algorithms exist for solving those constraints. The issue of solving algorithms is itself separate from any embedding of the algorithms into a programming language. Indeed one of the main conceptual differences between CSP and CLP is that the former is a collection of algorithms and techniques which may be implemented in any language, whereas the latter is a class of languages, with computation rules, scope rules, input-output, and so on. Therefore, we will now present some background material under three headings: theory concerning constraints and constraint solving, languages, and algorithms. We then conclude this chapter with some definitions which will be used throughout the thesis.

2.2 Theory of constraint systems and constraints

2.2.1 Constraint systems

Whenever a constraint-based system is being developed, a domain is chosen, for example the real numbers, or booleans, or strings. It is assumed that a theory already exists for this domain. Therefore notions of what constitutes a relation over the domain, and what should be done with a collection of relations, are assumed to be defined. For example, systems of linear equations and inequalities over real numbers had been used for a long time in Linear Programming, before CLP($\mathcal{R}$) was developed [43]. Therefore it was intuitively clear what a 'constraint' was (a linear equation or inequality), and what should be done to it (ideally, reduced to a set of equations between a single variable and a real number, as shown in the example below). Similarly with booleans: we can assume that constraints are just well-formed formulae, and constraint solving means finding an assignment of truth values to variables such that all the formulae are true.

Therefore there has not been all that much attention paid to what might seem to be a very important question: what *is* a constraint. A straightforward answer, such as can be found in the next section, describes constraints in terms of their role in a constraint system, which just pushes the question one step away; and the theory behind any given constraint system was always assumed to be taken wholesale from an existing domain. (This is similar to the assumption of some given equality theory as a basis for logic programming languages [40].)

In fact it is possible to abstract away from these assumptions, as has been done by Jaffar and Lassez. Their extended technical report version [40] of their seminal paper [41] provides a uniform general treatment of constraint systems, based on many-sorted logic. However, its rigorous and highly technical discussion of multiple constraint domains simultaneously is too complex for our present purposes. In Appendix A we present a simplification of their work, restricted to a constraint system over just one domain. Such single-domain systems suffice for the presentation of the ideas in the rest of this thesis, and make the presentation there much clearer. However, not even the level of detail presented in the appendix is necessary to understand this thesis. Therefore, in the rest of this section we will follow the example of the field and just assume as 'given' the theory underlying a constraint system over any particular domain.

2.2.1.1 Example

A standard part of school mathematics is Gaussian elimination to find values for x and y from the following equations (calculation of the answer is left as an exercise for the reader):

$$x + y = 4$$
$$x - y = 2$$

This sort of problem is known as 'solving two equations in two unknowns' and clearly it is desirable to be able to solve more complicated instances using a computer. One interesting aspect of constraint programming is hidden in this wish, but is explicit in the example

$$X \leq 3, X \geq 3$$

From two pieces of partial or non-precise information, we can derive $X = 3$, i.e. one piece of explicit knowledge. Thus specifiers or users can express as much or as little information as they have even if they cannot do so in a precise form, and information from two different sources can be distilled. This is not impossible in other computer science methodologies, but it is can be done both easily and elegantly using constraint solving.

2.2.1.2 Another example constraint domain

Throughout this thesis small examples will be used to clarify the presentation, generally modelled using finite domains or reals as these are the simplest to present in a concise manner.

However this should not be taken to imply that no other domains are possible. The literature contains discussion of strings, booleans, and rational and infinite trees [16, 17, 18, 77]. We now present a slightly different domain, based on sets and subsets. This differs from those already seen because the domain is only partially ordered (by subset inclusion).

Consider the problem of organising interpreters for a meeting of an international organisation. Let us assume that there are three delegates, one French, one Greek, and one from Holland, and that the pool of interpreters is modelled using sets, F for those who speak French, G, and H. Let us further assume that we have two domain-specific constraints $x \in S$ and $S_1 \subseteq S_2$, as well as equality over both elements and sets. Available operations include set intersection, union, and so on. Constraint conjunction is indicated by commas and query variables are assumed to be existentially quantified, as is standard in logic programming and CLP. Unlike CLP, elements are in lower case, sets are in upper case, i.e. x, y, z are variables ranging over individuals, X, Y, Z are variables ranging over sets. The constraint 'x can interpret between languages X and Y' can be expressed easily using the intersection of X and Y. The first possibility is to find up to three individuals each of whom can translate between two of the languages:

$$x \in F \cap G, y \in F \cap H, z \in G \cap H$$

However it is quite possible that some of these intersections are empty. Therefore we may need to find a pair of translators who have a common intermediate language, X, Y, or Z:

$$x_1 \in F \cap X, x_2 \in X \cap G, y_1 \in G \cap Y, y_2 \in Y \cap H, z_1 \in G \cap Z, z_2 \in Z \cap H$$

We might wish to use as few interpreters as possible. For example, if $X = Y = Z =$ English, then we would only need three interpreters. This can be expressed as minimising the cardinality of the set $\{x_1, x_2, y_1, y_2, z_1, z_2\}$, or by ordering all the possible solutions by subset inclusion and picking the first.

If the original formulation was in fact satisfiable, then the second version will have solutions in which $X = F$ or $X = G$. If $X = F$ then the first constraint becomes trivial (for non-empty F) $x_1 \in F \cap F$, and the second constraint is more important $x_2 \in F \cap G$. This will allow x_1 to take any value in F, and if x_1 does not happen to equal x_2, perhaps because we have not in fact attempted to minimise the cardinality, then there will be 'noise' in the answer.

A more important problem is the assumption that only a pair of translators will be needed to connect two languages; in fact we might need a long sequence. This is where a constraint programming language is superior to an isolated solver, as the transitive closure of the relationship should be expressed, rather than just the relationship itself. Such recursive 'ancestral' or 'path' predicates can be written in two lines of a (constraint) logic programming language, the base case being the relationship itself. The following uses standard logic programming notation:

```
% Outputs a list of interpreters, such that Lang1 can be
%        translated into Lang2

interpret([Interpreter], Lang1, Lang2) :-
         speaks(Interpreter, Lang1),
         speaks(Interpreter, Lang2).
interpret([Int|Interpreters], Lang1, Lang2) :-
         speaks(Int, Lang1),
         speaks(Int, OtherLang),
         interpret(Interpreters, OtherLang, Lang2).
```

□

2.2.2 Constraints

A constraint is an element of a constraint system as described above. Constraints are interpreted with respect to some structure **R**, such as the real numbers, or booleans, or strings, etc. The definition of a constraint has a syntactic aspect, which allows one to consider constraints simply as tokens; this is appropriate in certain limited circumstances, but is not especially interesting for our purposes. We can say that a constraint has operational significance as a consequence of its interactions with other constraints, as defined in the theory of the particular domain.

Declaratively, generalising as much as possible without losing the link with an interpretation, we can say that an n-ary constraint is a relation over the domain, i.e. a subset of the cartesian product of the domain with itself n times. The subset denoted by a constraint may be expressed in a shorthand, such as the finite representation $X > 5$ of an infinite subset of the reals, or it may be explicitly enumerated, as is often the case in Finite Domain problems such as those found in CSPs. In this thesis we will discuss 'relaxing' or weakening constraints; given that a constraint is a subset of a domain, we can say that relaxation is effected by taking the union of the constraint with some new values. This is the standard mechanism we will use, but it is not necessary to dwell on it — we could just consider relaxation as a black-box operation taking a constraint as input and producing its relaxation as output. Consequently, whenever it is more convenient we can choose to treat constraints as basic, ignoring their internal nature as subsets of the domain.

2.2.3 Constraint Satisfaction Problems

A constraint satisfaction problem (CSP) is defined to consist of a pair $\langle V, C \rangle$ where V is a set of variables, each with a domain (extension), and C is a set of constraints. Constraints are relations over the variables in V. In CSPs, they are usually treated extensionally, i.e. a binary constraint is just considered as a set of pairs. Solving a CSP involves finding one value from the domain of each variable such that all the constraints are satisfied simultaneously.

Generally, it is standard practice in the field of CSP to restrict discussion to binary constraints over variables with finite domains. A constraint c between two variables x and y can be denoted c_{xy}. It can be shown that an n-ary relation (and hence constraint), $n > 2$, can always be represented by a set of binary relations (hence constraints). This is straightforward for finite relations, but more complicated when they are infinite. The domains of the variables in V are usually considered as unary constraints, but they can also be represented as binary constraints between a variable and itself: the value v is in the domain of a variable x iff c_{xx} contains (v, v). (This suggestion is mentioned by Freuder in [31], where he uses it to simplify the presentation of his theory PCSP.)

We can consider a CSP as a graph, with each variable represented by a node and each constraint represented by an edge. The constraints are labels for the edges. This representation of CSPs has been very fruitful in the past, see for example [20, 32, 55, 63], but is not relevant for most of this thesis. (The exception is in Section 7.3.1.2 where it provides the intuition for the defence of a certain choice when transforming PCSP into HCLP.)

2.3 Languages — Constraint Logic Programming

Traditional logic programming languages such as Prolog have their domain restricted to uninterpreted terms constructed from the constant and function symbols of the program, generally called the Herbrand Universe. The only test of satisfiability, failure of which might lead to back-tracking, is a test of syntactic equality, implemented by the unification algorithm. Interpretation of numeric constant symbols as numbers in some theory of mathematics can be achieved through impurities embedded in the language, such as the `is/2` predicate which is an interface to an imperative language implementation of arithmetic. Constraint Logic Programming [40, 41] extends logic programming by containing one or more interpreted domains of constraints. CLP is parameterised by the domain of discourse $\mathcal{D}$ over which the constraints range. Satisfiability in $\mathcal{D}$ of a set of constraints is tested by algorithms specialised to the domain. Many CLP languages have been implemented, including CLP($\mathcal{R}$), CHIP, and Eclipse [1, 22, 43]. There are a number of survey papers and books available [34, 42, 74, 75].

Jaffar and Lassez show that for every domain $\mathcal{D}$ with the two properties of solution compactness and satisfaction completeness, CLP($\mathcal{D}$) has many desirable characteristics. These include the existence of equivalent declarative and operational semantics, and in fact all the important properties of pure logic programming hold for the CLP scheme generally. (See [40, 41] for more details, and for definitions of solution compactness and satisfaction completeness.)

We have already discussed certain aspects of constraints and constraint systems in Section 2.1. Here we briefly present constraints in the context of logic programming. A constraint is a relation over a domain $\mathcal{D}$, the choice of which determines the predicate symbols $\Pi_{\mathcal{D}}$ of the language. $\Pi_{\mathcal{D}}$ is partitioned into two, the built-in constraint symbols $\Pi^c_{\mathcal{D}}$ and the symbols for user-defined predicates $\Pi^p_{\mathcal{D}}$. A constraint is an expression of the form $c(t_1, \ldots, t_n)$ where c is an n-ary symbol drawn from $\Pi^c_{\mathcal{D}}$ and each t_i is a term. Constraints can be written using other syntactical sugar, such as the standard infix mathematical order relations, but should always formally be considered as expressions of the predicate type.

CLP clauses are of the form

$$p(\mathbf{t}) :- b_1(\mathbf{t}), \ldots, b_{m+n}(\mathbf{t}).$$

where the b_i's are a mixture of c_i's and q_i's, and where $p, q_1, \ldots, q_m$ are predicate symbols from $\Pi^p_{\mathcal{D}}$, $\mathbf{t}$ denotes a list of terms, and $c_1, \ldots, c_n$ are constraints from $\Pi^c_{\mathcal{D}}$. Declaratively, the order of the c_i's and q_i's is not important, but operationally it is usually better to collect the constraints as early as possible, so that assignments of values by the predicates of the logic programming part of the program are only made if they are consistent with the constraints. The advantage of this is that pruning happens early. This is called "constrain then generate" as opposed to the traditional Prolog methodology of "generate then test". Constraints may be thought of as active tests: if failure will eventually occur in a branch of the search tree, it is more efficient if it happens early.

2.4 Algorithms

During the execution of a program, sets of constraints are collected and tested for satisfiability. In the logic programming paradigm unsatisfiability will cause failure, and hence backtracking. However, the user does not need to know how this satisfiability checking is done, which leads to the 'black box' methodology. In fact, the checking is done by different algorithms depending on the domain of the constraints.

For example, finite domain constraints are solved by a variety of propagation algorithms which seek to establish various levels of 'arc-consistency' [55, 61]. One way is to select for each variable (node of a graph) a value which remains in its domain. Use the constraints on that variable (arcs or edges in the graph with the node at one end) to remove incompatible values from the domains of the other variables to which the first one is related (thus enforcing arc-consistency). Check that the other domains have at least one value remaining in their domains. If not, the choice for the original variable was incorrect, and the original value should be removed from its domain and another one chosen (backtracking).

There is a trade-off between the amount of consistency enforced at each node and the number of nodes visited. These trade-offs are embodied in different algorithms, named from AC-1 to AC-7. There are also trade-offs concerning how to decide which variable to start with, and which value should be initially chosen. Extensive empirical research suggests that the variable with the smallest current domain should be chosen first, known as the first-fail heuristic, but that it is not usually worth considering which value to give it. Recently there has also been a theoretical analysis of which algorithms should be used. See the following papers and the references that they contain for more on propagation algorithms: [53, 56, 57, 65, 73].

Another example is the case of linear equations and inequalities over the reals, which can be resolved using the Simplex algorithm, a version of which is implemented in CLP($\mathcal{R}$) [43]. The Simplex algorithm is based on symbolic manipulation, similar in philosophy to the method which might be used in the example earlier ($x + y = 4$, $x - y = 2$). However, it is limited to linear systems. Algorithms exist for polynomials but their theoretical complexity is much worse, and also trigonometric and other complicated functions cannot be included. Therefore, interest has also been shown in interval methods [76]. These work by starting from a maximum and minimum possible value for each variable, and then propagating this information via the constraints, in a manner not wholly dissimilar from arc-consistency. It is possible that accuracy will be lost, but the advantage of this method is that it can be used when the constraints contain inequalities, polynomials, and complicated functions, in fact almost anything which can be expressed mathematically.

Ideally, the treatment in this section and the previous one should have been conceptually uniform. However, historically languages and solvers developed separately, languages within the logic programming community and solvers in AI and OR (Operations Research). Therefore they have their own separate terminology. In fact, this is not as important as it might be, because most of the issues in the two areas are orthogonal.

In the rest of this thesis we will continue to refer to some of the concepts outlined above. Some precision will be required and therefore we end this chapter by presenting various useful definitions.

2.5 Why over-constrained systems arise

Over-constrained systems may arise in various ways:

1. **mistake in *specifying* or *applying* the system:** in a language such as Pascal, a large class of errors will lead to a failure of the program to compile. For example, if we mis-measure an aspect of the world (e.g. a task in fact takes 1 hour, but we measure it as taking 25 hours), or if we mis-type a measurement when entering it into a computer (e.g. typing 11 hours instead of 1), then a strict use of Pascal's data-types may catch our mistake, especially the first one.

   However, in a constraint-based language, similar mistakes will not prevent compilation, but will usually cause queries simply to fail. The logical reading of such a failure is that a set of constraints is unsatisfiable, i.e. the system is over-constrained, even though we might characterise it differently, just as a 'bug'.

2. **composition:** the composition of two self-consistent specifications or models is inconsistent ('feature interaction'), for example some English electrical equipment and an American plug — different voltages

3. **expressivity:** standard (i.e. non-preferential) constraint languages are not rich enough. Specifiers sometimes realise that some of their constraints are less important than others. If they only utilise the most important ones, they get far too many solutions. But if they add in all the less important preferences, they get no solutions at all. The problem goes from under-determined to over-determined in one step. If they have access to a preference system such as HCLP instead of CLP, they can express both important and less important constraints, without worrying that the latter will impinge upon the solvability of the former.

   Consider the following example from Prof. Barry Richards of Imperial College [Private Communication]:

   > We were once asked to model the timetable for an airline. The model we created from the information supplied to us had no solutions; there were certain hard constraints which could not be "squared" with the data. We checked and checked our work for mistakes, until eventually we approached the client and admitted that we had a problem which we could not solve. They replied that they knew the model had no solutions, but all the appropriate staff knew that they had to "relax" certain constraints to preserve the timetable!

   Note that it is clearly impractical to expect the airline to label every single constraint with a strength of preference just to avoid this local inconsistency. It would also be very difficult to discover a single global distance function (or cost or objective function) which could lead to the relaxation of only these few constraints.

In our opinion the first reason is difficult to avoid: humans make mistakes. Good programming language design may catch some errors, but probably not the majority; however this is outside the scope of this thesis. Instead, we concentrate on composition and expressivity.

2.6 Example α

In this section we present the natural language specification of a simple over-constrained system which we then model using binary finite domain constraints. We will re-use this example at intervals throughout the thesis, so that the similarities and differences between various approaches become clear.

Consider the problem of choosing matching clothes (example adapted from Freuder and Wallace [33]). A robot wishes to wear a shirt, some shoes, and some trousers, and wants them all to match each other. There are various choices for the different items and various constraints between them. We can easily model this using three finite domain variables with a number of binary constraints between them. If we use the letter S to denote the variable for shirts, then we can use F for shoes (footwear) and T for trousers. The domain of the shirt variable will be $S :: \{r, w\}$ for red and white respectively, and similarly shoes and trousers will have domains $F :: \{c, s\}$ for cordovans and sneakers, and $T :: \{b, d, g\}$, for blue, denim, and grey. A constraint that shirts must match footwear will be denoted C_{SF}, and so on. Then, using Freuder and Wallace's assumptions about which clothes go with which, the complete problem can be expressed as follows (we will call this model α)[1]:

$$S :: \{r, w\},\ F :: \{c, s\},\ T :: \{b, d, g\}$$

$$C_{ST} :: \{(r, g), (w, b), (w, d)\},\ C_{FT} :: \{(s, d), (c, g)\},\ C_{SF} :: \{(w, c)\}$$

This problem is over-constrained; it has no solutions. We can see this by choosing the red shirt, and tracing the implications of this choice. We must choose the grey trousers, which forces us to choose the cordovans as footwear. But according to C_{SF}, the cordovans only go with the white shirt. Contradiction. We can trace the effects of choosing the white shirt in the same way, also arriving at a contradiction.

We can ask why this situation has arisen. It is not because the robot or its designer have made a mistake in specifying or applying a model. We do not in fact know what has led the designer here, but it could be because the problem of choosing shoes has been composed with the problem of choosing shirts and trousers. Or it could be because the model can only express one uniform predicate that a certain item of clothing matches a certain other, with the same level of preference for each instance of this notion. Whatever the reason, we need to consider some way of relaxing or weakening the problem until solutions can be found. Various possibilities are discussed throughout this thesis.

2.7 Definitions

2.7.1 Valuations, solutions, and satisfaction

The definitions in this and subsequent subsections are not necessarily all standard, i.e. they may differ from usage elsewhere in the literature, but they are useful for our purposes. Some of the definitions and comments were kindly suggested by Michael Maher [Private Communication].

A **valuation** is an assignment of one value from its domain to each variable in a constraint problem.

A valuation is said to **satisfy** a constraint if the constraint is true in that valuation.

[1] In fact, unless there are elements in the domain of a variable which do not appear in any constraint, it is redundant to state individual variable domains explicitly: we can always reconstruct them by saying that $U :: \{i \mid (i, j) \in C_{UV}$ or $(k, i) \in C_{WU}$, for all $j, k, V, W\}$. Therefore the formal definition of the problem only includes the second of the two lines above.

A **solution** to a problem is a valuation which satisfies all the constraints in the problem.

A constraint problem may have many solutions which can be represented in shorthand using the various constraint symbols allowed in the input, and not just equality. For example CLP($\mathcal{R}$), which has the real numbers as one of its domains, expresses solutions using $<$, $\leq$, etc. The constraint problem $\{X > 3, X > 4\}$ has infinitely many solutions, which CLP($\mathcal{R}$) represents by $\{X > 4\}$.

2.7.2 Entailment, consistency, and contradiction

$\sigma \models c$ A set of constraints σ **entails** a constraint c, written $\sigma \models c$, if, in every model (valuation) where all the constraints in σ are true, c is also true.

$\sigma \not\models c$ A set of constraints σ **does not entail** a constraint c, written $\sigma \not\models c$, if there exists at least one model in which all the constraints in σ are true but c is not true.

$\sigma \wedge c \models \bot$ A set of constraints σ **contradicts** a constraint c, written $\sigma \wedge c \models \bot$, if there does not exist any model in which all the constraints in σ are true and c is also true[2].

$\sigma \wedge c \not\models \bot$ A set of constraints σ **is consistent with** a constraint c, written $\sigma \wedge c \not\models \bot$, if there exists at least one model in which all the constraints in σ are true and c is also true.

[**derived**] A set of constraints σ is **inconsistent** or **over-constrained** if it contains a subset $s \subseteq \sigma$ and a constraint $c \in \sigma$ such that s contradicts c. This can be written $\sigma \models \bot$, but in fact we will not subsequently need to use a symbolic description.

Note that the definition of 'does not entail' is not the same as the definition of 'is consistent with'.

We abuse language by stating that some individual constraint entails (or contradicts or is consistent with) another, when we should really say that the singleton set containing the constraint entails (or contradicts or is consistent with) the other.

As we require (Appendix A) that equality is part of the language of all constraint domains, an assignment can be represented as an equality constraint. Therefore a valuation can be considered as a set of constraints, and hence as a constraint problem. When considered as a problem in this way, a valuation has precisely one solution, namely itself. Consequently, we may say that a valuation (qua valuation) satisfies a particular constraint, or we may abuse jargon slightly by saying that a valuation (qua constraint set) *entails* a constraint, to the same effect.

As a valuation has precisely one solution, if a valuation is consistent with a constraint then it entails it, and if a valuation does not entail a constraint it must contradict it.

[2]We could also write this as $\sigma \cup \{c\} \models \bot$, but we choose not to, as union sometimes represents conjunction (as here, when we are combining problems) but sometimes represents disjunction (when combining solutions).

This is one of the reasons why we are often only concerned with the distinction between entailment and contradiction, and can ignore the other two definitions.

2.7.3 Metrics, metric spaces, metric comparators

A **metric space** is a mathematical concept arising in topology (see e.g. [59]). A metric space is defined to be a set $\mathcal{S}$ with a function d defined over $\mathcal{S} \times \mathcal{S} \rightarrow \mathbb{R}$ such that:

$d(x,y) \geq 0$: the distance between any two points is a non-negative real number

$d(x,y) = 0 \Leftrightarrow x = y$: the distance between two points x and y is zero iff they coincide

$d(x,y) \leq d(x,z) + d(z,y)$: the 'triangle inequality' holds — the direct distance between two points is never greater than the indirect distance via a third point

In the theory of Partial Constraint Satisfaction Problems (PCSP, see Chapter 4 and [31, 33]), Freuder and Wallace use the word **metric** to mean a function over the space of problems which returns some notion of the distance of one problem from another. One can see why they chose this name, but in fact it is slightly misleading, as they do not need the third condition listed above, and conceivably might not even need the second[3]. More importantly, this term might cause confusion with the HCLP notion of a 'metric comparator' (see below). Consequently, in this thesis in the context of PCSP we will use the term **distance function** instead of 'metric'.

In the theory of Hierarchical Constraint Logic Programming (HCLP, see Chapter 3 and [8, 83, 85]), Borning, Wilson, and others use the term **metric comparator** to mean a mechanism for comparing two solutions of a constraint problem, where the constraint domain has what they call a metric over it. (Again, the authors do not in fact need the triangle inequality to hold in the spaces which interest them.)

A distance function (PCSP-metric) measures the distance of one Constraint Satisfaction Problem from another in problem space, where the second is a relaxation or weakening of the first. An HCLP metric comparator measures how well a valuation satisfies a constraint, when the constrained variables range over a metric domain (a value space, as opposed to a problem space). Metric comparators are explained in more detail in our presentation of HCLP in Chapter 3; here we just to wish to emphasise that they are not the same as PCSP metrics.

2.7.4 Predicate and metric error functions in HCLP

In HCLP, comparators are divided into the two classes 'predicate' and 'metric', depending on whether they use a predicate 'error function' or a metric one. The concept of an error function captures the notion of the error of a valuation θ with respect to a

[3]Maher feels that these criticisms are unfair; the word 'metric' is often used simply to mean 'concerning measurement' without implying formalisation as a metric space [Private Communication].

particular constraint c, and is written as $e(c\theta)$ or $e(c,\theta)$. Error functions are defined as returning a non-negative real number; they return 0 only if the constraint is true in that valuation. In the literature, e is presented as having one argument $c\theta$, i.e. the result of applying θ to c. We prefer to think of e as taking two arguments, written $e(c,\theta)$, and then stating explicitly what it means to apply a valuation to a set of constraints. We do this below, separately for the two classes of error function. (It is just a minor aesthetic issue. [Sebastian Hunt, Private Communication].)

The simplest error function returns the value 0 if the constraint is satisfied by the valuation, and 1 otherwise. Borning and Wilson call this a 'predicate error function' which is slightly misleading as it is not a predicate in the logical sense; it returns 0 or 1, not true or false. However, we will continue to use their terminology to be consistent with the literature. The informal definition of this error function given by Borning and Wilson is completely clear, but is not backed-up with a formal one. Therefore we offer the following:

$$e(c,\theta) = \begin{cases} 0, & \text{if } \theta \models c \\ 1, & \text{if } \theta \wedge c \models \bot \end{cases}$$

As a valuation has precisely one model, the two cases in this definition are sufficient to cover all four definitions in the list presented in Section 2.7.2, as is discussed there.

If the constraint domain has a distance function defined over it which satisfies the first of the conditions for a metric space, and if a particular valuation θ does not entail some constraint c, then we can ask by how much c remains unsatisfied. This calls for the use of a 'metric error function', another notion which is defined clearly but informally in the standard literature. (We provide a formal definition in the next paragraph.) As an example, consider the constraint c = '$X \geq 3$' and the valuation θ = '$X = 1$'. If we wish to know how well or badly θ partially satisfies c, we must chose a valuation which completely satisfies c, and calculate its distance from θ. Formally speaking, in the previous sentence 'partially satisfies' actually means 'does not satisfy', and 'completely satisfies' just means 'satisfies', but the motivation for this usage is clear. c can be completely satisfied by many different valuations such as $X = 3$, $X = 4$, $X = 5$, etc., and so the error for the valuation $X = 1$ might be 2 (i.e. $3 - 1$), 3, 4 etc. We cannot choose the appropriate valuation *a priori*, but it is clear that in general we ought to take the minimum of the errors. (For example, when asking how far a point is from a line, we are assumed to want the distance to the *nearest* part of the line, i.e. the minimum distance.)

Another issue is that c may constrain more than one variable. We will denote the tuple of all the variables in c by $\mathbf{x}$ (bold x). Let $\theta|_{\mathbf{x}}$ and $\phi|_{\mathbf{x}}$ denote the projection of two valuations θ and ϕ respectively on to the variables in $\mathbf{x}$, and let ∂ be the distance between two points in valuation space. If $\mathbf{x}$ only contains one variable, we can imagine that $\partial(u,v) = |\,u - v\,|$, as was done in the example in the previous paragraph. But if there is more than one variable in c we must decide how to combine their individual distances. (For example, we could use the distance along each dimension separately, or we could consider the diagonal distance, etc.) Having chosen ∂, and considering all of the above, we can formally define the metric error function $e(c,\tau)$ as follows:

$$e(c,\theta) = min\{\partial(\theta|_{\mathbf{x}}, \phi|_{\mathbf{x}}) \mid \phi \models c\}$$

Different definitions for a number of the terms that we have defined in this section are given by Satoh and Aiba [69]; the net effect is similar to ours, but they have many differences of detail.

Chapter 3

HCLP — Hierarchical Constraint Logic Programming

The Hierarchical Constraint Logic Programming (HCLP) scheme of Borning, Wilson, and others [8, 83, 85] greatly extends the expressivity of the general CLP scheme [41]. A semantics has been defined for HCLP [83, 84] and some instances of it have been implemented [60, 83]. There is also related work by Satoh [68].

The presentation in this section is adapted from Wilson's thesis [83], which should be referred to for more details and examples. Other references are also available e.g. Borning, Freeman-Benson and Wilson [7], and Wilson and Borning [85]; the second of these includes a discussion of the logic programming aspects of HCLP, such as goal-reduction, disjunctions of clauses, and so on.

3.1 Formulation

The Hierarchical CLP scheme includes both required and optional constraints. The HCLP scheme is parameterised not only by the constraint domain $\mathcal{D}$ but also by the 'comparator' $\mathcal{C}$, which is used to compare and select from the different ways of satisfying the soft constraints.

An HCLP clause has the form

$$p(\mathbf{t}) \text{ :- } b_1(\mathbf{t}), \ldots, b_{m+n}(\mathbf{t}).$$

where the b_i's are a mixture of $l_i c_i$'s and q_i's, and where $\mathbf{t}$ is a list of terms, $p, q_1, \ldots, q_m$ are atoms and $l_1 c_1, \ldots, l_n c_n$ are labelled constraints i.e. constraints annotated with a strength level l. A program is a bag (multiset) of rules, and a query is a bag of atoms. The strengths of the different constraints are indicated by a non-negative integer label. Constraints labelled with a zero are *required* (hard), while constraints labelled j for some $j > 0$ are optional (soft), and are preferred over those labelled k, where $k > j$. A program can include a list of symbolic names, such as *required*, *strongly-preferred*, etc., for the strength labels, which will be mapped to the natural numbers by the interpreter.

If the strength label on a constraint is omitted, it is assumed to be *required*. As can be seen, this standard definition assumes that the strength levels are totally ordered, which is an assumption we will continue to make in the rest of this thesis. However, partially-ordered hierarchies are briefly discussed in Section 3.5.5.

Goals are executed as in CLP, except that initially non-required constraints are accumulated but otherwise ignored[1]. If there is more than one solution to a goal, the accumulated hierarchy of optional constraints is then solved, thus refining the valuations. The method used to solve the non-required constraints will vary from domain to domain, and for different comparators within a given domain.

The constraint store σ (a set) is partitioned into the set of required constraints S_0 and the set of optional ones S_i. The solution set for the whole hierarchy is a subset of the solution set of S_0, such that no other solution could be 'better', i.e. for all levels up to k, this solution is at least as good as all others, and for level S_{k+1} this solution is better, in terms of some comparator. Backtracking and incomparable hierarchies give rise to multiple possible solution sets, each a subset of the solution to S_0.

3.2 Comparators

3.2.1 Definitions

A *constraint hierarchy* is a finite bag of labelled constraints. Given a constraint hierarchy $\mathbf{H}$, $\mathbf{H}_0$ is a sequence of the required constraints in $\mathbf{H}$, in some arbitrary order, with their labels removed. $\mathbf{H}_1$ is a sequence of the strongest non-required constraints in $\mathbf{H}$ (with their labels removed), and so on up to the weakest constraints $\mathbf{H}_n$, where n is the number of non-required (optional) levels in the hierarchy. For completeness, Wilson also defines $\mathbf{H}_k = \{\}$ for all $k > n$.

A valuation function for a bag of constraints over the domain $\mathcal{D}$ maps the free variables in (some of) the constraints to elements of $\mathcal{D}$. A *solution* of a hierarchy is a set of valuations for all the free variables in the hierarchy. The first requirement of HCLP is that all the valuations in the solution set satisfy the required constraints. In addition, each valuation must satisfy the optional constraints at least as well as any other valuation in the solution set. In other words, there can be no valuation which satisfies the required constraints and which is "better" than the one being considered. There are a number of different ways to compare valuations, which give rise to different definitions of "better". The methods are called *comparators*. Solution sets and various comparators are formally defined in the following sections.

[1] This is an implementation detail but, while it is not essential, it helps us to understand the differing roles of required and optional constraints. Menezes et al. use an alternative 'optimistic' strategy [60].

3.2.2 Summary

Certain comparators can be used with any domain. For example, a 'predicate' comparator prefers one solution to another if it satisfies more constraints at some level (and an equal number of constraints at all previous levels). However if the domain has a metric (such as the real numbers) it is possible to ask how far from the preferred answer a solution is, in which case one might prefer fewer constraints to be exactly satisfied if the distance of the answer from a given point can be minimised. Weights can be used within a particular level of the hierarchy in order to influence the solution, but a heavily weighted constraint in a given level is completely dominated by the lightest constraint in any more important level. Wilson calls this property 'respecting the hierarchy' [83]. Because weights cannot have an effect outside their own level of the hierarchy, and in order to simplify the subsequent presentation, we will not consider them further in this thesis. They can, however, be easily incorporated into all our theories and formalisms.

In the rest of this thesis we mainly consider the *unsatisfied-count-better* (UCB) comparator, which is quite simple to understand and which can be defined over any domain[2]. Basically, one valuation is better than another if it leaves fewer constraints unsatisfied (or equivalently, if it satisfies more constraints).

Definition:
A solution θ is *unsatisfied-count-better* than a solution σ if it satisfies as many constraints as σ does in levels $1 \dots k-1$, and at level k it satisfies strictly more constraints than σ.

Quite a lot of the other work on HCLP considers a different comparator, *locally-predicate-better* (LPB), which is slightly less discriminating than UCB but is easier to implement in certain situations. Locally-predicate-better is concerned not just with the *numbers* of constraints satisfied by a particular valuation, but by the particular constraints themselves. See Section 3.5.1 for a more detailed comparison of LPB and UCB.

Definition:
A solution θ is *locally-predicate-better* than a solution σ if it satisfies every constraint that σ does in levels $1 \dots k-1$, and at level k it satisfies a strict superset of the constraints satisfied by σ. (If θ and σ satisfy different constraints then they are incomparable and both will appear in the solution set.)

The following sections contain a more detailed presentation of comparators.

3.2.3 Error functions

To compare valuations, HCLP begins by considering how well a particular valuation satisfies a single constraint. The error function $e(c\theta)$ returns a non-negative real number which indicates how well a valuation θ satisfies constraint c, where $c\theta$ denotes the result

[2]Thanks to Michael Maher for emphasising the difference between UCB and LPB and for noting that UCB is more appropriate for our work [Private Communication].

of applying the valuation θ to c. $e(c\theta)$ must have the property that $e(c\theta) = 0$ if and only if $c\theta$ holds. The simple error function which returns 0 if the constraint is satisfied and 1 if it is not can be used in any domain $\mathcal{D}$. Its formal definition was given in Section 2.7.4. A comparator that uses this error function is called a *predicate* comparator. If a domain is a metric space, other error functions can also be used, defined in terms of the domain's metric. (In fact, the restriction to metric spaces is too strong. See Section 2.7 for more details.) For example, the error for $X = Y$ could be defined to be $|X - Y|$ i.e. the distance between X and Y. The use of non-predicate error functions leads to *metric* comparators. Metric comparators are domain dependent, as their error functions may depend on the particular properties of $\mathcal{D}$.

The error function $\mathbf{E}(C\theta)$ maps e over a sequence of constraints $C = [c_1, \ldots, c_k]$:

$$\mathbf{E}(C\theta) = [e(c_1\theta), \ldots, e(c_k\theta)]$$

An *error sequence* is a sequence $[\mathbf{E}(H_1\theta), \mathbf{E}(H_2\theta), \ldots, \mathbf{E}(H_n\theta)]$, mapping $\mathbf{E}$ over each level of the hierarchy.

(The error v_i for the i^{th} constraint can be weighted by a positive real number w_i. This is where weights are taken into account in HCLP.)

3.2.4 Combining functions

The errors at a given level of the hierarchy can be combined in various ways when comparing valuations. These alternatives are defined as *combining functions.* A combining function g is a function which is applied to sequences of reals, and returns a single value that can be compared with other values using the associated relations $<_g$ and $<>_g$ ("neither less than nor greater than"). For example, g might sum all the numbers in the sequence, or perhaps select the maximum. HCLP requires $<_g$ to be irreflexive, antisymmetric, and transitive and $<>_g$ to be reflexive and symmetric. (The notation $<>_g$ is used instead of $=$ because the relation is not necessarily transitive. $<>_g$ indicates that two valuations cannot be strictly ordered by $<_g$. This may be because they are equal, but for other comparators it may be because they are incomparable.)

The combining function $\mathbf{G}$ is a raised version of g and is applied to error sequences. It returns a sequence of values each of which can be compared using $<>_g$ and $<_g$ with the appropriate position in another sequence. Sequences as a whole may be compared lexicographically. The output of $\mathbf{G}$ is called a *combined error sequence.* Let $R = [\mathbf{E}(H_1\theta), \ldots, \mathbf{E}(H_n\theta)]$. Then

$$\mathbf{G}(R) = [g(\mathbf{E}(H_1\theta)), \ldots, g(\mathbf{E}(H_n\theta))]$$

The lexicographic ordering $<_{LG}$ is defined on combined error sequences $u_1, \ldots, u_n$ and $v_1, \ldots, v_n$ in the standard way:

$$\begin{aligned} u_1, \ldots, u_n \quad <_{LG} \quad & v_1, \ldots, v_n \\ & \text{if } \exists\, k \in 1 \ldots n \text{ such that} \\ & \qquad \forall\, i \in 1 \ldots k-1 \cdot u_i <>_g v_i \wedge u_k <_g v_k \end{aligned}$$

(In certain circumstances later in this thesis, one of the sequences might be shorter than the other. In this case a sequence is always considered lexicographically earlier than another sequence of which it is a strict prefix.)

We consider a mathematical mechanism for avoiding the use of lexicographic orders over sequences in Section 3.5.4.

3.2.5 Solutions to constraint hierarchies

Using all the above definitions, we can now present Wilson's definition of the solution set S of a constraint hierarchy H, by using the comparator defined by the combining function g, its associated function $\mathbf{G}$, and the lexicographic order defined by $<_{L\mathbf{G}}$. S_0 denotes the solution to the required constraints i.e. ignoring the optional constraints. The set S, whose calculation is the aim of the whole process, is all the valuations in S_0 for which no better valuation exists, as designated by the lexicographic ordering $<_{L\mathbf{G}}$:

$$\begin{array}{rcl} S_0 & = & \{\theta \mid \forall\, c \in H_0 \cdot e(c\theta) = 0\} \\ S & = & \{\theta \mid \theta \in S_0 \wedge \forall\, \sigma \in S_0 \\ & & \neg(\mathbf{G}([\mathbf{E}(H_1\sigma), \ldots, \mathbf{E}(H_n\sigma)]) <_{L\mathbf{G}} (\mathbf{G}([\mathbf{E}(H_1\theta), \ldots, \mathbf{E}(H_n\theta)])))\} \end{array}$$

3.2.6 Comparator taxonomy

Wilson defines a number of comparators, each of which embodies a different way of defining the set of solutions to a constraint hierarchy. There are three main classes of comparator: *global*, *local*, and *regional*[3]. Remember that the error sequences for the constraints at levels $H_1, \ldots, H_n$ are compared using a lexicographic ordering, i.e. if a solution θ is better than a solution σ, this is because there is some level k in the hierarchy such that for $1 \leq i < k$, $g(\mathbf{E}(H_i\theta)) <>_g g(\mathbf{E}(H_i\sigma))$, and at level k, $g(\mathbf{E}(H_k\theta)) <_g g(\mathbf{E}(H_k\sigma))$.

A local comparator considers each constraint individually. A solution θ must do exactly as well as σ for each constraint in levels $1 \ldots k-1$, and at level k, θ must do strictly better for at least one constraint, and at least as well as σ for all the others. A global comparator combines the errors for all constraints at a given level using the combining function g, and only then compares two valuations. A regional comparator considers each constraint at a given level individually, similarly to a local comparator. But two solutions that are incomparable at a stronger level may still be compared at a weaker level, unlike a local comparator, leading to one being discarded. Therefore, in general a regional comparator will discriminate more than a local one.

We can define various classes of comparators, each using a specific combining function g and relations $<>_g$ and $<_g$. Then within each class, the choice of error function will specify the individual comparator.

[3]It would be more intuitive if the names global and regional were swapped, but we will stick to the standard HCLP usage.

The class of global comparators includes *weighted-sum-better*, *worst-case-better*, and *least-squares-better*; the constraint errors within each level are combined by taking the weighted sum, the weighted maximum, and weighted sum of squares respectively. Typical local and regional comparators form the classes *locally-better* and *regionally-better*.

For weighted-sum-better, $g(\mathbf{v}) = \sum_{i=1}^{|\mathbf{v}|} w_i v_i$. Worst-case-better has $g(\mathbf{v}) = \max\{w_i v_i \mid 1 \leq i \leq |\mathbf{v}|\}$ and least-squares-better has $g(\mathbf{v}) = \sum_{i=1}^{|\mathbf{v}|} w_i v_i^2$. In all three cases, $<_g$ is defined as for the reals, and $<>_g$ is equivalent to $=$ for the reals.

For locally-better, $g(\mathbf{v}) = \mathbf{v}$ and $<>_g$ and $<_g$ are defined as follows:

$$\begin{aligned} \mathbf{v} <_g \mathbf{u} &\equiv \forall\, i \cdot v_i \leq u_i \wedge \exists\, j \text{ such that } v_j < u_j \\ \mathbf{v} <>_g \mathbf{u} &\equiv \forall\, i \cdot v_i = u_i \end{aligned}$$

Regionally-better has $g(\mathbf{v}) = \mathbf{v}$, and $<>_g$ and $<_g$ defined as follows:

$$\begin{aligned} \mathbf{v} <_g \mathbf{u} &\equiv \forall\, i \cdot v_i \leq u_i \wedge \exists\, j \text{ such that } v_j < u_j \\ \mathbf{v} <>_g \mathbf{u} &\equiv \neg((\mathbf{v} <_g \mathbf{u}) \vee (\mathbf{u} <_g \mathbf{v})) \end{aligned}$$

Independent of the choice of a global, regional, or local combining function, an error function also needs to be selected. *Locally-predicate-better* (LPB) is locally-better using the simple error function that returns 0 if the constraint is satisfied and 1 if it is not. *Locally-metric-better* is locally-better using a domain-dependent metric for computing the errors for each constraint for each valuation. We can define *weighted-sum-predicate-better*, *weighted-sum-metric-better*, etc, in a similar way.

If we consider weighted-sum-predicate-better with weights of 1 on every constraint, we get *unsatisfied-count-better* (UCB). It simply counts the number of constraints which are left unsatisfied by a valuation. (The order induced by UCB is the same as the one we would get by counting the number of constraints *satisfied* by each valuation, and choosing the highest score. However, in general, we wish to consider scores as errors, and so we would like *lower* scores to indicate more preferred solutions.)

Wilson gives reasons why the unweighted predicate versions of the other two global comparators are not very useful, and so she chooses to use the shorter names worst-case-better and least-squares-better (LSB) to refer to the metric variants.

Wilson uses the terminology $better(\theta, \sigma, H)$ as shorthand for

$$\mathbf{G}([\mathbf{E}(H_1\sigma), \ldots, \mathbf{E}(H_n\sigma)]) <_{LG} \mathbf{G}([\mathbf{E}(H_1\theta), \ldots, \mathbf{E}(H_n\theta)])$$

This terminology makes it easy to express the relational properties of comparators: they are all irreflexive i.e. $\forall\, \theta \,\forall\, H \cdot \neg better(\theta, \theta, H)$. Many of them are transitive:

$$\forall\, \theta, \sigma, \tau, \forall\, H \cdot \; better(\theta, \sigma, H) \wedge \; better(\sigma, \tau, H) \rightarrow \; better(\theta, \tau, H)$$

The regional comparators are not transitive, because their $<>_g$ relations are not transitive. However, *better* will always be transitive when the comparator's $<>_g$ relation is.

All comparators must be defined so as to give rise to a *better* that respects the hierarchy i.e. any number of weaker constraints can be violated if the alternative is to violate a single stronger constraint. More formally, if S_0 contains a valuation that completely satisfies all the constraints up to level k, then all the valuations in S must satisfy all the constraints up to level k:

$$\text{if } \exists\, \theta \in S_0 \wedge \exists\, k > 0 \text{ such that } \forall\, i \in 1 \ldots k \;\forall\, c \in H_i \cdot c\theta$$
$$\text{then } \forall\, \sigma \in S \;\forall\, i \in 1 \ldots k\, \forall\, c \in H_i \cdot c\theta$$

3.3 The disorderly property of HCLP

Wilson discusses a very simple but powerful example in her PhD thesis [83], which shows the non-monotonic, hence non-compositional, nature of any variant of HCLP which respects the hierarchy.

Wilson defines the 'orderly' property as follows: Let P and Q be constraint hierarchies, and let $S_{\{P\}}(\mathcal{C})$ be the set of solutions to the hierarchy P when comparator $\mathcal{C}$ is used. Then $\mathcal{C}$ is *orderly* if $S_{\{P \sqcup Q\}}(\mathcal{C}) \subseteq S_{\{P\}}(\mathcal{C})$, where we use $\sqcup$ to stand for constraint text union (constraint conjunction). A comparator which is not orderly is *disorderly*.

Proposition 1:
Any comparator which respects hierarchies over a non-trivial domain $\mathcal{D}$ is disorderly.

Proof (Wilson [83, Section 2.5.3]):
Let P = {`weak X = a`} and Q = {`strong X = b`} for two distinct elements of $\mathcal{D}$, `a` and `b`. (The existence of distinct elements is what makes $\mathcal{D}$ non-trivial.) $S_{\{P\}}(\mathcal{C})$ will contain the valuation which maps `X` to `a`, and if $\mathcal{C}$ respects the hierarchy then $S_{\{P \sqcup Q\}}(\mathcal{C})$ will contain the valuation which maps `X` to `b`. So $S_{\{P \sqcup Q\}}(\mathcal{C}) \not\subseteq S_{\{P\}}(\mathcal{C})$, so $\mathcal{C}$ is disorderly. □

The disorderliness of HCLP is very important, as we shall see in Part IV where we discuss compositionality in more detail.

3.4 Example α in HCLP

Each of the binary constraints in the example called α in Section 2.6 are satisfiable in themselves. It is only when we combine them that we arrive at an over-constrained system. HCLP resolves this situation by changing the relationship between different constraints, by treating some of them as more important than others.

The specifier of the problem might be asked to order the constraints by how important they are. The result might be that it is an absolute requirement that the shirt matches the trousers whereas it is not a requirement that the trousers match the shoes, but it is a strong preference. It would be nice if the shirt could also match the shoes, but that is only a weak preference:

required C_{ST}, strong C_{FT}, weak C_{SF}

Using any comparator will give the same results in this case[4]: (r, c, g) (red shirt, cordovans, grey trousers) and (w, s, d) (white, sneakers, denim). The blue trousers b do not match any item of footwear and so are less preferred according to the strong constraint. The weak constraint contradicts the others and so is ignored.

Note that it is possible for HCLP to fail to resolve an over-constrained system, if the inconsistency arises between required constraints. This is a useful property to have in, say, safety-critical situations, where a requirement really is a requirement. But in optimisation problems, where we just wish to get a 'reasonable' or 'best possible' answer, a failure might be irritating. The obvious answer is not to have any required constraints, labelling them all 'very-strong', say, but this might have efficiency implications for certain implementations.

3.5 Discussion of certain issues in HCLP

3.5.1 Comparing two comparators — LPB and UCB

In this thesis, we mainly discuss HCLP's *unsatisfied-count-better* comparator (UCB), whereas most of the standard literature uses *locally-predicate-better* (LPB). In this section we compare them from two points of view; how well they capture various aspects of a problem, and ease of implementation.

LPB considers one solution σ to be better than another, θ, if for all levels of the hierarchy from 0 to $k-1$ they satisfy precisely the same constraints, and at level k σ satisfies a strict superset of the constraints satisfied by θ. If the sets of constraints satisfied by each solution are neither a subset nor a superset of each other, then the two solutions are incomparable and are considered to be equally good. So if 99 people want a meeting at 2 p.m. and just one person of equal rank wants a meeting at 3 p.m., both times are acceptable. In our opinion this is ridiculous. UCB, however, would offer 2 p.m. as the sole solution, since a valuation which satisfies 99 constraints will be preferred over a valuation only satisfying a single constraint[5]. When HCLP is

[4]This is because comparators select solutions which are best for each constraint *within* a particular level. As there is only one constraint at each level, its solutions will always be preferred. It is the hierarchy that is responsible for inter-level effects.

[5]In fact, as an implementation detail, if we are considering UCB with sets rather than bags of constraints, 99 separate instances of the same constraint would collapse into one. (See Section 6.1.2 for a discussion of why we use bags and not sets.) However other examples can easily be formulated with the same undesirable behaviour that we criticise here. Consider the following set of constraints

required $0 \leq X \leq 10$, strong $X = 2$, strong $X \geq 5$, strong $X = 6$

LPB will offer *two* solutions: $X = 2$ and $X = 6$. Using UCB gives the intuitively correct answer of a implemented in a standard SLD-based logic programming system which returns one solution at a time, it is quite possible that the first solution offered to the user in this situation will be 3 p.m.

In fact, this is a clue as to why LPB is used so much: it has a good fit with SLD logic programming languages and so is very easy to implement in a backtracking environment. Treating the optional constraints as though they were a special type of predicate (ignoring the 'constrain-then-generate' paradigm), consider the problem's search tree: each leaf will describe a consistent set of constraints. The set of preferred solutions will be the equivalence class of all equally good solutions, i.e. *all* the leaves of the three. Therefore, if only one solution is required, the implementation may as well pick the leftmost branch.

Many of the applications of HCLP in graphics do in fact only need one solution. (A graphics environment [6] provided the original impetus to the development of HCLP.) In graphics, the set of constraints specifies the relationship between various objects on the screen. When one of them is dragged using the mouse, the others must follow. This requires constant re-execution of the same set of constraints, and each execution must return the 'same' solution otherwise objects might jump around the screen. Borning and Wilson's HCLP(LPB) interpreters satisfy this requirement operationally, even though it is not captured by the semantics of HCLP. It would be harder to use UCB under this additional requirement. [Thomas Schiex, Private Communication].

With UCB each leaf will have an error score, the number of constraints at that level which it leaves unsatisfied, and the preferred solutions will be the equivalence class of all leaves with minimal scores. Even if only one solution is required, in principle the whole space must be searched. (In practice, one can imagine using branch-and-bound methods, but in the worst case this will not lead to any benefit.) In Prolog terminology, this process can be though of as taking the `setof` solutions and then picking the best. Clearly, using `setof` is less efficient than simply accepting the first leaf as is done by LPB.

We feel that the trade-off between efficiency and what we consider to be accuracy should not be hidden in the choice of comparator. If a solution is genuinely acceptable to the user even if it only satisfies one of the constraints at a level and other solutions might satisfy more, this choice should be made explicitly by defining a new comparator. This comparator might be called *unsatisfied-count-reasonable* or perhaps *unsatisfied-count-n*. Clearly, for $n = 1$ only one branch of the search tree would need to be examined.

The meeting example partly explains why we prefer UCB: it gives what we consider to be the intuitively correct results. Related to this, UCB is sensitive to whether the collections of constraints are sets of bags. If bags are used, composition of preferences from different problems can be defined precisely, whereas LPB is insensitive to the choice. Composition, and our choice of bags instead of sets as in standard HCLP, are discussed elsewhere in this thesis.

However, we recognise that the situation is not black-and-white; one advantage of LPB

single solution $X = 6$. We will continue with the example in the main text as it makes LPB's behaviour so clear.

is that in certain circumstances an SLD-based implementation could provide the user with the first solution even if certain other parts of the computation, further to the right in the tree, would be infinite. An implementation of UCB could not tolerate such a situation. However, we feel that this advantage is not sufficiently important to override our other concerns.

3.5.2 Implementations of HCLP

DeltaBlue [29, 67] and SkyBlue [66] are designed for an HCLP approach to user-interface construction and are efficient for repeated solving of a relatively simple set of constraints (e.g. when a mouse moves the end of a line). But DeltaBlue cannot cope with multiple output variables or with cycles (see below), and SkyBlue copes with them only be invoking a separate solver (for example, one based on Gaussian elimination or the Simplex method). DeltaBlue has been shown to be optimal for its task by Gangnet and Rosenberg [35]. Both DeltaBlue and SkyBlue provide one solution chosen non-deterministically, as opposed to our work which provides all solutions. (Borning has said [Private Communication] that in general his applications only *need* the first solution.)

The notion of a cycle arises due to the operational nature of the solving attempted by these algorithms, in which a constraint such as $A = B + C$ is considered to be solvable in three different ways (by three different 'methods') namely $A{:=}B + C$, $B{:=}A + C$, and $C{:=}A + B$. Then the two constraints $A = B + C$, $B = C + D$ may or may not form a cycle, depending on the particular methods chosen to solve them. Most constraint problems outside the domain of user-interfaces will contain many cycles, and anyway one of the key advantages of the constraint paradigm is its declarative nature.

3.5.3 Labelling predicates as well as constraints

In this thesis, we are really concerned with constraint solving and not constraint logic programming (see Section 2.1). This split is acceptable because, we claim, solving is mostly orthogonal to programming. This would not be completely true if we allowed strength labels on predicates as well as on constraints. Standard HCLP, as introduced earlier in this section, assumes all predicates are *required* and only constraints can be optional. In the rest of this subsection, we will consider two possibilities for augmenting HCLP with labelled (optional) predicates. We do not develop this further in the rest of this thesis, as it is not necessary, as is shown in the next section.

3.5.3.1 Source level transformation

In standard HCLP a rule is of the form

$$a :- l_1 c_1, l_2 c_2, a_1, a_2, a_3.$$

where $l_1 c_1$ is a labelled constraint, but a_1, a_2 etc. are *unlabelled* atoms / calls to predicates. It might seem that this restriction limits the expressiveness of the language. But this is not generally the case, as can be shown by the following example[6]:

Consider the facts

```
is_happy(fred).
is_happy(ann).
is_married(fred).
```

We can ask if the query

```
?- ...is_happy(X), strong is_married(X) ...
```

can always be re-written

```
?- ...is_happy(X), is_married(NewVar), strong NewVar = X, ...
```

where the call to `is_married` is required to succeed, but *will* always succeed because it involves a brand new unconstrained variable.

The answer is 'yes', except in certain error situations: consider the case where there are no clauses defining `is_married(AnyVar)`, or if it is defined as

```
is_married(X) :- fail.
```

The transformed query will fail, even though the original query would not. This problem can be avoided if we are willing to transform the program as well as the query [Jean-Marie Jacquet, Private Communication]. Assuming an SLD computation rule, if we change the code for any predicate which has a strength label by adding a cut '`!`' to the end of the last clause in its definition, and then adding another clause afterwards, we can avoid failure in this case.

```
is_married(X) :- fail,!.
is_married(X) :- true.
```

If we do not wish to transform programs in this way, then the programmer must guarantee that he will avoid pathological situations, in which case our query transformation is sufficient.

Therefore the mechanism for transforming away any strength-labelled predicates is very simple: consider a predicate P, labelled with strength $\mathcal{L}$, with k arguments variables called $x_1, \ldots, x_k$. We must **invent** k new variables, say $y_1, \ldots, y_k$, not used anywhere else in the program. Then **delete** the strength label from the predicate P and **swap** new variables for old. Finally **create** k new equality constraints, each labelled with the strength $\mathcal{L}$, equating corresponding variables '$\mathcal{L}\ x_i = y_i$'. So

$$\mathcal{L}P(x_1, \ldots, x_k) \quad \text{becomes} \quad P(y_1, \ldots, y_k), \mathcal{L}x_1 = y_1, \ldots, \mathcal{L}x_k = y_k.$$

The program transformation is as follows: for each predicate P labelled with some strength-label, **re-write** the final clause of P by adding a cut after the last goal. Then **add** another clause, with body `true`.

[6]Due to Alan Borning [Private Communication]

Variables which are arguments to one predicate will often be arguments to another one as well. If we have two predicates $P_1(x)$ and $P_2(x)$ with different strength levels $\mathcal{L}_1$ and $\mathcal{L}_2$, and assuming that y and z are new variables, then after the transformation we will have

$$P_1(y),\ P_2(z),\ \mathcal{L}_1\, x = y,\ \mathcal{L}_2\, x = z.$$

Allowing predicates to be labelled as well as constraints requires redefinition of the semantics of comparators. For example, before the above transformation, violating $\mathcal{L}P(x_1, \ldots, x_k)$ would increase the UCB error by 1. After the transformation, the predicate $P(y_1, \ldots, y_k)$ would be guaranteed to succeed, but possibly more than one of the new equalities $\mathcal{L}x_1 = y_1, \ldots, \mathcal{L}x_k = y_k$ would be violated, leading to a different UCB error. Any redefinition of the comparators would have to take this into account. [Pierre-Yves Schobbens, Private Communication].

3.5.3.2 Min-strength algebras

In the example above, what if one of the *rules* defining `is_married` contains optional constraints? Then we have what Borning has called a min-strength algebra: "the strength of any subconstraint is the minimum of its strength and the strength on the original goal." [Private Communication].

Borning feels that the choice of a min-strength algebra or a simpler solution[7] should be made pragmatically. "The goal of putting strengths on things is to allow you to express preferences, and you need some reasonable, simple, well-defined way to define how they interact so that the programmer can use this to describe the problem to hand." Our choice in this thesis is to assume that the programmer will implicitly do source-level transformation, taking responsibility for resolving unusual situations such as those mentioned above.

These issues have been addressed to some extent by Borning and Freeman-Benson in their work on Kaleidoscope [27, 28]. There are also links between this issue and some work in non-monotonic logic — see e.g. Brewka [11].

3.5.4 Using ordinals instead of the lexicographic ordering

At certain stages in this thesis, integers alone are not enough to order various items, such as valuations or subsets of a set of constraints. We have to use *sequences* of integers, which necessitates the use of a lexicographic ordering. This is similar to their use in standard HCLP (see Section 3.2, especially Section 3.2.4). If we use $<_L$ to indicate a lexicographic order, we required that $[a, b] <_L [c, d]$ if $a < c$ or if $a = c$ and $b < d$, et cetera. However, there is an alternative, which we describe in this section.

[7]For example, Borning suggests addition: the words 'strong', 'medium', 'weak' etc. are just labels referring to strength levels 1,2,3. So one solution is just to sum the numbers. This is simple to understand (although probably no simpler than taking the minimum of the numbers). Borning does not claim that it has any semantic justification, and we agree that it is conceptually suspect.

Simplistically, the ordinal numbers are defined to allow orderings beyond 'infinity', which is denoted by ω (omega). See e.g. [37] for a formal presentation. It is known that there is an isomorphism between the lexicographic order over sequences of integers, and the ordinals with their standard order. The isomorphism is established by considering an encoding which, for example, converts the sequence $[a, b]$ into the ordinal $a\omega + b$. In general, a sequence of integers of length $k + 1$ such as $[n_k, n_{k-1}, n_{k-2}, \ldots, n_2, n_1, n_0]$ can be represented by the ordinal $n_k\omega^k + n_{k-1}\omega^{k-1} + n_{k-2}\omega^{k-2} + \ldots + n_2\omega^2 + n_1\omega + n_0$. The order over the ordinals so constructed is the same as the lexicographic ordering of the original sequences.

3.5.5 Partially ordered hierarchies and 'sequences'

For a brief overview of the theory of partial orders and lattices, see Section 6.2.

Borning and Wilson briefly mention the subject of partially ordered hierarchies in [7, pp. 242–244]. Instead of having `required`, `strong`, `weak`, one can have `required`, {`strong`, `tough`}, {`weak`, `wimpy`} etc., with the additional fact that `strong` is stronger than `weak`, `tough` is stronger than `wimpy`, but `strong` and `tough` are incomparable, as are `weak` and `wimpy`. They suggest that the solution to such a hierarchy is the union of all the solutions to the different possible flattenings of the partial order into a total order. In other words, calculate the solution on the assumption that the total order is `required`, `strong`, `tough`, `weak`, `wimpy`, then calculate the solution on the assumption that the total order is `required`, `tough`, `strong`, `weak`, `wimpy`, etc etc, then union all the solutions together. Quite a lot of work!

However, irrespective of how the solution is defined, it is clear that we can indeed imagine partially ordered hierarchies. Then, when calculating HCLP solutions, instead of comparing two (totally ordered) sequences (pointwise) using a lexicographic order to see which is more preferred, we must compare two partially ordered sets. To do this we can use a variant of the following 'natural' order:

$$(\sigma_1, \tau_1) \leq_{S \times T} (\sigma_2, \tau_2) \quad \text{if } \sigma_1 \leq_S \sigma_2 \text{ and } \tau_1 \leq_T \tau_2$$

This approach can be extended to our own work: in Chapter 7, when transforming a constraint hierarchy into a PCSP problem, we calculate a set of distance functions, one for each level of the hierarchy, and create a single unified distance function by placing the results of all these individual functions in a lexicographic sequence. Clearly, we could instead construct a partially ordered set of functions, and compare two possible solutions pointwise using the natural combination of all the results for the individual functions.

Similarly, in our integrated framework for HCLP and PCSP in Chapter 11, we lexicographically order the results of our $[\![\_]\!]$ operator. Again, this work could be easily extended to cover partially-ordered hierarchies and distance functions.

We will not pursue this possibility further, as it reduces the clarity of the presentation of the formalisms.

Chapter 4

PCSP — Partial Constraint Satisfaction Problems

All this section is taken from Freuder [31]. See also [33].

Freuder has developed a theory of Partial Constraint Satisfaction Problems (PCSPs) to weaken systems of constraints which have no solutions, or for which finding a solution would take too long. Instead of searching for a solution to a complex problem, Freuder says we should consider searching for a simpler problem which we know we can solve.

Freuder formalises the notion of a Partial CSP by considering three components

$$\langle (P, U), (PS, \leq), (M, (N, S)) \rangle$$

where P is a constraint satisfaction problem (see Section 2.2.3), U is a set of 'universes' i.e. a set of potential values for each of the variables in P, $(PS, \leq)$ is a problem space with PS a set of problems and $\leq$ a partial order over problems, M is a 'distance function' on the problem space, and (N, S) are necessary and sufficient bounds on the distance between the given problem P and some solvable member of the problem space PS.

A solution to a PCSP is a problem P' from the problem space and its solution, where the distance between P and P' is less than N. If the distance between P and P' is minimal, then this solution is optimal.

The necessary and sufficient bounds may be ignored, in which case the sufficient bound on the distance between the original and relaxed problems is assumed to be 0, and the necessary bound is infinity. In fact, only M and one of P and PS are generally important. (P and PS can be derived from each other.)

4.1 The problem space

A problem space *PS* is a partially-ordered set of CSPs where the order $\leq$ is defined as follows and *sols(P)* denotes the set of solutions to a CSP called P:

$P_1 \leq P_2$ iff $sols(P_1) \supseteq sols(P_2)$

Note that the ordering is over *problems*, but defined in terms of *solutions*. The problem space for a PCSP must contain the original problem P, which can provide the maximal element in the order, for standard problem spaces. In the most general case, *PS* can in fact contain Q such that $P \leq Q$ or such that P and Q are incomparable. But if we take the conjunction of all the constraints in all the problems in *PS* and create a single problem R, then R will definitely be the greatest element in the order. If P has no solutions, then *sols(P)* $= \{\}$, which is a subset of all other sets.

The obvious problem space to explore when trying to weaken a problem is the collection of all problems Q such that $Q \leq P$, but it may also be useful to consider only some of these Qs, i.e. those problems which have been weakened in a particular way which makes sense in the context of the system that we are trying to model.

4.2 Weakening a problem

There are four ways to weaken a CSP: (a) enlarging the domain of a variable, (b) enlarging the domain of a constraint, (c) removing a variable, and (d) removing a constraint. Consider the following example due to Freuder: if none of your shirts match your tie, you could buy a new tie (variable domain enlargement / augmentation), you could decide that a certain tie does, after all, go with a certain shirt (constraint augmentation), you could decide not to wear a tie (variable removal), or you could ignore clashes between ties and shirts (constraint removal). (As a comparison with these four methods, in HCLP we could decide that the constraint that shirts match ties is simply not very important.)

Freuder shows that these can all be considered in terms of (b) above i.e. enlarging constraint domains (adding extra pairs to the relation which defines the constraint). (a) As we have already decided to consider the domains of variables as binary constraints c_{xx}, domain enlargement can clearly be achieved by constraint augmentation. (d) Enlarging a constraint c_{xy} until it equals $x \times y$ (the cartesian product of the domains) has the same effect as removing it altogether. (c) Removing all the constraints on a variable achieves the aim of removing the variable itself. See [31].

4.3 One augmentation can create multiple solutions

It it possible to augment one constraint with one extra tuple and yet increase the number of solutions by more than one. For example, consider the three constraints $XY = \{(a, e), (b, e)\}$, $YZ = \{(d, f)\}$, and $XZ = \{(a, f), (b, f), (c, f)\}$. There are no solutions. Adding the single tuple (c, d) to the domain of the constraint XY will give rise to the solution $X = c, Y = d, Z = f$. But if instead we add the single tuple (e, f) to the domain YZ, we would immediately jump to having two solutions: $X = a$, $Y = e, Z = f$ and $X = b$, $Y = e, Z = f$.

This is similar to imagining two routes from London to Oxford, but none from Oxford to Birmingham. Adding just one road from Oxford to Birmingham means that there are suddenly two ways to get from London to Birmingham.

4.4 The distance function

Different distance functions[1] are possible, but one obvious one is derived from the partial order on the problem space. If $M(P, P')$ equals the number of solutions not shared by P and P', then when $P' \leq P$ the distance function measures how many solutions have been added by the relaxation of P. We call this the 'solution-subset' distance function. Another distance function is a count of the number of constraint values not shared by P and P', i.e. the number of augmentations of P needed to get to P'. This is referred to below as the 'augmentations' distance function.

A third distance function called MaxCSP is defined as seeking a solution "that satisfies as many constraints as possible" [33]. MaxCSP was studied extensively in the longer journal article on PCSP by Freuder and Wallace [33], and is in some ways the most important distance function, or the one that is most closely identified with PCSP. The journal article shows how PCSP(MaxCSP) fits into the framework of arc-consistency and other standard CSP algorithms.

Other distance functions can be defined, including ones based on HCLP-like strength labels. Furthermore, Freuder suggests that a distance function may be used which will tend to find weakened problems with certain properties, for example one whose constraint graph has a certain structure which makes solving it easier (e.g. see [20, 30, 32]).

4.5 Example α in PCSP

In Section 2.6 we presented an example, α, of choosing which clothes to wear, and then relaxed the problem by using HCLP in Section 3.4. We will now consider the same example as a PCSP. Remember that the original problem had no solutions, and was modelled as follows:

$$S :: \{r, w\},\ F :: \{c, s\},\ T :: \{b, d, g\}$$
$$C_{ST} :: \{(r, g), (w, b), (w, d)\},\ C_{FT} :: \{(s, d), (c, g)\},\ C_{SF} :: \{(w, c)\},$$

As mentioned in the discussion of PCSP, we only need to consider weakening caused by constraint augmentation. If we add a pair (x, y) to a constraint between variables

[1] Freuder uses the term 'metric' instead of 'distance function'. We explain why we have changed this in Section 2.7.

U and V, then we implicitly assume that x is either already in the domain of U or is added to it, and y is in, or is added to, V.

Initially, we will only augment constraints with pairs whose elements are in the domain of the variables concerned. Even with this restriction, we can still weaken the problem in a number of different ways. For example, consider the following variants on the original constraints (additional pairs are in bold):

$C'_{ST} :: \{(r,g),(w,b),(w,d),(\mathbf{w},\mathbf{g})\}$ $\quad C'_{FT} :: \{(s,d),(c,g),(\mathbf{s},\mathbf{b})\}$
$C''_{ST} :: \{(r,g),(w,b),(w,d),(\mathbf{r},\mathbf{b})\}$ $\quad C''_{FT} :: \{(s,d),(c,g),(\mathbf{c},\mathbf{d})\}$
$C'''_{ST} :: \{(r,g),(w,b),(w,d),(\mathbf{w},\mathbf{g}),(\mathbf{r},\mathbf{b})\}$ $\quad C'''_{FT} :: \{(s,d),(c,g),(\mathbf{s},\mathbf{b}),(\mathbf{c},\mathbf{d})\}$

$C'_{SF} :: \{(w,c),(\mathbf{w},\mathbf{s})\}$
$C''_{SF} :: \{(w,c),(\mathbf{r},\mathbf{s})\}$
$C'''_{SF} :: \{(w,c),(\mathbf{w},\mathbf{s}),(\mathbf{r},\mathbf{s})\}$

Note that the first two variants of each constraint, C' and C'', have one additional pair, while the third variant contains two additional pairs. The set of augmentations we have made is incomplete; other possibilities have not been enumerated for reasons of space. The largest variant of a constraint C_{UV} is $U \times V$, the cartesian product of the individual domains. If all the constraints are replaced by their cartesian products, (i.e. removed), we would expect to get 12 solutions (as there is a choice of two possibilities for two of the types of clothing, and three for one of them).

However, if we calculate all the possible solutions to all the combinations of different weakenings, we will get more than 12 because of duplicates. This is because any solution involving C''' will also appear in either C' or C''. But this is intentional: the PCSP problem space does indeed contain problems defined in terms of all three weakenings of each constraint C. In this case, we would expect more than 500 solutions when duplicates are counted.

Assuming a simple distance function, namely that we prefer solutions involving the smallest total number of augmentations, we discover that there are five equally good solutions, each with just one of the constraints receiving one extra pair of values. One solution is $(S,F,T) = (w,s,d)$, i.e. we can find a set of matching clothes if we decide that the white shirt does after all match the sneakers. This solution involves $C'_{SF} :: \{(w,c),(\mathbf{w},\mathbf{s})\}$. The other four solutions are (w,c,b), (w,c,d), (w,c,g), and (r,c,g), including some involving augmentations not listed above.

Chapter 5

The Ideal System

5.1 Disadvantages of current methods

There are various methods for resolving over-constrained systems in CLP and CSP, infeasible systems in linear programming in OR, and some of the knowledge representation formalisms of AI. They have different characteristics, and some of them have been the subject of quite a large amount of work, leading to the existence of robust implementations. Therefore they have many advantages. However, they also have certain drawbacks. These are discussed here in the context of each system; their negations are summarised alongside their advantages in a later section which lists the characteristics of the ideal system.

- In linear programming, Chinneck and Dravnieks have done some work on what they term IIS's (infeasible systems of linear equations and inequalities) [14, 15]. They relax each inequality i by adding a distinct new variable ϵ_i to it, and replace the original optimisation function with one which minimises the sum of the ϵ's. Any non-zero ϵ in the answer indicates one member of the minimal infeasible subset. If that ϵ is then removed and the system solved again, another member of the infeasible subset is identified. When all have been enforced in this manner, the next attempt fails. In fact their method is more general than this, in order to cover cases when there is more than one independently inconsistent subset, and in order to include equalities. Drawbacks of the approach are that it
  - produces answers which are **hard to understand** — all the minimally inconsistent subsets are produced, which are hard to understand as they are somewhat out of context. Taking all the maximally consistent subsets instead would produce a flood of very similar solutions.
  - is **computationally expensive** — if the system as a whole contains n constraints and the minimal infeasible subset contains s of them, it is necessary to solve $s+1$ different problems each of size n. Solving one of these problems is hard enough (the theoretical complexity is exponential; in practice most of the time much better results can be achieved, but it is still quite expensive),

and yet the worst case is when $n = s$ and so one must solve $n + 1$ problems of size n.

- The process supported by the system described by Maher and Stuckey in their paper "Expanding Query Power" [58] requires continuous user interaction while the solver is running. This interaction cannot be automated therefore the **human user does too much work** when interacting with the computer.

- The disadvantages of HCLP are considered in detail below (Section 5.3). One of them is that HCLP requires the user to label every constraint in advance; there might be thousands of them and therefore the **human user does too much work** before starting to interact with the computer. Furthermore, correct labelling requires a great deal of human expertise. HCLP is declarative, which is very useful, but not compositional.

  This issue of being required to consider the correct strength label for every single constraint could be resolved in another way, which we briefly mention in Section 5.4.

- PCSP requires the user to define the distance function. Occasionally this will be straightforward, especially if the user has no preferences for certain subsets of the constraints or variables, but just wants a global property satisfied. But in all other cases it may be difficult to define the appropriate function, just as it is difficult in OR to find the correct objective function. Therefore it may be **hard to model problems**. Furthermore, when it is found, such a function may not be declarative. This makes it much **harder to maintain** the model when the problem evolves. Other disadvantages of PCSP are discussed in Section 5.3.

In their survey paper, Jaffar and Maher briefly discuss preference systems [42]. They criticise some of the OR systems for having objective functions with non-logical behaviour. (They also mention some work which addresses this issue.) Jaffar and Maher feel that the two approaches of HCLP and OR each have advantages compared to the other, in certain areas, but neither is completely acceptable in general. The OR approach is useful when there is an obvious choice of objective function, but often this is not the case; therefore it may be difficult to represent one's intended preferences. HCLP provides a more abstract method of specifying preference (more declarative than encoding preference in some objective function) but it is harder to 'fine-tune' [to achieve some general or global objective], say the authors, and may produce far too many maximally preferred answers. Also, it is hard to detect inconsistencies among these preferences.

5.2 Characteristics of the ideal system

It is hard to define the 'perfect' preference system, if indeed such a thing exists. However, we now present a reasonably complete list of the features we think would be useful.

Important characteristics:

1. enables the declarative expression of preference (equivalent to: does not require machine code level of detail to specify distance functions). Declarative behaviour makes it easier to maintain systems, and to verify that they satisfy their specifications.
2. takes account of any user-provided strength labels present. Then the airline staff in Richards' example (Section 2.5, page 17) would have been able to encode their knowledge of local relaxations.
3. does not require all strength levels to be stated
4. the theory behind the system is compositional

Other characteristics:

5. output is easy to understand. The user would not be able to choose from a flood of n different relaxations, each containing one relaxed and $n-1$ unrelaxed constraints in the original, as may be the outcome with the linear programming approach mentioned previously.
6. allows the expression of required constraints which must not be violated. For example, when composing some English electrical equipment with an American plug, it is better that the equipment does not work than that it works but electrocutes someone. So the safe limits of the equipment should be treated as required constraints.
7. can cope when the set of default[1] constraints is itself over-constrained e.g. when the specifier of the original system had not considered the possibility of composition.
8. allows the relaxation of the general problem structure (as opposed to changing the meaning of individual constraints).
9. allows the relaxation of individual constraints (the opposite of item 8)

Hard-to-quantify characteristics:

Severity: With respect to a mistake in specifying the system leading to an over-constrained situation, there is a characteristic which preference systems might have which is quite hard to quantify, namely 'severity of response to errors'. If a mistake

[1] In HCLP, unlabelled constraints are assumed to be required. This syntactic convention is to preserve backwards-compatibility with CLP. It has the effect that an over-constrained system of unlabelled constraints cannot be resolved by HCLP; failure will occur just as in CLP. Therefore standard HCLP does not possess the characteristic in this list item. However, it is clear that a very minor change of convention would rectify this. In PCSP, of course, the default is that all constraints can be relaxed, which is equivalent to a convention that unlabelled constraints are all at the same non-required strength level.

is made when specifying the system and so an incorrect constraint is added to the representation, if it is weak HCLP will probably ignore it. If it is strong, HCLP will give it high preference, and so the specifier is likely to become aware of it. In PCSP with the standard distance function, all constraints are treated equally, and so it is possible that the error will remain unnoticed longer than in the HCLP case. We feel, therefore, that HCLP is more severe than PCSP; usually this is beneficial as we would like to be aware of our mistakes.

Implementability: There is no point in defining the perfect preference system if it is not possible to implement it, or if all implementations are guaranteed to be unusably inefficient. Both HCLP and PCSP are so general that it is possible to imagine parameters that would make them inefficient, for example a poor choice of comparator or distance function. But both of them have instances which have already been implemented in a sufficiently efficient manner [29, 33, 60]. Indeed Sannella has produced the SkyBlue system which contains an HCLP solver sufficiently efficient to be used for interactive graphics [66]. GOCS, our framework for preference systems which can combine all the advantages of HCLP and PCSP, is more general than HCLP and PCSP, and so poor choices of its various parameters will be detrimental to efficiency. However, appropriately chosen instances should in principle be as efficient as HCLP or PCSP.

As an example of a system which might be said to have poor implementability, consider Satoh's work on constraint hierarchies [68]. One of the criticisms that has been made of his work is that it is based on an undecidable fragment of second-order logic. In fact, first-order logic is itself only semi-decidable, so it is not clear whether the pertinent criticism of Satoh's work is decidability *per se*, or its implications for implementation. Note that his theories *have* in fact been implemented as part of the ICOT programme [69]. In fact, completely unimplementable systems are unlikely to be publicised.

5.3 Analysis of HCLP and PCSP

If we examine HCLP with the above characteristics in mind, we can see that it possesses **1**, **2**, 5, 6 and 8, as well as being more severe on errors than PCSP, and equally implementable, at least in principle. Note therefore that HCLP possesses two out of the four most important characteristics (printed in bold type).

PCSP benefits from the following characteristics in full: **3**, **4**, 7, and 9. Therefore PCSP possesses the other two of the four important characteristics.

PCSPs output may sometimes be easy to understand (characteristic 5), but this is not always the case. For example, if the problem contains n mutually inconsistent constraints and the distance function minimises some global property such as total number of constraint augmentations, the user is likely to be presented with n equally acceptable solutions. This may or may not be a 'flood', depending on the size of n, but certainly choosing among them may be very difficult[2].

[2] In one case we examined, a slightly larger version of Example α with four constraints, there were 16 equally good solutions. Thus the user was required to discriminate between 16 solutions, even though the original problem only contained four constraints.

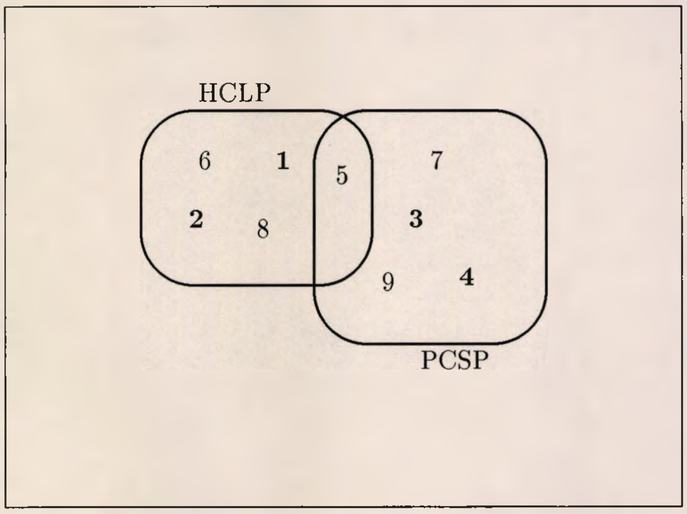

Figure 5.1: HCLP and PCSP have different advantages

HCLP can also suffer from this problem in theory, but only when all n constraints are at the same level of the hierarchy and a predicate comparator is used. These two conditions must occur simultaneously, and the less likely this situation is, the more HCLP benefits by comparison with PCSP on this point. GOCS allows but does not require the use of different strength levels, and so a flood can always be reduced by additional labelling in the relevant area, without needing to label every single constraint in the problem.

Proponents of PCSP might claim that it also possesses important characteristic 2, i.e. it can take account of any user-provided strength labels present, but this is only true if the distance function specifically highlights particular constraints. In fact it is reasonably easy to select particular *variables* and give them a large priority in the distance function, by giving them a very large score, but it is difficult to select particular *constraints* in this way. This is also why we claim that standard PCSP does not allow the expression of required constraints which must never be violated (characteristic 6). Conversely, in HCLP it is easy to give all *constraints* equal importance, e.g. label them all strong, but it is difficult to treat all *variables* symmetrically.

Summary: HCLP possesses characteristics **1**, **2**, 5, 6 and 8. PCSP possesses **3**, 4, (5), 7 and 9. We will claim in Part V that implementations of GOCS could possess *all* the separate advantages of these two systems.

5.4 Constraint schemas — one possible approach to one of the disadvantages of HCLP

In this section we briefly describe a possibility which we have *not* followed in this thesis. Our reasons for not doing so are mentioned towards the end of the section.

HCLP has many good characteristics. But one of its disadvantages is that the user has to label every single constraint. An obvious method of avoiding this difficulty would be to have patterns or 'constraint schemas' so that every constraint with a particular pattern is given the same strength label. For example:

`Var` $=$ `Num`	are required
`Var` $=$ `Num` $+$ `Var`	are strong
`Var` $\geq$ `Num`	are strong
`Var` $\geq$ `Num` $*$ `Var` $*$ `Var`	are weak
...	

This example is motivated by the observation that some equalities between a single name and a single number are not really constraints but are definitions of natural constants such as $\pi = 3.141$.

Another heuristic would be to have a list of variables: any constraint containing one or more of these variables is to be labelled 'required' while all the other constraints are optional. A variant of this could be related to the issue of composition of two problems: any constraint containing variables only in one of the two problems can be considered weaker than any 'glue' constraint containing variables from both. (Or the opposite course could be adopted, in which the glue is considered less important than the properties of the individual systems.)

The approach suggested here would mean that an expert in a given domain would have to create a set of constraint schemas just once, and then would not have to subsequently label all the constraints in individual problems.

(It might be necessary to put all the constraints into some canonical or normal form before applying schemas to them, but this is straightforward.)

This suggestion has the advantage of simplicity, but we did not follow up this idea in this thesis for two reasons: (a) it seemed unlikely to work, (b) even if the approach *was* successful, we would only have improved HCLP with respect to *one* of the characteristics of the ideal preference system. We now consider these two reasons in more detail.

(a) It seems difficult to discover a set of schemas which capture exactly or even approximately the labellings that we require. This is partly because there are so many different ways of modelling a problem using constraints. For example, in finite domain CLP, the day of the week on which an event occurs will probably be represented by a single variable ranging over seven possible values. In integer programming, however, there would be seven variables each with domain [0,1], representing the truth or falsity of a statement such as "The event is on Monday". Schemas written on the assumption that the first of these two representations was going to be used would not work for the second.

Furthermore, in the case of composition of two self-consistent systems which contradict each other, the constraints which clash are likely to have the same form or pattern. In other words, the infeasibility will arise *within* one of the schema classes. For example, in the case of electrical equipment, the British system will contain the constraint `Voltage`

`= 240` and the American system will contain the constraint `Voltage = 110`. These two constraints will belong to the same schema class irrespective of which of the heuristics mentioned above is used. Of course these problems can be overcome by adding strength labels to particular constraints manually, but we feel that this is only acceptable if the number of constraints needing this treatment is small.

(b) With respect to the list of characteristics of the ideal system, the approach mentioned here would only have one extra advantage over HCLP, namely not needing to label every constraint. Therefore there would still be a number of characteristics that it would not have:

Schemas will not detect mistakes in applying a given schema model to some situation for which it was not designed — there will always be a default labelling for constraints not covered by one of the schema patterns. Also, semantically different constraints which happen to have the same pattern as a schema will not be detected. Schemas will not detect mistakes in modelling a particular instance within the correct schema class (for example `Voltage = 2400`). Schemas do not aid expressivity, in fact they may reduce it; certainly they are less flexible than full HCLP. And finally, as discussed above, in our opinion schemas will only have limited success in resolving contradictions arising from composition.

For these reasons, and because schemas would not subsume PCSP, and because an alternative approach (GOCS) seemed likely to be more fruitful, we decided not to develop this idea further.

5.5 Issues we will not analyse or investigate

In this thesis we are not really concerned with the (logic) programming aspects of CLP. A collection of constraints arrives at the solver and is found to be over-constrained. We then consider what to do about it; but we are never concerned with the order in which the constraints arrived at the solver, how they got there, or whether or not the programming language contains back-tracking. Of course PCSP is not a Constraint Logic Programming language, and therefore it would introduce many irrelevancies into the comparison, transformation, and integration of it with HCLP if we were to discuss these issues, and also others such as parsing, debugging, parallelism, etc. This is not to suggest that these issues are unimportant, just that they are not only beyond the scope of this thesis, but are in fact orthogonal to it.

The conceptual separation we have tried to make is not perfect, and decisions at one level almost always have effects elsewhere. Therefore we occasionally touch on themes which are not directly to do with constraint solving. For example, standard HCLP allows strength levels to be used to label constraints but not predicates, and in Section 3.5.3 we briefly considered if this affects its expressivity or not. Also, our compositional variant of HCLP is more easily parallelised than HCLP itself. This is worth noting even though parallelism is not our main concern.

Finally, although we have claimed not to be interested in the manner or order in which constraints arrive at the store, in actual fact the issue of incrementality is very important; if 19 constraints have been processed and a twentieth arrives, we do not wish to have to discard all the work we have already done and start from scratch. Incrementality is a key motivation for the work on HCLP in Part IV, where its links to compositionality are developed. Part of the illustration of this involves a discussion of query composition in logic programming (Section 8.5), but what we do *not* develop there is a model of the precise order in which constraints arrive. We have abstracted away from that issue, and hence away from the details of, say, SLD implementations of (constraint) logic programming languages.

Chapter 6

Mathematics

6.1 Bags (Multisets)

6.1.1 Standard definitions

Bags, sometimes called multisets, are like sets except that duplicate elements are allowed i.e. $\wr a, a \int \neq \wr a \int$ (we use $\wr, \int$ to denote bags). In this chapter, we define various properties of bags, treating as basic the notion of number of occurrences of an element in a bag. In other words, a bag based on some set of elements S is a function $S \to \mathbb{N}$; see Gries & Schneider [36]. A brief overview of some of the properties of bags can be found in Knuth [52].

By definition, an element which occurs a times in a bag A, and b times in B, occurs $a+b$ times in the additive-union $A \uplus B$, max(a,b) times in $A \cup B$, which is the equivalent of set-theoretic union, and min(a,b) times in the intersection $A \cap B$. We will not generally use $A \cup B$ in this thesis, and so we feel free to refer to $A \uplus B$ as union, rather than the more clumsy 'additive-union'.

Let $e \# B$ denote the natural number n of occurrences of the element e in the bag B.[1]
Examples: $a \# \wr a, a \int = 2$. $b \# \wr a, a \int = 0$.

A shorthand is to label elements with a superscript indicating the number of occurrences. **Example:** $\wr a, a \int = \wr a^2 \int$. Of course $a \# \wr a^k \int \equiv k$.

The membership function $e \in A$ returns 'true' if $e \# A > 0$, 'false' otherwise. The empty bag contains zero occurrences of all elements: $\forall\, e \cdot e \# \wr \int = 0$.

We can now define union and intersection in terms of #; we will follow standard practice

[1]Schemes based on bags seem more appropriate for finite domains. But they can also be considered for domains where an extensional view would be impractical, such as the reals, if we allow the 'extension' of a constraint to be written as the union of disjoint ranges. For example, the 'extensional solution' of the constraints 'weak $X > 5$, weak $X < 10$' is '$(X \leq 5)$, $(5 < X < 10)$, $(5 < X < 10)$, $(10 \geq X)$' or, equivalently, '$(X \leq 5)$, $(5 < X < 10)^2$, $(10 \geq X)$'. Then # denotes the number of occurrences of a *range*, rather than a single element.

and omit the universal quantification $\forall\, e$:

Definition:

$$\begin{aligned} e\#(A \uplus B) &= e\#A + e\#B \\ e\#(A \cap B) &= \min(e\#A, e\#B) \end{aligned}$$

Examples:

$$\begin{aligned} \wr a, a \wr \uplus \wr a, b \wr &= \wr a, a, a, b \wr \\ \wr a, a \wr \cap \wr a, b \wr &= \wr a \wr \end{aligned}$$

Both bag union and intersection are associative and commutative. Note that bag intersection is a 'lifted' version of set intersection. In other words, if A and B are both bags with no duplicate elements ('set-like' bags), then so is $A \cap B$. This is obvious from its definition. We can define a function **set** which makes bags set-like as follows:

Definition:

$$e\#\mathbf{set}(A) = \begin{cases} 1, & \text{if } e\#A \geq 1 \\ 0, & \text{otherwise} \end{cases}$$

Note that **set** is idempotent i.e. $\mathbf{set}(\mathbf{set}(A)) = \mathbf{set}(A)$. Also, $\mathbf{set}(A \cap B) = \mathbf{set}(A) \cap \mathbf{set}(B)$ and $\mathbf{set}(A \uplus B) = \mathbf{set}(A \cup B) = \mathbf{set}(A) \cup \mathbf{set}(B)$.

The bag equivalent of the subset operation may be called sub-bag. We will use the symbol $\subseteq$, and define it in terms of $\#$ as follows:

Definition: $A \subseteq B \Leftrightarrow \forall\, e \cdot e\#A \leq e\#B$

We use the term 'power-bag' to mean the set of all possible sub-bags of a bag.

6.1.2 Why we use bags and not sets

Consider the following example: imagine David wants a meeting with Bernie at 2 p.m., but Bernie wants the meeting at 4 p.m. The solution **set** for David's constraint would contain the valuation $\{T = 2\}$, and Bernie's solution set would be $\{T = 4\}$. Taking the intersection of these two sets would give no solutions. In other words, we cannot calculate a preferred time for the meeting. But the meeting must occur at some stage, and as no preferred time has been offered within the constraint-based decision support system, an external choice must be made. This might be $T = 1$, say, or $T = 5$, etc, i.e. times which are not preferred by either David or Bernie. If $T = 2$ or $T = 4$ was chosen, *one* of the two people would at least be satisfied. Therefore intersection of sets is not ideal for modelling possible over-constrained systems. This is not surprising, as set intersection is the logical basis of CLP and CSP; if it *was* acceptable, there would have been no need for HCLP or PCSP.

The above example can be modelled using set *union*, giving two possible solutions which are as good as each other and better than the alternatives. However, we will now show that union of *sets* is not useful in general. Consider a variation of the first example. Now David is happy to have a meeting either at 2 p.m. or at 3 p.m., and Bernie would be satisfied with either 3 p.m. or 4 p.m. The union of the separate solution sets contains three valuations $\{T = 2, T = 3, T = 4\}$. Of course a meeting can only happen at one time, and so a choice must be made. Clearly, two out of the three choices represented by this set of solutions are incorrect: $T = 2$ or $T = 4$ would only satisfy one person, when a solution exists which would be acceptable to both. In this case set intersection would have found the correct answer.

It appears that we would like to calculate the intersection of the sets when it would not be empty, and the union otherwise. However, a disjunction ("... if $D \cap B = \{\}$, ... otherwise") in the definition of the solution is not elegant. Let us instead consider union of *bags*. Then the first example has solution $\wr T = 2, T = 4 \int$ and the second has solution $\wr T = 2, T = 3, T = 3, T = 4 \int$. The first solution is clearly correct. The second is not perfect: ideally we would like just $\wr T = 3, T = 3 \int$ or $\wr T = 3 \int$. However, at least the second solution distinguishes between $T = 3$ and the other possibilities. (See Chapter 9 for how to obtain $\wr T = 3, T = 3 \int$ from $\wr T = 2, T = 3, T = 3, T = 4 \int$.) Therefore union of bags is preferable to either union or intersection of sets.

Another possibility might be suggested: instead of using sets, use *weighted* sets. Then $T = 3$ would be more heavily weighted than $T = 2$ and $T = 4$, which in turn would be preferred over all other valuations. Then the solution set for the second example would be $\{(T = 2, 1), (T = 3, 2), (T = 4, 1)\}$. But this is just another way of representing bags! As mentioned in the previous section, formally a bag can be considered as a function from elements to $\mathbb{N}$. It does not matter if the result of the function on element e is expressed explicitly by (e, n), or implicitly by $e, \ldots, e$ (n occurrences). As weighted sets are slightly less standard, and rather more concrete, than bags, we choose to use the latter.

6.1.3 Bags other than of tuples of values

Much of the discussion elsewhere in this thesis is in terms of bags of *tuples of values* (i.e. bags of tuples of elements from the domains of variables). This approach is motivated by the fact that the CSP paradigm is committed to this style of representation, and so to be able to compare HCLP with PCSP it is necessary to work at this level. But we do not actually need to use bags of tuples; we could instead use bags of anything which has certain characteristics. We define such entities as SCCs ('soluble combinable [simple] constraints').

What qualifies as an SCC? It depends on the domain. An SCC is required to have a certain property (see below); in the context of finite unordered domains, tuples of elements do indeed have these properties, and we use them in this thesis for ease of presentation.

In a finite or infinite *ordered* domains, we can also use bags of disjoint intervals[2]. In a finite ordered domain, the intervals should be closed[3]. In an infinite ordered domain, the intervals should be closed at the value 3 (say) if the constraint being represented is $X \geq 3$, but should be open at that end if the constraint is $X > 3$.

The defining property of SCCs is that they can be combined (bag-union) together without invoking the constraint solver. For example it is clear that in finite domains $\wr a^2, b^3, c^4 \int \uplus \wr b^1, c^1, d^1 \int = \wr a^2, b^4, c^5, d^1 \int$ without constraint solving (just add the superscripts which indicate number of occurrences). For the real numbers, it is clear to the eye that $(0 \leq X \leq 10) \uplus (5 < X < 15) = (0 \leq X \leq 5)^1 \uplus (5 < X \leq 10)^2 \uplus (10 < X < 15)^1$; ranges of the sort on the right-hand side would be considered as SCCs as a computer can manipulate them "easily".

In the rest of this thesis we will just consider tuples of values, but we believe that everything we say applies to any representation which can be considered as SCCs.

6.2 Lattices

A *partially-ordered set* (poset) is a set over which a binary relation $\leq$ is defined which is reflexive, transitive, and antisymmetric.

For a poset S with partial order $\leq$, then $a \in S$ is an *upper bound* of a subset $X \subseteq S$ if for all $x \in X$, $x \leq a$. a does not need to be a member of the subset X. Similarly, b is a *lower bound* of X if for all $x \in X$, $b \leq x$.

A subset $X \subseteq S$ has a *least upper bound* or infimum $a \in S$ if a is an upper bound of X and for any upper bound α of X, $a \leq \alpha$. Similarly, a lower bound $b \in S$ is a *greatest lower bound* or supremum of X if b is a lower bound of X and for all lower bounds β of X, $\beta \leq b$. If a least upper bound (denoted $lub(X)$) exists, it is unique. Similarly for greatest lower bounds $glb(X)$.

A partially-ordered set is a *lattice* if and only if every pair of elements in the set has a least upper bound and a greatest lower bound. If every subset of the set also has a *glb* and a *lub*, then the lattice is said to be *complete.* All lattices with a finite number of elements are complete. All the partially ordered sets which we will consider are in fact complete lattices, but we will not need to use many of their properties. For more details, see one of the many books on lattice theory, e.g. [19].

[2]An interesting presentation of interval constraints can be found in [76].

[3]A *closed* interval is one where the end-points are included in the domain. An *open* interval does not contain its endpoints.

Part III

Transformation

In this part we present a transformation between HCLP and PCSP. This will show that problems which can be expressed in one of these two formalisms can also be expressed in the other.

Chapter 7

Transformations between HCLP and PCSP

7.1 Preliminary remarks

HCLP and PCSP are not identical in scope, therefore it is impossible to transform all of HCLP into PCSP. However, the work presented in the rest of this chapter is complete in the sense that we present transformations for every single aspect which can be transformed. First of all, however, we discuss those parts of HCLP which are outside the scope of PCSP, and make other preliminary remarks.

7.1.1 Differences which will not be transformed away

Firstly, CLP in general defines a class of programming languages, which place constraint solving in a logic programming framework, whereas CSP defines a set of problems, techniques, and algorithms. We could embed PCSP in a logic programming framework, and then a comparison with HCLP would make sense, or we can ignore the programming language aspects of HCLP, and compare the resulting theory of 'constraint hierarchies' with PCSP. In this chapter we will consider the latter approach, i.e. when we say 'HCLP' we really mean 'constraint hierarchies'.

Secondly, CSP techniques are always defined with finite domains whereas the CLP framework extends to continuous domains such as the real numbers. We will only attempt to transform HCLP(FD); however, we *will* transform metric comparators as well as predicate ones. Metric comparators required a notion of 'distance' between points in the domain, but there is no reason why this distance cannot be discrete (see Section 2.7.3).

Finally, in HCLP the required constraints are special; the difference between required and strong constraints is richer than the difference between, say, strong and weak. PCSP does not have this special class of required constraints [Borning, Private Communication]. This is discussed further in the next section.

7.1.2 PCSP with distinguished required constraints

In Section 4, we presented the standard formalisation of PCSPs as $\langle (P, U), (PS, \leq), (M, (N, S))\rangle$. We can modify this to allow us to denote a subset of the constraints in P as 'required', giving a theory which can be called $\mathrm{P_R}$CSP. Our additions are in italics:

$$\langle (P, R, U), (PS, \leq), (M, (N, S))\rangle$$

where P is a constraint satisfaction problem, *$R \subseteq P$ is a set of constraints*, U is a set of 'universes' i.e. a set of potential values for each of the variables in P, $(PS, \leq)$ is a problem space with PS a set of problems *each of which contains all the constraints in R*, and $\leq$ a partial order over problems, M is a 'distance function' on the problem space, and (N, S) are necessary and sufficient bounds on the distance between the given problem P and some solvable member of the problem space PS. A solution to a PCSP is a problem P' from the problem space and its solution, where the distance between P and P' is less than N, and *where all the constraints in R are satisfied*. If the distance between P and P' is minimal, then this solution is optimal.

In Section 4 we noted that Freuder states that the obvious problem space to explore when trying to weaken a problem is the collection of all problems Q such that $Q \leq P$, but we also noted that it may be useful to consider only some of these Qs, i.e. those problems which have been weakened in a particular way which makes sense in the context of the system that we are trying to model [32]. Therefore we note that $\mathrm{P_R}$CSP can be considered simply as selecting those Qs which satisfy all the constraints in R.

One way to select the appropriate part of the problem space is to choose a distance function which gives an infinitely large distance for all other parts. If distance functions are generally denoted by μ, from now on we will assume the existence of a particular function μ_∞, usually parameterised by a set of required constraints σ, which defines a distance of zero to any problem which satisfies all the constraints in σ, and a distance of infinity to all other problems. If T is some arbitrary problem drawn from the problem space, then

$$\mu_{\infty(\sigma)} = \begin{cases} 0, & \text{if } \sigma \subseteq T \\ \infty, & \text{otherwise} \end{cases}$$

In fact, this definition is sufficient, but not necessary, i.e., any problem given a distance of zero will indeed satisfy the required constraints, but some acceptable problems will not be given a distance of zero. For example, consider sets of constraints which are syntactically different from σ but logically equivalent. However, given that PCSP does not create new problems arbitrarily but by relaxing constraints from the original problem, this definition is acceptable.

Note that the obvious alternative, namely

$$\mu_{\infty(\sigma)}(T) = \begin{cases} 0, & \text{if } \mathit{sols}(T) \subseteq \mathit{sols}(\sigma) \\ \infty, & \text{otherwise} \end{cases}$$

is necessary but not sufficient. It measures any over-constrained problem T to be at zero distance from σ (as T's solutions are the empty set), even if T concerns completely different variables and constraints.

$\mu_{\infty(\sigma_r)}$ will be the first element of the sequence of functions $\mu = [\mu_r, \mu_s, \mu_w, \ldots]$ parameterised by the constraints at each level of the hierarchy. For example, if the comparator used is UCB, then $\mu = [\mu_{\infty(\sigma_r)}, \mu_{UCB(\sigma_s)}, \mu_{UCB(\sigma_w)}, \ldots]$.

The main conclusion of this section is that we can deal with the issue of required constraints in a straightforward and localised manner. Therefore, perhaps surprisingly, in the rest of this chapter we do not really need to emphasise the difference between PCSP and P_RCSP.

7.1.3 Distance function and comparator typing

Freuder discusses different PCSP distance functions, and also different types of distance functions (where 'type' is used in the sense of integers, reals, strings, pointers etc), although he does not explicitly use type terminology. One distance function is derived from the partial order on the problem space. Then $M(P, P')$ equals the number of solutions not shared by P and P'. Clearly, this distance function takes two inputs of the same type, and outputs a number. Another distance function suggested by Freuder is the one which counts the number of constraint *values* not shared by P and P'; this also has two inputs of the same type, and this would appear to be a general characteristic. Sometimes the initial problem P is used by the distance functions implicitly, which makes them look like they have just one input, namely P'. But this is not actually the case.

HCLP comparators, on the other hand, have as input a bag of labelled constraints and a set of valuations, and as output the set of incomparable valuations, a subset of the second input, such that no other valuations in the input are better. See Section 3.2 for a complete discussion of comparators.

Using $[\alpha]$ to represent a collection of elements of type α (the difference between a set, a bag, and a sequence is not important here), and using *con* as the type of unlabelled constraints and *lcon* as the type of labelled constraints, and *val* as the type of a valuation, we can give type definitions of a generic distance function μ and comparator C as follows:

$$\mu : [con] \times [con] \to num$$
$$C : [lcon] \times [val] \to [val]$$

Consequently, it might seem that when transforming comparators into distance functions, we ought to remember that distance functions have homogeneous input types whereas comparators don't. In fact, however, HCLP comparators are defined in terms of various functions, including error functions e which have as input a single constraint and a single valuation, and which return numbers (not sets of valuations). There is also $\mathbf{E}$, which is a version of e raised to collections of constraints, and a function g which combines all the errors for the individual constraints into an overall error for that valuation, using e.g. 'sum', or 'max' or 'least-squares'. Finally $\mathbf{G}$ raises the errors once again, to consider the different levels of constraints in the hierarchy. Labelling of constraints can just be though of as a mechanism for partitioning a collection into sub-collections (all the required constraints, then all the strong ones, etc). So we can see that [*lcon*] can be thought of as [[*con*]].

$$\begin{array}{lcll} e & : & con \times val \rightarrow num & \\ \mathbf{E} & : & [con] \times val \rightarrow [num] & \\ g & : & [con] \times val \rightarrow num & \\ \mathbf{G} & : & [[con]] \times val \rightarrow [num] & \text{or equivalently} \\ \mathbf{G} & : & [lcon] \times val \rightarrow [num] & \end{array}$$

The comparator $\mathcal{C}$ is a version of **G** raised over all possible valuations: the sequences of numbers calculated for each valuation with respect to all the constraints in the hierarchy can be compared lexicographically, in order to find the equal-best solutions (valuations) to the hierarchy as a whole.

As mentioned above, a particular CSP problem which is a PCSP relaxation of the original problem, may have more than one solution, and hence more than one valuation. Therefore the *val* which appears in the type definition for g would be replaced by [*val*], and so the *num* would be replaced by [*num*]. However, we introduce a step of taking the maximum. Clearly $max : [num] \rightarrow num$, and so the types of g and **G** can be respected.

Therefore if we split the comparator into its constituent parts, we may interpose the step of taking the maximum of a sequence of numbers, and thus arrive at an appropriate distance function. This is done in Section 7.2.2. In other words, despite initial appearances, the issue of differing types does not present an obstacle to the transformation progress.

7.1.4 Characterisation of HCLP and PCSP

In this section we present those aspects which are relevant for the transformation process. The relevant aspects for HCLP are

$$\langle \mathbf{H} = (\mathbf{H}_0, \mathbf{H}_1, \mathbf{H}_2, \ldots]), \mathcal{C} = (e, \mathbf{E}, g) \rangle$$

where **H** is a hierarchy of constraints, made up of all the required constraints $\mathbf{H}_0$, the strongly preferred constraints $\mathbf{H}_1$, weaker preferences $\mathbf{H}_2$ etc. The comparator $\mathcal{C}$ is used to compare different solutions; it is made up of various functions as described in the previous section, and results in a sequence of errors $[r, s, w, \ldots]$ giving the errors with respect to each level of the hierarchy (required, strong, weak, etc). These sequences are used to order different possible solutions lexicographically. The lowest element in the order indicates the best solution.

PCSP is formalised as a triple $\langle (P, U), (PS, \leq), (M, (N, S)) \rangle$, but we need only consider certain elements of it as follows: P is a constraint satisfaction problem, and M is a distance function which selects the consistent problem 'nearest' to P.

When transforming HCLP into PCSP, we will take all the constraints in $\mathbf{H}$ without their strength labels as being P. We will use the strength label information and the comparator to construct the appropriate distance function.

When transforming PCSP into HCLP, the constraints in the hierarchy will just be the constraints in P, and the distance function will be used to define their strength labels (i.e. which of the $\mathbf{H}_i$ should contain each constraint) and the comparator C.

In the case of the standard PCSP distance function, all the constraints from P must be placed in the same non-required level of the hierarchy, but it does not matter which one is used. Arbitrarily, we choose to label them 'strong' and so put them in $\mathbf{H}_1$.

7.2 Transforming HCLP into PCSP

7.2.1 Creating the distance function

The base problem P is *all* the constraints in the hierarchy, without their strength labels. U, PS, and (N, S) remain as they would for an original PCSP based on P. (By 'original PCSP' we mean one written down by a user, as opposed to one created by automatically transforming an HCLP problem.)

The distance function will be calculated from a combination of the HCLP comparator and the particular hierarchy of labelled constraints, and the hierarchy will lead to it being stratified into a lexicographic order.

The distance function μ derived from a hierarchy with n levels will be stratified into n parts, whose results will be ordered lexicographically (i.e. it will not calculate a single distance of the relaxed CSP from P). Each relaxation (each problem drawn from the problem space PS) will be annotated with a sequence[1] $[d_0, d_1, d_2, \ldots, d_{n-1}]$ each element of which is calculated by the respective distance function in $\mu = [\mu_0, \mu_1, \ldots, \mu_{n-1}]$. (The required level is formally called level 0, the strongest non-required level is 1, down to $n-1$ for the weakest level.) For example, in the case of a hierarchy containing only required, strong and weak constraints, each candidate problem will be annotated with a sequence $[r, s, w]$, where r is the distance according to μ_r, the part of the distance function derived from the required constraints, s is the distance according to μ_s, the part of the distance function derived from the strong constraints, and w is the weak distance, calculated by μ_w. We then order the various relaxations according to the lexicographical order of their sequences.

7.2.2 Transforming hierarchies into distance functions

The distance function calculates the distance of one of the problems, say T, in the problem space PS from the 'ideal' set of constraints which would have distance zero

[1] We use a sequence because the hierarchy's strength labels are totally ordered. We discussed partially-ordered hierarchies in Section 3.5.5.

(i.e. completely satisfy all the constraints in the original problem). In fact, as the original constraints might be inconsistent, it is possible that no such ideal set exists.

Let us define *sols*(T) to be the set of solutions to T. We assume that T is consistent (even though σ might not be), and so *sols*(T) will never be empty. Each member of *sols*(T) is a valuation, i.e. an assignment of a value from its domain to each variable in T. We can calculate how well a particular valuation satisfies the constraints σ using the machinery developed by Borning and Wilson in HCLP.

T may have more than one solution, and hence may give rise to more than one valuation, therefore we define the distance of T from σ to be the *maximum* of distances of each of the valuations in T. This is necessary because HCLP's comparators take as input the set of original constraints and a single valuation/possible solution. The output is the score for that particular valuation, which can then be used to place that valuation in an order. In PCSP, however, distance functions create an order over sets of constraints; a set of constraints can have many solutions, and so we have to choose the score of one of them. We choose the worst (largest) score, i.e. this set of constraints can never give an answer with a score worse than x. (The reasons for this choice are discussed in the next paragraph.) For example, if T is said to be a distance of 2 from σ, that means that any solution of T is a distance of *at most* 2 from σ. HCLP's comparators work by measuring errors, and so, by choosing the maximum of the errors for each valuation, we are considering the worst possible case. (The larger the distance, the greater the error. This is similar to the case in HCLP, where all the comparators give larger scores for less preferred valuations.)

We choose the maximum because any other choice would be unsound. Consider two possible CSPs, T_1 with solutions with scores 1,2,3,4,5, and T_2 with solutions all with score 4. If we order problems by the minimum or mean, say, T_1 will be selected in preference to T_2. All the solutions to T_1 are considered equally as solutions to the PCSP. So if we happen to choose the solution with score 5, we have in fact chosen a CSP such that a different choice would have had a better (more preferred) solution. This is clearly incorrect.

Therefore, using some HCLP terminology including denoting a general comparator by $\mathcal{C}$ (defined in terms of g, e, and $\mathbf{E}$), the PCSP distance function defined in terms of the set σ of constraints from one optional level of the hierarchy is:

$$\mu_{\mathcal{C}(\sigma)}(T) = \max\{g(\mathbf{E}(\sigma\tau) \mid \tau \in sols(T)\}$$

In other words, we treat all the constraints in σ as a sequence, apply a particular valuation τ to each of them, combine the errors using g, and then take the maximum of the errors for all the τ and treat it as the error for T.

The various distance functions, each parameterised by the constraints from a different level of the hierarchy, will lead to results which are lexicographically ordered, just as in HCLP. The main difference between standard HCLP and our work is that we interpose the step of taking the maximum error for each of the valuations in T between the application of g and placing in an order.

In the case of UCB, $g(\mathbf{v}) = \sum_{i=1}^{|\mathbf{v}|} v_i$ and $e = e_p$ is the simple predicate error function which returns 0 for each constraint in σ which is consistent with τ, and 1 for each inconsistent constraint. **E** is e raised over sequences, i.e. its input is a sequence of constraints, and its output in this case is a sequence of 0's and 1's. g then adds all these individual errors. Least-squares-better (LSB) has a more complicated $e = e_d$, which measures the error as a 'distance' in a metric space. g then sums the squares of these errors:

$$\begin{aligned}
\mu_{UCB(\sigma)}(T) &= \max\left\{\sum_{c\in\sigma} e_p(c,\tau) \mid \tau \in sols(T)\right\} \\
\mu_{LSB(\sigma)}(T) &= \max\left\{\sum_{c\in\sigma} e_d(c,\tau)^2 \mid \tau \in sols(T)\right\}
\end{aligned}$$

7.2.3 Logic description

```
transform-HCLP-PCSP( (LabelledConstraints, Comparator), (Constraints, DF) ) ⟵

    comparator-distance-function(Comparator, DFT) ∧
    partition-labelled-constraints(LabelledConstraints, [Required, Strong, Weak,...]) ∧
    remove-labels([Required, Strong, Weak,...],
            [UL-Required, UL-Strong, UL-Weak,...]) ∧

    distance-functions(([UL-Required, UL-Strong, UL-Weak,...], DFT),
            [df($df_\infty$, UL-Required), df(DFT, UL-Strong), df(DFT, UL-Weak),...]) ∧
    collect-distance-functions([df($df_\infty$, UL-Required), df(DFT, UL-Strong),
            df(DFT, UL-Weak)...], DF) ∧

    collect-unlabelled-constraints([UL-Required, UL-Strong, UL-Weak,...],
            Constraints).
```

Figure 7.1: HCLP into PCSP

In Figure 7.1 we present a first-order predicate logic (FOPL) description of the transformation from HCLP to PCSP. We assume that the logic is enriched with lists, and that all variables are universally quantified by quantifiers with maximal scope. The predicate has a collection of labelled constraints and a comparator as 'input', and 'outputs' a collection of unlabelled constraints and a distance function. Therefore any use of HCLP in a specification can be replaced by the use of *transform-HCLP-PCSP* followed by the use of PCSP.

In the figure, *DF* stands for a particular *distance function*, which will be of type *DFT* (where 'type' means a name such as MaxCSP). The appropriate distance function depends on the HCLP comparator in use. It is defined by the values paired by the relation *comparator-distance-function/2*, such as UCB and MaxCSP. The distance function can be stratified, with each strata being parameterised by a different level of constraints. Note also that *UL* stands for *unlabelled* constraints.

Initially, for clarity of presentation, this figure contains an implication $\longleftarrow$, and not an equivalence $\longleftrightarrow$. We will develop, below, an implication in the other direction, and then combine them to form an equivalence relation in a later section.

7.2.4 Example β

Consider a constraint hierarchy containing three strong constraints (or rather, three optional constraints all at the same arbitrary strength level) $A =$ '$X \geq 7$', $B =$ '$X \leq 3$', and $C =$ '$2 \leq X \leq 5$', where the variables are assumed to range over integers. Its HCLP solutions using UCB (unsatisfied-count-better) are $\{X = 2,\ X = 3\}$, each of which satisfies two out of three constraints. Its LSB (least-squares-better) solution would be $\{X = 5\}$, which has total square error of $2^2 + 2^2 + 0 = 8$.

The base set of the PCSP will be $P = \{A, B, C\}$, i.e. all the constraints in the hierarchy. P is inconsistent, so we will augment the domains of some of the constraints in P to get a consistent, weaker, problem. We will denote augmenting the domain of a less-than or greater-than constraint using $\vee$, e.g. $A_2 =$ '$X \geq 7 \vee X = 2$' means that we have relaxed A by adding the value 2 to its domain. (This could also be written $A_2 = A \cup \{2\}$.) There are many different relaxations of A, B, and C, including

$$\begin{array}{lll lll}
A_0 & :: & X \geq 7 \vee X = 0 & A_{024} & :: & X \geq 7 \vee X = 0 \vee X = 2 \vee X = 4 \\
A_2 & :: & X \geq 7 \vee X = 2 & A_{23} & :: & X \geq 7 \vee X = 2 \vee X = 3 \\
A_5 & :: & X \geq 7 \vee X = 5 & A_{235} & :: & X \geq 7 \vee X = 2 \vee X = 3 \vee X = 5 \\
\\
B_4 & :: & X \leq 3 \vee X = 4 & B_6 & :: & X \leq 3 \vee X = 6 \\
B_5 & :: & X \leq 3 \vee X = 5 & B_9 & :: & X \leq 3 \vee X = 9 \\
\\
C_0 & :: & 2 \leq X \leq 5 \vee X = 0 & \ldots & &
\end{array}$$

The distance of any set of constraints from the 'ideal' will be given by some distance function. We will consider the two previously defined distance functions, namely $\mu_{UCB(\{A,B,C\})}$ and $\mu_{LSB(\{A,B,C\})}$ (i.e. not the standard PCSP distance function, but two drawn from HCLP). We will also consider various sets drawn from the problem space, each containing one constraint derived from each of the constraints in the original set e.g. $\{A_2, B, C\}$, $\{A_5, B_6, C\}$, $\{A, B_5, C_0\}$, etc. Following PCSP, we will only consider consistent sets. In this case, the sequences will only contain one element, and so we do not really need to consider lexicographic orderings.

Consider the distance function $\mu_{UCB(\{A,B,C\})}$ and the set $T = \{A_2, B, C\}$. This set has only one solution when it is treated as a normal CSP, namely the valuation $\tau =$ '$X = 2$'. (Remember that PCSP induces an order over different *problems* and not different solutions, so the fact that a particular problem only has one solution is not important.) We can now apply this valuation to all the constraints in the original set: the error function will return 1 for $e(A, \tau)$ because $X = 2$ is inconsistent with the original A constraint, and 0 for each of B and C. The sum of all these errors is 1, and this is the only sum for the set T and hence the maximum. We can go through a similar procedure for some of the other members of the problem space, and we end up with the following table:

set	sols.	UCB errors		LSB errors	
		$\sum e$	max	$\sum e^2$	max
A_2, B, C	$X = 2$	1	1	25	25
A_{23}, B, C	$X = 2$	1	1	25	25
	$X = 3$	1		16	
A_{235}, B, C	$X = 2$	1	1	25	25
	$X = 3$	1		16	
A_5, B_5, C	$X = 5$	2	2	8	8
A_{235}, B_5, C	$X = 2$	1		25	
	$X = 3$	1	2	16	25
	$X = 5$	2		8	

The absolute values of the summed errors for LSB are much larger than those for UCB, but this is irrelevant; the only question is the order induced by each distance function separately. It can be seen that the first three sets are equally good as far as UCB is concerned. It does not matter that $\{A_2, B, C\}$ only produces one of the two possible solutions, as PCSP delivers an equivalence class of all those CSPs equidistant from $\{A, B, C\}$; the only way to guarantee that all equally good solutions have been found is by exploring all the members of the class. This is similar to the distinction between one-solution and all-solutions in CSPs or in logic programming.

The fourth set has the lowest error under μ_{LSB}, which indicates that $X = 5$ is the least-squares-better solution to the hierarchy (found by us after a straightforward differentiation of the error function, as opposed to an exhaustive search).

7.3 Transforming PCSP into HCLP

7.3.1 Transforming the augmentation distance function

7.3.1.1 General remarks

To transform PCSP with the standard distance function into HCLP, we take the constraints in P and give them all the same arbitrary non-required strength label, say 'strong'. Thus they will be placed in $\mathbf{H_1}$. Then we use the HCLP comparator unsatisfied-count-better (UCB). We claim that this is the correct comparator to use, i.e. we claim that the solutions calculated by HCLP using UCB are the same as those in PCSP, and the particular solutions which are best according to PCSP will also be best according to UCB. (The intuition is as follows: the number of unsatisfied constraints counted by UCB is the same as the number of constraints which would need a single domain augmentation to create a consistent CSP, thus UCB measures an equivalent distance to that measured in PCSP.)

Certain combinations of augmented constraints in the PCSP formulation, which duplicate solutions found at a closer distance, will not appear in the HCLP answer, but all

```
transform-PCSP-HCLP( (Constraints, DF), (LabelledConstraints, Comparator) ) ⟵

    partition-unlabelled-constraints(Constraints, [UL-Strong]) ∧
    add-labels([UL-Strong], [Strong]) ∧
    collect-labelled-constraints([Strong], LabelledConstraints) ∧
    create-comparator(DF, Comparator).
```

Figure 7.2: PCSP into HCLP — all constraints treated the same

the solutions to these combinations *will* appear. (Here is an analogy: if the list of PCSP solutions, in order from best to worst, is $[a, b, c, a, d, a, e]$, the list of HCLP solutions may be $[a, b, c, d, e]$. So although the lists are not equal, the fact that a should be chosen before b or d is present in both representations.)

A FOPL description of the transformation from PCSP to HCLP is given in Figure 7.2. It assumes that the distance function does not distinguish between different constraints, but treats them all the same, labelling them all 'strong'. It is clear that *partition-unlabelled-constraints(Constraints, List)* is the same as *collect-unlabelled-constraints(List, Constraints)* which was used in Figure 7.1. Also, *add-labels([UL-Strong], [Strong])* is the same as *remove-labels([Strong], [UL-Strong])*, and *collect-labelled-constraints([Strong], LabelledConstraints)* is equivalent to *partition-labelled-constraints(LabelledConstraints, [Strong])*. So we can re-write Figure 7.2 to get Figure 7.3. This use of the HCLP-to-PCSP predicates from Figure 7.1 in Figure 7.3 will facilitate combining the two implications into an equivalence relationship.

```
transform-PCSP-HCLP( (Constraints, DF), (LabelledConstraints, Comparator) ) ⟵

    collect-unlabelled-constraints([UL-Strong], Constraints) ∧
    remove-labels([Strong], [UL-Strong]) ∧
    partition-labelled-constraints(LabelledConstraints, [Strong]) ∧
    comparator-distance-function(Comparator, DF).
```

Figure 7.3: PCSP into HCLP — using HCLP-to-PCSP predicates

Two predicates defined in Figure 7.1 are not used in Figures 7.2 or 7.3, namely *distance-functions* and *collect-distance-functions*. This is because their inverses are null operations if all the PCSP constraints are to be treated as 'strong' when being transformed into HCLP. In fact, any refinement of these specifications would need to include information such that the predicates could be present but would be ignored when they are inappropriate. This is straightforward, as these predicates should be ignored when the non-strong levels are empty. Then the distance function is not complicated and may be represented by its name. We choose to avoid cluttering Figures 7.2 and 7.3, which would result from replacing the conjunction of these two predicates by a disjunction of the form

(*UL-Required* = *UL-Weak* = ⋯ = {} ∧ *DF* = *DFT*)
∨
(*distance-functions*(([*UL-Required*, *UL-Strong*, *UL-Weak*,...], *DFT*),
  [*df*(DF_∞, *UL-Required*), *df*(*DFT*, *UL-Strong*), *df*(*DFT*, *UL-Weak*),...]) ∧
collect-distance-functions([*df*(DF_∞, *UL-Required*), *df*(*DFT*, *UL-Strong*),
  df(*DFT*, *UL-Weak*)...], *DF*))

Apart from these two predicates, it can be seen that Figure 7.3 is just the same as Figure 7.1 read backwards. Of course, the order of conjuncts is not relevant in FOPL, but it is clearer to obey a natural reading order. Therefore, we can combine these two implications into an equivalence, including the disjunction mentioned above, as shown in Figure 7.4.

transform((*LabelledConstraints*, *Comparator*), (*Constraints*, *DF*)) ⟷

  comparator-distance-function(*Comparator*, *DFT*) ∧
  partition-labelled-constraints(*LabelledConstraints*, [*Required*, *Strong*, *Weak*,...]) ∧
  remove-labels([*Required*, *Strong*, *Weak*,...],
    [*UL-Required*, *UL-Strong*, *UL-Weak*,...]) ∧

  ((*UL-Required* = *UL-Weak* = ⋯ = {} ∧ *DF* = *DFT*)
  ∨
  (*distance-functions*(([*UL-Required*, *UL-Strong*, *UL-Weak*,...], *DFT*),
    [*df*(df_∞, *UL-Required*), *df*(*DFT*, *UL-Strong*), *df*(*DFT*, *UL-Weak*),...]) ∧
  collect-distance-functions([*df*(df_∞, *UL-Required*), *df*(*DFT*, *UL-Strong*),
    df(*DFT*, *UL-Weak*)...], *DF*))) ∧

  collect-unlabelled-constraints([*UL-Required*, *UL-Strong*, *UL-Weak*,...],
    Constraints).

Figure 7.4: HCLP-PCSP equivalence

7.3.1.2 Detailed defence of the choice of UCB when transforming the augmentation distance function

This section contains a detailed defence of our choice of UCB as the comparator to use in HCLP when transforming from PCSP using a distance function based on number of constraint augmentations. It addresses one possible key objection, but does not affect the presentation in subsequent sections of the chapter.

Consider those PCSP weakenings which involve more than one augmentation of a single constraint. We claim that the following complaint about our choice of UCB is unjustified: "UCB will just detect that a constraint had been violated by a valuation. It wouldn't detect that two different augmentations would be necessary for the constraint not to be violated." It is incoherent because two augmentations can never be necessary for a *single* constraint not to be violated. Two augmentations to a single constraint might, however, lead to an additional two or more solutions, but we can ignore this situation due to the following claim:

Claim: the additional solutions caused by $n \geq 2$ augmentations of a single constraint can be completely separated into n classes, each of which contains solutions caused by only one of the n augmentations. The CSPs represented by these singly-augmented constraints will all appear in the partial order induced by the distance function, and they will all appear earlier than the CSP containing the n-augmented constraint. Therefore, no solutions will be lost by ignoring all multiply-augmented constraints. Therefore, the fact that UCB only picks out those solutions which violate the smallest number of singly-augmented constraints, does not change the set of solutions computed. (All that would happen is that two solutions s_1 and s_2 will separately appear as, say, the equal-best solutions to the hierarchy, but their union will fail to appear as a second-best or third-best solution.)

Example: Let A' denote the constraint A with one extra tuple added to its domain, in the usual manner. Usually there will be more than one way to augment A; these alternatives may be indicated by A'_1, A'_2, etc. Let A'' generally denote two augmentations to A, and specifically $A''_{1,2}$ denote that the two augmentations are equivalent to $A'_1 \cup A'_2$. Then our claim is that all the solutions to the CSP $\{A''_{1,2}, B''_{3,4}, C''_{5,6}\}$ are present in the union of the solution sets $\{A'_1, B'_3, C'_5\} \cup \{A'_2, B'_3, C'_5\} \cup \{A'_1, B'_4, C'_5\} \cup \{A'_2, B'_4, C'_5\} \cup \ldots$. In other words, we can ignore multiple augmentations of a single constraint.

Intuition: Consider the CSP as a graph, with each variable represented by a node and each constraint represented by an edge (see also Section 2.2.3). The tuples which make up the constraint are labels for the edges. A solution to the CSP is a path through every edge in the graph, consistent with the labels. If we add a label to an edge, we are increasing by one the number of paths between the two nodes connected by that edge[2]. If instead we added a different label, we would again increase the number of paths between these two nodes by one. It is intuitively clear that adding these two labels simultaneously will add precisely two paths between the two nodes: any path can only take account of one of the two labels on the edge. We could have arrived at the same set of total paths through the graph by taking two copies of the original graph, adding one new label to each of them, finding the new paths caused by this single extra label, and then eventually taking the union of the two sets of paths.

Proof:
Consider various binary constraints over different pairs selected from n variables $X_1, \ldots, X_n$. We can define the *expansion* C^*_{ij} of each constraint C_{ij}, which originally related X_i and X_j, to a set of n-tuples by creating a tuple for each element of the cartesian product of the variables not originally involved in the constraint:

$$C^*_{ij} = \{(v_1, v_2, \ldots, v_i, v_j, \ldots, v_n) \mid (v_i, v_j) \in C_{ij}, v_k \in \mathrm{dom}(X_k), \\ 1 \leq k \leq n, k \neq i, k \neq j)\}$$

[2]The number of paths through the entire graph may increase by more than one. If there are k paths leading into the start node of the edge under consideration, and l paths leading away from the end node, then adding a path between the two nodes may increase the number of paths through the entire graph by up to kl, as discussed earlier in Section 4.3.

Example: if X has domain $\{a, b\}$, Y has domain $\{c, d\}$, and Z has domain $\{e, f\}$, and if $A_{XY} = \{(a, c), (b, d)\}$, then $A^*_{XY} = \{(a, c, e), (a, c, f), (b, d, e), (b, d, f)\}$.

It is clear that the solution to a CSP is precisely the intersection of the expanded versions of each of its constraints. Thus instead of considering the solution of the set of constraints $\{A_{XY}, B_{YZ}, C_{XZ}\}$, we can just consider $A^*_{XY} \cap B^*_{YZ} \cap C^*_{XZ}$.

(This is the same as the relational database idea of a 'join' between two relations. The join of A_{XY} and B_{YZ} on variable Y is denoted $A_{XY} \bowtie B_{YZ}$, and is defined as follows:

$$A_{XY} \bowtie B_{YZ} = \{(x, y, z) \mid (x, y) \in A_{XY}, (y, z) \in B_{YZ}\}$$

It is clear that $A^*_{XY} \cap B^*_{YZ} \cap C^*_{XZ} = A_{XY} \bowtie B_{YZ} \bowtie C_{XZ}$.)

If we add one pair to the domain of one of the constraints in a CSP, it is equivalent to adding a set of n-tuples to the domain of that constraint's expanded version, where the other places in the tuple are filled with all possible combinations of elements from the domains of all the other variables. Continuing with the example, let us assume, without loss of generality, that we have augmented constraint B. This leads to adding a set of n-tuples to B^*; let us call this set of additional tuples R. We can imagine adding a different pair to B which would lead to adding a different set to B^*, say R'. If we add both pairs to B at the same time, it is clear that we must add $R \cup R'$ to B^*. (If it is not clear, see Appendix B.1.)

Our claim is that we can ignore CSPs where one constraint has been multiply augmented; all their solutions will be present in the union of the solutions to CSPs with singly-augmented constraints. This is equivalent to claiming

$$A^* \cap (B^* \cup \underline{(R \cup R')}) \cap C^* = (A^* \cap (B^* \cup \underline{R}) \cap C^*)\underline{\cup}(A^* \cap (B^* \cup \underline{R'}) \cap C^*)$$

The proof is a straightforward exercise in the use of the distributivity laws of set theory ($J \cup (K \cap L) = (J \cup K) \cap (J \cup L)$ and its dual), with one use of the idempotence of set union ($K \cup K = K$). It is presented in Appendix B.2.

Therefore, using UCB as our comparator in the automatically generated HCLP version of a PCSP is acceptable. So our transformation from PCSP to HCLP holds. □

7.3.2 Transforming the MaxCSP distance function

MaxCSP is defined as seeking "a solution that satisfies as many constraints as possible" [33]. This is equivalent to saying that it tries to minimise the number of unsatisfied constraints. Clearly this creates the same ordering of solutions as the UCB comparator in HCLP, when all the constraints are at the same strength level in the hierarchy. Any more detailed discussion of transformations of MaxCSP would repeat some of the contents of the previous discussion of the augmentation distance function.

7.3.3 Transforming non-standard distance functions

We have shown above how to transform problems using the standard PCSP distance function into HCLP. We now consider three other possibilities, firstly where all the variables and constraints are treated equally by the distance function but the distance is not defined as minimum augmentation, secondly where some of the *variables* in the problem are highlighted, and finally where some of the *constraints* are highlighted. As an example we then present a description in logic of this third possibility.

7.3.3.1 Non-specific (homogeneous) distance functions

All the constraints are put at the 'strong' level of the hierarchy resulting from the transformation. The combining function embodied by the distance function must be transformed into an HCLP-like comparator, specifically into an error function for each constraint and a combining function which combines the errors at each level.

7.3.3.2 Distance functions which prefer a subset of the variables

In general CSPs are considered in terms of binary constraints. The theory can be extended, but complications are introduced. CLP, on the other hand, is indifferent to the arity of constraints. Therefore, if a PCSP problem has some kind of cost function which selects solutions which minimise the value of some function of (some of) the variables, we can simply treat it as another constraint. If the use of the cost function is expressed in the usual way ("Do not violate any constraints in order to minimise the function") then it can be labelled 'weak', while all the constraints in the original PCSP are labelled 'strong.' If it is acceptable to violate constraints in order to minimise the function, then the inverse strength labelling can be used.

7.3.3.3 Distance functions which prefer a subset of the constraints

This possibility can be transformed into HCLP in a very straightforward manner: the preferred constraints are labelled 'strong', while the others are labelled 'weak'. If there are multiple subsets with some order over them, then clearly more HCLP strength levels can be used.

7.3.3.4 Another FOPL description

Figure 7.5 contains a slightly more complicated FOPL description than the previous PCSP-to-HCLP figures in this chapter. The implication in Figure 7.3 is extended to include non-standard distance functions, i.e. those which treat some of the constraints as being more important than others. We assume that *Special* is a list of the distinguished constraints (or a list of lists if more than two classes of constraint are desired). Note that *within* the distinguished constraints the distance function is assumed to operate in the same manner as it does *within* the other constraints. If not, a single HCLP comparator could not be used.

```
transform-PCSP-HCLP( (Constraints, df(DF,Special)),
          (LabelledConstraints, Comparator) ) ⟵

   collect-unlabelled-constraints([UL-Strong, UL-Weak], (Constraints, df(DF,Special))) ∧
   remove-labels([Strong, Weak], [UL-Strong, UL-Weak]) ∧
   partition-labelled-constraints(LabelledConstraints, [Strong, Weak]) ∧
   comparator-distance-function(Comparator, DF).
```

Figure 7.5: PCSP into HCLP — non-standard distance functions

Special parameterises the distance function and will also be an additional parameter for various predicates, to allow the distinguished constraints to be placed in the 'strong' level while the others are considered to be 'weak'. This is why the previous specifications contained the variable *Strong* inside a list — so that the extension to the non-standard case would be easier. The changes to the subsidiary predicates necessitated by the extra parameters are obvious. However, this extension reduces the clarity of Figure 7.5, which is why we did not consider this case when defining the general equivalence between HCLP and PCSP. Nonetheless, it is clear that the equivalence could indeed be extended in this way.

7.4 Discussion and conclusions

7.4.1 Backwards and forwards

The transformations between HCLP and PCSP create a relation between them, i.e. we can imagine transforming an HCLP representation of a problem into PCSP, and then transforming the resulting PCSP problem into HCLP. If we end up with the same representation that we started with, we can say that the transformation

$$\text{HCLP} \longrightarrow \text{PCSP} \longrightarrow \text{HCLP}$$

is in fact an equivalence relation

$$\text{HCLP} \longleftrightarrow \text{PCSP}$$

We have shown this equivalence for the standard PCSP distance functions (Figure 7.4). Also, in principle we feel that the discussion and description in logic in Section 7.3.3 shows how to create such a relationship for non-standard distance functions.

7.4.2 Conclusions and benefits

We have developed a general methodology for transforming between HCLP and PCSP. We have clarified various issues, and provided a proof of correctness of the choice of

UCB for the augmentations distance function. We have shown that strength labels, associated with constraints in HCLP, contain information which is necessary to define the global distance function in PCSP. If we do not have an implementation of HCLP, we can replace a call to it by the two calls *transform, PCSP*, and vice-versa. Therefore problems which can be expressed in one of these two formalisms can also be expressed in the other.

HCLP and PCSP each have advantages when modelling problems, and each have advantages when implementing models and solving them. Using the work presented in this Part, the appropriate paradigm can be used for each of these steps, with a meaning-preserving transformation in between if necessary.

Part IV

Composition

In this part we discuss composition and how it relates to constraint-based systems. We then present our two-stage compositional variant of HCLP. Finally we discuss compositionality in PCSP, proving an important combining operator to be compositional and providing a heuristic condition for its general applicability.

Chapter 8

Compositionality

8.1 Introduction

Compositionality attracts attention for various reasons. At a very general level, it is related to the standard scientific principle of reductionism, i.e. describing a complex system in terms of the properties and interactions of its parts[1]. Systems with non-compositional models cannot be addressed in this manner.

Compositionality is a desirable property for programming paradigms to possess. It is beneficial from a software engineering view because it allows modularity, not just when writing programs but also when executing them. Therefore parallel and distributed implementation are made easier. At this level as well as at others, compositionality implies *de*compositionality, i.e. we can solve a complex problem by splitting it into simpler parts, solving them, and then composing the results into a complete solution.

Compositionality also makes it much easier to reason about the properties of a program, allowing different aspects to be considered separately. Furthermore, it is easier to make local optimisations, and as long as meaning is preserved at the local level it can be guaranteed that the global meaning is also unchanged. Of course, after the parts have been composed together, global optimisations can still be applied.

In the context of constraint programming, compositionality is a useful property for a system to have because it suggests that implementations will have the potential to be efficient. Standard HCLP is very expressive, but the efficiency of its implementations may be poor, and its semantics lacks certain desirable properties. Our proposal in this part of the thesis involves splitting HCLP into two parts to gain a more tractable semantics (Chapter 9). We then discuss compositionality in the context of PCSP; this has not been considered before, and the implicit assumption that PCSP is compositional

[1] In fact, in the philosophy of science reductionism is sometimes considered to have two aspects: (a) explanation of macroscopic effects in terms of microscopic causes, i.e. describing a system in terms of its parts, but also (b) explanation of *qualitative* change at the macroscopic level in terms of *quantitative* change at the microscopic (see e.g. Losee [54]). One could try and stretch this notion by saying that our bag-based approach in Chapter 9, in which the number of times a constraint appears can affect the solution to the problem, is an instance of (b), but that might be going too far.

is investigated in Chapter 10. We present a condition on the distance function which is sufficient to show that the standard PCSP distance functions are indeed compositional, when they have been interpreted in a certain way.

8.2 Compositionality

In constraint logic programming, efficiency is discussed with reference to 'incrementality', whereas in discussing semantics one characteristic that we look for is compositionality; these are the concerns of the rest of this section and the next one.

We say that a function or operator is compositional if it preserves certain structure that is relevant in a given situation. For example, set union is commutative and associative, but does not preserve number of occurrences. So union is compositional when we consider, say, a collection of raindrops merging together (as one drop plus another drop is still just one drop), but not when we consider the volume of the water involved. In the context of incremental implementations of CLP systems, we can say that a theory is compositional if the solution to the combination of two problems is the same as the combination of the solutions to the problems separately. More formally, if we have some kind of solution function or proof system ρ, if we can combine *problems* using $\sqcup_\rho$, and if we can combine *solutions* using $\circ_\rho$, then

$$\rho(A \sqcup_\rho B) = \rho(A) \circ_\rho \rho(B)$$

For our purposes, proving the compositionality of the ρ system will entail proving the associativity and commutativity of $\circ_\rho$, at least.

8.3 Incrementality

Whereas compositionality is a property of formal systems, incrementality is a (desirable) property of Constraint Logic Programming *implementations.* There is no precise definition, but what it means is that the work required to add an extra constraint to the solution of a large set of constraints and check its satisfiability[2] is proportional to the complexity of the addition, and not related to the size of the initial set. If a system is not incremental, then adding one more constraint to the solution of, say, 20 constraints, involves as much work as solving the system of 21 constraints from scratch.

In fact, even in an 'incremental' system the amount of work required to deal with an additional constraint will probably depend on more than just the constraint itself: the number of variables in the original set may be relevant, as well as other factors. This is not surprising, as too rigid a definition of compositionality would require $O(n)$ computational complexity, and certain varieties of constraint solving are in principle

[2]Smolka distinguishes between CLP, which requires an incremental test of constraint *satisfiability,* and the Concurrent Constraint paradigm, which requires an incremental test of constraint *entailment* as well [CompulogNet Newsletter].

exponential. (Restricted cases are not quite as computationally complex, but certainly remain much greater than linear.)

We wish to suggest that compositionality is weaker than incrementality: a theory which has compositional semantics may have a non-incremental implementation, often intentionally. For example, batch processing of queries in relational databases is much more efficient than line-at-a-time database accesses, notwithstanding the compositional nature of relational database theory. Conversely, a truly incremental implementation of a non-compositional formalism is difficult to define. What is more important than the distinction between incrementality and compositionality is the distinction between having both these properties and having neither, which is related to the distinction between 'sufficiently efficient' and 'unusably inefficient'. Both logic programming and constraint logic programming are in principle sufficiently efficient, and they have compositional theories (see Section 8.5), and so compositionality is assumed to hold in general, almost without being mentioned. Therefore, the focus in previous CLP work has tended to be on incrementality alone, rather than on its relationship with compositionality.

See Section 10 of Jaffar and Maher's survey [42] for more on incrementality, including various definitions and a detailed discussion of algorithmic complexity.

8.4 Partial compositionality

Throughout this thesis we are concerned with the application of preference systems to over-constrained problems. As the original problem is over-constrained, we cannot solve it precisely. Therefore we create various weakenings or relaxations of it, and then we place them in an order of preference. The first or equal-first elements in the order are the 'best' solutions we can hope for.

Within this area, one of our main themes is composition: can we find the solution to a combination of problems by combining the solutions to the individual problems. It turns out that we can usually combine the *elements* and that we can sometimes combine the *orders* as well; we characterise the appropriate orders later in this thesis.

The final task is to find the best element of the order, i.e. the best solution to the combined problem, in terms of the best elements of the individual solutions. Unfortunately, we cannot always succeed. If the best solution to one of the problems is incompatible with the best solution to the other, then the best solution to the combined problem will *not* be the combination of the best elements of the individual problems. Note that the best solution to the combination can indeed be the combination of the best individual solutions if they are not incompatible. More formally, if we combine problems using $\sqcup$ (constraint text [bag] union), and denote the relaxations of a problem P by $[P]$, and if we combine solutions using $\oplus$, and order them using $\leq$, then

$$\begin{array}{l}\mathrm{best}_{\leq}[P \sqcup Q] \neq \mathrm{best}_{\leq}[P] \oplus \mathrm{best}_{\leq}[Q] \\ \qquad \text{if } \mathrm{best}_{\leq}[P] \text{ is inconsistent with } \mathrm{best}_{\leq}[Q]\end{array}$$

This is a property of all preference systems even when the original problems are not (separately) over-constrained. If the desires modelled by P are not wholly compatible with those modelled by Q, it is necessary to compromise, and accept a solution which appears second-best to each party.

Therefore it is not sufficient for a compositional system to offer as solution only the best element; the entire order must be presented to the user. This is expensive in space, but cheaper computationally, as the alternative is to combine the problems and re-calculate.

As a result of this, in the rest of the thesis instead of saying that a theory is "compositional", we say that it is "as compositional as a preference system can be," by which we do not mean to imply that compositionality can have degrees, but that the theory is compositional under all the circumstances where any other theory could be.

8.5 Query composition in Logic Programming and CLP

Consider the two programs in Figure 8.1. They may be logic programs or constraint logic programs. The subscripts 1,2,3 are not part of the predicate names; they indicate that the query `?- p(X)` will have up to three solutions $\{\mathtt{P}_1, \mathtt{P}_2, \mathtt{P}_3\}$, one from each of the three clauses of `p`. The composed query `?- p(X)` $\sqcup$ `q(X)` will give rise to six computations, none or more of which may be successful. Therefore there ought also to be up to six solutions arising from $\circledast$, the composition, in some sense, of the elements of the cartesian product of the two sets of solutions. We could treat each of the $\{\mathtt{P}_i \circledast \mathtt{Q}_j\}$ combinations independently, and make a distinction between this multiplicity and multiple solutions all arising from one branch[3], but here we will just consider the collection of solutions as a whole, ignoring how they arose.

$\mathtt{p_1(X)}$ `:- ...` $\qquad$ $\mathtt{q_1(X)}$ `:- ...`

$\mathtt{p_2(X)}$ `:- ...` $\qquad$ $\mathtt{q_2(X)}$ `:- ...`

$\mathtt{p_3(X)}$ `:- ...`

`?- p(X).` $\qquad$ `?- q(X).`

$\{P_1,\ P_2,\ P_3\}$ $\qquad$ $\{Q_1,\ Q_2\}$

`?- p(X)` $\sqcup$ `q(X).`

$\{P_i \circledast\ Q_j \mid\ i = 1,2,3,\ j = 1,2\}$

Figure 8.1: Composing queries in logic programming

In standard logic programming, the composition of the solutions to two or more disjoint queries is the most general unifier (m.g.u.) of those solutions[4]. Calculating the m.g.u.

[3]For example, '$X > 4$' over the finite domain 0...10 will give rise to 6 solutions.

[4]Composition of the programs themselves, and of non-disjoint queries, is more complicated, requiring non-standard semantics [9].

of two solutions takes time related to the size of the solutions, and not related to the size of the original programs. Similarly, in constraint logic programming, solving the conjunction of two output sets of constraints will depend on the size of the output, not on the size or complexity of the input constraints. These two operations, of finding the m.g.u. and conjoining the constraints, are intensional, i.e. they deal with the abstract representation of the solutions. If we consider extensional solutions (models), then in both cases composition is just intersection/join.

In the next chapter we develop a compositional variant of HCLP.

Chapter 9

A Compositional Version of HCLP

We have previously discussed the difference between constraint theories, languages, and solving algorithms. In this chapter we present a new operational semantics for a language with exactly the same syntax as HCLP, and which gives the same results. Our semantics is based on bags, whereas HCLP is based on sets, and our work consists of two stages. Nonetheless, we will treat the work in this chapter as being a new solver for an old language, and not as a new language in its own right.

This approach may be contrasted with those of Parts III and V: the transformations in the former may be thought of as a pre-processing stage before using a standard ('old') implementation of standard languages, HCLP and PCSP. The integrated approach in the latter part is more radical, presenting a completely new language.

Remember that the non-compositionality of standard HCLP was shown in Section 3.3 using Wilson's proof of what she calls its 'disorderly' nature. We begin this chapter by presenting a new piece of mathematical machinery. Then we discuss the first stage of our version of HCLP, which we call BCH — which stands for Bags for the Composition of Hierarchies. BCH is compositional, but calculates a superset of the answers that would be expected from HCLP. We then present the second stage, which we call FGH — Filters, Guards, and Hierarchies. FGH removes precisely those BCH solutions which are superfluous according to HCLP, but re-introduces non-compositional behaviour. We then conclude the chapter by comparing BCH/FGH with standard HCLP and showing that it is equivalent.

9.1 Guards

The work presented in this chapter requires an alternative to bag intersection, with certain other properties, and so we now define a new infix binary operator $/\!\!/$ ('guarded by'). If A and B are both bags, then $A /\!\!/ B$ contains only those elements of A which are also in B, with the same multiplicity as in A. For set-like bags, $/\!\!/$ is the same as intersection (see Corollary 7 below).

Definition:

$$e\#(A \mathbin{/\!\!/} B) = \begin{cases} e\#A, & \text{if } e\#B > 0 \\ 0, & \text{if } e\#B = 0 \end{cases}$$

Examples:

$$\begin{aligned} \Lbag a, a, c \Rbag \mathbin{/\!\!/} \Lbag a, b \Rbag &= \Lbag a, a \Rbag \\ \Lbag a, a, c \Rbag \mathbin{/\!\!/} \Lbag b \Rbag &= \Lbag \Rbag \end{aligned}$$

We claim that BCH, to be defined below using guards, has a strong mathematical basis, including the fact that it is compositional. In order to defend this view, we need to prove that $/\!\!/$ is not a trivial or nonsensical operation. We do this by showing that it has various properties (such as associativity, distributivity through $\uplus$, etc.), and by showing that under certain circumstances it is equivalent to intersection. We present claims of associativity and partial commutativity here, as well as certain propositions showing the precise relationship between guard and intersection. Some of the propositions are immediate from the definition of $/\!\!/$, and the others have straightforward proofs. We do not present all the proofs, but enough to show that the others may be constructed without difficulty.

Convention: Union binds tighter than guard, i.e. $A \uplus B \mathbin{/\!\!/} C$ is to be interpreted as $(A \uplus B) \mathbin{/\!\!/} C$. Also, $/\!\!/$ is assumed to obey the standard left-associativity rule, i.e. $A \mathbin{/\!\!/} B \mathbin{/\!\!/} C$ is to be interpreted as $(A \mathbin{/\!\!/} B) \mathbin{/\!\!/} C$.

Proposition 2: Idempotence
$A \mathbin{/\!\!/} A = A$.

Proof:

$$e\#(A \mathbin{/\!\!/} A) = \left\{ \begin{array}{ll} e\#A, & \text{if } e\#A > 0 \\ 0, & \text{if } e\#A = 0 \end{array} \right\} = e\#A$$

□

Proposition 3: Empty Bag
$A \mathbin{/\!\!/} \Lbag \Rbag = \Lbag \Rbag \mathbin{/\!\!/} A = \Lbag \Rbag$.

The proof is obvious from the definition of $/\!\!/$.

Proposition 4: Set Distributivity
$\mathbf{set}(A \mathbin{/\!\!/} B) = \mathbf{set}(A) \mathbin{/\!\!/} \mathbf{set}(B)$.

Proof:
Reminder of the definition of **set**, which turns a bag into a set-like bag by collapsing multiple occurrences of an element into a single occurrence:

$$e\#\mathbf{set}(A) = \begin{cases} 1, & \text{if } e\#A \geq 1 \\ 0, & \text{otherwise} \end{cases}$$

Using this definition to expand the left-hand side of the proposition, we get:

$$e\#(\mathbf{set}(A /\!\!/ B)) = \begin{cases} 1, & \text{if } e\#A > 0 \text{ and } e\#B > 0 \\ 0, & \text{otherwise} \end{cases}$$

whereas the right-hand side gives

$$e\#(\mathbf{set}(A) /\!\!/ \mathbf{set}(B)) = \begin{cases} e\#\mathbf{set}(A), & \text{if } e\#\mathbf{set}(B) > 0 \\ 0, & \text{otherwise} \end{cases}$$

Clearly $e\#\mathbf{set}(A) = 1$ iff $e\#A > 0$, and $e\#\mathbf{set}(B) > 0$ iff $e\#B > 0$.

In fact, $\mathbf{set}(A) /\!\!/ \mathbf{set}(B)$ also equals $\mathbf{set}(A) /\!\!/ B$, and also, by the definition of $/\!\!/$, $A /\!\!/ B = A /\!\!/ \mathbf{set}(B)$. □

— — — — — — — — —

The next two propositions show the relationship between intersection and guarding, including the corollary that if both the bags are set-like, then guard is equivalent to intersection.

Proposition 5: Intersection Guard 1
$A \cap B \subseteq A /\!\!/ B$.

Proof:
$e\#(A \cap B)$ is defined to equal the minimum of $e\#A$ and $e\#B$, so the left-hand side of the relation to be shown can be written as follows:

$$e\#(A \cap B) = \begin{cases} 0, & \text{if } e\#A = 0 \\ 0, & \text{if } e\#B = 0 \\ e\#A, & \text{if } e\#B \geq e\#A, \text{ and both are non-zero} \\ e\#B, & \text{if } e\#B < e\#A, \text{ and both are non-zero} \end{cases}$$

The definition of $A /\!\!/ B$ involves $e\#A$ as long as $e\#B > 0$. So the right-hand side of the relation can be written:

$$e\#(A /\!\!/ B) = \begin{cases} 0, & \text{if } e\#A = 0 \\ 0, & \text{if } e\#B = 0 \\ e\#A, & \text{if } e\#B \geq e\#A, \text{ and both are non-zero} \\ e\#A, & \text{if } e\#B < e\#A, \text{ and both are non-zero} \end{cases}$$

The first three lines of the expanded definitions are equal. The fourth line of the left-hand side gives $e\#B$ which is less than $e\#A$, whereas the fourth line of the right-hand side gives precisely $e\#A$. So by the definition of the sub-bag relation, which is given in terms of $\leq$, $A \cap B \subseteq A /\!\!/ B$. □

Proposition 6: Intersection Guard 2
$\mathbf{set}(A) \cap B = \mathbf{set}(A) /\!\!/ B$.

Proof:
Note the proof of Proposition 5. The only line in which the two sides of the relation differ is the one where $e\#B < e\#A$ and both numbers of occurrences are non-zero. But if A is set-like, then $e\#A$ can only equal 0 or 1, so $e\#B$ cannot be less than $e\#A$ (as $e\#B$ is non-zero). So the fourth line cannot arise. Lines 1–3 *can* arise (when the non-strict inequality in line 3 of the proof of Proposition 5 becomes an equality). So the left-hand side equals the right-hand side in all the applicable cases. $\square$

Corollary 7: Guard Intersection Equivalence
Hence, if $A = \mathbf{set}(A)$ (and $B = \mathbf{set}(B)$) then $A \cap B = A \mathbin{[\!]} B$.

Proof:
Immediate, from Proposition 6. $\square$

— — — — — — — — —

It is clear that $[\!]$ is not commutative in general, i.e. $A \mathbin{[\!]} B \neq B \mathbin{[\!]} A$, but it is in the case of A and B set-like:

Corollary 8: Partial Commutativity
If $A = \mathbf{set}(A)$ and $B = \mathbf{set}(B)$ then $A \mathbin{[\!]} B = B \mathbin{[\!]} A$.

Proof:
Immediate, from Proposition 6, given the commutativity of $\cap$. $\square$

Proposition 9: Left (Sub-bag) Absorption
If $A \subseteq B$ then $A \mathbin{[\!]} B = A$.

The proof is obvious from the definitions of $[\!]$ and $\subseteq$. In fact, we only need the weaker pre-condition that $\mathbf{set}(A) \subseteq B$.

If A and B are both set-like another corollary of Proposition 6 holds:

Corollary 10: Right (Sub-bag) Absorption
If $A = \mathbf{set}(A)$ and $B = \mathbf{set}(B)$ and if $B \subseteq A$, then $A \mathbin{[\!]} B = B$.

Corollary 11: Summary: Set and Intersection Distributivity
$\mathbf{set}(A \mathbin{[\!]} B) = \mathbf{set}(A) \mathbin{[\!]} \mathbf{set}(B) = \mathbf{set}(A) \cap \mathbf{set}(B) = \mathbf{set}(A \cap B)$.

— — — — — — — — —

$[\!]$ is associative in all cases:

Proposition 12: Guard Associativity
$A \mathbin{[\!]} (B \mathbin{[\!]} C) = (A \mathbin{[\!]} B) \mathbin{[\!]} C$.

Proof:
Immediate, by expanding the definition of both sides. $\square$

Proposition 13: Union Distributivity
$(A \uplus B) /\!\!/ C = (A /\!\!/ C) \uplus (B /\!\!/ C)$.

Proofs of the previous proposition and the next one follow immediately from the definitions.

Proposition 14: Intersection Distributivity
$A /\!\!/ (B \cap C) = (A /\!\!/ B) \cap (A /\!\!/ C)$.

Proposition 15: Intersection Conversion
$A /\!\!/ (B \cap C) = A /\!\!/ (B /\!\!/ C)$.

By associativity of $/\!\!/$ we can also see that $A /\!\!/ (B \cap C) = (A /\!\!/ B) /\!\!/ C$.

$/\!\!/$ also has the property that $(A /\!\!/ B) \cap C = (A \cap C) /\!\!/ B$.

9.2 BCH — Bags for the Composition of Hierarchies

9.2.1 Compositionality of BCH

We now define an associative, commutative composition operator $\circ$ in terms of its effect on solutions to individual hierarchies $p_1, p_2, \ldots, p_k$, which we assume have each already been calculated separately (by some method to be discussed below). This operator defines what we mean when we refer to BCH or BCH$_\circ$:

Definition (BCH Compose):

$$
\begin{aligned}
\mathop{\circ}_i S_0(p_i) &= \bigcap_i S_0(p_i) \\
\mathop{\circ}_i S_n(p_i) &= \left(\biguplus_i S_n(p_i) \right) /\!\!/ \mathop{\circ}_i S_0(p_i) \qquad (n > 0)
\end{aligned}
$$

Note that all these levels are guarded by the solution to level 0, i.e. the required constraints. Solutions to p_i's optional constraints which contradict p_j's required constraints are removed by the guard, thus any valuation found lower down the hierarchy will be acceptable to all the required constraints[1]. But certain valuations are present which are not as preferred as others — these rules have no analogue for comparators in HCLP (but see below, Section 9.3). Consequently, as each remaining element of the S_i's satisfies the required constraints we can say that the rules 'respect the requireds'. But as elements may be present in S_2 which are not in S_1, say, these rules do *not* 'respect the hierarchy'.

Note that we sometimes abuse notation in the following manner: if p, q and r are the names of three problems, then we use $p \circ q = r$ as a shorthand for $\forall\, n \cdot S_n(p) \circ S_n(q) =$

[1]If the optional constraints contradict the required ones they are ignored. In this case in HCLP $S_1(p) = S_0(p)$ and in BCH $S_1(p) = \{\!\!\{\}\!\!\}$, both of which satisfy $S_1(p) \subseteq S_0(p)$.

$S_n(r)$. In other words, it may appear that $\circ$ is being applied to problems, but in fact it is being applied to their solutions.

It is clear that $\circ$ is commutative, due to the commutativity of intersection and union of bags. To show that $\circ$ preserves all that we require under composition, it is necessary to prove that it is associative.

Proposition 16: Compose Associativity
$(p \circ q) \circ r = p \circ (q \circ r)$

Proof:
Case S_0: obvious, by the associativity of $\cap$.

Case S_n $(n > 0)$: In the following, let P_0 be an abbreviation for $S_0(p)$, let p_n mean $S_n(p)$, and similarly for Q_0, R_0, q_n, and r_n.

$$
\begin{aligned}
& e\#((S_n(p) \circ S_n(q)) \circ S_n(r)) \\
&= e\#(((p_n \uplus q_n) /\!\!/ (P_0 \cap Q_0)) \uplus r_n) /\!\!/ (P_0 \cap Q_0 \cap R_0) && \{\text{by defn. of } \circ\} \\
&= \begin{cases} 0, \text{ if } e\#P_0 = 0 \text{ or } e\#Q_0 = 0 \text{ or } e\#R_0 = 0 \\ e\#((p_n \uplus q_n) /\!\!/ (P_0 \cap Q_0)) + e\#r_n, \text{ otherwise} \end{cases} && \{\text{by defn. of } /\!\!/\} \\
&= \begin{cases} 0, \ \ldots \\ e\#p_n + e\#q_n + e\#r_n \end{cases} && \{\text{see (1) below}\} \\
&= \begin{cases} 0, \ \ldots \\ e\#p_n + e\#((q_n \uplus r_n) /\!\!/ (Q_0 \cap R_0)) \end{cases} && \{\text{see (2) below}\} \\
&= e\#(((q_n \uplus r_n) /\!\!/ (Q_0 \cap R_0)) \uplus p_n) /\!\!/ (P_0 \cap Q_0 \cap R_0) && \{\text{by defn. of } /\!\!/\} \\
&= e\#(S_n(p) \circ (S_n(q) \circ S_n(r))) && \{\text{by defn. of } \circ\}
\end{aligned}
$$

(1) Non-zero ('otherwise') case: as $e\#P_0 \neq 0$ and $e\#Q_0 \neq 0$
(2) Non-zero ('otherwise') case: as $e\#Q_0 \neq 0$ and $e\#R_0 \neq 0$

□

The above proof is based on one due to Sebastian Hunt. Previously we used a proof we had developed independently ourselves, based on some of the properties discussed in Section 9.1, but it was about twice as long as the version presented here. Sebastian Hunt proved the proposition directly from the definition of $/\!\!/$, as opposed to proving it from properties which were themselves proved using that definition as we had done. This was not done at our request, but of course we prefer to use a proof which is shorter and clearer.

9.2.2 BCH solutions to individual hierarchies

The previous section was concerned with composing solutions to hierarchies; but where do these solutions come from in the first place? One possibility is to assume that HCLP has been used initially, with BCH only being invoked to compose the previously calculated answers in order to avoid starting HCLP from scratch on the composed program texts. All the theory in this chapter is well-defined for this situation (treating the various solution sets $S_1, \ldots, S_n$ as set-like bags), however in the rest of this section we discuss the second possibility, namely that BCH is used for everything. When we use BCH to solve an individual hierarchy we label the rules $\mathrm{BCH}_{\mathsf{indiv}}$; when used to compose solutions the terms BCH or $\mathrm{BCH}_{\circ}$ are used.

Note that the answers for a single hierarchy defined by $\mathrm{BCH}_{\mathsf{indiv}}$ will not be the same as those calculated by HCLP (the latter will be a subset of the former, as discussed below). The answer to a composition of solutions of course depends on the individual solutions; as these will differ depending on whether $\mathrm{BCH}_{\mathsf{indiv}}$ or HCLP was used to find them, the answers to the composition will also differ. But given the solutions to the sub-problems, then the solution to their composition calculated by BCH is completely determined.

We now define what we mean by $\mathrm{BCH}_{\mathsf{indiv}}$: it is the method of calculating the solution to an individual hierarchy, defined as follows. The solution to a hierarchy P is defined to be a tuple $\langle S_0(P), S_1(P), \ldots, S_n(P)\rangle$ where each S_i is the solution to the constraints at level i, guarded by S_0. To solve the required constraints and find S_0, we invoke the appropriate CLP solver. In fact, for finite domains this is equivalent to taking the intersection/join of the extensions of the individual constraints. To 'solve' the optional constraints, take the (bag- additive-) union[2] of the extensions of the individual constraints. Then guard each of the S_i, $i > 0$, with S_0, thus removing solutions which violate the required constraints. We can state this formally, using $H_0(P)$ to indicate all the required constraints in P, $H_i(P)$ to indicate all the optional constraints at level i of the hierarchy, and $\biguplus_j \mathrm{dom}(c_j)$ to indicate the union of the domains or extensions of the constraints c_j:

Definition ($\mathbf{BCH}_{\mathsf{indiv}}$):

$$\begin{aligned}
\mathrm{BCH}_{\mathsf{indiv}}(P) &= \langle S_O(P), S_1(P), \ldots, S_n(P)\rangle \\
\text{where } S_0(P) &= \left(\bigcap_i \mathrm{dom}(c_i),\, c_i \in H_0(P)\right) \\
\text{and } S_j(P) &= \left(\biguplus_i \mathrm{dom}(c_i),\, c_i \in H_j(P)\right) /\!\!/ S_0(P) \qquad\qquad (j > 0)
\end{aligned}$$

This definition of the solution to an individual hierarchy may seem extravagant, but it becomes less so in the context of filter functions such as $\mathbf{f}_{\max}$ (Section 9.3.2).

[2] In Section 7.3.1.2, we mentioned that when we said 'intersection' of constraint domains we really meant the intersection of what we called the 'expansion' of the constraint domains, i.e. the relational database concept 'join'. Similarly, here we do not really mean union, unless we are dealing with expansions, but the bag counterpart of the relational database concept of restricted relational product $\times_{\mathsf{rest}}$ (both triples are included):

$$X \underset{\mathsf{rest}}{\times} Y \quad = \quad \{(a,b,d) \mid (a,b) \in X, (c,d) \in Y\} \;\cup\; \{(a,c,d) \mid (a,b) \in X, (c,d) \in Y\}$$

However, using 'union' should not be misleading.

9.2.2.1 Relationship between HCLP and $\text{BCH}_{\text{indiv}}$

In the above discussion each of the non-required levels contains the union of the extensions of the constraints at that level. HCLP would also produce at most the union of the extensions (if all the constraints were consistent with all stronger levels, but also mutually contradictory). At the other extreme, if all the constraints were mutually consistent, then the HCLP solution would be the *intersection* of the extensions. If we removed the labels from all the constraints at a given level and solved them using CLP, the result would be the (possibly empty) intersection of the extensions. Therefore the solutions for each level satisfy the following relation:

$$\text{CLP} \subseteq \text{HCLP} \subseteq \text{BCH}_{\text{indiv}}$$

9.2.2.2 Empty solutions for a level

In HCLP, the key difference between required and non-required constraints is that the former can cause failure to occur. In other words, the required constraints may have an empty solution set, but no weaker constraints can cause a failure if stronger constraints have been satisfied[3]. In our composition rules, in $A \mathbin{[\!]} B$, A represents the solutions for a weaker level than B, and yet the definition of $[\!]$ allows the possibility of $A \mathbin{[\!]} B$ being empty even if B is not (in the case that $A \cap B = \lbag\rbag$). Thus our rules appear to allow failure to arise from optional constraints. In fact, as we define the solution S to be the tuple $\langle S_0, S_1, \ldots, S_n \rangle$ and not just its final element S_n (for use by FGH; see Section 9.3), it is not a problem if one of the elements of S is empty: the solution that is offered to the user is no longer the final element of the tuple, but all the non-empty elements. Thus this aspect of HCLP is not present in our logic, but is left until the interpretation.

9.2.2.3 No constraints at a level

If there are no constraints at the *required* level, the solution set is the entire domain (the cartesian product of the domains of all the variables) which we denote by saying $S_0 = \mathcal{U}$. It can be seen that guarding any bag with this universal set will not have any effect: $\forall A \cdot A \mathbin{[\!]} \mathcal{U} = A$. If there are no constraints at one of the optional levels, the BCH solution for that level will be $\mathcal{U} \mathbin{[\!]} S_0 = S_0$. Therefore, we can see that the rules we have defined here will work for hierarchies with arbitrarily many or few constraints at any given level.

9.2.2.4 Equivalence of $\text{BCH}_{\text{indiv}}$ and BCH_{o}

Proposition 17:
As long as all levels contain constraints, and if $\sqcup$ *represents constraint text union (i.e.*

[3]In fact we need a slightly stronger condition in HCLP; if S_0 is non-empty *and finite* then we can be certain that S_n will not be empty. The extra condition is necessary to deal with certain pathological cases, such as the hierarchy 'required $X > 0$, strong $X = 0$' with a metric comparator [7].

constraint conjunction), then $BCH_{indiv}(A) \circ BCH_{indiv}(B) = BCH_{indiv}(A \sqcup B)$

Proof:
The definition of $\mathrm{BCH_{indiv}}$ requires the extensions of each constraint at a level to be combined using union ($\uplus$). The definition of $\circ$ also combines equivalent levels from two hierarchies using union. The distributivity of $\uplus$ through $/\!\!/$ guarantees that the order in which the various combinations are done is unimportant. Hence the proposition is true. Note that the left-hand-side of the proposition defines the $\mathrm{BCH_{\circ}}$ solution (i.e. using $\circ$ to compose two previously calculated hierarchies) and so we can see that the difference between $\mathrm{BCH_{indiv}}$ and $\mathrm{BCH_{\circ}}$ is that $\mathrm{BCH_{indiv}}$ defines the meaning of $\langle\_\rangle$, whereas $\mathrm{BCH_{\circ}}$ composes $\langle\_\rangle$'s, however they are defined. □

9.2.2.5 Relationship between HCLP and $\mathrm{BCH_{\circ}}$

By a similar argument to that on the relationship between HCLP and $\mathrm{BCH_{indiv}}$ (see Section 9.2.2.1 above), and considering Proposition 17, we can see that HCLP $\subseteq$ $\mathrm{BCH_{\circ}}$.

9.2.3 Complexity and incrementality of BCH

9.2.3.1 Complexity of BCH

The following is a summary; for detailed calculations, see Appendix C.1.

Using a naïve representation, the complexity of composing k hierarchies each with n solutions to their required constraints (*not* necessarily the number of solutions from n constraints) and l levels of optional constraints is $O(lkn^2)$ in the worst case. Using a more sensible representation, and noting that k and l are likely to be fixed at small values, the total complexity of BCH is $O(n)$.

This result is expressed in terms of basic element comparison operations, i.e. whether a member of one bag is identical with a member of another bag. Comparing these results to the complexity of HCLP is difficult, partly because no analysis of the latter exists in the literature; but note that whatever results might be obtained are likely to be in terms of constraint checks, an inherently more difficult operation than simple element comparison.

9.2.3.2 Incrementality of BCH

More interesting than standard complexity results is the question of incrementality. If we have already composed k hierarchies using BCH, what is the extra work required to compose one additional hierarchy?

Let us assume that the previously composed system has a total of n elements in S_0, and hence at most n distinct elements in each of the optional levels. Let us assume that the new hierarchy has at most m elements per level. Let l be the maximum of the number of optional levels in the two systems. Calculating the new S_0 requires the intersection of two sets, of size m and n. This will take $O(mn)$ at worst, and the resulting set will have size at most $\max(m,n)$. For each optional level, finding the union of two bags takes $O(m+n)$, and then guarding with S_0 takes at most $O(\max(m,n).(m+n))$. If we make the assumption that m is as large as n, this expression becomes $O(n^2)$.

Therefore when we include the calculations on l optional levels the total complexity is $O(mn + l.\max(m,n).(m+n)) = O(l.\max(m,n).(m+n)) = O(ln^2)$ (setting $m = n$). This could be improved by guarding each level of the previously composed system with the required level of the new hierarchy *before* taking the union of the optional levels, and by other methods. But even so, it is clear that the work required will be approximately quadratic rather than linear in m. Therefore, perhaps, we should not claim that BCH is incremental. But note that k, the number of systems which were originally composed together, before the addition of the new hierarchy, is not mentioned in these results. In other words, the work done by BCH may depend on the number of *solutions* in the previously composed system, but not on the number of original hierarchies. We are able to avoid starting from scratch, re-calculating the composition of k systems, which is not the case in HCLP. So we have achieved the goal set in Chapter 8.

9.3 FGH — Filters, Guards and Hierarchies

9.3.1 Filter functions

This section describes FGH, which takes the results of BCH and uses them to calculate solutions equivalent to those of HCLP. Unlike BCH, FGH respects the hierarchy, and so is disorderly and non-monotonic, hence non-compositional.

FGH uses an extra mathematical construct, a class of filters, functions from bags to bags, which remove some of the elements from the input bag. It is necessary to introduce them here because we will use them in the next section to define rules for removing less preferred valuations from BCH solutions. Particular examples are discussed later, but in general filter functions will be denoted by **f**, and **F** will be used to denote the raised version (i.e. **f** applied to each member of a tuple of bags). Note that the guard operator *//* is concerned with the relationship between *different strength levels* in a hierarchy, whereas filter functions select solutions *within* a given level. (Similarly, in HCLP the comparators compare solutions within a given level.) More details can be found in the next section, where we examine one particular filter at length.

The whole process described in the rest of this section can be summarised as showing that the following equation holds, and asking if the right-hand side is more efficient or has better semantics than the left-hand side:

$$\text{HCLP}(P \sqcup Q) = \mathbf{F}(P \circ Q)$$

The answer is given in Section 9.5.2, below.

We now define an operation F which removes elements from bags in a tuple, forming new bags. Let P be a program with a BCH solution $\langle S_0(p), S_1(p), \ldots\rangle$. Note that $S_1(p) \subseteq S_0(p)$, $S_2(p) \subseteq S_0(p)$, etc., but it is not necessarily the case that $S_2(p) \subseteq S_1(p)$, etc. As we have already used $p \circ q$ to refer to a BCH operation we will describe the FGH extension as $\mathbf{F}(p \circ q)$, i.e. $\mathbf{F}(S_n(p) \circ S_n(q))$ for each level n of the hierarchy, where $\mathbf{F}(p)$ is defined as follows:

Definition F:

$$\begin{array}{lcll}
\mathbf{F}\langle S_0(p), S_1(p), \ldots, S_n(p)\rangle & & & \\
 & = & \langle \mathbf{F}(S_0(p)), \mathbf{F}(S_1(p)), \ldots, \mathbf{F}(S_n(p)\rangle & \\
\mathbf{F}(S_0(p)) & = & S_0(p) & \\
\mathbf{F}(S_n(p)) & = & \mathbf{f}\,(S_n(p) /\!\!/ \mathbf{F}(S_{n-1}(p))) & (n > 0) \\
 & & & \\
sols(\mathbf{F}(P)) & = & \textbf{set} \text{ applied to the final non-empty bag in the tuple} & \\
 & & \qquad \langle S_0(p), S_1(p), \ldots, S_n(p)\rangle &
\end{array}$$

where **f** is a filter function, applied to the result of guarding a level with the previous level (itself filtered). Note that we overload the symbol **F**, allowing it to operate both on tuples and on members of tuples.

The reason for filtering each level is to remove solutions which are less preferred than others. The reason for guarding S_n with level S_{n-1} is to remove solutions which are acceptable to level n, but less preferred by the stronger constraints of level $n-1$. Of course, we also need to remove solutions which are less preferred by levels $n-2$, $n-3$, etc., but this is dealt with by the fact that S_{n-1} has itself been guarded by S_{n-2}, and so it is not necessary to guard S_n by S_{n-2} explicitly.

Note that in the definition of **F** itself we have not included an application of **set** which would reduce the number of occurrences of each element to 1. We consider it to be the very last action that should be performed on the bags of solutions.

9.3.2 The filter $\mathbf{f}_{\max}$

The rules defined in the previous section are parameterised by filter functions. We now define one particular filter function $\mathbf{f}_{\max}$, which is the most interesting when compared to HCLP's 'unsatisfied-count-better' comparator. Other comparators and their related filters are discussed later, in Section 9.4. $\mathbf{f}_{\max}$ removes those elements of a bag which do not occur a maximal number of times. In other words, if some elements occur once in a given bag, and some elements occur twice, $\mathbf{f}_{\max}$ defines the bag containing only those elements occurring twice.

Definition:

$$e \;\#\; \mathbf{f}_{\max}(A) = \begin{cases} e\#A, & \text{if } e\#A = max\{i\#A \cdot i \in A\} \\ 0, & \text{otherwise} \end{cases}$$

Example:

$$\begin{aligned}\mathbf{f}_{\max}\wr a, a, b\wr &= \wr a, a\wr \\ \mathbf{f}_{\max}\wr a, b\wr &= \wr a, b\wr\end{aligned}$$

$\mathbf{F}_{\max}$ is defined as $\mathbf{f}_{\max}$ raised to the level of tuples of bags in the obvious way (similar to Definition **F**, above).

Example:
F was defined in terms of a single hierarchy p, but p may arise from the BCH composition of two or more hierarchies. For this example, let us assume that we have two hierarchies q and r with the following solution sets: $S_0(q) = \{a, b, c\}$ and $S_0(r) = \{b, c, d\}$. Let $S_1(q) = \{b, c\}$ and $S_1(r) = \{b, d\}$. Then $S_0(q) \circ S_0(r) = \{b, c\}$ and $S_1(q) \circ S_1(r) = (\wr b, b, c, d\wr /\!\!/ (S_0(q) \circ S_0(r))$.

$$\begin{aligned}\mathbf{F}_{\max}(S_1(q) \circ S_1(r)) &= \mathbf{f}_{\max}(\wr b, b, c, d\wr /\!\!/ \wr b, c\wr) \\ &= \mathbf{f}_{\max}(\wr b, b, c\wr) \\ &= \wr b, b\wr\end{aligned}$$

So the BCH solution is $\langle \wr b, c\wr, \wr b, b, c\wr\rangle$ and the FGH($\mathbf{f}_{\max}$) solution is $\langle \wr b, c\wr, \wr b, b\wr\rangle$; therefore the preferred solution to p is the value b as it is the only element of the final non-empty bag in the solution tuple, where the final bag has been affected by all the levels of optional constraints. Thus BCH allows us to explore the *space* of solutions, including different values in each bag and all the bags in the tuple, while FGH produces its *best* possibilities, i.e. the best value(s) of the final non-empty bag of the solution tuple.

For simplicity, the example above concerned possible values for a single variable (i.e. $X = a$ or $X = b$, for some variable X). But it is clear that everything can be defined in terms of possible values for a vector $\mathbf{X} = [X_1, X_2, \ldots, X_n]$. In this case, # counts the number of occurrences of a particular vector and not simply a particular value for a single variable. For example, the bag $\wr\langle a, b\rangle, \langle a, c\rangle\wr$ contains two distinct 'elements', corresponding to $[X_1 = a, X_2 = b]$ and $[X_1 = a, X_2 = c]$.

9.3.3 Complexity of FGH

It is clear that **F** requires l filtering steps for a hierarchy with l levels, using any given **f**.

If $\mathbf{f}_{\max}$ is used, for each level it is necessary to sort by number of occurrences (greatest first). If there are up to k occurrences of each of n distinct elements in a level, sorting will take $O(kn \log kn)$ simple comparison operations (or $O(n \log n)$ if a sensible representation is used). Keeping a note of the maximum number of occurrences of an element found so far, it is easy to select the first part of the output bag, i.e. those elements occurring precisely k times. This process must be repeated for each level, and so the complexity of FGH using $\mathbf{f}_{\max}$ is $O(lkn \log kn)$, where in practice k and l are likely to be fixed at quite small values, leading to a complexity of $O(n \log n)$.

All the above is in addition to the complexity of the BCH step, which was found to be $O(lkn)$, or $O(n)$ under some sensible assumptions (see Section 9.2.3.1). So we have a total complexity of $O(n) + O(n \log n) = O(n \log n)$.

9.3.4 Non-compositionality of FGH

We can show that the FGH scheme is not compositional in general by showing that it is not compositional for one filter function, $\mathbf{f}_{\max}$. An alternative would be to show that it is disorderly, just like HCLP. We follow both these possibilities in the next two subsections.

9.3.4.1 Non-compositionality of FGH($\mathbf{f}_{\max}$)

We can give a simple example why FGH($\mathbf{f}_{\max}$) is not compositional as follows: consider two bags $\wr a, b \wr$ and $\wr b, c \wr$. Composing them with BCH gives $\wr a, b, b, c \wr$, and then filtering with $\mathbf{f}_{\max}$ will result in $\wr b, b \wr$. If the third bag contains $\wr a, c \wr$, then composing it with $\wr b, b \wr$ followed by filtering will give the incorrect answer $\wr b, b \wr$. However, if we had composed it at the BCH stage (getting $\wr a, a, b, b, c, c \wr$), filtering would have resulted in a different answer. So FGH is not associative, hence not compositional.

Therefore if we wish to add a constraint to the results of filtering a tuple of bags, we cannot do it directly. We must compose the BCH solution tuple for the new constraint (which will be very simple to calculate) with the BCH solution of the original hierarchy, and then re-filter. Thus FGH is not really incremental in its own right. Of course, BCH *is* incremental, and it is relatively cheap to re-filter using FGH[4], and so we still have some gain compared to HCLP, which must restart right from the very beginning.

9.3.4.2 The disorderly property of FGH

Let us consider again the example which was used in the proof of the disorderly nature of HCLP (Section 3.3). The two hierarchies were $P = \{$`weak X = a`$\}$ and $Q = \{$`strong X = b`$\}$ and the solutions calculated by BCH were as follows, where as usual $\sqcup$ stands for constraint text union:

$$\begin{aligned}
\mathrm{BCH}(P) &= \langle \mathcal{U}_X, \mathcal{U}_X, \wr a \wr \rangle \\
\mathrm{BCH}(P \circ Q) &= \langle \mathcal{U}_X, \mathcal{U}_X \uplus \wr b \wr, \mathcal{U}_X \uplus \wr a \wr \rangle \\
\mathrm{BCH}(P \sqcup Q) &= \langle \mathcal{U}_X, \wr b \wr, \wr a \wr \rangle
\end{aligned}$$

[4]The computational complexity is increased by only a logarithmic factor, assuming a complexity of $O(n)$ for BCH and $O(n \log n)$ for FGH.

The FGH solutions are:

$$\begin{aligned}
\mathbf{F}_{\mathsf{max}}(P) &= \langle \mathcal{U}_X, \mathcal{U}_X, \Lbag a \Rbag \rangle \\
\mathbf{F}_{\mathsf{max}}(P \circ Q) &= \langle \mathcal{U}_X, \Lbag b, b \Rbag, \Lbag b \Rbag \rangle \\
\mathbf{F}_{\mathsf{max}}(P \sqcup Q) &= \langle \mathcal{U}_X, \Lbag b \Rbag, \Lbag \Rbag \rangle
\end{aligned}$$

So for this example $\mathbf{F}_{\mathsf{max}}(P \sqcup Q) \subseteq \mathbf{F}_{\mathsf{max}}(P) \not\subseteq \mathbf{F}_{\mathsf{max}}(P \circ Q)$, which does not yet demonstrate the disorderliness of FGH.

Now consider a different hierarchy $R = \{$`weak X = a, strong X = c`$\}$. Then

$$\begin{aligned}
\mathrm{BCH}(R) &= \langle \mathcal{U}_X, \Lbag c \Rbag, \Lbag a \Rbag \rangle \\
\mathbf{F}_{\mathsf{max}}(R) &= \langle \mathcal{U}_X, \Lbag c \Rbag, \Lbag \Rbag \rangle
\end{aligned}$$

$$\begin{aligned}
\text{Also } \mathrm{BCH}(R \circ Q) &= \langle \mathcal{U}_X, \Lbag b, c \Rbag, \mathcal{U}_X \uplus \Lbag a \Rbag \rangle \\
\text{and } \mathrm{BCH}(R \sqcup Q) &= \langle \mathcal{U}_X, \Lbag b, c \Rbag, \Lbag a \Rbag \rangle
\end{aligned}$$

$$\begin{aligned}
\text{Therefore } \mathbf{F}_{\mathsf{max}}(R \circ Q) &= \langle \mathcal{U}_X, \Lbag b, c \Rbag, \Lbag b, c \Rbag \rangle \\
\text{and } \mathbf{F}_{\mathsf{max}}(R \sqcup Q) &= \langle \mathcal{U}_X, \Lbag b, c \Rbag, \Lbag \Rbag \rangle
\end{aligned}$$

As $\Lbag b, c \Rbag \not\subseteq \Lbag c \Rbag$, we can see that $\mathbf{F}_{\mathsf{max}}(R \sqcup Q) \not\subseteq \mathbf{F}_{\mathsf{max}}(R)$, thus demonstrating the disorderliness of FGH.

9.4 Other filter functions

Although the previous sections have concentrated on $\mathbf{f}_{\mathsf{max}}$, other filter functions are possible. It is interesting to define filter functions which mimic the various HCLP comparators; whereas $\mathbf{f}_{\mathsf{max}}$ is derived filters from *unsatisfied-count-better* (UCB) one can imagine other filters based on metric comparators such as *least-squares-better* (LSB). These might not simply filter existing elements from bags, but could actually create new ones. For example, if $A = \Lbag 2, 5, 5 \Rbag$, $\mathbf{f}_{\mathsf{max}}(A) = \Lbag 5, 5 \Rbag$, but $\mathbf{f}_{lsb}(A) = \Lbag 3, 3, 3 \Rbag$ (same cardinality as A).

9.4.1 The filter $\mathbf{f}_k$ can be equivalent to intersection

Definition:

$$e \;\#\; \mathbf{f}_k(A) = \begin{cases} e\#A, & \text{if } e\#A \geq k \\ 0, & \text{otherwise} \end{cases}$$

Example:

$$\begin{array}{lcll} \mathbf{f}_1\wr a,a,b\wr & = & \wr a,a,b\wr & \\ \mathbf{f}_2\wr a,a,b\wr & = & \wr a,a\wr & \\ \mathbf{f}_n\wr a,a,b\wr & = & \wr\wr & \text{for all } n>2 \end{array}$$

The filter $\mathbf{f}_k$ selects all elements which occur at least k times. We will now show how union followed by filtering with $\mathbf{f}_k$ is equivalent to intersection, for set-like bags.

Consider the bag-intersection of k set-like bags. (Remember that a bag is set-like if each of its elements occurs once.) An element e will appear in the intersection if and only if it occurs in each of the bags being intersected, in other words, if there are a total of k occurrences of e, one in each bag. Therefore e will appear k times in the bag union of all these bags. Therefore it will be selected by $\mathbf{f}_k$. Formally:

$$\mathbf{f}_k\left(\biguplus_{i=1}^{k} p_i\right) = \bigcap_{i=1}^{k} p_i \qquad \text{for set-like } p_i$$

9.4.2 The filter $\mathbf{f}_{\#n}$

We now define another simple alternative to $\mathbf{f}_{\max}$, called $\mathbf{f}_{\#n}$:

Definition:

$$e \# \mathbf{f}_{\#n}(A) = \begin{cases} n, & \text{if } e\#A \geq n \\ 0, & \text{otherwise} \end{cases}$$

$\mathbf{f}_{\#n}$ can be used to insist that any acceptable solution has a certain level of support. For example, if two (set-like) hierarchies have been composed together using BCH, $\mathbf{f}_{\#2}$ requires candidate solutions to have support from both of them. This is similar to $\mathbf{f}_k$, but the difference is that $\mathbf{f}_k$ leaves the number of occurrences of an element untouched if it exceeds k, whereas $\mathbf{f}_{\#k}$ reduces the occurrences to precisely k.

A special case of $\mathbf{f}_{\#}$ is when $n = 1$. $\mathbf{f}_{\#1}$ can also be called $\mathbf{f}_{\text{set}}$, for the obvious reason:

$$e \# \mathbf{f}_{\#1}(A) = e \# \mathbf{f}_{\text{set}}(A) = \begin{cases} 1, & \text{if } e\#A \geq 1 \\ 0, & \text{otherwise} \end{cases}$$

Example:

$$\begin{array}{lcll} \mathbf{f}_{\#1}\wr a,a,b\wr & = & \wr a,b\wr & \text{(compare with } \mathbf{f}_1\text{, above)} \\ \mathbf{f}_{\#2}\wr a,a,b\wr & = & \wr a,a\wr & \\ \mathbf{f}_{\#n}\wr a,a,b\wr & = & \wr\wr & \text{for all } n>2 \end{array}$$

The conclusion of the paper on incremental HCLP by Menezes, Barahona and Codognet [60] discusses weakening the nature of the comparator – instead of saying a solution is acceptable if it satisfies more constraints than the alternative solutions (at level k) and as many at levels less than k, their new criterion is a solution is acceptable if it satisfies at least a certain number of preferences (a certain 'threshold' they say). Clearly, this idea can easily be encoded by the use of $\mathbf{f}_{\#n}$ with n less than the number of constraints in the problem.

In Section 3.5.1 we discussed the differences between two HCLP comparators UCB and LPB (locally-predicate-better). We noted that LPB will offer certain 'solutions' which intuitively do not seem to be as good as other possible solutions. We stated that if the user decides to accept a certain level of inaccuracy for efficiency reasons, this decision should be explicit and not hidden in the choice of comparator. Clearly, $\mathbf{f}_{\#n}$ allows such a choice to be made explicitly.

The ease with which the ideas in the two previous paragraphs fit into the framework of filter functions suggests to us that this formalism is very rich and general. Notwithstanding this, the reason that most of this chapter concentrates on $\mathbf{f}_{\mathsf{max}}$ is its closeness to UCB, the comparator on which we focus throughout this thesis.

9.5 Equivalence of BCH/FGH and HCLP

9.5.1 Example γ — comparison with HCLP

Figure 9.1 contains an example of the use of both BCH and FGH, and a comparison with HCLP. Part of the example (P) is taken from Wilson's thesis [83, Sect.2.2]. Note that BCH does not discriminate enough, by itself, to capture the behaviour of standard HCLP. This is not surprising, and motivates the use of filter functions. Also note that the combination of FGH and BCH does indeed give precisely the same answers as HCLP.

9.5.2 Relationship between HCLP and FGH

HCLP is parameterised by a comparator $\mathcal{C}$, and FGH is parameterised by a particular filter function. Therefore it is not possible to state a relationship as general as, say, "The HCLP solution set for a hierarchy is identical to its FGH solution"[5].

Notwithstanding the above, we *can* claim the following, where UCB stands for the *unsatisfied-count-better* comparator:

Proposition 18: FGH applied to BCH is equivalent to HCLP
$sols(\text{FGH}(\mathbf{f}_{\mathsf{max}})(H)) = \text{HCLP}(\text{UCB})(H)$

Proof:
Reminder: a solution θ is *unsatisfied-count-better* than a solution σ if it satisfies as

[5]More precisely, we cannot state "The HCLP solution set for a hierarchy H is identical to the result of applying set to the final non-empty bag of the FGH tuple of solutions for H".

	P	Q	R
required	$X > 0$	$0 < X < 20$	$0 < X < 23$
strong	$X < 10$	$X > 5$	$X > 15$
weak	$X = 4$	X div 7[a]	X div 8
Sols.[b]			
S_0	$X > 0$	$0 < X < 20$	$0 < X < 23$
S_1	$0 < X < 10$	$5 < X < 20$	$15 < X < 23$
S_2	$X = 4$	$X = 7, X = 14$	$X = 16$

[c]	HCLP[d]	BCH	FGH[e]
S_0	$0 < X < 20$	$0 < X < 20$	$0 < X < 20$
S_1	$5 < X < 10$ $15 < X < 20$	$(0 < X \leq 5)^1$ $(5 < X < 10)^2$ $(10 \leq X \leq 15)^1$ $(15 < X < 20)^2$	$(5 < X < 10)^2$ $(15 < X < 20)^2$
S_2	$X = 7$, $X = 8$, $X = 16$	$X = 4$, $X = 7$, $X = 8$, $X = 14$, $X = 16$	$X = 7$, $X = 8$, $X = 16$

[a] i.e. X is divisible by 7, shorthand for the domain constraint X::[0,7,14,...]
[b] HCLP solutions for each hierarchy individually
[c] Solutions for the combined hierarchies
[d] Calculated taking into account error sequences [83]
[e] Using $\mathbf{f}_{\mathrm{max}}$

Figure 9.1: Compositions using HCLP, BCH, and FGH

many constraints as σ does in levels $1 \ldots k-1$, and at level k it satisfies strictly more constraints than σ.

As discussed in Section 9.2.2.5, we assume that the the members of the BCH solution tuple will contain only those elements which are acceptable to the required constraints, and will contain at least all the elements which are preferred by one or more optional constraints.

$\mathbf{f}_{\mathrm{max}}$ selects all elements occurring at least n times, for some n, and filters out all elements occurring less then n times. An element appears n times at a given optional level because it is preferred by n constraints at that level. An element only appears less than n times if it is preferred by less than n constraints. Therefore it would not have been selected by UCB. $\Box$

It might appear that this proof is incomplete in the case that a level is empty, i.e. the elements appear 0 times. However, this is not the case. If the final bag in the tuple is empty, then it is not considered part of the FGH solution — the definition of *sols*($\mathbf{F}$) is given in terms of the final *non-empty* bag. Therefore we only need to consider the case when a bag, say S_{n-1}, is empty, but a subsequent bag is not. But $\mathbf{F}(S_n) = \mathbf{f}(S_n /\!\!/ \mathbf{F}(S_{n-1}))$. Now by the definition of guard, $\forall\, x \cdot x /\!\!/ \Lbag\Rbag = \Lbag\Rbag$. Therefore if $S_{n-1} = \Lbag\Rbag$ then S_n is also empty. Contradiction. Therefore the above proof is complete.

Let us define a *CLP(all)* solution to be one which satisfies *all* constraints. Now any BCH solution satisfies at least one of the optional constraints. An HCLP solution satisfies as many as possible of the optional constraints, as does an FGH solution. *CLP(required)* refers to the solutions found by ignoring the optional constraints, hence it satisfies all the required constraints but none of the optional ones. Given these definitions, and combining the result of Proposition 18 with that of Section 9.2.2.1, we can say:

$$\mathrm{CLP(all)} \subseteq \mathbf{set}(\mathrm{FGH}) = \mathrm{HCLP(UCB)} \subseteq \mathbf{set}(\mathrm{BCH}) \subseteq \mathrm{CLP(required)}$$

9.6 Conclusions and benefits

We have developed a compositional variant of HCLP based on bags, which stores the intermediate solutions to a hierarchy in a tuple $\langle S_0, S_1, \ldots S_n \rangle$. We are able to avoid invoking the constraint solver to recalculate solutions from scratch, due to the simplicity and elegant mathematical properties of our scheme. This scheme allows the exploration of the solution space, and can be implemented in an incremental manner.

We have defined a new binary infix relation over bags, called 'guard'. We have also defined a class of filter functions over bags, and placed them in a non-compositional framework which respects the theory of HCLP. We have examined one filter function in detail, and shown that it can be used to calculate the same solutions to a hierarchy as would be obtained by HCLP using the unsatisfied-count-better comparator. Thus we have separated HCLP into its compositional and non-compositional parts. The choice of a filter can be left until after BCH has provided the super-bag of solutions; if there are many, a more discriminating filter function can be used.

Most of our presentation has been expressed in terms of finite domains of integers, but it is clear that our work can be extended to any of the usual constraint domains, such as reals, especially if they can be represented by what we term SCCs (see Section 6.1.3).

In general, constraint satisfaction is of exponential complexity, compared to which guarding and filtering are cheap. In addition to being efficient, these operations are simple to understand, and calculate the answers we would obtain from HCLP while avoiding its computational expense and complex semantics.

In the next chapter we extend our discussion of compositionality to PCSP.

Chapter 10

Composition in PCSP

In this chapter we address the following question: if we have two problems S and T which we use as the basis for two PCSPs, can we find the PCSP based on the combined problem, say ST, by composing the two PCSPs we have already calculated? A shorthand for this question is to ask whether or not PCSP is compositional. If it is, we can avoid the situation which we always wish to avoid, namely the requirement of combining the problems themselves and starting from scratch.

The results of a PCSP consist of two separate items. (The other parts of the definition of a PCSP are not important for a discussion of compositionality.) The first is a set of CSPs, each of which is a relaxation of the original problem. We call these the elements of the PCSP. The second item is an ordering of these elements induced by the distance function. If the original problem was called S, then we abuse notation by calling the PCSP based on S, i.e. the pair (element, order), S as well. Thus we assess compositionality by asking if the elements in ST can be easily derived from the elements of S and T, and if the order is easily derived from the two original orders.

10.1 The elements

A solution to a CSP is a valuation which satisfies all the constraints at once. If we have two CSPs A and B, then a solution to their conjunction is clearly a valuation which satisfies all the constraints in A and at the same time satisfies all the constraints in B. We could take the union (conjunction) of the set of constraints in A and the constraints in B and find its solutions. Alternatively, we could take the join of the solution set of A and the solution set of B (equivalently: take the intersection of the 'expansions' of A and B, as defined in Section 7.3.1.2). It is clear that these two methods will produce the same solutions to the combined problem. Hence it is clear that the CSP paradigm is compositional.

The elements of the PCSP based on S are all relaxations of the original problem S. Any solutions for S will also be solutions for each of the elements. By monotonicity of conjunction of constraints, any solution of the problem ST (the conjunction of S and

T) will be a solution for one of the elements of S (ignoring the values of any variables in T and not in S). Any solution of ST will be a solution for all the elements of T (again, after projecting out any variables not in T). Therefore the pairwise joins of the solutions of all the different elements in S and T will equal the solutions of all the different relaxations in ST.

As we do not need to consider the elements further, we will now turn our attention to how they are placed in an order by a distance function.

10.2 The order

10.2.1 Ordering problems and ordering solutions

In PCSP the distance function is defined as ordering *problems*. There is one particular distance function which fits in well with the partial order over the problem space which is a separate part of the standard definition of PCSP. It is based on the number of *solutions* not shared by two problems (see Section 4.4 or [31]). However Freuder notes that it may be expensive to compute. He suggests an alternative, the 'standard' or 'augmentations' distance function, based on the difference between the total number of constraints augmented in each *problem*. This can be calculated without needing to solve all the elements. We are interested in composing solutions, and so it might seem that we ought only to work with solution-space distance functions. However, this is an unnecessary restriction: when ordering problems it may be expensive to consider their solutions, but when ordering solutions we already know the various characteristics of the problems they solve.

In the following we will discuss the order itself, rather than the distance functions by which they are generated, allowing for a general, uniform treatment. Then we shall instantiate our general conclusions with particular example distance functions.

10.2.2 Composing orders

An order is a relation. The statement $x \leq y$ is shorthand for the statement $(x, y) \in \leq$, where $\leq$ is treated as the name of a set of pairs. Therefore two instances of the same semantic concept give rise to two different orders (e.g. the concept 'subset' gives rise to two different orders $\subseteq_S$ and $\subseteq_T$ when we consider the subset order over S as opposed to the subset order over T). We can compose two orders which do not share the same conceptual background using application-specific information, but examples of this nature are too concrete to discuss here. However, in the general case composing two conceptually different orders, for example subset over sets which happen to contain numbers on the one hand, and a numeric order over the sum of the numbers in each set on the other, does not seem coherent semantically.

In the following we use $\otimes$ to denote the result of combining two problems and calculating the combined PCSP from scratch. The order thus obtained is denoted $\leq_{S \otimes T}$, where

S and T are the two problems to be composed. We use $\oplus$ to denote some combining operation over relations, i.e. a function which takes two sets of pairs as inputs and produces a set of pairs as output. As we have already shown in the previous section that the elements of ST can be easily derived from the elements of S and T, we claim that the following proposition is self-evident:

Proposition 19: Composition — General Condition
If $\exists \oplus$ associative and commutative $\cdot \leq_{S \otimes T} = \leq_S \oplus \leq_T$, then PCSP is compositional for $\leq_S$ and $\leq_T$.

To be more precise, the set equality in the above proposition can be unpacked:

$$((s_1, t_1), (s_2, t_2)) \in \leq_{S \otimes T} \quad \text{iff} \quad ((s_1, t_1), (s_2, t_2)) \in \ (\leq_S \oplus \leq_T)$$

10.2.3 The 'solution subset' distance function

Freuder suggests a metric based on number of solutions not shared by two problems (see Section 4.4 or [31]). He notes that if $P' \leq P$ this distance function measures how many solutions have been added by the relaxation of P[1]. However, because we are dealing with orders induced by the distance functions rather than the function itself, we are interested in the underlying subset inclusion relation over solution sets. If neither $P \leq P'$ nor $P' \leq P$ holds, the subset inclusion order makes P and P' incomparable. This might seem to be a difference between it and Freuder's distance function based on number of solutions not shared. In fact, we feel that his distance function is not well-defined. Specifically, if P has solution set $\{a\}$ and P' has solution set $\{b\}$, then they are at distance 2 from each other. If Q has no solutions but Q' has solution set $\{c, d\}$, then they are also separated by a distance of 2. But which is *better*, P, P', or Q? It is not clear. Therefore we choose to show the compositionality of the related solution subset inclusion ordering instead, which we do in the next proposition.

Proposition 20: Solution subset is compositional
An appropriate $\oplus$ exists when $\leq_S = \subseteq_S$ and $\leq_T = \subseteq_T$.

Proof:
The order, calculated from scratch, for $S \otimes T$ will be $(s_1, t_1) \leq_{S \otimes T} (s_2, t_2)$ iff $s_1 \cap t_1 \subseteq s_2 \cap t_2$, where the s_i and t_i are sets of solutions. It is an obvious theorem of set theory that $a \cap b \subseteq c \cap d$ iff $a \cap b \subseteq c$ and $a \cap b \subseteq d$. Keeping this theorem in mind, consider the following definition for $\oplus$:

$$X \oplus Y = \{(z, x_2 \cap y_2) \mid (z, x_2) \in X,\ (z, y_2) \in Y\}$$
where z is assumed to be of the form $z = z_1 \cap z_2$,
but this is not in fact necessary

It is clear that $(u, v) \in X \oplus Y$ iff $(u, v) \in X \otimes Y$. $\square$

It is also clear that $\oplus$ is commutative. We now prove that it is associative, i.e. $X \oplus (Y \oplus Z) = (X \oplus Y) \oplus Z$.

[1] Recall that in PCSP $\leq$ is an order over the problem space, defined by $P_1 \leq P_2$ iff $sols(P_1) \supseteq sols(P_2)$.

(We implicitly assume that the various orders X, Y, and $X \oplus Y$ are defined over elements of the respective problem spaces i.e. collections of elements.)

Proof:

$$\begin{array}{ll}
\text{Now} & (B,C) \in X \oplus (Y \oplus Z) \\
\text{iff} & (B,C) \in \{(A, P \cap Q) \mid (A,P) \in X, (A,Q) \in Y \oplus Z\} \\
\text{iff} & \exists\, P,Q \cdot (C = P \cap Q \ \wedge\ (B,P) \in X \ \wedge\ (B,Q) \in Y \oplus Z) \\
\text{iff} & \exists\, P,Q \cdot (C = P \cap Q \ \wedge\ (B,P) \in X \ \wedge \\
 & \qquad \exists\, R,S \cdot (Q = R \cap S \ \wedge\ (B,R) \in Y \ \wedge\ (B,S) \in Z)) \\
\text{iff} & \exists\, P,Q,R,S \cdot (C = P \cap Q \ \wedge\ (B,P) \in X \ \wedge \\
 & \qquad Q = R \cap S \ \wedge\ (B,R) \in Y \ \wedge\ (B,S) \in Z) \\
\text{iff} & \exists\, P,R,S \cdot (C = P \cap R \cap S \ \wedge\ (B,P) \in X \ \wedge\ (B,R) \in Y \ \wedge \\
 & \qquad (B,S) \in Z) \qquad\qquad\qquad\qquad (\mathbf{1})
\end{array}$$

Let us leave the above derivation at this point. We will now arrive at a similar expression starting from $(X \oplus Y) \oplus Z$ (modulo different names for existentially qualified variables).

(We could just continue the above derivation and put it back together to arrive at the right-hand side, but the following is clearer.)

$$\begin{array}{ll}
\text{Now} & (B,C) \in (X \oplus Y) \oplus Z \\
\text{iff} & (B,C) \in \{(A, P \cap Q) \mid (A,P) \in X \oplus Y, (A,Q) \in Z\} \\
\text{iff} & \exists\, P,Q \cdot (C = P \cap Q \ \wedge\ (B,P) \in X \oplus Y \ \wedge\ (B,Q) \in Z) \\
\text{iff} & \exists\, P,Q \cdot (C = P \cap Q \ \wedge\ \exists\, R,S \cdot (P = R \cap S \ \wedge \\
 & \qquad (B,R) \in X \ \wedge\ (B,S) \in Y) \ \wedge\ (B,Q) \in Z) \\
\text{iff} & \exists\, P,Q,R,S \cdot (C = P \cap Q \ \wedge\ P = R \cap S \ \wedge \\
 & \qquad (B,R) \in X \ \wedge\ (B,S) \in Y \ \wedge\ (B,Q) \in Z) \\
\text{iff} & \exists\, Q,R,S \cdot (C = R \cap S \cap Q \ \wedge \\
 & \qquad (B,R) \in X \ \wedge\ (B,S) \in Y \ \wedge\ (B,Q) \in Z) \qquad (\mathbf{2})
\end{array}$$

We can write (**1**) and (**2**) above one another as follows:

$$\begin{array}{ll}
\exists\, P,R,S \cdot (C = P \cap R \cap S \ \wedge\ (B,P) \in X \ \wedge\ (B,R) \in Y \ \wedge\ (B,S) \in Z) & (\mathbf{1}) \\
\exists\, R,S,Q \cdot (C = R \cap S \cap Q \ \wedge\ (B,R) \in X \ \wedge\ (B,S) \in Y \ \wedge\ (B,Q) \in Z) & (\mathbf{2})
\end{array}$$

and we can see that they are identical, modulo different names for existentially quantified variables. □

Note that the earliest member of the subset inclusion order is the empty set. If the user would like to have the 'best' solution to the PCSP, it will be the earliest *non-empty* member of the order. If the earliest non-empty member of S is intersected with the earliest non-empty member of T, the intersection might be empty, and therefore not a candidate solution for the combined PCSP ST. Therefore, not withstanding the above proof of compositionality for the order as a whole, the user is still required to do an extra task, namely to ignore or delete the empty members of the order. However, checking if a set is empty does not require much work, and therefore we do not consider it to be a problem, in general.

The $\oplus$ that we defined in above, and the proof of its associativity, did not depend on the order being subset inclusion. The order was represented simply as the pairs which are members of the sets X, Y, and Z. Therefore the proof of associativity holds for other orders too, as long as we use this composition operator. But we must ask if this particular operator is *useful* for any orders apart from subset inclusion.

We now define a generalisation of the subset ordering. An order is 'subset closed' iff $\forall\, u, x_1, x_2 \cdot (u \subseteq x_1 \wedge x_1 \leq x_2) \rightarrow u \leq x_2$. It seems that the $\oplus$ defined above is useful for subset closed orders as well as for subset itself. In terms of this definition, it is not surprising that $\subseteq$ is subset closed, as partial orders must be transitive. It is even less surprising when we note that stating that $\leq$ is subset closed is equivalent to saying that $\leq$ contains $\subseteq$. This observation, and the following proof, are due to Sebastian Hunt [Private Communication].

Proof:
Assume that $\leq$ is subset closed. Then, as $\leq$ is reflexive, $\forall\, u, x \cdot u \subseteq x \rightarrow u \leq x$.

Conversely, assume that $\forall\, u, x \cdot u \subseteq x \rightarrow u \leq x$. If $u \subseteq x$ and $x \leq y$, then $u \leq y$, as $\leq$ is transitive. So $\leq$ is subset closed. □

10.2.4 The MaxCSP distance function

Freuder and Wallace define the MaxCSP distance function as seeking a solution that satisfies as many constraints as possible (see Section 4.4 or [33]). This can also be thought of as seeking a solution which violates as few constraints as possible. For two solution sets X and Y to some problem, the order induced by MaxCSP can be defined as follows:

$X \leq_{MaxCSP} Y$ iff X satisfies more constraints than Y

It is clear that a set of solutions U satisfies more constraints than a set of solutions V whenever $U \subseteq V$. Therefore

$U \subseteq V \rightarrow U \leq_{MaxCSP} V$

Therefore MaxCSP is subset closed, using Hunt's equivalent definition.

This suggests that MaxCSP is compositional, using an $\oplus$ operation similar to the one defined for the subset inclusion distance function. However, instead of defining the composed order directly, let us define a combining operator for the scores calculated by the distance function, and use the standard ordering over the integers to order the solutions according to their scores. We will work at the level of the sets of constraints which gave rise to the solution set, rather than at the level of the solution set itself, as this is more in keeping with the definition of MaxCSP.

Let s be a set of constraints which is a relaxation of the original OCS S, and similarly for t and T. Let s satisfy n_s of the constraints in S, and let t satisfy n_t of the constraints in T. When we previously discussed the elements of the composed PCSP we used intersection over solution sets, which gives the same results as conjunction or union of sets of constraints. So let us say that $s \sqcup t$ is a set of constraints which is a relaxation of the problem $S \sqcup T$. Clearly, the solution set for $s \sqcup t$ will be the intersection (join) of the solution sets for s and t. Its place in the order $\leq_{S \oplus T}$ will be determined by its score $n_{s \sqcup t}$. Clearly, the larger the score of $s \sqcup t$, the earlier in the order $\leq_{S \oplus T}$.

The score $n_{s \sqcup t}$ of the conjunction of constraints $s \sqcup t$ will not be the simple addition of the scores n_s and n_t due to the possibility of some of the constraints being in both sets, and hence counted twice. Let us use $\sqcap$ to mean intersection of sets of constraints (not sets of solutions). For any x, $s \sqcap x$ will be one of the relaxations of S, so that $s \leq_S s \sqcap x$, as solutions to $s \sqcap x$ will satisfy fewer constraints than solutions to s. Specifically, $s \sqcap t$ will be one of the relaxations of S, and therefore its score $n_{s \sqcap t}$ will in principle be available. In the light of all the above, it is clear that the appropriate compositional operator $\oplus$ for the MaxCSP distance function will be defined as follows:

$$n_{s \sqcup t} = n_s \oplus n_t = n_s + n_t - n_{s \sqcap t}$$

It would be preferable if $\oplus$ could be defined only in terms of n_s and n_t, but that is not possible.

We have abused notation in this section, using $\oplus$ as an operation over scores instead of over orders or relations, but we feel that this presentation is sufficiently precise, and much clearer.

10.2.5 The 'augmentations' distance function

Another distance function mentioned by Freuder is based on the number of constraint augmentations not shared by two problems, i.e. the number of relaxations it would be necessary to make to P to arrive at P'. In Section 4 we called this the 'augmentations' distance function. In [31] Freuder does not discuss the semantics of this distance function when neither $P \leq P'$ nor $P' \leq P$ holds. Therefore a question similar to the one in Section 10.2.3 (re. P, P', and Q') arises. Let us avoid this question by redefining the distance function as counting number of constraint augmentations between two problems related by the order $\leq$ over the problem space[2]. This ensures that the induced

[2]Other plausible assumptions could be made which would turn this distance function into MaxCSP, especially in the light of the detailed defence of the choice of UCB for transforming PCSP into HCLP (see Section 7.3.1.2), but we wish to avoid repetition of the previous section 10.2.4.

order is subset closed. Let us further assume that S and T do not contain the same constraints: they may contain constraints over the same variables, but let us assume that if C_{XY} is one of the constraints in S, then T may contain D_{XY} but not another copy of C_{XY}. Augmenting D_{XY} with some tuple (x, y) is considered to be completely separate from adding (x, y) to the domain of C_{XY}, even though the same tuple is added. This avoids the question of double counting of the same augmentation of the same constraint. (An alternative is to use bags, of course, as we do elsewhere in this thesis.)

This still leaves an important question. If we augment some constraint in S once, and we also augment a constraint in T once, but the augmentations do not lead to the same solution, the two solutions may not be in the intersection of the two elements. Should we nonetheless include the augmentations in the total? We feel that the answer is 'yes', as it may often be the case that an augmentation has no effect, even when considering S alone.

Bearing in mind all these assumptions, we claim that the *number* of constraint augmentations no longer matters, and we can just consider the induced order. In this case we claim that the $\oplus$ defined in the proof of Proposition 20 is also appropriate here, given that the order is indeed subset closed.

10.3 Conclusions

We have shown that PCSP is compositional when the best-known distance function MaxCSP is used, and also when modifications of the two other obvious distance functions are used.

We have defined a heuristic condition for the applicability of a compositional combining operator for distance functions.

Part V

Integration

In Part III of this thesis we discussed transformations between two different constraint paradigms, HCLP and PCSP. In Part IV we created a compositional variant of the solver part of a language, HCLP.

In this part we define our own framework, GOCS, *as a compositional integration of certain concepts from HCLP and PCSP. Therefore in this part we spend some time at the level of languages, before considering implementations and other solver-level issues.*

Then we present a large example in order to compare the GOCS *approach with that of HCLP and PCSP.*

Chapter 11

GOCS: A New Preference System Integrating HCLP and PCSP

11.1 Introduction

Both HCLP and PCSP have desirable characteristics: HCLP allows a fine-grained and declarative expression of preferences, but commits the user to labelling all the constraints in the problem. PCSP only requires the specification of a single global distance function, which can be used for optimisation or to find instances with a particular problem structure, but it can be very difficult to find the correct function.

The links between CLP and CSP are well-known, and have greatly benefited both paradigms. However, no such link has been developed between HCLP and PCSP. General frameworks have been developed of which PCSP and (indirectly) HCLP are particular instances [4, 5, 71], but they do not provide a direct link between them. In this chapter we present a general framework of which HCLP and PCSP are instances, and in which it is also possible to use both approaches simultaneously: an easy-to-find global function can be used, fine-tuned with a small number of labels, or more labels can be used with the global function relegated to a secondary role. Furthermore our framework is *compositional*, in contrast with [4, 5, 71] where the issue is not even raised.

Our motivation is partly theoretical (similar to [4, 5, 71]) but mainly an attempt to capture the two formalism's orthogonal approaches to over-constrained systems (OCSs) in a single framework. Remember that the two orthogonal approaches are as follows: HCLP reorganises the *structure* of an over-constrained problem, by specifying relationships between constraints; PCSP keeps a flat structure to the problem, but changes the *meaning* of the individual constraints by adding elements to the domain.

11.2 A general framework for over-constrained systems

We now present GOCS, a general framework for dealing with over-constrained systems (OCSs) i.e. sets or bags of constraints with no solutions. In this section we discuss some general characteristics which are desirable for such a system.

This part of our work has similarities of purpose with the work of Bistarelli, Rossi, and Montanari [5], and Schiex, Fargier, and Verfaillie [71]; these two theories are compared in Bistarelli et al. [4]. However, we go one stage further than these papers: we show how the different approaches of HCLP and PCSP can be used *simultaneously*. Therefore the various advantages of each of these systems are available.

11.2.1 Characteristics

We can relax an OCS in two ways: either we can leave out some of the constraints thus relaxing the system as a whole, or we can relax some of the constraints individually. The first way is effectively what HCLP does, where the choice of constraints to omit is guided by the strength labels. It can be considered as the choice of one of the consistent members of the power-bag of the original bag. The second method, of which PCSP is an example, generates a collection of consistent bags all of which have the same size as the original, where some of the constraints are not identical to the original ones, but are derived from them. We can also consider this method in the form of an enriched notion of power-bag, as will be shown below.

An important question is how to place all the bags in an order, such that the first element in the order is the 'best' relaxation of the original problem. It is straightforward to define a function which calculates a score for each member of a collection of bags which shows how good or bad that bag is with respect to the required, strong, weak, etc. constraints in an HCLP formulation of the problem. These scores can be drawn from any partially ordered set; HCLP itself uses a comparator to calculate sequences of numbers $[r, s, w, \ldots]$, which are then ordered lexicographically. It is also straightforward to define a function which calculates a score based on the number of constraints in a bag which are relaxations of the original constraints, as in PCSP. (PCSP uses what its developers call 'metrics' and what we call 'distance functions', to avoid confusion with HCLP's 'metric comparators'.) But the question is how to combine these two functions, and thus relate these two notions of 'score' which may be semantically different even if they have syntactic similarities.

We allow a great deal of generality in answering this question, giving some characteristics that the function must obey, but otherwise not restricting the user too much. However, we also give a very simple example to aid the presentation, in which one unit of relaxation (one augmentation, in PCSP terminology) is equated with one move down the hierarchy of strength labels. In other words, using our example combined function (with a lexicographic order on sequences of integers), a required constraint which has been relaxed once is given the same score, and hence the same position in the order, as a strong constraint which has not been relaxed at all.

Another restriction is placed on any general framework for OCSs: if the original problem is not in fact over-constrained, then it must be considered to be the best possibility; it may not be ignored in favour of one of its relaxations.

Compositionality is very important in CLP, both for theoretical reasons and due to its relationship with incrementality of implementations. This has been discussed in much greater detail in Part IV of this thesis. GOCS is as compositional as it is possible for a system to be when it includes preferences and partiality; it is a fundamental feature of this class of theories that the 'best' solution (or the best consistent subproblem) of an over-constrained problem P, when combined with the best solution to another problem Q, may not be the best solution to the combined system. For example, they may not even be consistent with each other! However, if they *are* consistent with each other, then the best solution is indeed the combination of the best sub-solutions (see Section 11.3.3).

11.2.2 Formal definition of GOCS

We define GOCS using bags. Our motivation for this is that bags allow us to weight preferences during composition. For example, if 19 people strongly want a meeting at 9 a.m. and one person strongly wants it at 10 a.m., a set representation of the composition of the preferences would have each of 9 a.m. and 10 a.m. occurring once, and thus appearing to be equally acceptable, which is clearly incorrect.

For each constraint c in the original problem, we define its set of relaxations $Rel(c)$ as being all the bags each containing one possible relaxation (augmentation by adding tuples) of c, as well as c itself, and also the empty bag $\lbag\rbag$ which denotes the complete removal of c from the problem[1]:

$$Rel(c) = \{\lbag\rbag, \lbag c\rbag, \lbag c'\rbag, \lbag c''\rbag, \ldots\}$$

Rel produces a set, not a bag. This is because we only want one relaxation of any given constraint; bags are not relevant when relaxing one particular problem, only when composing two previously-relaxed problems.

Note that c might be labelled with an HCLP-style strength level. In this case, all the constraints in the members of $Rel(c)$ keep that same label. Note also that we define a bag of labelled constraints to be *ul*-consistent if the underlying set of unlabelled constraints is consistent (where *ul* stands for 'unlabelled', and where consistency is indicated by not entailing a contradiction):

$$\lbag(l_1, c_1), (l_2, c_2), \ldots, (l_n, c_n)\rbag \not\models_{ul} \bot \text{ if and only if } \{c_1, c_2, \ldots, c_n\} \not\models \bot$$

We can now define our enriched notion of power-bag, in terms of all the consistent members of the cartesian product of all the *Rel*'s of the constraints in the original

[1] In fact, Freuder has shown that in PCSP it is not necessary to treat constraint removal separately from constraint augmentation [31], but it is useful to do so for our purposes, to aid the comparison with HCLP.

problem P:

$$[\![P]\!] \;=\; \{\overline{\sigma} \mid \sigma \in \underset{c\in P}{\times} Rel(c),\, \overline{\sigma} \not\models_{ul} \bot\}$$

We use $\overline{\sigma}$ to indicate the flattening of a tuple into a bag. The cartesian product of n sets is an n-tuple. Here, each element of the tuple is a singleton bag; the elements of all the bags are distinct, except that the empty bag may be repeated. So if $\sigma = (\lbag\rbag, \lbag b\rbag, \lbag c'\rbag, \lbag\rbag, \lbag e''\rbag, \ldots)$ then $\overline{\sigma} = \lbag b, c', e'', \ldots\rbag$. The issue of flattening a tuple in this way is trivial, and clutters up the presentation, so elsewhere we will simply pretend that the elements of the cartesian product of n sets of distinct singleton bags are simply bags containing at most n distinct elements.

Of course P is a *bag* of (labelled) constraints, thus allowing us to account for the presence of a constraint in more than one of the problems to be composed. $[\![P]\!]$ is derived from a cartesian product, the elements of which can be isomorphically mapped to bags of labelled constraints. Therefore, abusing notation, we can define its type to be $[\![\_]\!] : \mathbb{M}(L \times C) \rightarrow \mathbb{P}(\mathbb{M}(L \times C))$ where $\mathbb{M}$ represents a power-bag, $\mathbb{P}$ is powerset, and $L \times C$ represents pairs of strength labels drawn from L and unlabelled constraints with type C.

Example:
Consider some problem R comprising three (possibly labelled) constraints a, b, c. Then

$$\begin{aligned}
Rel(a) &= \{\lbag\rbag, \lbag a\rbag, \lbag a'\rbag, \lbag a''\rbag, \ldots\}\\
Rel(b) &= \{\lbag\rbag, \lbag b\rbag, \lbag b'\rbag, \lbag b''\rbag, \ldots\}\\
Rel(c) &= \{\lbag\rbag, \lbag c\rbag, \lbag c'\rbag, \lbag c''\rbag, \ldots\}
\end{aligned}$$

$$\begin{aligned}
Rel(a) \times Rel(b) \times Rel(c) \;&=\; \{(\lbag\rbag, \lbag\rbag, \lbag\rbag), (\lbag a\rbag, \lbag\rbag, \lbag\rbag), \ldots (\lbag a\rbag, \lbag b, \lbag c\rbag),\\
&\qquad \ldots, (\lbag a'\rbag, \lbag b'', \lbag c'''\rbag), \ldots\}\\
\text{then flatten:} \;&=\; \{\lbag\rbag, \lbag a\rbag, \ldots, \lbag a, b, c\rbag, \ldots, \lbag a', b'', c'''\rbag, \ldots, \}
\end{aligned}$$

then remove *ul*-inconsistent bags, to give, say,

$$[\![R]\!] = \{\lbag\rbag, \lbag a\rbag, \ldots, \lbag a', b'', c'''\rbag, \ldots, \}$$

$\square$

Given two bags of constraints P and Q, we can calculate $[\![P \uplus Q]\!]$. But we can also define a function $\oplus : \mathbb{P}(\mathbb{M}(L \times C)) \times \mathbb{P}(\mathbb{M}(L \times C)) \rightarrow \mathbb{P}(\mathbb{M}(L \times C))$ which combines the results of applying $[\![\_]\!]$ to P and Q separately:

$$X \oplus Y = \{(\sigma \uplus \theta) \mid \sigma \in X,\, \theta \in Y,\, (\sigma \uplus \theta) \not\models_{ul} \bot\}$$

The following equality is a claim of compositionality for Gocs; the proof is in Section 11.3:

$$[\![P \uplus Q]\!] = [\![P]\!] \oplus [\![Q]\!]$$

11.2.3 Ordering the elements of $[\![\_]\!]$

In HCLP, it is easy to decide which of two constraints or collections of constraints is more important: just use the strength labels and whichever comparator C the user has selected. In PCSP, it is also easy: just use whichever distance function the user has chosen. In GOCS the user must also select the particular function h to calculate scores for the members of $[\![P]\!]$, and must start by choosing the distance function δ which scores each member of $Rel(c)$. The scores must be drawn from some ordered set, but apart from that, the only restrictions we place on δ are that any unrelaxed original constraint must be given a score of 0, and if one element of $Rel(c)$ has been augmented more than another, it must be given a larger score (a greater distance from c). The definition of $[\![\_]\!]$ does not include the scores which annotate all its members, but as they are always easily calculated, and depend on nothing but the choice of h and the members of P, we can talk as though $[\![\_]\!]$ does include them.

δ may treat all the constraints in a problem equally, but it may also discriminate: it may define the increase in distance between a once-augmented constraint and a twice-augmented constraint to be the same as the distance between a twice-augmented constraint and a constraint which has been augmented three times, or it may treat them differently.

The score function $h : \mathbb{M}(L \times C) \rightarrow (\pi, \leq_\pi)$ has as input a bag of labelled constraints which are related to each other by a distance function, and outputs a member of some set π which is pre-ordered by $\leq_\pi$. It is a pre-order rather than a partial order because two different bags may have the same score, and hence the same place in the order; whenever we talk about the 'best' element in the order, we mean the equivalence class of all the equal-best elements. Our example throughout this chapter is an h that has as output a sequence of integers, ordered lexicographically. (See [4, 5, 71] for discussion of the mathematical properties of various functions in different formulations of HCLP and PCSP.)

11.2.4 Example α in GOCS

See Section 2.6 for the original description of this example, and Sections 3.4 and 4.5 for its treatment in HCLP and PCSP respectively. Remember that the HCLP specifier had decided to assign the following strength labels to the constraints: required C_{ST}, strong C_{FT}, and weak C_{SF}.

Using the HCLP convention that unlabelled constraints are 'required', the relaxations of the three constraints will be:

$$\begin{aligned}
Rel(C_{ST}) &= \{\wr\int, \wr C_{ST}\int, \wr C'_{ST}\int, \wr C''_{ST}\int, \wr C'''_{ST}\int, \ldots\} \\
Rel(\text{strong } C_{FT}) &= \{\wr\int, \wr\text{strong } C_{FT}\int, \wr\text{strong } C'_{FT}\int, \wr\text{strong } C''_{FT}\int, \ldots\} \\
Rel(\text{weak } C_{SF}) &= \{\wr\int, \wr\text{weak } C_{SF}\int, \wr\text{weak } C'_{SF}\int, \wr\text{weak } C''_{SF}\int, \ldots\}
\end{aligned}$$

Let us consider a very simple choice for h, namely that it returns a sequence of integers $[r, s, w, \ldots]$ which count the number of unaugmented required, strong, and weak constraints present in each of the bags in $[\![\alpha]\!]$. These sequences will be ordered lexicographically, with larger (later) elements being more preferred than earlier ones, unlike HCLP where scores always measure errors and so bags with lower scores are preferred. Then for example the empty bag will be annotated with a score $[0,0,0]$ and so will be the worst element in the order, and $\wr$weak $C_{SF}\int$ will score $[0,0,1]$. Given that the original problem is of size 3, by the definition of $[\![\_]\!]$ each sequence must sum to at most 3; the very best element that the order might contain will be $[3,0,0]$. Let us also define a very simple relationship between strength labels and augmentation, namely that one augmentation is equivalent to one movement down the hierarchy. For example $\wr$strong $C'_{FT}\int$ will be treated by h in the same way as an unaugmented weak constraint. (Note that in this example the number of dashes does *not* indicate the number of augmentations, as there is more than one way to add a single tuple to the each of the constraints in α.) Then, for example, $\wr$strong C'_{FT}, weak $C_{SF}\int$ will get a score of $[0,0,2]$, as it contains two weak-equivalent constraints. In later sections we will refer to this score function as max-ucb as it is a combination of HCLP's UCB comparator and the PCSP distance function called MaxCSP.

The annotated, ordered, version of $[\![\alpha]\!]$ will contain many members; the maximal ones will be $\wr C_{ST}$, strong C_{FT}, weak $C'_{SF}\int^{[1,1,0,1]}$ and $\wr C_{ST}$, strong C_{FT}, weak $C^{+}_{SF}\int^{[1,1,0,1]}$, where C^{+}_{SF}, not listed earlier, is C_{SF} augmented with the single tuple (r, c). The solutions to these two bags of constraints are (w, s, d) and (r, c, g) respectively, the same as were calculated by HCLP by omitting C_{SF} completely (Section 3.4). Therefore it might seem that in this case we have not gained anything by augmenting constraints. But in fact what this example shows is that we can use strength labels with confidence in this framework: we labelled sufficiently many of the constraints that we completely determined the answer, without the distance function interfering. However, if we have a problem with thousands of constraints, and do not wish to label each one individually, we could just label those parts of the problem over which we want most control.

11.3 Composition

Proposition 21:
$[\![P]\!] \oplus [\![Q]\!] = [\![P \uplus Q]\!]$

Proof:

Case $\subseteq$:
Consider a typical element of $[\![P]\!] \oplus [\![Q]\!]$ which, by the definition of $\oplus$, must be of the form $\sigma \uplus \theta$ for some $\sigma \in [\![P]\!], \theta \in [\![Q]\!]$. By the definition of $[\![\_]\!]$, $\sigma \uplus \theta$ contains various constraints, each drawn from $Rel(c)$ for some $c \in P$, or some $c \in Q$, or in both. c will also be an element of $P \uplus Q$, and so $[\![P \uplus Q]\!]$ will contain all consistent bags each of which contains one member of $Rel(c)$. This is true for all the constraints in $\sigma \uplus \theta$. By the definition of $\oplus$, $\sigma \uplus \theta$ must be consistent. Therefore it also satisfies the consistency condition in the definition of $[\![\_]\!]$.

Case $\supseteq$:

Consider some $\psi \in [\![P \uplus Q]\!]$. It will have the form $\psi = \wr c_1^*, \ldots, c_n^* \int$ where $c_i^* \in Rel(c_i)$ for $c_i \in P$ or $c_i \in Q$ or both. Let us define two bags σ and θ such that $\sigma = \uplus \wr c_i^* \in \psi \mid c_i^* \in Rel(c_i), c_i \in P \backslash Q \int$ and $\theta = \uplus \wr c_i^* \in \psi \mid c_i^* \in Rel(c_i), c_i \in Q \int$. Clearly $\psi = \sigma \uplus \theta$. Now ψ is consistent, and so $\sigma \subseteq \psi$ and $\theta \subseteq \psi$ must also be consistent, and so $\sigma \in [\![P]\!]$ and $\theta \in [\![Q]\!]$. Therefore $\psi = \sigma \uplus \theta \in [\![P]\!] \oplus [\![Q]\!]$.

Therefore we have shown that $\oplus$ is compositional. □

11.3.1 Compositionality of scoring functions

The proof shows that we can compose the solutions to problems without needing to compose the problems and re-calculate solutions from scratch. But can we also re-use the scores that we calculated for each member of $[\![P]\!]$ and $[\![Q]\!]$ separately? The answer is yes, as long as we can define some function $\oplus_h : \pi \times \pi \to \pi$ which has the property that $h(\sigma \uplus \theta) = h(\sigma) \oplus_h h(\theta)$. It is important for compositionality that $\oplus_h$ does not depend on P and Q themselves, but only on σ and θ.

We can define $\oplus_h$ for our example h using pointwise addition:

$$[r_\sigma, s_\sigma, w_\sigma, \ldots] \oplus_h [r_\theta, s_\theta, w_\theta, \ldots] = [r_\sigma + r_\theta, s_\sigma + s_\theta, w_\sigma + w_\theta, \ldots]$$

It is clear that this definition has the desired property, and so we can combine scores without needing to recalculate them, without affecting compositionality. All that is necessary is to check that the conjunction of the constraints $\sigma \uplus \theta$ is consistent. This consistency condition is important, and it has a key effect on the possibility of total compositionality, which is discussed in the next section.

11.3.2 Non-compositionality / non-monotonicity of pre-ordering

If the best element in the lexicographic ordering of $[\![P]\!]$ is inconsistent with the first element in $[\![Q]\!]$, then their union will not be *ul*-satisfiable and so it will not be an element of $[\![P \uplus Q]\!]$. Therefore, a fortiori, it will not be the first element in the ordering of $[\![P \uplus Q]\!]$.

So it is possible that the first element in the lexicographic ordering of $[\![P \uplus Q]\!]$ will not necessarily be formed from the union of the first element of the ordering of $[\![P]\!]$ and the first element of the ordering of $[\![Q]\!]$. This is where the non-monotonic nature of HCLP is present (and all other systems for expressing preferences have the same characteristic). If we had chosen to define the 'solution' to be only the first element of the order, then the solution of $[\![P \uplus Q]\!]$ would not necessarily equal the solution of $[\![P]\!]$ or $[\![Q]\!]$. This is why we chose the 'solution' to be the entire order, leaving the selection of the best element to a subsequent stage.

11.3.3 Summary

GOCS defines a class of preference systems, i.e. constraint solvers for relaxing and resolving over-constrained systems. GOCS takes as input a bag of constraints P and outputs the solution to the 'best' relaxation of that bag. GOCS is represented by a tuple

$$\langle [\![P]\!], (h, \leq_h), (\oplus, \oplus_h) \rangle$$

where $[\![P]\!]$ indicates the invocation of GOCS on some problem P, h is a score function with scores ordered by $\leq_h$, $\oplus$ composes bags of constraints (which are relaxations of different problems), and $\oplus_h$ composes scores.

The best relaxation, and hence solution, is found as follows. h gives a score to each possible relaxation of the original bag; these scores are then ordered by $\leq_h$ to find the equivalence class of best relaxations. If the original problem is soluble, i.e. it is not over-constrained, then it must be found to be the best. This is a restriction we impose on h and $\leq_h$.

The operator $\oplus$ composes relaxations of different problems, avoiding the necessity of composing the problems themselves and then relaxing, which would waste any work already done in calculating the individual relaxations. Depending on the definition of h and $\leq_h$, it may also be possible to define a function $\oplus_h$ which composes the scores of the various relaxations.

$$[\![P \uplus Q]\!] = [\![P]\!] \oplus [\![Q]\!]$$

$$\begin{array}{rl} \text{if } \exists \oplus_h \text{ such that} & h(\sigma \uplus \theta) = h(\sigma) \oplus_h h(\theta) \\ \text{then} & \text{best}_{\leq_h}([\![P \uplus Q]\!]) = \text{best}_{\leq_h}([\![P]\!]) \oplus_h \text{best}_{\leq_h}([\![Q]\!]) \\ & \quad \text{when } \text{best}_{\leq_h}([\![P]\!]) \uplus \text{best}_{\leq_h}([\![Q]\!]) \not\models_{ul} \perp \end{array}$$

This is only useful if $\oplus_h$ preserves the best elements. In other words

$$\begin{array}{ll} \text{if} & \text{best}_{\leq_h}[\![P]\!] \text{ has score } \sigma \\ \text{and} & \text{best}_{\leq_h}[\![Q]\!] \text{ has score } \tau \\ & \\ \text{then} & \text{best}_{\leq_h}([\![P \uplus Q]\!]) = (\text{best}_{\leq_h}[\![P]\!]) \oplus (\text{best}_{\leq_h}[\![Q]\!]) \\ & \quad \text{and has score } \sigma \oplus_h \tau \end{array}$$

11.4 Programming language aspects of GOCS

This thesis is mainly about constraint *solving* and not constraint logic *programming*. However, we will now sketch an implementation of a subset of GOCS, partly to clarify the framework and partly to aid the comparison with HCLP.

GOCS is very general and declarative, and so certain restrictions are likely to be imposed by any implementation. One key issue is whether the entire solution space is searched before any answers are returned, or whether some kind of 'first solution' is returned quickly. We will assume that the first successful branch of the search tree is *not* displayed, but simply used as a bound for a branch-and-bound search of the remaining space. This guarantees correctness (only good answers are returned) at the expense of speed. An alternative would be to display the first bound, and then only continue the search if it is not acceptable to the user.

In this section we will use concrete syntax based on that of Eclipse [22] including features inspired by its Propia library. An alternative would be to use the style of Fages et al. [25].

11.4.1 Score functions

The user chooses a score function in a fashion similar to setting a compiler or debugger directive:

```
?-   set_score_function(max-ucb).
```

Various score functions are possible; one example is `max-ucb`, as described earlier in Section 11.2.4. Others are mentioned later in this section. Mixing two or more score functions in one query would only make sense if we also defined a score-combining function. The results of such a process would quickly become unintuitive for the user, and so we only allow one score function to be in use in any given query.

11.4.2 The relaxation operator

The following table lists some of the possible forms for the *Rel* operation in GOCS. Each line indicates which constraint or group of constraints is to be relaxed, and what specific type of relaxation (or 'method') should be used. All methods can be used with all the ways of denoting a collection of constraints. If an unlabelled constraint can be satisfied as is, then it must not be relaxed.

	`relax` *What How*
1	`relax` *constraint* `outward`
2	`relax` *constraint* `with` *predicate*
3	`relax` *List-of-variables* `above`
4	`relax all below`
5	`relax all slack`
6	`relax none`
7	`relax all hclp(`*strength*`)`

Rel is only formally defined as operating on constraints. However, we can add some syntactic sugar to reduce the amount of typing that the user must do. Specifically, `relax` followed by a list of variables and a method applies that method to all the constraints containing at least one of the variables from the list. (In practice, it may be more useful to change the semantics of this construct only to relax constraints *all* of whose variables are in the list.) The scope of the relaxation is from where it appears textually until the end of the query, or until the next occurrence of `relax`. Lines 4 and 5 show how to relax all the constraints in a query with the same method. Line 6 shows how to terminate the scope of a relaxation.

Relaxing `outward` means that the domain of the constraint is augmented with values which are close to the existing values. A simple example is the constraint $X = 3$ where X is a finite domain variable. Using Eclipse syntax (where # indicates finite domain), we can write `relax X #= 3 outward`. If this constraint needs to be relaxed, the values 2 and 4 are tried first, then 1 and 5, and so on. Thus if we would ideally like to fly somewhere on a Tuesday, say, the `outward` method will try Monday and Wednesday before Sunday or Thursday.

Relaxing $X = 3$ `above` would try values greater than 3 rather than less than 3, and vice-versa for `below`.

`outward`, `above`, and `below` can also be defined for n-ary constraints and over non-integer domains, as long as the domain is ordered (such as the days of the week). An `outward` relaxation of a constraint with tuple `(a,b)` would include `(a,a)`, `(a,c)`, and `(b,b)`. This might appear slightly ad-hoc, but it is under the control of the user, via the order he defines over the domain.

The `hclp(`*strength*`)` method simply either includes a constraint or ignores it completely. For the purpose of deciding whether to ignore a constraint when other explicitly labelled contradictory constraints are present, and for calculating scores, it is necessary to decide what strength level the relaxed constraints should be considered to have.

`relax slack` is relevant for constraints over numeric domains, and is based on a standard method in linear programming. It means that each constraint has an additional non-negative variable denoted ϵ. Constraints of the form $\sum a_i x_i \geq 0$ become $\sum a_i x_i + \epsilon \geq 0$. $\sum a_i x_i \leq 0$ becomes $\sum a_i x_i - \epsilon \leq 0$, and equalities have two extra variables, one added and one subtracted. Then one can minimise the sum of the epsilons (if $\sum \epsilon_i = 0$ then the original problem is recovered) by using score function `slack-min-sum`. An alternative is to minimise the maximum value of any slack variable, so that the solutions offered are 'near misses', by using `slack-min-max`. Other possibilities also exist, such as least-squares.

Using `slack-min-sum` or `slack-min-max` gives rise to the question of how to combine slack variables with strength labels. One possibility is to say that a sum of slacks (respectively, a maximum slack) greater than 10 is equivalent to violating a strong constraint, whereas a sum (max) between 5 and 10 is equivalent to violating a medium constraint, etc. These equivalences between slacks and strength labels are somewhat arbitrary, but are acceptable if they give the user the desired answers in his specific applications; indeed, the application expert is the only person who can define these equivalences in the first place.

The method indicated in line 2 of the table is the one closest to the standard PCSP notion of augmentation. We allow the programmer to define a predicate which takes a variable tuple as input and returns a list of lists as output. The elements of the outer list represent equivalence classes of relaxations, i.e. a set of relaxations which are to be treated as equivalent weakenings of the constraint by the score function. The first class represents the unweakened unaugmented constraint. Note that the output list is not named in the method, i.e. its effect is communicated directly to the environment, similar to the style of Fages et al. [25]. For example

```
relax X #= 3 with two_step_up(X)
```

invokes the relaxation predicate `two_step_up` which might be defined as follows:

```
two_step_up(Num, [ [Num], [Num+2], [Num+4] ].
```

The increase in error score (or the decrease in satisfaction rating) when moving from one equivalence class to the next is defined in the score function, not in the predicate. Of course the score function must also define the relationship between strength-labels and relaxations.

This example is just a fact, but a (recursive) predicate can also be used, if a longer (or infinite) set of relaxations is required.

11.4.3 Example

We continue to use the HCLP convention that unlabelled constraints are *required* but with the additional proviso that this is only true if they are also not inside the scope of a relaxation operation. Consider the following example query:

```
1    ?-   set_score_function(max-ucb).
     yes.

2    ?-   [X,Y,Z] ::  [0..10],
3            X + Y #> Z
4            p(X,Z),
5            strong X #> 5, weak Y #< X
6            relax all outward,
7            X #= 3,
8            q(Y,Z),
9            strong Y #= 4,
10           relax none,
11           r(X,Y,Z).
```

Remember that the scope of the relaxation in line 6 only goes downwards, textually. Therefore the constraints in lines 2 and 3 are required. The constraints in line 5 have been labelled with strength levels but are not inside the scope of any relaxation, and therefore can only be either enforced or completely ignored, in the standard HCLP manner. There is no difference between the constraints in lines 7 and 9 in terms of how they will be relaxed. The only difference caused by one of them having a strength label is in the score that will be calculated.

The two predicates `p/2` and `q/2` are assumed to impose some constraints (we will ignore precisely what these constraints are) but, more importantly, they are assumed to include disjunctions. The existence of different constraint hierarchies due to disjunctive rules (i.e. rules with more than one clause which is satisfiable) leads to the following problem. Imagine there are two branches of the computation, one with 10 soft constraints of which 5 are satisfied, and one with 5 constraints of which 4 are satisfied. If we simply count number of satisfied constraints as part of the score, then we should treat the first branch as being 'better', even though it satisfies a smaller proportion of its constraints. A similar problem arises if the number of *un*satisfied constraints is counted, if one branch satisfies 98 out of 100 constraints but the other satisfies 1 out of 2. Using the *proportion* of constraints satisfied is possible, but also gives rise to problems (for other examples). This issue also arises in HCLP, and has been addressed in [83, 84], under the heading 'inter-hierarchy comparison'. In GOCS we also have disjunctions due to the different possible relaxations for a constraint, both within and between score function equivalence classes, but this does not cause any extra complications of this sort.

When a clause is selected whose head is unifiable with any current bindings on the variables, a new instance of the body of the clause is created with new variables. These are then unified with the actual parameters as appropriate. Unification can be thought of as imposing an equality constraint, which might be within the scope of some relaxation method. However, they should *not* be treated as relaxable. In fact, if an Eclipse-like syntax is used it is easy to make the distinction between these equations and those imposed by the user: equality constraints over finite domains are indicated by `#=`, and over rational domains by `$=`, whereas unification equalities are denoted simply by `=`. Therefore there is no problem distinguishing between them, and not relaxing these 'rename-apart' equalities.

A typical execution strategy for the above query is as follows: follow the constraints in textual order, imposing them without relaxing or ignoring them, but making note of any strength labels. If a labelled constraint is inconsistent with the current store and is not within the scope of a relaxation, then ignore it, but note the fact that a constraint of that strength has been ignored. Select one of the clauses for `p(X,Z)` for which the head is unifiable with any current bindings of variables (all of them are unifiable in this case as no variable has yet been bound), create a choice point, and continue imposing the constraints found in the body of `p(X,Z)`, and so on, recursively.

When line 5 is reached, impose the labelled constraints, as they are consistent. If such a constraint is not consistent, then just ignore it completely, calculating the appropriate score. If the constraint in line 7 is consistent, impose it. If not, choose one of the relaxations in the earliest consistent equivalence class defined by the relaxation method, e.g. $X = 2$ or $X = 4$. Create a choice point if there are other consistent possibilities, whether they are due to the members of that equivalence class, or of other equivalence classes in the list.

Line 8 may also create a choice point, leading to other constraints being imposed, and other predicates being called. Note that all constraints inside `q(Y,Z)` are assumed to be inside the scope of the `relax` on line 6, unless explicitly labelled. This gives rise to similar questions to those raised by HCLP with strength labels on predicates as well as constraints (see 'min-strength algebras' etc. in Section 3.5.3).

The constraint in line 9 has a strength label as well as being inside the scope of the `relax` on line 6, and so it can be relaxed, or it can be completely ignored, if that would result in a better score. If the score function treats relaxing a constraint as a smaller change than completely ignoring it, it will probably not be ignored on the first sweep through the search space. Once the initial bound has been found, it may in fact contradict so many constraints elsewhere that a better score arises from ignoring it completely in the HCLP style. Again, if it is relaxed it creates a choice point.

Line 10 indicates that the predicate `r/3` is not in the scope of the `relax` on line 6, and so any constraints imposed inside it will be assumed to be required, unless themselves explictly labelled or relaxed.

In Chapter 12 there is another example of a query in this syntax, in the context of a real-world example. It shows how we can use GOCS to avoid the HCLP necessity of labelling every single constraint in a problem, while still maintaining the possibility of local control.

It may be clear from the above how HCLP and PCSP can be treated as instances of this scheme, by only using the appropriate half of the syntax (either strength labels or relaxation operations). A more formal demonstration of the subsumption of HCLP and PCSP by GOCS is given in the next section.

11.5 HCLP and PCSP as instances of GOCS

11.5.1 HCLP in GOCS

We can show that HCLP is an instance of GOCS by showing how we can imitate HCLP's two important characteristics which are (a) *all* required constraints must be satisfied, and (b) constraints are either included or omitted[2], without augmentation or relaxation. We now consider these two issues in more detail.

(a) If all the required constraints *can* be satisfied at once, there is no problem: the appropriate h will give them scores showing that they are the preferred elements in the order. So we only need to consider the case when it is impossible to satisfy all the requirements at once. Then HCLP would declare 'failure', and implementations might initiate back-tracking, etc. One method would be to parameterise h by r, the number of required constraints in the original problem, and give the worst possible score 'bottom' to any bag which did not contain them all. Then an additional constraint could be placed on what we mean by 'solution'; instead of simply being the best element in the order, it would have to be the best non-bottom element. But parameterising h in this way would mean that it would not satisfy the property mentioned earlier as being sufficient for compositionality of $\oplus_h$. An alternative, therefore, is simply to parameterise the pre-order $\preceq_h$ instead, so that the solution would be considered to be the best element scoring at least r.

Another alternative is, perhaps, even better: a 'required' constraint is one which must be used in the final solution, and which cannot be relaxed. In other words, if c is required, then

$$Rel(c) = \{\wr c \wr\}$$

[2]This is not strictly true when considering metric comparators.

This forces c to be included in each member of $[\![\_]\!]$. If the required constraints are themselves over-constrained, all these elements will be inconsistent, and so Gocs will produce an empty solution set.

(b) If we define a scoring function h which includes a distance function δ which gives an infinite increase in distance for all augmentations of the original constraint, this will have the same effect as removing them from $Rel(c)$, in which case $Rel(c) = \{\Lbag\Rbag, \Lbag c\Rbag\}$ and so the members of $[\![P]\!]$ will be the same as the power-bag of P. This would restrict us to either including a constraint or omitting it completely, thus satisfying the other characteristic of HCLP.

Thus we can implement HCLP in Gocs either by changing the definition of $Rel(c)$ to be just $\{\Lbag c\Rbag\}$ for required constraints and $\{\Lbag\Rbag, \Lbag c\Rbag\}$ for optional constraints, or by using a particular score function h. The former approach is closer to the philosophy behind HCLP itself, but means that Gocs has to be parameterised by both Rel and h. The latter approach fits more clearly into the Gocs's own framework. Note that the modifications to Rel can be simulated by careful choice of h; therefore we will not choose definitively between them here.

Given Proposition 21 (compositionality up to the point when the results are ordered — see also Section 8.4), then the above reconstruction of HCLP is also compositional up to the same point. Of course, this is in contrast to the standard presentation of HCLP (Chapter 3).

11.5.2 PCSP in Gocs

Unlike HCLP, PCSP does not have strength labels. Also, it does not distinguish a special set of constraints as being 'required'. Therefore, it is clear that we can treat PCSP with Gocs by ignoring the label on each constraint, or treating them all as being, say, 'strong'.

PCSP is parameterised by a distance function δ, whose definition can be used for the Gocs score function h. The three standard distance functions are based on number of solutions not shared by two problems (the 'solution subset' distance function), number of values not shared by their constraints (the 'augmentations' distance function), and number of constraints satisfied by a solution (MaxCSP) [33].

Solution-subset requires all the solutions to be generated before the order can be decided, and so generally it is not used. The links between the augmentations distance function and constraint augmentation as represented by $Rel(c)$ are very clear: one can almost derive the distance simply by counting the number of dashes on each constraint! The order $\leq_h$ can easily be derived from the order over the problem space already present in the definition of PCSP. In this case, it is just the standard order over the integers. In the present context MaxCSP is similar to the augmentations distance function, in as much as each constraint violated can be forced to be satisfied by augmenting precisely one constraint. Hence a similar argument holds.

Given Proposition 21, the above reconstruction of PCSP is compositional. This is not really surprising, as the distance functions normally discussed in PCSP all have the subset-closure property we mentioned earlier (Chapter 10); compositionality has not been, and has not needed to be, a focus of research in PCSP.

11.6 Comparative complexity of GOCS

GOCS is very general, in two different senses: firstly, there is a large choice of score functions. Secondly, for each score function GOCS can subsume both the HCLP and the PCSP approach to problem relaxation. Due to the first sort of generality, it is possible to express very hard problems. In this case, it would not be surprising if GOCS was computationally expensive, but that is due to the nature of the problem and not the nature of GOCS.

However, what is more interesting is what price must be paid for the second sort of generality. Specifically, if it is possible to express something in HCLP in such a way that it can be solved reasonably efficiently, we would hope that expressing it in GOCS does not worsen its complexity significantly. Therefore, we will now consider the complexity of the example score function h, similar to UCB and MaxCSP, which we introduced in Section 11.2.4. Our argument will be informal, and based on the assumed complexity of UCB in a standard HCLP implementation.

For each level of the hierarchy, UCB requires the best subset of that level. Therefore if there are n constraints at that level, UCB requires 2^n subsets to be examined in the worst case. GOCS requires the same amount of work, ignoring augmented constraints. Once the maximal collection of unaugmented constraints is found, GOCS then augments each of the excluded constraints at that level with one extra tuple of values. This should take time linear in the number of constraints to be augmented, and therefore can be ignored when compared to $O(2^n)$. Of course, the collections at all but the first level must be checked for consistency within the context of the best solutions found for the levels of greater preference, both for UCB and GOCS.

Note that augmenting constraints will not affect the possible solutions to subsequent levels of the hierarchy — any constraint which might be affected by the presence of the augmented constraint would have conflicted with the stronger level anyway. This may not be very clear, so consider the following example.

Example:
Consider three constraints at the strong level, $X < 8$, $X < 9$, and $X > 9$, and one medium constraint $2 \leq X \leq 5$. The best solution to the strong level will satisfy two out of the three constraints, $X < 8$ and $X < 9$, and so the third constraint $X > 9$ will be augmented, and therefore treated as being medium by our example score function. $X > 9$ can be augmented in many different ways consistent with the solution to the strong level, for example $X > 9 \vee X = 1$, $X > 9 \vee X = 2$, and $X > 9 \vee X = 3$. (If it is augmented with more than one value, it will move more than one level down the strength hierarchy, and so will not affect the medium level.) Any collection of medium constraints which contains an augmentation which contradicts $2 \leq X \leq 5$ will receive a worse score than those containing augmentations consistent with $2 \leq X \leq 5$. All

the possible medium levels in which the augmentation falls between 2 and 5 will be given the same score, and so all will appear in the set of maximal solutions to the problem as a whole (and therefore it does not matter that the single augmentation $X > 9 \vee 2 \leq X \leq 5$ is not present at the medium level). Therefore, pushing the contradictory constraint down a level and augmenting it, instead of ignoring it in the style of HCLP, will not affect the possible solutions to subsequent levels of the hierarchy. $\square$

Once a consistent collection has been found, it takes at most linear time to calculate its score. It could even take constant time, if information has been accumulated during the consistency-checking stage. Clearly, both UCB and GOCS can use the best score found so far for branch-and-bound pruning of the search space, if only the best solutions are required. So if there are k levels with an average of n constraints at each level, the complexity of GOCS with the particular score function we have been discussing is $O(k2^n) = O(2^n)$, which is the same as the UCB complexity. Any improvements to the UCB result should translate directly into the GOCS case. Therefore, it seems that we do not pay a large price for encoding the HCLP approach in GOCS.

PCSP in general treats all constraints equally. Therefore, if there are kn constraints in the problem as a whole, all at the same level, any exponential complexity will be in terms of $O(2^{kn})$, which is worse than $O(k2^n)$. It is clear that encoding a pure PCSP approach in GOCS will not incur additional computational complexity.

The interesting question is to analyse the complexity of a mixed approach, which has both HCLP and PCSP aspects at the same time. See, for example, the GOCS treatment of the example in the next chapter. If half the constraints have strength labels, and the other half do not, the score function which we are discussing will treat the unlabelled unaugmented constraints as being 'strong'. Therefore the strong level will be quite large, containing at least $kn/2$ constraints (in fact $kn/2$ plus any which are actually labelled strong), and so will have complexity $O(2^{kn/2})$ or more.

Clearly, using $\ll$ for 'much less than', we can say that $O(2^n) \ll O(2^{kn/2}) \ll O(2^{kn})$, and so GOCS with this score function is in principle no worse than a straightforward PCSP approach, and possibly better.

11.7 Conclusions

11.7.1 GOCS considered as a preference system

In Section 5.2, we listed some of the characteristics of the ideal preference system. We will now go through that list and see to what extent GOCS achieves the ideal.

With respect to the four important characteristics C1 – C4, we claim that GOCS enables the **declarative expression of preferences** (C1) and **takes account of any user-provided strength labels present** (C2), as does HCLP. We also claim that GOCS **does not require all strength levels to be stated** (C3) and is **compositional** (C4), and shares these two characteristics with PCSP. Therefore GOCS has all four of those characteristics that we feel are most important.

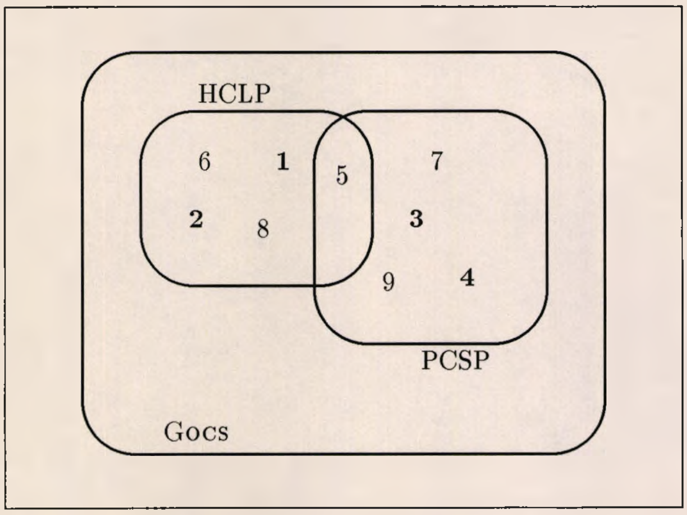

Figure 11.1: Gocs combines all the advantages of HCLP and PCSP

Gocs allows preference to be expressed both globally and locally, via the combination of the score function h and the definition of *Rel*. Therefore, the user should get as many solutions as match his criteria, but not more, thus the output should be reasonably **easy to understand** (C5). Gocs **allows the expression of required constraints which must not be violated** (C6) but can also **cope when the set of required-by-default constraints is itself over-constrained** (C7). These two characteristics depend on the definition of *Rel* and cannot generally hold at the same time.

By design, Gocs **allows the relaxation of the general problem structure** (C8) as does HCLP, but also **allows the relaxation of individual constraints** (C9), like PCSP.

With respect to the characteristics that we found hard to quantify, the **severity** of Gocs depends on the precise choice of *Rel*. Given that it does not impose an overhead compared to UCB or PCSP (Section 11.6), we can say that Gocs is indeed **implementable**.

Therefore we feel justified in saying that Gocs possesses *all* the separate advantages of HCLP and PCSP, as claimed in Section 5.3. See Figure 11.1.

11.7.2 Conclusions

We have developed a general framework for over-constrained systems. It has HCLP and PCSP as special cases, and it allows the use of aspects of both systems at the same time, thus giving the user the flexibility to consider the priority of as many or few constraints as is desired. Furthermore, it embodies the two different approaches to problem relaxation: changing the structure of the *problem* in the style of HCLP, or changing the meaning of *individual constraints* as done by PCSP.

HCLP and PCSP have complementary advantages and disadvantages: HCLP allows fine-grained control but forces the user to label every single constraint, whereas PCSP does not force the user to label all constraints, but does not allow as much detailed control. GOCS can use as much HCLP-like labelling information as it is given, but does not require more. Therefore the user has all the benefits of HCLP and PCSP without the disadvantages.

Compositionality is very important, and yet many other systems for dealing with partial and preferential information do not even attempt to limit their non-compositional and non-monotonic aspects. Our framework is always compositional with respect to the consistent problems derived from the OCS by constraint relaxation, and as long as a certain condition is met it is also compositional with respect to the scores assigned to each possible solution.

11.7.3 Benefits

The combination of various of the characteristic of an ideal preference system which GOCS possesses leads to efficient use of time by specifiers of systems — they can concentrate on global issues for most of the time, but are still able to fine-tune the model without needing to consider every single local interaction. We demonstrate this in the next chapter, which presents a detailed comparative treatment of a real-world example, as modelled in HCLP, PCSP, and GOCS.

Chapter 12

Example — The Radio Link Frequency Assignment Problem

12.1 Introduction and background

The “Radio Link Frequency Assignment Problem” (RLFAP) is to assign frequencies to communication links in such a way that no interference occurs. It is a real-world problem, which we will specify and solve using HCLP, PCSP, and GOCS.

Our aim is to show that it is easier to specify OCSs in GOCS than in either HCLP or PCSP. HCLP requires every single constraint to be given a strength label, whereas PCSP requires any local information to be encoded in the global distance function. This will be hard to do, at the least, and the result will usually not be very declarative.

12.1.1 General problem description

RLFAP problems contain two types of constraint:

- $|x - y| > k$ — the absolute difference between two frequencies should be greater than some given number k
- $|x - y| = k$ — the absolute difference between two frequencies should exactly equal k

According to the problem documentation, “The first type of constraint is enough to make the problem NP-hard since it enables the expression of the *Graph k-Colourability* problem” [13].

The CELAR suite is a set of 11 instances of the generic RLFAP problem, which is regarded as a set of benchmarks. These benchmarks are not merely artificial, as they are drawn from a real-world military application. We will present a cut-down version of one of the

CELAR instances, known as Problem 6. This shorter version, which we call Example ζ, is over-constrained, as are Problem 6 itself and also some of the other problems.

Having discovered that a particular instance is over-constrained, we must inform the customer, an army officer. We can offer him a choice: he must either tell us which constraints are important and which are not (in the style of HCLP), or he must provide a general cost/objective function, and allow us to relax constraints arbitrarily in order to minimise the cost.

The original specifier of the CELAR suite probably followed a similar path to the one outlined here. The army selected the first option (HCLP-style), as can be seen by examining the CELAR instances: each constraint has been given a weight[1]. The smallest problem in the CELAR suite has more than 1200 constraints, and so choosing the first option entails a large amount of work for the specifier. We hope to show that if the army had been offered a third choice, namely GOCS, they might very well have chosen it instead.

Notwithstanding what actually happened, we will now consider our Example ζ as a problem to be solved, which we will tackle in HCLP, PCSP, and then GOCS. First of all, we present some common background which will be needed in all three formulations.

12.1.2 Description as a CSP

Each variable represents the frequency to be assigned to one link. The frequencies available vary from link to link, but all of them are represented by one of seven non-continuous ranges of integers, i.e. finite domains. The values in the domains range from 16 to 792. The largest domain (greatest cardinality) has size 48, i.e. the most flexible link can choose one of 48 different frequencies to use between 16 units and 792 units. The variables are identified by numbers (non-contiguous integers), which we will precede with an X to obey standard CLP variable-naming conventions.

Throughout this chapter, we will use the syntax of the Eclipse finite domain constraint solving library [22]. Note that Eclipse does not have a built-in absolute value constraint; the `abs/1` predicate to be found in Eclipse can only be used with `is/2`, which is not declarative, not back-trackable, and does not propagate information forwards. However, Micha Meier has provided the body of the following procedure, as a simple implementation of such a constraint [Private Communication].

We have added a procedural wrapper in which the third argument is only used for pattern matching in clause selection. Also, we have reduced the limits in the second clause: for full generality `Diff` should have domain $[-\mathit{LargeInt}..\mathit{MNum}, \mathit{PNum}..\mathit{LargeInt}]$, for some large integer value which is the default limit of a finite domain. In Eclipse, *LargeInt* is usually 1,000,000. However, in the CELAR problem suite, the largest value in any of the domains is 792, and so we can reduce the limits accordingly.

[1] In fact, weights and HCLP-style strength labels are not the same semantically (as discussed below in Section 12.2). But from the user's point of view, assigning weights to all the constraints involves as much work as assigning strength labels.

The code for $|X - Y| = Num$ and $|X - Y| > Num$ is:

```
abs(X,Y,=,Num) :-
     MNum is -Num,
     Diff :: [MNum, Num],
     X - Y #= Diff.

abs(X,Y,>,Num) :-
     MNum is -Num-1,
     PNum is Num+1,
     Diff :: [-793..  MNum, PNum..793],
     X - Y #= Diff.
```

12.1.3 Other background information

Each constraint in each problem in the CELAR suite is labelled with one of the letters D, C, F, P or L[2]. Note that in Problem 6 all the D-type constraints are equalities (i.e. of the form $|x - y| = k$), and are *required*. All the F and P constraints are strong, whereas the C and L constraints are either medium, weak, or very weak. This suggests that the approach of Section 5.4, where we discussed constraints schemas, might to some extent be applicable here.

There are 200 variables in Problem 6, and 1322 constraints. This is the equal-smallest problem in terms of number of variables, and second smallest in terms of constraints. For comparison, the smallest problem has 200 variables and 1235 constraints, whereas the largest problem has 916 variables and 5744 constraints.

In the CELAR problem suite, there are four sorts of solution to be found:

SAT: find a valuation satisfying all constraints (both hard and soft);

CARD: if the problem is satisfiable, find a valuation minimising the total number of frequencies used

SPAN: if the problem is satisfiable, find a valuation minimising the largest frequency used

MAX: if the problem is over-constrained, find a valuation that minimises the total cost of violations, using the following optimisation function (we have simplified it by omitting references to 'mobility' [13]):

$$cost = a_1 n_1 + \ldots + a_4 n_4$$

where the a_i are problem specific ($a_1 = 1000$ down to $a_4 = 1$ for Problem 6), and n_i stands for the number of violated constraints with weight a_i.

[2]Apparently these letters refer to the original meaning of the constraints, but they are not directly useful in a CSP context [13].

We chose Problem 6 because it is the smallest instance of type MAX, which is clearly the problem class most closely related to UCB and MaxCSP.

Note that the MAX optimisation function, when treated as a PCSP distance function, is subset-closed.

Proof:
Reminder: an order $\leq$ is subset-closed if for all sets of solutions u, x_1, x_2, $(u \subseteq x_1 \wedge x_1 \leq x_2) \rightarrow u \leq x_2$. In Section 10.2.3 this was shown to be equivalent to saying that $\leq$ contains $\subseteq$ [Hunt].

Assume that $u \subseteq x$. So the problem U for which u is the solution set contains more constraints than the problem X_1 for which x_1 is the solution set. So the violation cost of U is less than the violation cost of X_1. Therefore $u \leq x_1$. So $u \subseteq x_1 \rightarrow u \leq x_1$, and so $\leq$ is subset closed. □

12.1.4 Our modifications of Problem 6

There are 200 variables in Problem 6 as a whole. 18 of them are identified by (non-contiguous) numbers between X13 and X146. Another 22 range from X215 to X274, and the rest vary from X275 to X880. We decided to focus on the variables between X215 and X274, to reduce the size of the problem for clarity of presentation. We will refer to these as the X200 variables.

There are 1322 constraints in the problem as a whole, but if we select only those involving the X200's, we get 313, of which 18 are *required*. This problem is still over-constrained, including multiple separate infeasible subsets. Note, however, that simply selecting constraints involving certain variables changes the meaning of the problem. For example, if the original problem contained the four constraints $|X13 - X201| > 24$, $|X14 - X202| > 36$, $|X201 - X202| = 0$, and $|X13 - X14| = 0$, we would simply ignored the fourth one when trying to reduce the size of the problem. This removes one infeasibility. However, as we have already mentioned, our cut-down version of the problem remains over-constrained. Furthermore, the point of this exercise is not to solve Problem 6, but to demonstrate the difference in effort of using HCLP, PCSP, and Gocs — the *concept* of an RLFAP is important here, and not the solution to any particular instance.

In fact, even reducing 1300 constraints to 300, of which 15 must all be ignored to get solutions, is too much to present here. Therefore we have deleted more and more constraints. We will now present a version with 44 constraints, which still has two separate inconsistencies. We have already referred to it, using the name Example ζ.

12.2 Example ζ in HCLP

As mentioned earlier, if all the constraints in ζ are considered to be mandatory, the problem is over-constrained. Some typical constraints are:

```
abs(X271, X272, =, 238)
abs(X273, X274, =, 238)

abs(X271, X274, >, 293)
abs(X271, X292, >, 480)
```

An HCLP-style response would be to label each of the constraints with a strength. Apparently, it is standard for the equality constraints in this problem to be considered to be *required* or its equivalent in all the various frameworks which can be used to relax OCSs. However, this first step is insufficient: if we ignore all the optional constraints, there are far too many solutions.

We could treat all the optional constraints as being equal, and select a comparator such as UCB which would try and minimise the number of violated constraints, without differentiating between them. This is in fact similar to the PCSP approach which we will consider in the next section, and which is not ideal for this problem[3].

In order to avoid the problems which will be encountered by the PCSP approach (below), it is necessary to use more than one strength label for the optional constraints. This gives rise to three problems:

- In general, there are far too many constraints. Asking an army officer to choose a strength label for each constraint individually is not realistic.
- Even if the client is willing to do so, it is difficult for him to be consistent: HCLP strength labels are global, and although the officer might know that for one particular radio it is better to choose x than y, and for another radio it is better to choose u than v, the relative strength of x and u is difficult to define. Partially ordered constraint hierarchies could be used (Section 3.5.5) but calculating solutions to them is *very* expensive computationally.
- If the work is split among a number of officers or groups of clients, another problem of consistency arises, in addition to the one mentioned in the previous item. Individual's judgements may vary, and this may even be intentional (cheating).

We invite the reader to consider the example provided by Professor Richards discussed in Section 2.5, or any large-scale problem of interest — would it be feasible to ask your client for every single constraint to be labelled with a strength?

The second and third problems above could be avoided by making the problem modular. Each group of clients could specify their own local preferences, without the problems entailed by global strength labels. Then the sub-problems could be combined. Unfortunately, standard HCLP is not compositional, and so this approach cannot be used.

In fact, the Celar set of Rlfap problems includes a weight for each optional constraint. In Problem 6, and hence in ζ, the constraints have a weight of 1000 for the strongest,

[3]It is interesting that the simplest HCLP-style approach is so closely related to a simple PCSP model. As shall be seen below, the more complex PCSP approach to these benchmarks taken by others is itself similar to HCLP.

then 100, 10, and 1 for the weakest. The cost function is to minimise the weighted number of violated constraints. In the problem description files these numbers are not used; instead, each constraint has a 'weight index', where an index of 1 refers to a weight of 1000, down to an index of 4 for a weight of 1. Figure 12.1 shows all the constraints in ζ along with their weight indices. For completeness, Figure 12.2 shows the domain of each variable.

```
abs(X13, X14, =, 238)                    abs(X15, X16, =, 238)
abs(X81, X82, =, 238)                    abs(X83, X84, =, 238)
abs(X85, X86, =, 238)                    abs(X143, X144, =, 238)

abs(X145, X146, =, 238)                  abs(X215, X216, =, 238)
abs(X233, X234, =, 238)                  abs(X235, X236, =, 238)
abs(X237, X238, =, 238)                  abs(X261, X262, =, 238)

abs(X263, X264, =, 238)                  abs(X265, X266, =, 238)
abs(X267, X268, =, 238)                  abs(X269, X270, =, 238)
abs(X271, X272, =, 238)                  abs(X273, X274, =, 238)


abs(X269, X271, >, 59)      % 1          abs(X269, X272, >, 186)     % 2
abs(X269, X273, >, 436)     % 3          abs(X269, X274, >, 293)     % 4

abs(X270, X272, >, 59)      % 1
abs(X270, X271, >, 186)     % 2   (constraint A)

abs(X270, X274, >, 117)     % 2          abs(X270, X273, >, 293)     % 4

abs(X271, X273, >, 436)     % 3   (constraint B)
abs(X272, X273, >, 293)     % 4   (constraint C)

abs(X271, X556, >, 490)     % 1          abs(X271, X292, >, 480)     % 2
abs(X271, X629, >, 153)     % 2          abs(X271, X713, >, 12)      % 2
abs(X271, X714, >, 168)     % 2          abs(X271, X717, >, 21)      % 2
abs(X271, X719, >, 21)      % 2          abs(X271, X787, >, 216)     % 2

abs(X271, X555, >, 349)     % 3          abs(X271, X608, >, 10)      % 3
abs(X271, X630, >, 10)      % 3          abs(X271, X274, >, 293)     % 4
abs(X271, X583, >, 224)     % 4          abs(X271, X584, >, 53)      % 4
abs(X271, X718, >, 158)     % 4          abs(X271, X720, >, 158)     % 4
```

Figure 12.1: ζ required equality constraints, and optional constraints with weights

Very loosely speaking, we can say that the constraints with index 1 are *strong*, those with index 2 are *medium*, then *weak* and *very weak*. However, **this is not correct:** the cost function would select eleven medium constraints rather than one contradictory

CELAR Domain 1:

> $[X215, X216, X261, X262, X263, X264, X265, X266, X267, X268,$
> $X269, X270, X271, X272, X273, X274] ::$
> $[16, 30, 44, 58, 72, 86, 100, 114, 128, 142, 156, 254, 268, 282, 296,$
> $310, 324, 338, 352, 366, 380, 394, 414, 428, 442, 456, 470, 484, 498,$
> $512, 526, 540, 554, 652, 666, 680, 694, 708, 722, 736, 750, 764, 778, 792]$

CELAR Domain 3 [*sic*]:

> $[X233, X234, X235, X236, X237, X238] ::$
> $[30, 44, 58, 72, 86, 100, 114, 128, 142, 268, 282, 296, 310, 324, 338,$
> $352, 366, 380, 428, 442, 456, 470, 484, 498, 512, 526, 540, 666, 680,$
> $694, 708, 722, 736, 750, 764, 778]$

The above shows the domains for the variables of interest to us (X200).
All the others are assigned Domain 1.

Figure 12.2: ζ variable domains

strong constraint. HCLP comparators 'respect the hierarchy', which means that an infinite number of medium constraints should be violated rather than one strong.

When trying to solve ζ, it turns out that there are three constraints of interest, labelled A, B, and C in Figure 12.1:

A :	`abs(X270,X271,>,186)`	with index 2	hence weight 100
B :	`abs(X271,X273,>,436)`	with index 3	hence weight 10
C :	`abs(X272,X273,>,293)`	with index 4	hence weight 1

There are multiple solutions to ζ if A is commented-out, all with cost 100. There are also multiple solutions if A is enforced but B and C are commented-out, with cost 11 $(10 + 1)$. There are no solutions with A enforced and also either B or C enforced, i.e., we cannot obtain solutions of cost 1 or 10. Therefore the best solution is to relax B and C, with cost 11.

The above result uses the weights supplied by the original specifier, who probably expended a great deal of effort (or whose client expended a great deal of effort) to define all the weights. However, it appears that the effort was indeed necessary, as the 'obvious' solution, treating all the optional constraints as the same, would be to violate A, which is clearly incorrect from the user's point of view. This issue is important in our discussion of PCSP, which follows.

12.3 Example ζ in PCSP

Let us start by assuming that the PCSP approach is being tried *ab initio*, i.e. without knowledge gained in the HCLP stage above. Let us also assume that we have a version of PCSP which allows required constraints, but that we are not using the detailed weight information involving the four indices.

We can select a standard PCSP distance function such as MaxCSP, which maximises the number of satisfied constraints, i.e. it minimises the number of violated constraints. Then the best solution is one in which A alone is violated. But when this solution is shown to the user, he will not accept it. He will at least say "There are other optional constraints which have been satisfied by this solution which I would prefer to be violated rather than A." In this particular example, the problem could be solved by moving A into the class of required constraints, but that is not generally acceptable. For example, in a more complex instance it might lead to a constraint with weight index 1[4] being violated instead of A which has index 2.

If it is not possible simply to re-label A 'required', we could modify the distance function to prefer A. Such a distance function could impose the appropriate order on solutions σ_1 and σ_2 by stating that all other things being equal,

$$\sigma_1 < \sigma_2 \qquad \text{if } \sigma_1 \models A \text{ and } \sigma_2 \cup A \models \bot$$

However, this might give rise to solutions which satisfy A but which violate stronger optional constraints. Furthermore, it does not scale well. If σ_1 is earlier than σ_2 according to one statement like the above, but $\sigma_2 < \sigma_1$ according to another such statement, then the problem has become one of multi-criteria optimisation, which is just as hard as any of the other approaches we use.

An alternative which *does* scale is to ask for more information to establish a total order over all the optional constraints, and to select a distance function which takes account of it. The order can be defined using strength labels or using weights, but in either case we find the same problems that we discussed with respect to HCLP in the previous section.

12.4 Example ζ in Gocs

12.4.1 Formulation

Once more, we start at the beginning, accepting that some constraints are required, but not using the weight information. Using some simple score function h, such as `max-ucb`, the one similar to MaxCSP and UCB discussed in Section 11.2.4 and following, Gocs calculates solutions which violate A. As the user feels that there are constraints which

[4] I.e. a constraint which we know with 'hindsight' to have weight index 1. Clearly, the user may not express his preferences in this manner, but equally clearly these preferences *do* exist, otherwise the weights found in the actual benchmark files would be completely arbitrary.

should be violated rather than A (as well as stronger constraints which should *not* be violated merely in order to satisfy A) we must then ask him to identify some of them.

The user may select a number of constraints (more than just B and C[5]), but will not need to consider the relationship between *all* the optional constraints. Therefore, the HCLP problem does not arise; at worst, probably, the user might label all the constraints involving $X271$ before happening to consider B and C. Eventually, perhaps after he has iteratively weakened other constraints without changing the over-constrained nature of the problem, he may say that A is more important than B and C put together.

Then the GOCS formulation will have different relaxation operators *Rel* for the required constraints, for the unlabelled constraints, and for those that we specifically label in a local fashion. As discussed in Section 11.5, we can consider the relaxation for a required constraint to be simply

$$Rel(c) = \{\lbag c \rbag\}$$

The relaxation of an optional constraint will be

$$Rel(c) = \{\lbag \rbag, \lbag c \rbag, \lbag c' \rbag, \lbag c'' \rbag, \ldots\}$$

whether or not it is labelled with a strength. If it *is* labelled (or has a weight attached to it), then the label (weight) stays the same in all members of *Rel*.

We label A, B, and C in order to define a local ordering over them, and therefore it does not matter if we call A 'strong' and B 'medium', or A 'medium' and B 'weak'. However, just to maintain consistency of presentation with respect to the four weight indices, we could label the three constraints 'medium', 'weak', and 'very weak'.

Remember that in PCSP, c' denotes an augmentation of the domain of constraint c with an extra tuple of values. Finding a solution in which c' is present, and such that there is no equivalent solution with c instead of c', is equivalent to finding a solution in which c is violated.

Let us use D to denote the bag containing one occurrence of each of all the other 41 constraints in the problem apart from A, B, and C. D is consistent in itself, and so will appear in all the maximally preferred solutions to ζ. $[\![\zeta]\!]$ will be annotated with sequences of 5 integers, the scores for the required, strong, medium, weak, and very-weak levels (or more than 5 integers if the weaker levels are augmented). D contains constraints at all the strength levels, but its scores for each level will be the same in all cases of interest to us, and so we just place d in the first (required) position in the sequence in order to distinguish solutions containing D from the empty bag. In fact $d = 18$ with this score function, but that is not interesting here. Remember that for this particular score function, one augmentation is equivalent to one movement down the strength hierarchy, so a very-weak constraint (position 5 in the sequence) scores 1

[5]Or even a set of constraints which does not include B and C at all, but this will not result in solutions which satisfy $\mathcal{T}$, and so the user may be approached again, until a superset of $\{B, C\}$ has been identified.

in the **sixth** position if it has been augmented once.

$$\begin{aligned}[\![\zeta]\!] = \{\ & \lbag\rbag^{[0,0,0,0,0]},\ \lbag D\rbag^{[d,0,0,0,0]},\\ & \lbag A, D\rbag^{[d,0,1,0,0]},\ \lbag A, B', C', D\rbag^{[d,0,1,0,1,1]},\\ & \lbag B, D\rbag^{[d,0,0,1,0]},\ \lbag C, D\rbag^{[d,0,0,0,1]},\ \lbag B, C, D\rbag^{[d,0,0,1,1]}, \ldots\}\end{aligned}$$

A reminder of the actual constraints named A, B, B', etc, is given on the next page.

Therefore GOCS chooses $\lbag A, B', C', D\rbag^{[d,0,1,0,1,1]}$ as it is last in the lexicographic order. This is clearly the correct solution.

In order to be explicit about the best bag in the order, let us consider one particular choice of B' and C'. We also repeat the definitions of the unaugmented constraints from Figure 12.1. Remember that D stands for all the other constraints apart from A, B, and C:

A :	`abs(X270,X271,>,186)`
B :	`abs(X271,X273,>,436)`
B' :	`abs(X271,X273,>,436)` $\vee$ `(X271,X273) = (708,652)`
C :	`abs(X272,X273,>,293)`
C' :	`abs(X272,X273,>,293)` $\vee$ `(X272,X273) = (470,652)`

In fact, the solution to $\lbag A, B', C', D\rbag^{[d,0,1,0,1,1]}$ is also one of the solutions for $\lbag A, D\rbag^{[d,0,1,0,0]}$, as augmenting B is equivalent to violating it, as mentioned earlier. In other words, solve the problem, completely ignoring B and C. As it happens, one solution includes $X271 = 708$, $X272 = 470$, and $X273 = 652$. Any constraint between $X271$ and $X273$ which was ignored, such as B, can be included in the original problem in augmented form, as long as the tuple $(708, 652)$ is added to its domain, and similarly for constraints between $X272$ and $X273$.

$\lbag A, D\rbag$ has many different solutions, whereas each particular augmentation of B and C will only give rise to one solution. However, if we take the union of the solutions to all the different possible augmentations of B and C we will obtain the same set as for $\lbag A, D\rbag$. Choosing one particular augmentation of each is similar to choosing only the 'first' solution to $\lbag A, D\rbag$.

The fact that the complete removal of B or C from the problem changes 0 solutions into a large number is slightly surprising, and may be a side-effect of our reduction of the size of the problem. Whatever the reason, such a large number of solutions might be considered to be a 'flood' and so to be a criticism of GOCS. Of course, the HCLP and PCSP representations of the problem would also have this number of solutions, but that is not an acceptable excuse.

So, given that there are too many solutions, what can be done? The answer is simple: impose extra constraints, say E. In HCLP it would be necessary to create the conjunction $\lbag A, D, E\rbag$ and re-calculate from scratch. This is not necessary if GOCS is used. Calculate the solutions to $[\![E]\!]$ and then compose with $[\![\zeta]\!]$. As an aside, this shows that compositionality is not merely a theoretical issue, but is genuinely useful.

Therefore, perhaps we should not have said that *avoiding* a flood of answers is a characteristic of the ideal system, as it may be due to the nature of the problem. Instead, we could say that being able to *manage* or *reduce* a flood, once it has occurred, is a useful property. GOCS is successful with respect to this formulation, as well as with respect to the original characteristic.

12.4.2 Interaction with an implementation

We now re-present some parts of the previous subsection using the instance of the GOCS framework described in Section 11.4.

Using the fact mentioned in the previous section that we only need to ignore the violated constraints instead of augmenting them, and using the convention that unlabelled constraints are required, we can write the initial (unsatisfiable) query as follows:

```
?-   abs(X13, X14, =, 238), abs(X15, X16, =, 238), abs(X81, X82, =, 238),
abs(X83, X84, =, 238), abs(X85, X86, =, 238), abs(X143, X144, =, 238),
abs(X145, X146, =, 238), abs(X215, X216, =, 238), abs(X233, X234, =, 238),
abs(X235, X236, =, 238), abs(X237, X238, =, 238), abs(X261, X262, =, 238),
abs(X263, X264, =, 238), abs(X265, X266, =, 238), abs(X267, X268, =, 238),
abs(X269, X270, =, 238), abs(X271, X272, =, 238), abs(X273, X274, =, 238),
    % Point 1.  (See comments below)
abs(X269, X271, >, 59), abs(X269, X272, >, 186), abs(X269, X273, >, 436),
abs(X269, X274, >, 293), abs(X270, X272, >, 59),
abs(X270, X271, >, 186),     % A
abs(X270, X274, >, 117), abs(X270, X273, >, 293),
abs(X271, X273, >, 436),     % B
abs(X272, X273, >, 293),     % C
abs(X271, X556, >, 490), abs(X271, X292, >, 480),
abs(X271, X629, >, 153), abs(X271, X713, >, 12), abs(X271, X714, >, 168),
abs(X271, X717, >, 21), abs(X271, X719, >, 21), abs(X271, X787, >, 216),
abs(X271, X555, >, 349), abs(X271, X608, >, 10), abs(X271, X630, >, 10),
abs(X271, X274, >, 293), abs(X271, X583, >, 224), abs(X271, X584, >, 53),
abs(X271, X718, >, 158), abs(X271, X720, >, 158).

  no solution.
?-
```

Using HCLP would require the labelling of *all* the constraints after Point 1, above. Using PCSP(MaxCSP) without weights would lead to constraint A being violated, contrary to the wishes of the user. However, the following GOCS query only requires the user to choose a score function and relaxation method, and to label the three constraints. (As discussed previously, we can obtain the same solutions using various other labellings, such as 'strong' for A and 'medium' for B and C, which is in fact what a user would have been most likely to use.) We define the default strength to be 'medium' in order to be able to raise the priority of certain constraints above the average by labelling them 'strong'.

```
?-  set_score_function(max-ucb),
abs(X13, X14, =, 238), abs(X15, X16, =, 238), abs(X81, X82, =, 238),
abs(X83, X84, =, 238), abs(X85, X86, =, 238), abs(X143, X144, =, 238),
abs(X145, X146, =, 238), abs(X215, X216, =, 238), abs(X233, X234, =, 238),
abs(X235, X236, =, 238), abs(X237, X238, =, 238), abs(X261, X262, =, 238),
abs(X263, X264, =, 238), abs(X265, X266, =, 238), abs(X267, X268, =, 238),
abs(X269, X270, =, 238), abs(X271, X272, =, 238), abs(X273, X274, =, 238),
relax all hclp(medium),
abs(X269, X271, >, 59), abs(X269, X272, >, 186), abs(X269, X273, >, 436),
abs(X269, X274, >, 293), abs(X270, X272, >, 59),
medium abs(X270, X271, >, 186),
abs(X270, X274, >, 117), abs(X270, X273, >, 293),
weak abs(X271, X273, >, 436),
very-weak abs(X272, X273, >, 293),
abs(X271, X556, >, 490), abs(X271, X292, >, 480),
abs(X271, X629, >, 153), abs(X271, X713, >, 12), abs(X271, X714, >, 168),
abs(X271, X717, >, 21), abs(X271, X719, >, 21), abs(X271, X787, >, 216),
abs(X271, X555, >, 349), abs(X271, X608, >, 10), abs(X271, X630, >, 10),
abs(X271, X274, >, 293), abs(X271, X583, >, 224), abs(X271, X584, >, 53),
abs(X271, X718, >, 158), abs(X271, X720, >, 158).
```

(solutions as in Section 12.4.1)

In order to achieve the desired answer, the first query was modified with the addition of three strength labels, a score function and a relaxation method. These give the user the flexibility and expressiveness to obtain the solution he wants, without being forced to do unnecessary work and without losing declarativity.

12.5 Conclusions

In HCLP, the necessity of labelling every single constraint leads to many disadvantages. Simple PCSP approaches avoid these problems, but are not sufficiently expressive to capture the user's requirements. More complex PCSP approaches either suffer similar problems to those of HCLP, or require complex distance functions which are also difficult to work with. GOCS allows the use of a simple global score function, with a small number of strength labels to resolve local issues. Thus the client obtains the solution he wants, without a great deal of unnecessary effort. This is the concrete and visible benefit of having the characteristics of an ideal preference system.

Part VI

Discussion

In this part we draw together the conclusions and benefits of the three previous parts, and discuss related issues and further work.

Chapter 13

Conclusions

13.1 Overview

Over-constrained systems arise for different reasons. We believe that two of the most important reasons are related to **compositionality** and **expressiveness**. We also believe that the design of preference systems, i.e. systems for relaxing OCSs, should be driven by the reasons OCSs arise, as well as more standard computer science issues such as efficiency.

We have explored compositionality in the context of two of the standard preference systems, HCLP and PCSP. We have also transformed between them, to show that in principle they have similar expressivity, i.e. a problem expressed in one formalism can always be expressed in the other. However, they both lack expressivity in a slightly different sense, namely in making it easy for the user to specify as much information as he wishes, without needing to express more.

Therefore we have developed our own framework, GOCS, which has HCLP and PCSP as instances and so is at least as expressive as they are, in the first sense. We believe that GOCS is more expressive than both of them, in the second sense, and makes better use of the specifier's time. Furthermore, GOCS is as compositional as any preference system can be.

Thus our two motivations, having been addressed separately (expressiveness and transformation in Part III, composition in Part IV) have been brought together in our discussion of GOCS in Part V. That Part is called 'Integration' which was explained as referring to the integration of HCLP and PCSP approaches to over-constrained systems. It can now also be seen as having integrated our two major concerns, expressiveness and compositionality, in one general framework.

13.2 Our general approach

We tackled the two issues of expressivity and compositionality separately, and then in combination. Part III investigated if HCLP and PCSP were equally expressive, Part IV dealt with compositionality in each of the two theories separately. Part V introduced GOCS which has HCLP and PCSP as special cases and so is at least as expressive as they are, and is also as compositional as a preference system can be.

Throughout, we abstracted away from *programming language* issues, to concentrate on constraint *solving*. We used standard mathematics such as bags and partial orders. We did *not* use non-monotonic logic, although preference systems have non-monotonic aspects. This was intentional: the mathematical and logical tools we used are very standard, and have classical properties. Note the contrast with other work which has used second-order logic.

We have made use of logical issues such as soundness and completeness. We may not have explicitly labelled the discussion with those terms, but had them in mind when showing that BCH/FGH produces the same answers as HCLP, and that problems which can be expressed in HCLP can also be expressed in PCSP and vice versa. We have also discussed 'computer science' issues such as computational complexity.

We have used many simple examples, chosen to illustrate different points of the argument. Most of them placed clarity above realism, but we also treated an important real-world problem in detail.

Therefore the reader has been presented with enough examples to be able to understand our arguments, and enough formality to be able to judge their accuracy.

13.3 Related work

To the best of our knowledge, no-one else has considered precisely the issues addressed in this thesis. However, there are various pieces of work which touch on different areas we have discussed.

As mentioned in Chapter 3, the semantics of HCLP has been considered by Borning and Wilson [83, 84]. Fages et al. [25] and Satoh and Aiba [68, 69] have developed alternative methods of integrating hierarchies with constraints, and consider semantics as well as implementations. Implementations and algorithms embodying standard HCLP semantics have been developed by Freeman-Benson, Sannella et al. [29, 66, 67], and certain optimality properties have been proven by Gangnet and Rosenberg [35].

The work of Menezes, Barahona, and Codognet [60] is closely related to our interest in incremental versions of HCLP. We discuss this at greater length in 'Further work', Section 13.7.

There has been a great deal of work in the general area of PCSP, including implementations and detailed experimental comparisons. See e.g. Wallace and Freuder [80, 81, 82].

There is less theoretical work, except as discussed below. However, there *is* theoretical work on similar themes, such as Schiex on 'possibilistic' CSP [70], and Fargier et al. on 'probabilistic' CSP [26].

The main theoretical work on PCSP is also the closest work we have found to our general concern of integrating HCLP and PCSP in a single framework. Bistarelli, Schiex et al. [4, 5, 71] consider two different mathematical structures (semirings and triangular co-norms) and show that embedding certain operators in these structures produces HCLP[1], whereas operators with other properties give rise to PCSP, and also various other preferential and fuzzy frameworks. One interesting aspect of the co-norm framework [4, 71] is that predictions can be made about the computational complexity of all possible algorithms for, say, PCSP(MaxCSP), just by a consideration of certain mathematical properties of the operators which represent it.

However, each of these theories requires a *different* set of operators, and so this work does *not* integrate the different approaches to relaxing OCSs that can be found in GOCS. In other words, the PCSP-semiring is precisely as expressive as PCSP, and there is no other semiring which contains PCSP as an instance. Furthermore, there is no discussion of compositionality.

Although compositionality is a mathematical property, our interest in it is partly concerned with software engineering issues, which are also the motivation for our discussion of expressivity. Much of the other work mentioned earlier in this section is concerned with the implementation of *existing* theories (HCLP and PCSP) rather than modifying them for semantic, engineering, or other reasons. Therefore, we can characterise without too much inaccuracy the difference between this thesis and related work as follows: whereas others discuss either algorithms (Sannella, Wallace, et al.) or highly mathematical theoretical issues (Bistarelli, Schiex et al.), we are more concerned with 'usability', driven by how over-constrained systems arise in the real-world.

13.4 Our contribution

We have analysed the different reasons why over-constrained systems arise (Section 2.5), and how they relate to the ideal characteristics of the preference systems that we would like to use to resolve them (Chapter 5).

We have developed a general methodology for transforming between HCLP and PCSP (see Part III, Chapter 7). We have shown that strength labels, associated with constraints in HCLP, contain information which is necessary to define the global distance function in PCSP. We have demonstrated that a call to HCLP can be replaced by a call to the transformation predicate followed by applying PCSP to the output, and vice versa.

In Part IV, Chapter 8 we discussed compositionality and its relationship with incrementality. We have shown that non-monotonic systems such as preference systems can

[1]This is not actually shown directly, but it is clear that it is possible, if necessary using methods similar to those we have used in Part III.

never be completely compositional, and have discussed those situations in which it can and cannot. We provided an analogy and contrast with composition in standard logic programming and CLP, which *are* monotonic, as long as the composed queries and predicates are disjoint.

We have developed a compositional variant of HCLP. It is based on bags, and trades increased space against reduced time when compared to standard HCLP. Our variant is defined in terms of simple mathematical constructs, leading to an elegant scheme with clear logical properties. See Chapter 9.

In this context, we have defined a new binary infix relation over bags, called 'guard' (Section 9.1). We have also defined a class of filter functions over bags, and placed them in a non-compositional framework which respects the theory of HCLP (see Section 9.3). We have examined one filter function in detail, and shown that it can be used to calculate the same solutions to a hierarchy as would be obtained by HCLP using the unsatisfied-count-better comparator. Thus we have separated HCLP into its compositional and non-compositional parts. The choice of a filter can be left until after BCH has provided the super-bag of solutions; if there are many, a more discriminating filter function can be used.

Most of our presentation has been expressed in terms of finite domains of integers, but it is clear that our work can be extended to any of the usual constraint domains, such as reals, especially if they can be represented by what we term SCCs (see Section 6.1.3).

We have shown that PCSP is compositional when the best-known distance function MaxCSP is used, and also when modifications of the two other obvious distance functions are used (Chapter 10). Furthermore, we have defined a condition for the applicability of a compositional combining operator for distance functions ('metrics' in standard PCSP terminology).

In Part V, Chapter 11, we developed a general framework for over-constrained systems, called GOCS. It has HCLP and PCSP as special cases, and it allows the use of aspects of both systems at the same time, thus giving the user the flexibility to consider the priority of as many or few constraints as is desired. Furthermore, it embodies the two different approaches to problem relaxation: changing the structure of the *problem* in the style of HCLP, or changing the meaning of *individual constraints* as done by PCSP.

HCLP and PCSP have complementary advantages and disadvantages: HCLP allows fine-grained control but forces the user to label every single constraint, whereas PCSP does not force the user to label all constraints, but does not allow as much detailed control. GOCS can use as much HCLP-like labelling information as it is given, but does not require more. Therefore the user has all the benefits of HCLP and PCSP without the disadvantages.

We presented a detailed example based on a real-world problem, and modelled it in HCLP, PCSP, and GOCS (see Chapter 12). As already mentioned, in HCLP the necessity of labelling every single constraint leads to many disadvantages. Simple PCSP approaches avoid these problems, but are not sufficiently expressive to capture the user's requirements. More complex PCSP approaches either suffer similar problems to those of HCLP, or require complex distance functions which are also difficult to work with. GOCS allows the use of a simple global score function, with a small number of strength labels to resolve local issues. Thus the client obtains the solution he wants, without a great deal of unnecessary effort. This is the concrete and visible benefit of having the characteristics of an ideal preference system.

Compositionality is very important, and yet many other systems for dealing with partial and preferential information do not attempt to limit their non-compositional and non-monotonic aspects, and usually do not even discuss this issue. GOCS is always compositional with respect to the consistent problems derived from the OCS by constraint relaxation, and as long as a certain condition is met it is also compositional with respect to the scores assigned to each possible solution (Section 11.3).

13.5 Publications from this thesis

Some of the work in this thesis has been presented at various conferences and workshops:

- The work in Chapter 9 was presented essentially unchanged at the *Workshop on Over-Constrained Systems* at *CP'95* in Cassis, near Marseilles. Selected papers from this workshop, including the one mentioned here, have been published as a book by Springer. See references [44, 46] for the full citations.

  Some of the background and literature review completed in the context of this thesis have informed a *Brief Overview of Over-Constrained Systems*, which is the introduction to the same book published by Springer. See [45].

- An earlier version of the work in Chapter 9 was presented at *IJCAI'95* in Montreal. See reference [49].

- The transformation presented in Part III is discussed in a paper to be presented at *CP'96* in Boston; for details see [51].

- GOCS, presented in Part V, has been described in a paper to be presented at the Non-Standard Constraints workshop at *ECAI'96* in Budapest. See [50].

  A poster about GOCS is to be presented at JICSLP'96 in Bonn.

Among other publications there is a paper which was presented at a constraints workshop at *ECAI'94* in Amsterdam. The theory it discusses does not form part of this thesis, but is an alternative attempt to address similar issues. See [48].

13.6 Benefits

Our analysis of the characteristics of the ideal preference system should be useful for others wishing to develop their own theories and formalisms. Even if they disagree with some of the detailed points in our list of characteristics, they will benefit from analysing why the over-constrained systems arise in the first place, as a key determinant of the structure of the framework they wish to use to resolve them.

HCLP and PCSP each have advantages when modelling problems, and each have advantages when implementing models and solving them. Using the work presented in Part III, the appropriate paradigm can be used for each of these steps, with a meaning-preserving transformation in between if necessary.

Using our compositional variant of HCLP, we are able to avoid invoking the constraint solver to recalculate solutions from scratch. This scheme allows the exploration of the solution space, and can be implemented in an incremental manner. Furthermore, we have separated HCLP into its compositional and non-compositional parts, which is interesting for all those working with non-monotonic theories. The choice of a filter can be left until after BCH has provided the super-bag of solutions; if there are many, a more discriminating filter function can be used.

In general, constraint satisfaction is of exponential complexity, compared to which guarding and filtering are cheap. In addition to being efficient, the operations we define and use are simple to understand, and calculate the answers we would obtain from HCLP while avoiding its computational expense and complex semantics.

We have defined a compositional combining operator for certain distance functions ('metrics'), and provided a discussion which will be useful when new distance functions are being defined.

The combination of various of the characteristic of an ideal preference system which GOCS possesses leads to efficient use of time by specifiers of systems — they can concentrate on global issues for most of the time, but are still able to fine-tune the model without needing to consider every single local interaction. We demonstrated this with a detailed treatment of a real-world example, modelled in HCLP, PCSP, and GOCS. It also applies to the motivational example described by Richards, quoted in Section 2.5.

We conclude this section by repeating the benefits announced in Section 1.5:

for the theoretician:

- a framework in which to discuss, compare, and contrast HCLP and PCSP simultaneously
- transformations between HCLP and PCSP
- a compositional variant of HCLP, showing at what stage non-monotonic (disorderly) behaviour is introduced
- a proof of compositionality for PCSP distance functions with subset and subset-closed derived orders

for HCLP implementors and users:

- a two-stage implementation of (a variant of) HCLP, the first stage being compositional and incremental. Therefore possibility of more efficient implementations

- ability to delay choice of HCLP comparator until after the first stage, when an idea of the approximate number of solutions will be available

for implementors of preference systems generally:

- an analysis of the reasons why over-constrained systems arise, leading to a list of desirable characteristics for preference systems to have
- transformations between HCLP and PCSP, thus allowing an implementor of one of them to re-use an implementation of the other
- clarification of the two different approaches to relaxation of problems, combined with unification of two important methods for choosing which relaxation is best

for the user:

- the combination of HCLP and PCSP approaches. No need to label every constraint, no need to construct a sophisticated, imperative, distance function
- hence easier and more expressive specification of constraint systems, and quicker and easier debugging and maintenance.

13.7 Further work

In Part III we have shown that problems which can be expressed in HCLP can also be expressed in PCSP, and vice versa. One interesting question which arises from this equivalence of expressiveness is whether the two paradigms are also equivalent in terms of computational expense. Therefore, an investigation of the comparative algorithmic complexity of HCLP and PCSP would have a good fit with the usability issues we have discussed in this thesis.

Our work in Part IV on BCH/FGH, a compositional variant of HCLP, briefly discussed metric comparators. We expect that their inclusion in FGH would be straightforward, but deeper investigation is necessary to confirm this view.

More interesting, as it touches on concerns with a wider scope, is the possibility of formally including constraint deletion and dynamic constraint satisfaction in BCH/FGH. Deletion and dynamicity are difficult to handle in an elegant manner in most formalisms, but should be made easier by the modular nature of our tuple representation of solutions.

A different line of investigation would consider whether BCH/FGH provides a good implementation method for (a subset of) GOCS.

As has been mentioned earlier in this thesis, Menezes, Barahona and Codognet have developed what they call an incremental compiler for HCLP [60]. They use an optimistic implementation strategy which assumes that most optional constraints will not contradict the required constraints. However, as they leave the semantics of HCLP unchanged, i.e. non-compositional, their implementation is not truly incremental in all cases. Specifically, when some constraints *are* contradictory, they have to invoke a specialised backwards phase. The circumstances in which this happens are similar to those in which all preference systems become non-compositional (see Section 8.4 of this thesis). However, contradiction *always* affects the system of Menezes et al., whereas in fact it need only affect the composition of *best* solutions (Section 8.4). This might be due to the 'lazy' evaluation strategy they adopt — the explicit BCH/FGH representation of all solutions by bags of tuples can be considered an 'eager' strategy[2].

The reason for the above seeming digression is to suggest a possible line of further work. Menezes, Barahona and Codognet deal with many issues at the implementation level which we have dealt with explicitly in the semantics of BCH/FGH. Therefore they do not require the user to learn a new theory (although in our opinion BCH/FGH is not difficult to understand for those familiar with HCLP). A comparison of the benefits available by following each of these two approaches would be very interesting — is the support for efficient implementations built in to the semantics of BCH/FGH actually useful? Or is it better to focus on existing theories and try and optimise them at the implementation level. Certainly, the latter reduces the proliferation of languages, and avoids the situation where there are more languages than research groups! Notwithstanding this feeling, we hope that the work on BCH/FGH is useful to implementors of standard HCLP systems.

The last chapter in Part IV provides a proof of compositionality for certain PCSP distance functions, and suggests a larger class for which the same composition operator is appropriate. We would like to investigate other operators and the conditions for them to be compositional as well. It is possible that the work of Bistarelli et al. [5], Schiex et al. [71], and their combined work [4] will provided useful pointers. Those papers do not discuss compositionality, but *do* provide interesting mathematical characterisations of HCLP, PCSP, and other theories.

It would be interesting to place BCH/FGH and Gocs (Part V) in the context of the papers by Bistarelli, Schiex, et al., and to discover if what might be called the software engineering concerns of this thesis have counterparts in the mathematics of [4, 5, 71].

In Part V we gave a description of Gocs, a general framework which can provide the basis for the design of systems to solve OCSs.

Further work suggested by this part includes the extension of the *Rel* operation to take account of metric comparators. Also, it would be interesting to see if the subset-closure property we use as a heuristic condition for the applicability of a compositional combining operator for PCSP distance functions is useful in the context of Gocs score functions.

Construction of a particular concrete implementation is out of the scope of this thesis, but nevertheless we see it as being an activity that should be undertaken in the future when the need arises for a practical system embodying all the benefits of Gocs.

[2]This characterisation of the difference between [60] and BCH/FGH was suggested by an anonymous reviewer of [49], a paper based on an earlier version of Chapter 9, which we presented at IJCAI.

Appendices

Appendix A

Constraint Systems

This appendix presents a simplification to one domain of the many-sorted foundation for constraint systems developed by Jaffar and Lassez in [40].

Let SYMB denote a collection of symbols, i.e. a carrier set of elements or tokens which are assumed to *name* different entities, and so can be treated as *being* different entities. (The symbols in SYMB will be used as the elements of the domain $\mathcal{D}$ over which the constraints range. The difference is that SYMB is considered as a completely uninterpreted collection of symbols, whereas $\mathcal{D}$ is assigned some kind of interpretation.) The *signature* of an n-ary function (predicate, variable) symbol f is a sequence of $n+1$ (respectively n, 1) repetitions of the name of SYMB. Let Σ denote a collection of function symbols and their signatures, and let Π denote a collection of predicate symbols and their signatures. Π is required to contain the equality symbol, which must be present in all constraint systems but which does not need a signature. $\tau(\Sigma)$ denotes the ground terms of the language, and $\tau(\Sigma \cup V)$ denotes terms possibly containing variables drawn from a set of variables V. An *atom* is of the form $p(t_1, \ldots, t_n)$ where p is an n-ary symbol in Π and $t_i \in \tau(\Sigma \cup V)$, for $1 \leq i \leq n$. Thus we use three different sets of symbols for three different purposes: as elements of the domain, as predicate or function symbols, and as variables.

We can define a *structure* $\mathbf{R}(\Sigma, \Pi)$ over the alphabets of function and predicate symbols Σ and Π, consisting of (i) the non-empty set $\mathcal{D}$, (ii) an assignment to each n-ary function $f \in \Sigma$ of a function of type

$$\underbrace{\mathcal{D} \times \mathcal{D} \times \ldots \times \mathcal{D}}_{n} \rightarrow \mathcal{D}$$

and (iii) an assignment to each n-ary $p \in \Pi$, except for the equality symbol, of a function

$$\underbrace{\mathcal{D} \times \mathcal{D} \times \ldots \times \mathcal{D}}_{n} \rightarrow \{\text{TRUE}, \text{FALSE}\}$$

(There is a difference between signatures and these type definitions in the many-sorted case, which is why we distinguish between them. For single sorts, there is no important difference, as can be seen here.)

An *atomic constraint*, or more precisely, an *atomic* (Π, Σ)*-constraint*, is an atom over the alphabets Σ and Π; a constraint is a finite set of atomic constraints, intuitively considered as a conjunction. TRUE and FALSE are distinguished constraints, the former corresponding to the empty constraint.

A $\mathbf{R}(\Pi, \Sigma)$*-valuation* on an expression over Π and Σ is a mapping from each distinct variable in the expression into $\mathcal{D}$. If θ is a valuation for the *term* t, then $t\theta$ denotes the appropriate element of $\mathcal{D}$. If θ is a valuation for the atomic *constraint* c, then $c\theta$ denotes the proposition that $c\theta$ is equivalent to TRUE or FALSE. (More precisely, either $\mathbf{R}(\Pi, \Sigma) \models c\theta$ or $\mathbf{R}(\Pi, \Sigma) \models \neg c\theta$.) If C denotes a set of atomic constraints, then $\mathbf{R}(\Pi, \Sigma) \models C\theta$ means that $\mathbf{R}(\Pi, \Sigma) \models c\theta$ holds for *all* $c \in C$. Whenever this is the case, we can say, following Jaffar and Lassez, that C is $\mathbf{R}(\Pi, \Sigma)$*-solvable* and that θ is a *solution* of C.

Example:
Let us define a constraint system over the real numbers $\mathcal{R}$. Consider the symbols SYMB $= \{1, 1.1, 1.01, \ldots, 2, 3, \ldots\}$. We can use them to name the elements of the domain $\mathcal{D} = \mathcal{R}$. Σ includes binary function symbols $+, \times$, etc, and Π includes $<, \leq, =, >, \geq$. The set of variables V can be defined as all alphanumeric sequences beginning with an upper-case letter. In this case the structure $\mathbf{R}(\Pi, \Sigma)$ consists of (i) the domain $\mathcal{D} = \mathcal{R} =$ SYMB, (ii) an assignment to each n-ary function $f \in \Sigma$ of a function of type $\mathcal{R} \times \mathcal{R} \times \ldots \times \mathcal{R} \rightarrow \mathcal{R}$, and (iii) an assignment to each n-ary $p \in \Pi$ of a function $\mathcal{R} \times \mathcal{R} \times \ldots \times \mathcal{R} \rightarrow \{\text{TRUE}, \text{FALSE}\}$.

Appendix B

Proofs and clarifications

B.1 Clarification of part of Section 7.3.1

In Section 7.3.1.2 we wrote the following:

If we add one pair to the domain of one of the constraints C in a CSP, it is equivalent to adding a set of n-tuples to the domain of that constraint's expanded version C^*, where the other places in the tuple are filled with all possible combinations of elements from the domains of all the other variables.

Continuing with the example in Section 7.3.1.2, let us assume that we have augmented constraint B. This leads to adding a set of n-tuples to B^*; let us call this set of additional tuples R. We can imagine adding a different pair to B which would lead to adding a different set to B^*, say R'. If we add both pairs to B at the same time, then we must add $R \cup R'$ to B^*.

This is true because of the following: if the first pair added to constraint C_{ij} is (α, β) (giving, say, C^1) and the second pair is (γ, δ) (giving C^2), and the doubly-augmented constraint is called C^3,

$$\begin{aligned}
\text{then} \quad C_{ij}^{1*} &= C_{ij}^* \cup \{(v_1, v_2, \ldots, \alpha, \beta, \ldots, v_n) \mid (v_k, k \neq i, k \neq j) \in \mathrm{dom}(X_k)\} \\
C_{ij}^{2*} &= C_{ij}^* \cup \{(v_1, v_2, \ldots, \gamma, \delta, \ldots, v_n) \mid (v_k, k \neq i, k \neq j) \in \mathrm{dom}(X_k)\}
\end{aligned}$$

$$\begin{aligned}
\text{and} \quad C_{ij}^{3*} &= C_{ij}^* \cup \{(v_1, v_2, \ldots, v_i, v_j, \ldots, v_n) \mid (v_i, v_j) \in \{(\alpha, \beta), (\gamma, \delta)\}, \\
&\qquad (v_k, k \neq i, k \neq j) \in \mathrm{dom}(X_k)\}
\end{aligned}$$

then clearly

$$C_{ij}^{3*} = C_{ij}^{1*} \cup C_{ij}^{2*}$$

□

B.2 Proof of a claim in Section 7.3.1

In Section 7.3.1.2 we made a set-theoretic claim about certain 'expansion sets' A^*, B^*, and C^*, and additional expansion sets R and R'. The fact that they are expansion sets is not relevant to the particular claim that we wish to prove. The claim was that the following equality holds:

$$A^* \cap (B^* \cup (R \cup R')) \cap C^* = (A^* \cap (B^* \cup R) \cap C^*) \cup (A^* \cap (B^* \cup R') \cap C^*)$$

This equality can be demonstrated using distributivity and idempotence properties. To reduce the clutter, we will write A instead of A^*, and so on. Furthermore, we will replace $C \cap R$ by CR, and $C \cap R'$ by CR', to make the distribution steps clearer. Finally, we will replace $A \cap B \cap C$ by ABC.

$$\begin{array}{lll}
A \cap (B \cup (R \cup R')) \cap C & & \\
= & (A \cap B \cap C) \cup (A \cap (R \cup R') \cap C) & \{\text{distributivity}\} \\
= & ABC \cup (A \cap (R \cup R') \cap C) & \{\text{local convention}\} \\
= & ABC \cup (A \cap (C \cap (R \cup R'))) & \{\text{assoc. and commut.}\} \\
= & ABC \cup (A \cap ((C \cap R) \cup (C \cap R'))) & \{\text{distributivity}\} \\
= & ABC \cup (A \cap (CR \cup CR')) & \{\text{local convention}\} \\
= & ABC \cup ((A \cap CR) \cup (A \cap CR')) & \{\text{distributivity}\} \\
= & ABC \cup ABC \cup ((A \cap CR) \cup (A \cap CR')) & \{\text{idempotence}\} \\
= & (ABC \cup (A \cap CR)) \cup (ABC \cup (A \cap CR')) & \{\text{assoc. and commut.}\} \\
= & ((A \cap B \cap C) \cup (A \cap (C \cap R))) \cup & \\
  & \quad ((A \cap B \cap C) \cup (A \cap (C \cap R'))) & \{\text{undoing convention}\} \\
= & (A \cap (B \cup R) \cap C) \cup (A \cap (B \cup R') \cap C) & \{\text{distributivity}\}
\end{array}$$

□

Appendix C

Complexity

C.1 Details of the complexity of BCH

For the required constraints, finding the intersection of k sets each of which contains n solutions[1] (*not* necessarily the number of solutions from n constraints) will take time[2] proportional to at most $\mathrm{O}(kn^2)$, with a naïve representation. (In fact, using a sensible data representation it can be done in $\mathrm{O}(kn)$.) The resulting set will have size at most n.

For each level of optional constraints, finding the union of k bags of size n will take $\mathrm{O}(kn)$ in the naïve case, although a more sophisticated representation could reduce this to $\mathrm{O}(k)$ (constant-time append). Guarding the resulting bag of size kn with a required-level set of size n will take $\mathrm{O}(kn^2)$ in the naïve case, or $\mathrm{O}(kn)$ with a sensible choice of representation. So each optional level will take $\mathrm{O}(kn)$, or $\mathrm{O}(kn^2)$ at worst[3].

So if there are l optional levels, the total complexity of BCH will be $\mathrm{O}(lkn^2)$ in the worst case, or $\mathrm{O}(lkn)$ under various sensible assumptions. In practice k and l are likely to be fixed at quite small values, in which case the complexity becomes $\mathrm{O}(n)$ in the number of solutions. Therefore these results are significantly better than constraint solving, which is exponential in general, and often n^4 or worse for practical cases (although those results are concerned with both domain size and number of constraints).

[1] Or n disjoint ranges (or, more generally, SCCs) in the case of continuous domains such as the reals (respectively: the domain of the SCCs).

[2] Measured in terms of basic comparison operations on elements.

[3] Note that if S_0 for a particular hierarchy contains n elements, all the optional levels of that hierarchy can contain at most n distinct elements. Each element may occur more than once, but this is unimportant if we use the alternative representation of bags — $\Lbag a^2 \Rbag$ as opposed to $\Lbag a, a \Rbag$.

Bibliography

[1] Abderrahmane Aggoun and Nicolas Beldiceanu. Overview of the CHIP Compiler System. In Koichi Furukawa, editor, *ICLP'91: Proceedings 8th International Conference on Logic Programming*, pages 775–789. MIT Press, 1991. (A longer version is in [2]).

[2] Frédéric Benhamou and Alain Colmerauer. *Constraint Logic Programming: Selected Research.* MIT Press, Cambridge, MA, 1993.

[3] Françoise Berthier. Solving Financial Decision Problems with CHIP. In J.-L. Le Moigne and P. Bourgine, editors, *CECOIA 2: Proceeedings 2nd Conference on Economics and Artificial Intelligence*, pages 233–238, Paris, June 1990.

[4] Stefano Bistarelli, Hélène Fargier, Ugo Montanari, Francesca Rossi, Thomas Schiex, and Gerard Verfaillie. Semiring-based CSPs and Valued CSPs: Basic Properties and Comparison. In [47].

[5] Stefano Bistarelli, Ugo Montanari, and Francesca Rossi. Constraint Solving over Semirings. In Chris Mellish, editor, *IJCAI'95: Proceedings International Joint Conference on Artificial Intelligence*, Montreal, August 1995.

[6] Alan Borning. The Programming Language Aspects of ThingLab, A Constraint-Oriented Simulation Laboratory. *ACM Transactions on Programming Languages and Systems*, 3(4):353–387, October 1981.

[7] Alan Borning, Bjorn Freeman-Benson, and Molly Wilson. Constraint Hierarchies. *Lisp and Symbolic Computation*, 5:223–270, 1992.

[8] Alan Borning, Michael Maher, Amy Martindale, and Molly Wilson. Constraint Hierarchies and Logic Programming. In Giorgio Levi and Maurizio Martelli, editors, *ICLP'89: Proceedings 6th International Conference on Logic Programming*, pages 149–164, Lisbon, Portugal, June 1989. MIT Press.

[9] Annalisa Bossi, Maurizio Gabbrielli, Giorgio Levi, and Maria Chiara Meo. An Or-Compositional Semantics for Logic Programs. In J.-M. Jacquet, editor, *Constructing Logic Programs*, chapter 10, pages 215–240. Wiley, 1993.

[10] S. Breitinger and H. C. R. Lock. Using Constraint Logic Programming for Industrial Scheduling Problems. In C. Beierle and L. Plümer, editors, *Logic Programming: Formal Methods and Practical Applications*, chapter 9, pages 273–299. Elsevier, 1995.

[11] Gerhard Brewka. Preferred Subtheories: An Extended Logical Framework for Default Reasoning. In *IJCAI'89: Proceedings 11th International Joint Conference on Artificial Intelligence*, pages 1043–1048, August 1989.

[12] Gerhard Brewka, Hans Werner Guesgen, and Joachim Hertzberg. Constraint Relaxation and Nonmonotonic Reasoning. Technical Report TR-92-002, ICSI, Berkeley, 1992.

[13] CELAR: Centre d'Electronique de l'Armement. RLFAP: Radio Link Frequency Assigment Problems. Available from `listserver@saturne.cert.fr` and from `http://web.cs.city.ac.uk/archive/constraints/constraints.html`, September 1994. (Made available in the framework of the European EUCLID project CALMA: Combinatorial Algorithms for Military Applications. See also `http://www.win.tue.nl/win/math/bs/comb_opt/hurkens/calma.html`.).

[14] John Chinneck. Finding Minimal Infeasible Sets of Constraints in Infeasible Mathematical Programs. Technical Report SCE-93-01, Department of Systems and Computer Engineering, Carleton University, Ottawa, 1993.

[15] John Chinneck and Erik Dravnieks. Locating Minimal Infeasible Constraint Sets in Linear Programs. *ORSA Journal on Computing*, 3(2):157–168, Spring 1991.

[16] Philippe Codognet and Daniel Diaz. Boolean Constraint Solving Using clp(FD). In *ILPS'93: Proceedings 3rd International Logic Programming Symposium*, pages 525–539, Vancouver, 1993.

[17] Alain Colmerauer. Prolog II Reference Manual and Theoretical Model. Technical report, Groupe Intelligence Artificielle, Université Aix – Marseille II, October 1982.

[18] Alain Colmerauer. An Introduction to Prolog III. *Communications of the ACM*, 33(7):69–90, July 1990.

[19] B. A. Davey and H. A. Priestley. *Introduction to lattices and order.* Cambridge University Press, 1990.

[20] Rina Dechter and Judea Pearl. Tree Clustering for Constraint Networks. *Artificial Intelligence*, 38:353–366, 1989.

[21] Daniel Dennett. Cognitive Wheels: The Frame Problem of AI. In C. Hookway, editor, *Minds, Machines, and Evolution: Philosophical Studies*, pages 129–151. Cambridge U.P., 1984.

[22] Munich ECRC. Eclipse 3.5 User Manual, 1995.

[23] M. Anton Ertl and Andreas Krall. Optimal Instruction Scheduling Using Constraint Logic Programming. In J. Maluszyński and M. Wirsing, editors, *PLILP'91: Proceedings 3rd International Symposium on Programming Language Implementation and Logic Programming*, LNCS 528, Passau, Germany, August 1991. Springer.

[24] O. Evans. Factory Scheduling Using Finite Domains. In G. Comyn, N. E. Fuchs, and M. J. Ratcliffe, editors, *Logic Programming in Action*, LNCS 636, pages 45–53. Springer, 1992.

[25] François Fages, Julian Fowler, and Thierry Sola. Handling Preferences in Constraint Logic Programming with Relational Optimization. In *PLILP'94*, Madrid, September 1994.

[26] Hélène Fargier and Jérôme Lang. Uncertainty in Constraint Satisfaction Problems: A Probabilistic Approach. In *ESQARU'93*, Grenada, November 1993.

[27] Bjorn Freeman-Benson. Constraint Imperative Programming. Technical Report 91-07-02, University of Washington, Seattle, July 1991. (PhD Dissertation).

[28] Bjorn Freeman-Benson and Alan Borning. The Design and Implementation of Kaleidoscope'90: A Constraint Imperative Programming Language. In *Proceedings IEEE Computer Society International Conference on Computer Languages*, pages 174–180, Oakland, April 1992.

[29] Bjorn Freeman-Benson, Molly Wilson, and Alan Borning. DeltaStar: A General Algorithm for Incremental Satisfaction of Constraint Hierarchies. In *Proceedings 11th IEEE Phoenix Conference on Computers and Communications*, pages 561–568, Scottsdale, Arizona, April 1992.

[30] Eugene Freuder. A Sufficient Condition for Backtrack-Free Search. *Journal of the ACM*, 29(1):24–32, January 1982.

[31] Eugene Freuder. Partial Constraint Satisfaction. In *IJCAI'89: Proceedings 11th International Joint Conference on Artificial Intelligence*, pages 278–283, August 1989.

[32] Eugene Freuder. Exploiting Structure in Constraint Satisfaction Problems. In B. Mayoh, E. Tyugu, and J.Penjaam, editors, *Constraint Programming: Proceedings 1993 NATO ASI Parnu, Estonia*, NATO Advanced Science Institute Series, pages 54–79. Springer, 1994.

[33] Eugene Freuder and Richard Wallace. Partial Constraint Satisfaction. *Artificial Intelligence*, 58:21–70, 1992.

[34] Thom Frühwirth, Alexander Herold, Volker Küchenhoff, Thierry Le Provost, Pierre Lim, Eric Monfroy, and Mark Wallace. Constraint Logic Programming: An Informal Introduction. In G. Comyn, N. E. Fuchs, and M. J. Ratcliffe, editors, *Logic Programming in Action*, LNCS 636, pages 3–35. Springer, 1992. (Also available as Technical Report ECRC-93-5).

[35] Michel Gangnet and Burton Rosenberg. Constraint Programming and Graph Algorithms. In *2nd International Symposium on Artificial Intelligence and Mathematics*, January 1992.

[36] David Gries and Fred Schneider. *A Logical Approach to Discrete Math.* Springer, 1994.

[37] Paul Halmos. *Naive Set Theory.* Springer, 1974.

[38] Nevin Heintze, Spiro Michaylov, and Peter J. Stuckey. CLP($\mathcal{R}$) and Some Electrical Engineering Problems. In Jean-Louis Lassez, editor, *ICLP'87: Proceedings 4th International Conference on Logic Programming*, pages 675–703, Melbourne, Victoria, Australia, May 1987. MIT Press. Also in Journal of Automated Reasoning vol. 9, pages 231–260, October 1992.

[39] Tien Huynh and Catherine Lassez. A CLP($\mathcal{R}$) Options Trading Analysis System. In Robert A. Kowalski and Kenneth A. Bowen, editors, *JICSLP'88: Proceedings 5th International Conference and Symposium on Logic Programming*, pages 59–69, Seattle, Washington, U.S.A., 1988. MIT Press.

[40] Joxan Jaffar and Jean-Louis Lassez. Constraint Logic Programming. Technical Report 86/74, Monash University, Victoria, Australia, June 1986.

[41] Joxan Jaffar and Jean-Louis Lassez. Constraint Logic Programming. In *POPL'87: Proceedings 14th ACM Symposium on Principles of Programming Languages*, pages 111–119, Munich, 1987. ACM.

[42] Joxan Jaffar and Michael Maher. Constraint Logic Programming: A Survey. *Journal of Logic Programming*, 19/20:503–581, 1994.

[43] Joxan Jaffar, Spiro Michayov, Peter Stuckey, and Roland Yap. The CLP($\mathcal{R}$) Language and System. *TOPLAS: ACM Transactions on Programming Languages and Systems*, 14(3):339–395, July 1992.

[44] Michael Jampel. A Compositional Theory of Constraint Hierarchies (Operational Semantics). In Michael Jampel, Eugene Freuder, and Michael Maher, editors, *OCS'95: Workshop on Over-Constrained Systems at CP'95*, Cassis, Marseilles, 18 September 1995.

[45] Michael Jampel. A Brief Overview of Over-Constrained Systems. In [47].

[46] Michael Jampel. A Compositional Theory of Constraint Hierarchies. In [47].

[47] Michael Jampel, Eugene Freuder, and Michael Maher, editors. *Over-Constrained Systems.* LNCS 1106. Springer, August 1996.

[48] Michael Jampel and David Gilbert. Fair Hierarchical Constraint Logic Programming. In Manfred Meyer, editor, *Proceedings ECAI'94 Workshop on Constraint Processing*, Amsterdam, August 1994.

[49] Michael Jampel and Sebastian Hunt. Composition in Hierarchical CLP. In Chris Mellish, editor, *IJCAI'95: Proceedings International Joint Conference on Artificial Intelligence*, Montreal, August 1995.

[50] Michael Jampel, Jean-Marie Jacquet, and David Gilbert. A General Framework for Integrating HCLP and PCSP. In Walter Hower and Zsofi Ruttkay, editors, *ECAI'96 Workshop on Non-Standard Constraints*, Budapest, 1996.

[51] Michael Jampel, Jean-Marie Jacquet, David Gilbert, and Sebastian Hunt. Transformations between HCLP and PCSP. In Eugene Freuder, editor, *CP'96: Proceedings of the 2nd International Conference on Principles and Practice of Constraint Programming*, LNCS, Cambridge, MA, August 1996. Springer.

[52] Donald Knuth. *The Art of Computer Programming.* Addison-Wesley, 1969. Volume 2: Seminumerical Algorithms.

[53] Grzegorz Kondrak and Peter van Beek. A Theoretical Evaluation of Selected Backtracking Algorithms. In Chris Mellish, editor, *IJCAI'95: Proceedings International Joint Conference on Artificial Intelligence*, Montreal, August 1995. (Outstanding Paper award).

[54] John Losee. *A Historical Introduction to the Philosophy of Science.* Oxford University Press, Oxford, third edition, 1993.

[55] Alan Mackworth. Consistency in Networks of Relations. *Artificial Intelligence*, 8(1):99–118, 1977.

[56] Alan Mackworth and Eugene Freuder. The Complexity of Some Polynomial Network Consistency Algorithms for Constraint Satisfaction Problems. *Artificial Intelligence*, 25:65–73, 1985.

[57] Alan Mackworth and Eugene Freuder. The Complexity of Constraint Satisfaction Revisited. *Artificial Intelligence*, 59(1–2):57–62, February 1993. Special Volume on Artificial Intelligence in Perspective.

[58] Michael Maher and Peter Stuckey. Expanding Query Power in Constraint Logic Programming Languages. In Ewing Lusk and Ross Overbeek, editors, *NACLP'89: Proceedings North American Conference on Logic Programming*, pages 20–37, Cleveland, Ohio, October 1989.

[59] Bert Mendelson. *An Introduction to Topology.* Allyn and Bacon, Boston, second edition, 1972.

[60] Francisco Menezes, Pedro Barahona, and Philippe Codognet. An Incremental Hierarchical Constraint Solver. In Paris Kanellakis, Jean-Louis Lassez, and Vijay Saraswat, editors, *PPCP'93: First Workshop on Principles and Practice of Constraint Programming*, Providence RI, 1993.

[61] Pedro Meseguer. Constraint Satisfaction Problems: An Overview. *AICOM*, 2(1):3–17, 1989.

[62] Ugo Montanari. Networks of Constraints: Fundamental Properties and Applications to Picture Processing. *Information Science*, 7(2):95–132, 1974.

[63] Ugo Montanari and Francesca Rossi. Constraint Satisfaction, Constraint Programming and Concurrency. In Paris Kanellakis, Jean-Louis Lassez, and Vijay Saraswat, editors, *PPCP'93: First Workshop on Principles and Practice of Constraint Programming*, Providence RI, 1993.

[64] Michel Page and Mahfoud Boudis. Constraint Programming in Economics. In *AIEM3: Third International Workshop on Artificial Intelligence in Economics and Management*, Portland, Oregon, August 1993. (Model of Water Usage in CLP($\mathcal{R}$)).

[65] Patrick Prosser. Forward Checking with Backmarking. In Manfred Meyer, editor, *Proceedings ECAI'94 Workshop on Constraint Processing*, Amsterdam, August 1994.

[66] Michael Sannella. The SkyBlue Constraint Solver and Its Applications. In Paris Kanellakis, Jean-Louis Lassez, and Vijay Saraswat, editors, *PPCP'93: First Workshop on Principles and Practice of Constraint Programming*, Providence RI, 1993.

[67] Michael Sannella, John Maloney, Bjorn Freeman-Benson, and Alan Borning. Multi-way versus One-way Constraints in User Interfaces: Experience with the DeltaBlue Algorithm. *Software—Practice and Experience*, 23(5):529–566, May 1993.

[68] Ken Satoh. Formalizing Soft Constraints by Interpretation Ordering. In *ECAI'90: Proceedings European Conference on Artifical Intelligence*, 1990.

[69] Ken Satoh and Akira Aiba. Computing Soft Constraints by Hierarchical Constraint Logic Programming. Technical Report TR-610, Institute for New Generation Computer Technology, Tokyo, January 1991. (Also *Journal of Information Processing*, 7, 1993).

[70] Thomas Schiex. Possibilistic Constraint Satisfaction Problems or "How to handle soft constraints". In *8th International Conference on Uncertainty in Artificial Intelligence*, Stanford, July 1992.

[71] Thomas Schiex, Hélène Fargier, and Gerard Verfaillie. Valued Constraint Satisfaction Problems: Hard and Easy Problems. In Chris Mellish, editor, *IJCAI'95:*

Proceedings International Joint Conference on Artificial Intelligence, Montreal, August 1995.

[72] Archana Shankar, David Gilbert, and Michael Jampel. Transient Analysis of Linear Circuits Using Constraint Logic Programming. In Mark Wallace, editor, *PACT'96: Proceedings of the 2nd International Conference on Practical Applications of Constraint Technology*, London, April 1996. Practical Applications Company, Blackpool, UK. (Also available as City University Technical Report TCU/CS/1995/17.).

[73] Barbara Smith and Stuart Grant. Sparse Constraint Graphs and Exceptionally Hard Problems. In Chris Mellish, editor, *IJCAI'95: Proceedings International Joint Conference on Artificial Intelligence*, Montreal, August 1995.

[74] Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, Cambridge, MA, 1989.

[75] Pascal Van Hentenryck. Constraint Logic Programming. *Knowledge Engineering Review*, 6(3):151–194, September 1991.

[76] Pascal Van Hentenryck. Constraint Solving for Combinatorial Search Problems: A Tutorial. In Ugo Montanari and Francesca Rossi, editors, *CP'95: Proceedings of the 1st International Conference on Principles and Practice of Constraint Programming*, pages 564–587. Springer, September 1995.

[77] Clifford Walinsky. CLP(Σ^*): Constraint Logic Programming with Regular Sets. In Giorgio Levi and Maurizio Martelli, editors, *ICLP'89: Proceedings 6th International Conference on Logic Programming*, pages 181–196, Lisbon, Portugal, June 1989. MIT Press.

[78] Mark Wallace. Applying Constraints for Scheduling. In B. Mayoh, E. Tyugu, and J.Penjaam, editors, *Constraint Programming: Proceedings 1993 NATO ASI Parnu, Estonia*, NATO Advanced Science Institute Series, pages 161–180. Springer, 1994.

[79] Mark Wallace, editor. *PACT'95: Proceedings of the 1st Conference on Practical Applications of Constraint Technology*, Paris, April 1995. Practical Applications Company, Blackpool, UK.

[80] Richard Wallace. Directed Arc Consistency Preprocessing as a Strategy for Maximal Constraint Satisfaction. In Manfred Meyer, editor, *Proceedings ECAI'94 Workshop on Constraint Processing*, Amsterdam, August 1994.

[81] Richard Wallace. Cascaded Directed Arc Consistency and No-Good Learning for the Maximal Constraint Satisfaction Problem. In Michael Jampel, Eugene Freuder, and Michael Maher, editors, *Over-Constrained Systems*, LNCS 1106. Springer, August 1996.

[82] Richard Wallace and Eugene Freuder. Conjunctive Width Heuristics for Maximal Constraint Satisfaction. In *AAAI'93: Proceedings of the 11th National Conference on Artificial Intelligence*, Washington, DC, 1993. American Association for Artificial Intelligence.

[83] Molly Wilson. *Hierarchical Constraint Logic Programming.* PhD thesis, University of Washington, Seattle, May 1993. (Also available as University of Washington Technical Report 93-05-01).

[84] Molly Wilson and Alan Borning. Extending Hierarchical Constraint Logic Programming: Nonmonotonicity and Inter-Hierarchy Comparison. In Ewing Lusk and Ross Overbeek, editors, *NACLP'89: Proceedings North American Conference on Logic Programming*, pages 3–19, Cleveland, Ohio, 1989.

[85] Molly Wilson and Alan Borning. Hierarchical Constraint Logic Programming. *Journal of Logic Programming*, 16(3):277–318, July 1993.

[86] Roland H. C. Yap. Restriction Site Mapping in CLP($\mathcal{R}$). In Koichi Furukawa, editor, *ICLP'91: Proceedings 8th International Conference on Logic Programming*, pages 521–534, Paris, June 1991. MIT Press.

[87] Karine Yvon. A Solver for Fair Hierarchical Constraint Logic Programming. BSc Final Year Project, Department of Computer Science, City University, London, June 1995.