Mapping Inferences
Constraint Propagation and Diamond Satisfaction

ILLC Dissertation Series DS-2002-05

INSTITUTE FOR LOGIC, LANGUAGE AND COMPUTATION

For further information about ILLC-publications, please contact

Institute for Logic, Language and Computation
Universiteit van Amsterdam
Plantage Muidergracht 24
1018 TV Amsterdam
phone: +31-20-525 6051
fax: +31-20-525 5206
e-mail: illc@wins.uva.nl
homepage: http://www.illc.uva.nl/

Mapping Inferences
Constraint Propagation and Diamond Satisfaction

Academisch Proefschrift

ter verkrijging van de graad van doctor aan de
Universiteit van Amsterdam
op gezag van de Rector Magnificus
prof.mr. P.F. van der Heijden
ten overstaan van een door het college voor
promoties ingestelde commissie, in het openbaar
te verdedigen in de Aula der Universiteit
op maandag 2 december 2002, te 11.00 uur

door

Rosella Gennari

geboren te Pavia, Italië.

Promotie commissie:

Promotor: Prof.dr. K.R. Apt
Co-promotor: Dr. M. de Rijke
Institute for Logic, Language and Computation
Faculteit der Natuurwetenschappen, Wiskunde en Informatica
Universiteit van Amsterdam
Nederland

Overige leden:
Prof.dr. J.F.A.K. van Benthem
Dr. F. de Boer
Prof.dr. M. van Lambalgen
Prof.dr. J-J.Ch. Meyer
Prof.dr. F. Rossi
Dr. Zs.M. Ruttkay
Dr. L. Torenvliet

Printed and bound by Ipskamp, Enschede.

ISBN: 90–5776–095–9

To my greatest fixpoints

The story is extant, and writ in very choice Italian.
W. Shakespeare, *Hamlet*, Act III, Sc. 2.

Contents

Acknowledgments

Chi cerca trova, seek and you shall find, is what pulled me out of my home town five years ago. Then I landed at the ILLC in Amsterdam, attracted by the Master of Logic program. One year later, on completion of that masters program, I decided to remain in the stimulating and international environment of the ILLC, and thus embarked on a longer program: the Ph.D. In its beginning, I was supposed to do a certain form of belief revision; as a matter of fact, in my second Ph.D. year, I did this by drastically changing my research topic, and thus started working on the problems gathered in this thesis.

From that moment, Krzysztof Apt and Maarten de Rijke took over the role of my Ph.D. advisors, providing me with support and guidance in their respective fields. Krzysztof opened the world of logic programming and constraint satisfaction problems to me, introducing me to both communities. Maarten has an inexhaustible supply of energy — some times I suspect that he owns the recipe of the *Panoramix* (Getafix) potion — and I profited from it: ranging from his expertise in automated reasoning and modal logics, to his help in amending even the "samenvatting" of this thesis. I also wish to thank Carlos Areces, for he helped me in the research transition during my second Ph.D. year, and his secure guidance through modal logics.

Also, Martin Stokhof made my passage to the LIT group of the ILLC easier, and helped me find my way through the unavoidable bureaucracy. Krzysztof also introduced me to CWI, where I found another stimulating working environment. There, I am also indebted to Lieke Schultze, who painstakingly made me utter my first (and last) words in Dutch, and the CWI librarians for their efficiency and patience with my delays.

However, my Ph.D. did not only mean Amsterdam, scientific readings and writing, but also scientific travels, ranging from Washington to Singapore! In particular, among those that I met in these trips, I wish to thank Geertjan Bex, Georg Gottlob, Francesco Scarcello for many interesting discussions and unforgettable social dinners.

Again at the intersection of science and friendship, a number of persons have been a valuable support to me during these Amsterdam years. I list some of them in my preferred, reversed, alphabetical order, just to keep things simple: Renata Wassermann, Riccardo Re, Alessandra Palmigiano, Gwen Kerdiles, Juan Heguiabehere, Caterina Caracciolo, Sebastian Brand, Raffaella Bernardi, Marco Aiello. In particular, I thank Juan and Sebastian for their technical support and comments, that I fully exploited for the tests in Sections 8.5 and 9.5 in this thesis; Marco for organising my visit to the IRST in Trento and the meetings on Temporal and Spatial Reasoning with Vera Stebletsova and Yde Venema. I am also indebted to Yde for some precious clarifications concerning lattice theory. All the members of the Constraint group at CWI, and the LIT group at the ILLC provided useful discussions and recreation hours. I am also grateful to Breanndán Ó Nualláin, who volunteered to amend the whole thesis of my Italian; an offer I could not exploit due my absent-mindedness... But he did a great job, at high speed, with a draft of these acknowledgments!

A number of people are also responsible for many "happy days" in these Amsterdam years. I list some of them in alphabetical order: Katia Bertoli, Elena Brosio, Bart Dirks, Catarina Dutilh Novaes, Anne Frenkel, Jelle Gerbrandy, Giacomino Grassi, Lorenzo Grassi, Roeland Merks, Gabriella Morandi, Monica Naef, Ronald de Wolf.

They say that out of sight means out of mind; but it is also true that every rule has its exception, and my family proved to be a great one; on many occasions, they provided me the serene atmosphere necessary to pursue any research work. Last but not least, Piero Spinnato has his great share in this; he also wrote up the first version of the "samenvatting", made useful comments and remarks to the thesis, and did more, much more than this. As a partial compensation for this, the present thesis is dedicated to my family and Piero.

My final thanks go to the committee members for accepting to read this thesis, and my coauthors for allowing me to reuse the material of our joint papers for some parts of this thesis.

Amsterdam, October 14^{th} 2002 — Rosella Gennari

Chapter 1

Introduction

Writing my Ph.D. thesis consisted of several tasks. Above all it meant trying to recast my research work in a homogeneous setting. As a result of this, two main research fields converge in the present thesis: the field of *constraint satisfaction* (Part I) and that of *automated theorem proving in modal logics* (Part II). Underlying the material in both Parts I and II is a persistent shared concern with knowledge *representation* and *reasoning*, on the chosen representation, in an *efficient* manner. This is the main link between both parts, which is in rationale and methodology rather than subject matter.

The material of this thesis is thus organised into three main parts, as explained below. This introduction is meant to be a guide for the reader through the tree parts of the present thesis. In the remainder of this chapter, I attempt to explain what follows: who could benefit from reading this thesis, and why so; the structure of this thesis, i.e., how its parts and chapters are organised.

1.1 Who Could Benefit from This Thesis

Before starting the actual writing, I was suggested to always bear in mind an idealised, non-expert reader for the thesis, the main reason being that the areas of constraint programming and modal logics, both treated in the present thesis, seem to pertain to two separate communities. As a matter of fact, the phrase "modal logic" never occurs in the two main manuals for constraint programming and satisfaction problems, see [MS98, Tsa93]; a similar fate is shared by the words "constraint satisfaction problem" and "constraint programming" in manuals for modal logics, for instance see [BdRV01].

Part I of the thesis is devoted to constraint satisfaction problems and, in particular, to a theoretical analysis of a class of algorithms, devised to boost the search for solutions to constraint satisfaction problems by inferring constraints. Bearing in mind an idealised reader, non-expert of the constraint literature, I devote part of Chapter 2 to introduce some basic concepts and fix the notation.

Part II of the thesis is concerned with boosting automated theorem proving for basic modal logics. The non-expert readers of modal logics should find enough background material in Chapter 7 to enjoy the remainder of this second part. In particular, the background material could be useful to constraint programmers who would like to see, in Chapter 9, how a constraint solver for basic modal logics can be developed.

As anticipated at the start of this chapter, in both Parts I and II there is a persistent shared concern with knowledge representation and reasoning, on the chosen representation, in an efficient manner. This is the main link between both parts of this thesis; even though Chapter 9 establishes a close connection between algorithms for solving constraints (as explained in Part I) and modal reasoning (as explained in Chapter 7, Part II). In **Part III**, I elaborate on these issues and provide some conclusions.

Thus the material presented in the three parts of this thesis can be of interest to an ensemble of various researchers such as:

1. the programmer who wishes to grab a general and uniform view of the so-called constraint propagation algorithms, many of which are already implemented in all constraint programming environments;

2. in particular, the constraint programmer, who might be interested in what constraint programming can do for modal automated theorem proving, or wish to get familiar with non-standard constraint problems, for which the task is to return an optimal partial solution;

3. the logician or linguist with an interest in automated theorem proving or constraint satisfaction problems;

4. the computer scientist, interested in satisfiability problems;

5. the relational database theorist who wishes to explore the similarities and differences between her/his field and that of constraint satisfaction problems;

6. in general, everybody who is interested in Artificial Intelligence — e.g. Temporal or Spatial Reasoning, Scheduling, Planning, Reasoning under Uncertainty.

In the following subsection, I briefly explain the structure of this thesis: i.e., how its parts and chapters are organised and what their dependencies are. This is meant to facilitate, to each reader, the creation of a personal reading path.

1.2 Structure of This Thesis

Parts

As explained above, this thesis is tripartite. The parts are rather voluminous, so a brief outline is given at the start of each of them.

Part I can be read independently of the remainder of the thesis. In addition, Chapters 7 and 8 in Part II can be read without any prior knowledge of the first part. Instead, Chapter 9 in Part II requires some knowledge of Chapter 2 in Part I and Section 4.2. Part III can be read only after the other two parts.

Furthermore, both Parts I and II begin with a preliminary chapter (Chapters 2 and 7, respectively), where the terminology is fixed and the background material is explained; in these preliminary chapters, a series of examples are proposed to the non-expert reader, and each key definition is accompanied by a motivating toy example, easy to grasp and remember.

Chapters

Instead of providing a detailed overview of the whole thesis in this introduction, I decided to provide each of the remaining chapters with a rather detailed, mainly non-technical introduction. Those introductions are organised in three subsections as explained below: *motivations*; *outline*; *structure*.

An analogous choice holds for the conclusions: in Parts I and II, each chapter is concluded by a synopsis of the presented material, and its connections with the remainder of this thesis; when pertinent, there is also a discussion on the chapter results.

Motivations. At the beginning of each chapter, I go through the effort of motivating why the proposed material could be of interest to the reader. For instance, Chapter 2 pertains to the so-called constraint satisfaction problems; thus that chapter begins by informally introducing the topic and surveying some of its current applications. Similarly, Chapter 7, at the start of Part II (Diamond Satisfaction), introduces the non-expert reader to modal logics by surveying some of the areas where those logics can be traced, and have been successfully applied.

Outline. Each introduction continues with a preliminary account of the main points that are covered in the chapter, in a brief and non-technical manner; ideally, this should give a glimpse of the chapter contents, without the burden of too many technical details at a first reading.

Structure. Finally, at the end of each introduction, the structure of the chapter is illustrated so that the expert reader can easily navigate through this.

Origins of the chapters. Some examples and definitions in Chapter 2 are taken from [Gen98]. Chapter 3 presents a new version of material first presented in [Gen00] and [Gen02]. Chapter 5 is largely based on the following articles: [BGR02], first appeared as [BGR00], both written with S. Bistarelli and F. Rossi; [Gen01a], whose longer version is [Gen01b]. Chapter 8 is based on the joint paper [AGHdR00], written with C. Areces, J. Heguiabehere and M. de Rijke. Chapter 9 presents the results of on-going work with S. Brand and M. de Rijke.

Part I

Constraint Propagation

"Chi bene incomincia è già a metà dell'opera", disse il primo. ("A good start is half the battle", said the first.)
G. Rodari, Vecchi Proverbi, from *Favole al Telefono*, Einaudi, 1971.

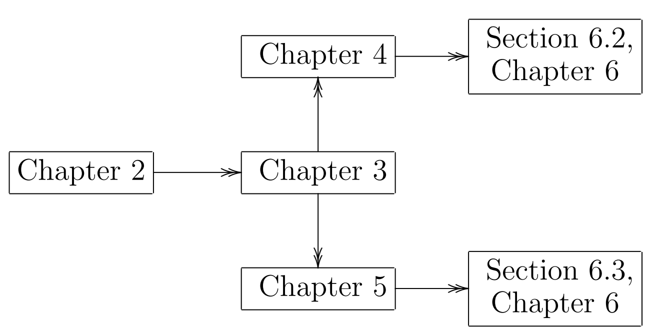
This part concerns itself with constraints satisfaction problems, and mainly a class of efficient approximate algorithms for them.

Constraint satisfaction problems, which constitute the core of this thesis, are introduced in **Chapter 2**.

A theoretical analysis of those algorithms follow in **Chapter 3**. First this theory is applied to a number of efficient algorithms for constraint satisfaction problems in **Chapter 4**. Then "non-standard" constraints are briefly introduced in **Chapter 5**, and the theoretical analysis of Chapter 3 is extended to the correlated algorithms.

Finally, in **Chapter 6**, we collect and characterise the functions that are used in the theoretical analysis of all those algorithms, for both the classical (see Section 6.2) and the non-standard case (see Section 6.3).

The following diagram shows the dependencies of the chapters or sections in this first part.



Chapter 2

Constraints

2.1 Introduction

2.1.1 Motivations

Constraint programming originated from the logic programming community and has become a flourishing programming paradigm, which is implemented in a number of heterogeneous environments, like: ECL^{i}PS^e (see [Agg95]), CONSTRAINT HANDLING RULES (see [Frü98]), OZ (see [Smo95]), CHIP (see [ADH$^+$87]). As stated in [MS98], constraint programming, "based on a strong theoretical foundation, [...] is now becoming the method of choice for modelling many types of optimization problems, in particular, those involving heterogeneous constraints and combinatorial search"; thus "constraint programming has recently been identified by the ACM [Association for Computing Machinery] as one of the strategic directions in computing research".

The central notion of constraint programming is that of a so-called constraint satisfaction problem, which in a nutshell consists of a finite collection of constraint relations over a finite number of domains.

The only task left to the constraint programmer is to formulate a given problem as a constraint satisfaction problem. Then the problem is "solved" by the constraint programming system, by means of general or domain specific methods. Here "solving" can mean finding values, one from each domain, that "satisfy" the problem constraints, or the optimal values, with respect to some criteria, for the problem constraints. The latter task has grown into an independent area, and we shall introduce and study it separately in Chapter 5.

Problems that can be expressed in a natural way by means of constraints are, for instance, those that lack efficient solving algorithms; like the MAP COLOURABILITY PROBLEM or the 3-SAT problem and, in general, combinatorial problems that are computationally intractable. Yet constraint programming has grown further than this and it can be traced in areas like Temporal Reasoning, Scheduling, Planning and Spatial Reasoning, see [DAA95].

For most of the aforementioned problems, the search for a solution usually involves a number of fruitless computations before any solution can be generated. Nowadays, constraint programming incorporates a variety of methods that are conceived for avoiding some fruitless explorations of the solution search space. In Chapter 4, we survey a number of these methods.

Moreover, the relational aspect of constraints often allows the programmer to re-use the same program for different purposes. In contrast, traditional programming languages do not usually provide support for specifying relations among the various components of programs. It is then the programmer who bears the burden to specify and maintain those relations in a dynamic situation.

The use of relations for programming is also popular in the database community, see the database relational model as for instance in [AHV95, Ull80, UW97]. Indeed, there are similarities between the two fields, as outlined in Chapter 6. However, in a relational database, relations are usually extensionally characterised, namely as tables; the task here is to efficiently query the database and retrieve all solutions/answers to the query. On the other hand, constraints can be assigned intensionally, like linear equations over real numbers are; the task is to satisfy all the problem constraints, and the solving algorithms are designed to cleverly produce a solution to the problem.

2.1.2 Outline and Structure

In this chapter we introduce the so-called constraint satisfaction problems, and the basic notions necessary for the comprehension of the remainder of Part I.

Section 2.2 presents the core notions of constraints, constraint satisfaction problems and their solutions; the terminology that we adopt usually follows the standard one; whenever we introduce some new conventions or terms, we signal these and explain their use.

A number of motivating examples are proposed in Section 2.3: these range from combinatorial problems (see Subsection 2.3.1) to problems that arise in areas like Image Understanding (see Subsection 2.3.3), Spatial and Temporal Reasoning, Planning (see Subsection 2.3.4). Some of those examples return in the remainder of this thesis: precisely when we explain the algorithms of Chapter 4, and introduce non-classical constraints in Chapter 5.

The algorithms that occur in the remainder of this thesis (Chapter 4, Sections 5.4 and 5.5) are explained via transformations of problems, as presented in Section 2.4, and via basic operations and orderings on problems, as illustrated in Section 2.5.

2.2 Constraint Problems and Their Solutions

In a constraint programming environment, problems are cast in terms of variables, domains and constraints: each variable is associated with a domain of interpretation, from which it takes its possible values; constraints on variables restrict the allowed domain values for variables. We define variables, domains, constraints and, finally, constraint satisfaction problems precisely in Subsection 2.2.1.

The notion of a solution to such problems is explained in Subsection 2.2.2. Intuitively, a solution to a problem assigns values, to each domain variable, according to the constraints imposed by the problem. For instance: we need to schedule a series of meetings on a certain date, so we open our agenda on that date and start filling in columns, corresponding to different day hours; these are the variables of our problem, and the events that can take place in those hours are the variable values. A constraint for this problem could be that there are no meetings that take place at the same hour.

In conclusion, the choice of variables and domains *defines* the search space for solutions; in the above case, the set of meetings that we want to schedule. Constraints *characterise the structure* of the solution search space; e.g., the fact that there cannot be overlapping meetings.

2.2.1 What Constraint Satisfaction Problems Are

Variables

To define a constraint satisfaction problem, we need a finite sequence of n distinct variables, say $r := \langle x_1, \ldots, x_n \rangle$. Consider a non-empty sequence $s := \langle x_{i_1}, \ldots, x_{i_m} \rangle$ of r variables such that, either $i_j < i_{j+1}$ for each $1 \leq j < m$, or $m = 1$; then s is a *scheme* of r of *length* m.

We shall usually denote the scheme of variables of a constraint satisfaction problem by X; then r, s, t, or the same with subscripts or superscripts, will usually denote schemes of X.

Besides, in this thesis, we shall write a scheme $s := \langle x_{i_1}, \ldots, x_{i_m} \rangle$ as $s := x_{i_1}, \ldots, x_{i_m}$ to avoid an overload of notation. Notice that, if a scheme r contains n variables, then there are $2^n - 1$ possible schemes of r.

Example **2.2.1.** The scheme $s := x_1, x_2, x_3$ gives rise to 7 schemes: i.e., $s_1 := x_1$, $s_2 := x_2$, $s_3 := x_3$; $s_4 := x_1, x_2$, $s_5 := x_1, x_3$, $s_6 := x_2, x_3$; s itself.

There are two useful operations on schemes: if s and t are two different schemes of r, then $s \cup t$ denotes the scheme of r on the variables in s and t; we call this operation the *join* of s and t. Vice versa, if t is a scheme of s, different from s, then $s - t$ is the scheme on the variables of s minus those in t, and we shall refer to it as the *projection of t out of s*.

EXAMPLE **2.2.2.** Let us consider $r := x_1, x_2, x_3$, and its schemes $s := x_1, x_2$, $t := x_2, x_3$. Then the join of s and t is r itself; the projection of t out of s is x_1.

Domains

Each variable x_i in a scheme $r := x_1, \ldots, x_n$ is interpreted over a domain, usually denoted with D_i. The Cartesian product of all variable domains

$$D := D_1 \times \cdots \times D_n$$

is called the *domain* of r, whereas the set of pairs $\langle D_i, x_i \rangle$ is denoted by

$$\boldsymbol{D} := \{\langle D_1, x_1 \rangle, \ldots, \langle D_n, x_n \rangle\} . \tag{DS}$$

The set $\boldsymbol{D}$ in (DS) is referred to as the *domain set* of r. We shall usually adopt a more compact notation and write a domain set in the form

$$\boldsymbol{D} = D_1, \ldots, D_n$$

every time meaning (DS) as above.

Given a scheme $s := x_{i_1}, \ldots, x_{i_m}$ of r, we denote the Cartesian product

$$D_{i_1} \times \cdots \times D_{i_m}$$

with $D[s]$. Notice that, if s is a singleton as x_i, then $D_i = D[x_i]$. Similarly, if $d_i \in D_i$ for every $D_i \in D$, let d be the tuple

$$(d_1, \ldots, d_n).$$

Then $d[s]$ denotes the tuple $(d_{i_1}, \ldots, d_{i_m})$.

In case $D[s]$ and $d[s]$ reduce to singletons, we shall blur the above distinctions, that is we shall feel free to write D_i instead of $D[x_i]$, as well as d_i in place of $d[x_i]$. Also, if s is the singleton of x_i, then the tuple $d[s]$ corresponds to d_i.

In a number of cases that we investigate in the present thesis, domains are finite. However, there are situations in which domains are allowed to be infinite, as in the case of linear inequalities over real numbers; see the following example.

EXAMPLE **2.2.3.** Given a scheme $r := x_1, x_2, x_3$, let the variables of r range over real numbers. Precisely, let D_1 be the closed interval $[0, 1]$, whereas D_2 and D_3 are equal to $[1, 3]$. Then D is the Cartesian product $D_1 \times D_2 \times D_3$. While $\boldsymbol{D}$ is the following set:

$$\{\langle [0, 1], x_1 \rangle, \ \langle [1, 3], x_2 \rangle, \ \langle [1, 3], x_3 \rangle\} .$$

Notice that, being D_2 equal to D_3, if we cast $\boldsymbol{D}$ as a set of domains, we would lose the association between variables and their domains.

Constraints

In this thesis, constraints are usually described as relations; so each constraint is associated to a scheme of variables, like in the database relational model.

Formally, given a scheme r and a domain D over r, let $s := \{x_{i_1}, \ldots, x_{i_m}\}$ be a scheme of r. Then a *constraint over* s, written as $C(s)$, is a subset of the Cartesian product $D[s]$. If k is the length of s, then $C(s)$ is a k*-ary constraint*.

The above definition of constraint has the advantage of being easily generalised to non-standard constraints, as presented in Chapter 5. Moreover, it naturally captures the intuitive meaning of constraints: i.e., that of relating variable values and thereby restricting possible assignments, as we shall make precise in Subsection 2.2.2.

In the literature, constraints as relations can be represented extensionally, for instance as tables, or intensionally. The following is an example of the latter representation.

Example **2.2.4.** Consider the following system of equations over real numbers:

$$\begin{cases} 2x_1 + x_2 + 2 & = 3x_4 - 1 \\ x_2 & = 3x_1 \\ x_1 + x_2 & = 7x_3 \\ x_4 & = x_3 + x_2 + 1 \end{cases}$$

Variables are the unknown x_i, for $i = 1, \ldots, 4$. Domains are equal, for instance, to $\mathbb{R}$. Each equality in the above system is regarded as a constraint on the related variables; for example, the first equality is interpreted as a constraint $C(x_1, x_2, x_4)$ on the scheme x_1, x_2, x_4.

In the case of Example 2.2.4, an extensional representation of constraints would be impossible: it would have to list all the allowed triples of real numbers.

We conclude this part with some definitions. Consider k different constraints on k schemes of r, say $C_i(s_i)$ for $i = 1, \ldots, k$. Then the set of pairs

$$\boldsymbol{C} := \{\langle C(s_1), s_1\rangle, \ldots, \langle C(s_k), s_k\rangle\} \tag{CS}$$

is referred to as a *constraint set* of r. If the number of involved constraints is of relevance, then we call the above set a k*-constraint set*. As in the case of domain sets, we shall usually adopt a more compact notation and denote constraint sets as (CS) in the following way:

$$\boldsymbol{C} := C(s_1), \ldots, C(s_k).$$

We have now all the ingredients, namely variables, domain and constraint sets, to formalise the notion of constraint satisfaction problem as below.

Constraint Satisfaction Problems

A *constraint satisfaction problem*, briefly CSP, is a tuple $P := \langle X, \boldsymbol{D}, \boldsymbol{C} \rangle$ defined as follows:

1. X is a scheme;

2. $\boldsymbol{D}$ is a domain set of X;

3. $\boldsymbol{C}$ is a constraint set of X.

Whenever we need to be more specific and highlight the scheme X of a CSP, we shall talk of a *CSP over* X. So, by writing the *CSPs over* X, we refer to all the CSPs that have the same scheme X.

2.2.2 Global Satisfaction

Assignments

Given a scheme X of n variables and $D := D_1 \times \cdots \times D_n$, a domain over X, a tuple $d \in D$ is a *total assignment* or *total instantiation*. The name is motivated by the fact that each tuple $d \in D$ gives rise to a function that assigns a single value in D_i, namely $d\,[i]$, to each variable $x_i \in X$, thereby instantiating it; and vice versa as well. We shall blur this distinction and consider assignments as functions whenever convenient.

Given a scheme s of X, a total assignment for s, namely a tuple of $D\,[s]$, is an s *assignment* over D. If the scheme s of X is not relevant, we generically talk of *assignment*.

Indeed, if r is a scheme of s, every s assignment d gives rise to an r assignment, namely $d\,[r]$. In the literature, this is usually referred to as *assignment restriction*, since assignments are usually defined as functions. Vice versa, every r assignment e can be extended to an s assignment, possibly more than one, in an arbitrary way. So, if $e \in D\,[r]$, then $d \in D\,[s]$ is an r *extension* of e *to* s iff $d\,[r] = e$.

Example **2.2.5.** The system of equations in Example 2.2.4 gives rise to a CSP. An example of a total assignment is the tuple $d := (0, 1, 2, 3)$. If s is the scheme x_1, x_2, x_3, then $d\,[s]$ is the s assignment $(0, 1, 2)$.

Consistency and solutions

Let us consider a CSP $P := \langle X, \boldsymbol{D}, \boldsymbol{C} \rangle$, and let $C(s)$ be a constraint on a scheme s of P. Suppose that the scheme t extends s. If a t assignment d over D is such that $d\,[s]$ belongs to $C(s)$, then we say that d *satisfies* or is *consistent with* the constraint $C(s)$.

Informally, a solution to the CSP $P := \langle X, \boldsymbol{D}, \boldsymbol{C} \rangle$ is a total assignment of which each projection, over a scheme s of X, satisfies every constraint of P over

s. Formally, consider a total assignment d over D; if, for every $C(s)$ over s of P, the tuple $d\,[s]$ satisfies $C(s)$, then we say that d *satisfies* or is *consistent with* P. The assignment d is commonly referred to as a *solution to* P.

Thus a CSP P is *consistent* or *satisfiable* iff there exists a total assignment that satisfies it. The subset $Sol(P)$ of the domain set D denotes the *set of solutions to* P.

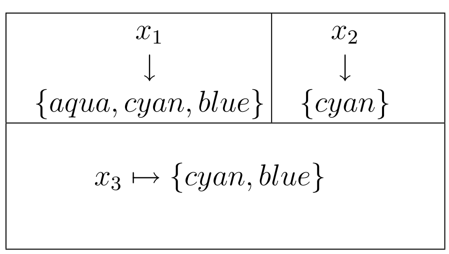
Example **2.2.6.** Let us consider the systems of equations in Example 2.2.4, that gives rise to a CSP. The total assignment $d := (0, 0, 0, 0)$ is not a solution because of the fourth equation. If s is the scheme x_1, x_2, x_3, then $d\,[s]$ is an s assignment; this can be extended to an assignment that is a solution to the system, namely $e := (0, 0, 0, 1)$. In fact, it is not difficult to check that the assignment e satisfies all equations/constraints of the problem.

2.3 Examples

We started this chapter by claiming that CSPs are ubiquitous. In the remainder of this section, we present some examples to support our claim.

2.3.1 Map Colourability

The first example we discuss is combinatorial. A planar map, like the one sketched below, can be represented by means of a graph $G := \langle V, E\rangle$ and a finite set D_i of colours, one for each vertex x_i in V. Hence the map colourability problem (see [GJ79]) consists in colouring the graph vertices so that no two adjacent vertices are painted with the same colour.



The instance of the map colourability problem which corresponds to the above map is the following:

1. the set of vertices is $V := \{x_1, x_2, x_3\}$;
2. the set of colours are, respectively: $D_1 := \{aqua,\ blue,\ cyan\}$ for x_1; then $D_1 := \{cyan\}$ for x_2; finally $D_3 := \{blue,\ cyan\}$ for x_3;
3. the only arcs in the graph are (x_1, x_2), (x_2, x_3), (x_1, x_3).

The encoding of this as a CSP is straightforward:

1. variables correspond to the vertices in V;
2. each colour set corresponds to a variable domain;
3. constraints are posted between those variables that are connected by an arc in the graph: i.e., $C(x_i, x_j)$ states that x_i and x_j have different colours, for $1 \leq i < j \leq 3$.

A solution to the MAP COLOURABILITY PROBLEM is an assignment of colours to all the variables that satisfy all the given constraints. In the case of the depicted map above, a solution is as follows: x_1 is *aqua*, x_2 is *cyan* and x_3 is *blue*.

2.3.2 Satisfiability Problems

In Chapter 9, we shall also deal with *propositional satisfiability*. A conjunction ϕ of propositional disjunctions of the form

$$p \vee \neg q \vee r \tag{2.1}$$

can be encoded as a CSP as follows (other encodings have been devised in the literature, see for instance [Wal00]):

- variables are proposition letters, such as p, q and r;
- domains only contain the Boolean values 0 (false) and 1 (true);
- constraints are posted between those variables/letters that occur in the same disjunct. For instance, (2.1) corresponds to a constraint on the variables p, q and r, that only rules out the tuple $(0, 1, 0)$ from the set $\{0, 1\}^3$.

A solution is thus an assignment, to all the variables/letters in the formula ϕ, that satisfies ϕ.

2.3.3 Image Understanding

Computer vision is an important AI area, that arose as part of robotics. Nowadays, its applications have moved beyond robotics; for instance, we encounter computer vision methods in the interpretation of satellite data.

Computer vision involves image analysis and understanding. A prototypical problem in this sense (see [DAA95]) is the *scene labelling problem*. The task is to reconstruct objects in a three dimensional scene by means of their bidimensional representations. This problem was first encoded as a CSP by Waltz; see [Wal75] as quoted in [DAA95]. The original problem is transformed from one of labelling lines to one of labelling junctions between lines. Waltz's procedure relies on two physical constraints to make the problem tractable:

- a number of combinations of line labellings at a junction are not physically realisable;
- each line connects two junctions; thereby the labellings at the two junctions must both assign the same label to the line.

Thus the goal is to find a physically consistent set of labellings for junctions. The procedure by Waltz dramatically reduces the search space size by means of the algorithms that will be presented in Section 4.2. We invite the reader to consult [DAA95] for a more detailed account of this.

Here we focus on another typical problem of image understanding, successfully tackled as a CSP in [Aie02]: document understanding. In Aiello's thesis (*ib.*), given a set of labellings that identify the basic components of a document, a CSP solver is used to reconstruct a reading order for the document. For instance, the labelled layout of Figure 2.1 is encoded as the following CSP:

1. each rectangular box in the drawing is associated with one variable;
2. variable domains collect pairs of real numbers, interpreting the upper-left corner and the lower-right corner of a rectangular box;
3. constraints are expressed through the bidimensional Allen relations. For instance, in Figure 2.1, x_1 and x_2 are related as follows: $x_1 \, equals_Y \, x_2$ states that the projections of the two documents on the Y axis coincide; whereas $x_1 \, precedes_X \, x_2$ states that the horizontal component of x_1 precedes that of x_2.

A solution to the above CSP is a reading order that satisfies all the Allen constraints of the problem.

2.3.4 Temporal Reasoning

Another example we consider pertains to Qualitative Temporal Reasoning and Scheduling, see [Gen98]. Suppose to have a series of tasks, each one taking a continuous interval of time, and to be all accomplished the same day, as in the following simple plot.

> Arie fidgets in his pocket, searching for his small agenda. Much to his surprise, he realises to have lost it. All he can vaguely remember is to have an important meeting that day in Amsterdam, in the meeting room B2.24. He knows that he should meet Bert a long time before Barbara starts her meeting with Kees in the B2.24 room. Arie remembers Cees talking about his own meeting with Dick in that room, and that this should be over by the time Dora meets Alfons there. Besides, the meeting of Barbara and Kees should be before the meeting of Dora and Alfons in the B2.24 room. Arie perfectly knows that

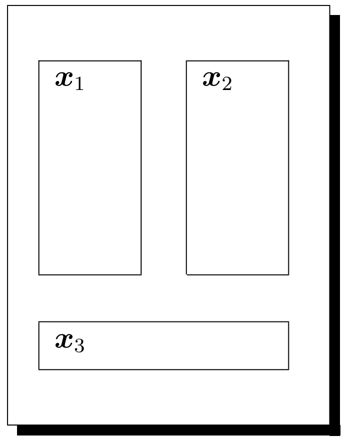


Figure 2.1: A labelled layout of a document page.

> all these events take place sequentially in the B2.24 room, with an interval of at least ten minutes in between any two meetings. Arie also remembers that everybody has precisely one meeting that day. Are Arie's memories consistent?

This story can be encoded as a CSP with four variables, one for each meeting in the story. So, let us state the following: x_1 means "Arie and Bert's meeting"; x_2 is "Barbara and Kees' meeting"; x_3 denotes "Cees and Dick's meeting"; x_4 represents "Dora and Alfons' meeting". Each variable is interpreted over the real line. Reading the story, we encounter four constraints between those variables; the constraints are formalised through the Allen relation *precedes* and its inverse, $follows$:

1. $C(x_1, x_2)$ is $x_1\, precedes\, x_2$; this states that Arie and Bert's meeting takes place before Barbara and Kees' meeting;

2. $C(x_3, x_4)$ is $x_3\, precedes\, x_4$; this encodes the fact that Cees and Dick's meeting takes place before Dora and Alfons' meeting;

3. $C(x_2, x_4)$ is $x_2\, precedes\, x_4$; this translates the fact that Barbara and Kees' meeting takes place before Dora and Alfons' meeting;

4. $C(x_1, x_4)$ is $x_1\, precedes \vee follows\, x_4$; this translates the fact that Arie and Bert's meeting has to take place before or after Dora and Alfons' meeting.

Then a solution is found when all the above constraints are satisfied.

A more challenging application for constraint programming, involving Temporal Reasoning, is provided by planning. We refer the reader to [KvB97] for a clear introduction to planning as CSP, and a comparison of the constraint programming planner CPlanner with other state-of-the-art planners.

2.4 Equivalent Problems

A number of CSP algorithms, like those that we present in Chapter 2, are better understood if we assume that the input problem has at most one constraint per scheme, or even precisely one per scheme. These two properties are obtained, respectively, by means of two procedures, named 'normalisation' and 'completion'. We describe those procedures in this subsection, and show that neither of them add or remove solutions with respect to the original problem; thus they preserve equivalence, which is precisely defined as follows — we remind that $Sol(P)$ denotes the set of all solutions to P, see p. 13.

DEFINITION **2.4.1.** Consider two CSPs $P := \langle X, \boldsymbol{D}, \boldsymbol{C} \rangle$ and $P' := \langle X, \boldsymbol{D}', \boldsymbol{C}' \rangle$ on the same scheme X. Then P and P' are *equivalent CSPs* if they have the same solution set: i.e., $Sol(P) = Sol(P')$.

EXAMPLE **2.4.2.** Consider the CSP in Example 2.2.4 and the CSP with the same scheme and domains, that has as constraints the following equations over real numbers:

$$\begin{cases} 5x_1 + 2 & = 3x_4 - 1 \\ x_2 & = 3x_1 \\ 4x_1 & = 7x_3 \\ x_4 & = x_3 + 3x_1 + 1 \end{cases}$$

The former CSP and the latter are equivalent, since they have the same solutions over real numbers. In fact, the latter CSP is obtained from the former by replacing the occurrences of x_2 by $3x_1$ in the first, third and fourth equations; this is an equivalence-preserving transformation, which is taught in high schools.

The above definition can be easily extended to compare CSPs on different schemes; yet, we shall not be in need of such extension in the present thesis.

2.4.1 Normalisation

If we transform a CSP so that it has at most one constraint on each scheme of variables, we obtain a normal form for it, as explained in the following definition.

DEFINITION **2.4.3.** Consider a CSP $P := \langle X, \boldsymbol{D}, \boldsymbol{C} \rangle$. The *normalisation of* P is a CSP $P' := \langle X, \boldsymbol{D}, \boldsymbol{C}' \rangle$ that shares the variable scheme X and domain set $\boldsymbol{D}$ with P. Then, for each scheme s of X such that there exist $k \geq 1$ constraints $C_1(s), \ldots, C_k(s)$ on s in P,

- there exists precisely one constraint $C'(s)$ in $\boldsymbol{C}$,
- and $C'(s)$ is the constraint on s that is equal to $\bigcap_{i=1}^{k} C_i(s)$.

The problem P is *normalised* if it satisfies the above requirements.

The above definition is consistent due to the following fact.

FACT **2.4.4.** *Every CSP has precisely one normalisation.* □

EXAMPLE **2.4.5.** The Temporal Reasoning problem in Subsection 2.3.4 is normalised, since there is at most one constraint on each scheme.

Notice that, in the above example, the original CSP has only binary constraints and so does its normalisation. Indeed, intersection does not modify the arity of constraints, hence the following fact.

FACT **2.4.6.** *A CSP has a k-ary constraint iff its normalisation does.* □

The above fact is trivial but not to overlook; in fact, some CSP algorithms only deal with constraints of a fixed arity, like binary ones. Thus the above fact ensures that normalisation does not modify the nature of a CSP, so to speak; passing to the normalisation is just to simplify the description of the algorithms.

Solving a CSP also means finding a total instantiation that is consistent with every constraint of the problem. Therefore, the following question is of primary concern: do we add or remove any solution by normalising a CSP? The answer is clearly negative, and its simple proof is outlined in the following lemma; it relies on the fact that an instantiation is consistent with a CSP it if satisfies all its constraints.

LEMMA **2.4.7.** *A CSP P and its normalisation are equivalent CSPs.*

PROOF. Let d be a tuple in D that is consistent with all the constraints $C(s)$ of P. Then consider a scheme s of X and analyse the following three cases.

1. If there is only one constraint $C(s)$ in P, then the same constraint on s and no other one is in the normalisation of P; hence d is consistent with the constraint $C(s)$ of the normalisation of P.

2. If there is more than one constraint on s in P, then $d\,[s]$ has to be consistent with all of them, thereby with their intersection as well.

3. Finally, if there are no constraints in P on s, then there are no constraints in the normalisation of P either.

The other implication follows by inspecting the same three cases and assuming that d is consistent with all the constraints of the normalisation of P. □

2.4.2 Completions

Several constraint propagation algorithms can be better described by assuming a stronger working hypothesis than normalisation: i.e., that the input problem has precisely one constraint on each scheme of variables, so that the problem is complete in the following sense.

DEFINITION **2.4.8.** Consider a CSP P and its normalisation $P^N := \langle X, \boldsymbol{D}, \boldsymbol{C} \rangle$. The *completion of* P is the problem $\bar{P} := \langle X, \boldsymbol{D}, \bar{\boldsymbol{C}} \rangle$ whose constraint set $\bar{\boldsymbol{C}}$ enjoys the following properties:

- for each scheme s of X, if $C(s)$ belongs to $\boldsymbol{C}$ of P^N, then it is also the only constraint on s in $\bar{\boldsymbol{C}}$;
- if P^N has no constraints on s, then $\bar{\boldsymbol{C}}$ has precisely one constraint $\bar{C}(s)$ on s, that is $D[s]$.

We say that a CSP P is *complete* iff $P = \bar{P}$.

The completion of a CSP is obtained by normalising the problem, and adding the necessary constraints as in the above definition.

While normalisation does not alter the nature of a CSP (i.e. if it is binary its normalisation is binary too), its completion instead modifies it. For instance, if P is a binary CSP, the choice of $\bar{P}$ is by no means optimal: it may have too many constraints with respect to those in P. Some CSP algorithms (see Subsection 4.4.2), require the input problem to be complete but only up to constraints of arity at most k, so to speak. Hence, we refine the above definition as follows.

DEFINITION **2.4.9.** Consider a CSP $P := \langle X, \boldsymbol{D}, \boldsymbol{C} \rangle$ on $n > 0$ variables, and let k be a natural number, not greater than n and different from 0.

- The CSP $\bar{P}_k$ is the k *completion of* P if the constraints of $\bar{P}_k$ are all the k-ary constraints of $\bar{P}$. The problem P is k *complete* iff $P = \bar{P}_k$.
- While $\bar{P}^s_k$ is the *strong* k *completion of* P iff the constraints of $\bar{P}^s_k$ are all the i-ary constraints of $\bar{P}$ for every $0 < i \leq k$. The problem P is *strongly* k *complete* iff $P = \bar{P}^s_k$.

EXAMPLE **2.4.10.** The Temporal Reasoning problem in Subsection 2.3.4 is 2 complete, since it has precisely one binary constraint on each scheme of x_1, x_2, x_3, x_4 of length 2.

The above definitions of completions are consistent due to the following fact.

FACT **2.4.11.**
(i). Every CSP $P := \langle X, \boldsymbol{D}, \boldsymbol{C}\rangle$ on n variables has precisely one k completion and one strong k completion, for every $k \leq n$.
(ii). The completion of $P := \langle X, \boldsymbol{D}, \boldsymbol{C}\rangle$ is the strong n completion of P, for n equal to the cardinality of X. □

As in the case of normalisation, the completion of a CSP P is equivalent to P. The proof of the following lemma is analogous to that of Lemma 2.4.7.

LEMMA **2.4.12.**
(i). A CSP P and its k completion are equivalent problems, for every $k \geq 0$ that is not greater than the number of variables in P.
(ii). A CSP P and its strong k completion are equivalent problems, for every $k \geq 0$ that is not greater than the number of variables in P.
(iii). A CSP P and its completion $\bar{P}$ are equivalent problems. □

2.5 Combining and Comparing Problems

2.5.1 Basic Operations

Most algorithms for solving or simplifying CSPs (see Chapter 4) can be described by means of functions and their iterations. These functions are obtained by means of some basic functions on relations, common to most of those algorithms, and in addition some specific ones. We introduce the basic functions as below, since this will allow us to obtain a more general and compact notation to describe all the algorithms presented in Chapter 4.

Consider a domain D over $X := x_1, \ldots, x_n$ and a scheme s of X. Given $B \subseteq D\,[s]$ and a scheme $t := x_{j_1}, \ldots, x_{j_k}$ of s, the *projection* of B over t is defined as follows:

$$\Pi_t^s(B) := B_{j_1} \times \cdots \times B_{j_k}.$$

When the reference to s is clear, we shall write Π_t instead of Π_t^s. So, for instance, $D\,[t]$ is equal to $\Pi_t(D)$ for every scheme t of X; in particular $D = \Pi_X(D)$. We extend the operation on tuples $d \in D\,[s]$ in the obvious way, and call $\Pi_t(d)$ the *projection of the tuple d on t*, for t a scheme of s.

We shall abuse notation and write $\Pi_i(B)$ and $\Pi_i(d)$, respectively, whenever the scheme reduces to the singleton scheme x_j, and refer to it as the *projection over the variable x_j*.

There is a sort of inverse operation to projection of domains and tuples, namely their join. To define this, let us consider two schemes $s := x_{i_1}, \ldots, x_{i_m}$ and

$t := x_{j_1}, \ldots, x_{j_k}$ on X, and let r denote the scheme $s \cup t$. Then, if $B \subseteq D[s]$ and $E \subseteq D[t]$, the *join* of B and E, denoted by

$$B \bowtie E,$$

is the subset of $D[r]$ of tuples d such that $d[s] \in B$ and $d[t] \in E$. This implies that, if r' stands for the scheme of the variables which are common to s and t, then $d \in D[r]$ yields $d[r'] \in B[r'] \cap E[r']$. The join of tuples is defined similarly and, if $d \in D[s]$, $e \in D[t]$, their join is denoted by $d \bowtie e$.

Example **2.5.1.** Consider the scheme $X := x_1, x_2, x_3$, and the domains $D_1 := \{\texttt{apple}, \texttt{banana}\}$, $D_2 := \{\texttt{chocolate}, \texttt{sugar}\}$, $D_3 = \{\texttt{dentist}\}$. If s is the scheme x_1, x_2, then $D[s] = D_1 \times D_2$; i.e., the set

$$\{(\texttt{apple}, \texttt{chocolate}), (\texttt{banana}, \texttt{chocolate}), (\texttt{apple}, \texttt{sugar}), (\texttt{banana}, \texttt{sugar})\}.$$

Consider the subset $B := \{(\texttt{apple}, \texttt{chocolate}), (\texttt{banana}, \texttt{chocolate})\}$ of $D[s]$. Then the projection of B over x_2 is the set that only contains `chocolate`, whereas its projection over x_1 collapses into D_1. Similarly, if we consider the tuple $d := (\texttt{banana}, \texttt{chocolate}, \texttt{dentist})$ from the Cartesian product $D_1 \times D_2 \times D_3$, then $d[s]$ is $(\texttt{banana}, \texttt{chocolate})$. While, if t is the scheme x_1, x_3, the tuple $d[t]$ is $(\texttt{banana}, \texttt{dentist})$.

Now, consider again the above subset B of $D[s]$, the scheme $t = x_1, x_3$, and the subset $E := \{(\texttt{apple}, \texttt{dentist})\}$ of $D[t]$. Then the join $B \bowtie E$ is the set

$$\{(\texttt{apple}, \texttt{chocolate}, \texttt{dentist})\};$$

whereas the join of E and the subset $\{\texttt{dentist}\}$ of D_3 is E itself.

2.5.2 Basic Orderings

As we shall clarify in Chapter 3, all the algorithms that we present in Chapter 4 transform a given CSP into another, but variables are neither added nor removed during this transformation process, so that only domains and constraints are modified.

Besides, those algorithms neither insert new values in the input domains, nor add domain elements to the input constraints. In other words, the algorithms in Chapter 4 transform CSP domains or constraints along a certain partial order, without backtracking in the order; in Chapter 3, we shall provide mathematical contents to these still vague claims. However, in the present section, we start introducing the main orders along which those algorithms transform CSPs.

The upward closure of CSPs

The notions of completion and its variants will allow us to easily define the orderings on CSPs that we shall encounter when dealing with constraint propagation.

Given two CSPs P_1 and P_2 on the same variable set, let us consider their completion $\bar{P}_1 := \langle X, \boldsymbol{D}_1, \boldsymbol{C}_1 \rangle$ and $\bar{P}_2 := \langle X, \boldsymbol{D}_2, \boldsymbol{C}_2 \rangle$. Then we write

$$P_1 \sqsubseteq P_2,$$

if the following statements both hold:

- for each $x_i \in X$, we have $D_{1,i} \supseteq D_{2,i}$, where $D_{1,i}$ is the domain of x_i in P_1 and $D_{2,i}$ the domain of x_i in P_2;
- for each $C_1(s)$ in $\bar{P}_1$ and $C_2(s)$ in $\bar{P}_2$, the relation $C_1(s) \supseteq C_2(s)$ holds.

Therefore, two CSPs on the same variable scheme are comparable through $\sqsubseteq$ iff the domains and constraints of their respective completions are comparable through the $\supseteq$ relation. Notice that, here and in the remainder of this thesis, we consistently choose to adopt $\sqsubseteq$ instead of its reverse $\sqsubseteq$ to denote the above relation. The motivation for this choice is that it has become standard in the mathematics and computer science literature (see [DP90]) to deal with partial orders or pre-orders with bottom, or complete partially ordered sets (CPOs) with bottom, least common fixpoints etc.; in other words, to use the $\sqsubseteq$ relation.

Algorithms as in Chapter 4 receive in input a CSP P, and transform it into a CSP P' such that the relation $P \sqsubseteq P'$ holds. Therefore, it is sensible to circumscribe the set of problems those algorithms can produce, so to speak, starting from the input CSP P, as specified in the following definition.

Definition **2.5.2.** Consider a CSP $P := \langle X, \boldsymbol{D}, \boldsymbol{C} \rangle$ and the family $P\!\uparrow$ of all problems P' on X for which the relation $P \sqsubseteq P'$ holds. We call the family $P\!\uparrow$ the *upward closure* or *closure* of P.

The above definition is commonly known in the theory of partial orders and pre-orders as the upward closure with respect to the given relation, see [DP90]. The first of the following results holds in general for all upward closures; the other follows immediately from Definition 2.5.2.

Fact **2.5.3.** *Consider a CSP P and its closure $P\!\uparrow$. Then the following statements hold:*

1. *if $P_1 \sqsubseteq P_2$ and $P_1 \in P\!\uparrow$, then $P_2 \in P\!\uparrow$;*
2. *if $P_1 \sqsubseteq P_2$, then $P_2\!\uparrow \subseteq P_1\!\uparrow$.* □

EXAMPLE **2.5.4.** Consider the CSP P with three variables, x_1 x_2 and x_3, whose domains are equal to $\{0,1\}$, and with only two constraints, defined as follows: $C(x_1, x_2)$ states that $x_1 \neq x_2$, so it only contains the pairs $(0,1)$ and $(1,0)$; $C(x_2, x_3)$ states that $x_2 \neq x_3$, so it contains the same pairs as $C(x_1, x_2)$. The completion $\bar{P}$ has constraints on all the schemes of x_1, x_2, x_3. Therefore, a part from the P constraints, $\bar{P}$ also has the following additional constraints:

- unary constraints on the given variables, $C(x_1)$, $C(x_2)$ and $C(x_3)$, all equal to $\{0,1\}$;
- $C(x_1, x_3)$ is equal to $\{0,1\}^2$; i.e., it contains all pairs of 0 and 1;
- $C(x_1, x_2, x_3)$ is the set $\{0,1\}^3$.

Problems in $P\!\uparrow$ can differ in domains or constraints from P. An instance of a CSP in $P\!\uparrow$ is the problem that is as P, except that its constraint $C(x_1, x_3)$ is equal to $\{(0,1),(1,0)\}$ — corresponding to the inequality $x_1 \neq x_3$. Another example is the $P\!\uparrow$ problem that has all domains empty, and constraints as P.

Unfortunately, the family $P\!\uparrow$ is too large, as the above example suggests: it still contains too many subproblems which are not related to any of the algorithms in Chapter 4. Thereby, in the remainder of this subsection, we carve out those subfamilies of $P\!\uparrow$ that are related to specific classes of algorithms in Chapter 4: i.e., *domain orderings* and *constraint orderings*.

Domain orderings

There are some algorithms for CSPs, such as arc consistency ones in Section 4.2, that only modify domains. Thus, consider a CSP $P := \langle X, \boldsymbol{D}, \boldsymbol{C} \rangle$, and the closure $P\!\uparrow$. Let $\mathcal{F}(P)$ be a subfamily of $P\!\uparrow$, and assume that all problems in $\mathcal{F}(P)$ differ at most in their domains, but have the same constraints: i.e.,

$$\text{if } P' \in \mathcal{F}(P) \text{ and } P' = \langle X, \boldsymbol{D}', \boldsymbol{C}' \rangle, \text{ then } \boldsymbol{C}' = \boldsymbol{C}. \tag{2.2}$$

Then, if P belongs to $\mathcal{F}(P)$, a partial ordering

$$\langle \mathcal{F}(P), \sqsubseteq, P \rangle$$

is *a domain ordering* over P. Notice that P is the bottom (i.e., the least element) of such orderings.

When the family $\mathcal{F}(P)$ contains all the problems P' that satisfy the property (2.2), we refer to the structure $\langle \mathcal{F}(P), \sqsubseteq, P \rangle$ as *the domain ordering* on P.

The CSP algorithms presented in Section 4.2 are explained via iterations of functions that only modify CSP domain sets. So, given a domain ordering over P, say $\langle \mathcal{F}(P), \sqsubseteq, \bot \rangle$, we name a function of the form

$$f : \mathcal{F}(P) \mapsto \mathcal{F}(P) \tag{2.3}$$

a *domain function*. However, all the problems in a domain ordering differ at most in their domains. Therefore, it is natural to regard a domain function as defined on the domains of $\mathcal{F}(P)$; i.e., if we introduce the family of domains

$$\mathcal{D}(P) := \{\boldsymbol{D'} : \text{ there exists } P' \in \mathcal{F}(P) \text{ such that } P' := \langle X, \boldsymbol{D'}, \boldsymbol{C} \rangle\}$$

and restrict the ordering $\sqsubseteq$ on $\mathcal{F}(P)$ to the domains in $\mathcal{D}(P)$, then we can equivalently regard a function as in (2.3) as a function of the form

$$f : \mathcal{D}(P) \mapsto \mathcal{D}(P)$$

on the structure $\langle \mathcal{D}(P), \sqsubseteq, \boldsymbol{D} \rangle$, in which $\boldsymbol{D}$ is the domain set of P. Notice that $\boldsymbol{D}$ is the bottom of such orderings.

Example **2.5.5.** Consider the problem P in Example 2.5.4. The domain ordering on P only contains problems that have the same constraints as P, and differ in their domains. Thus problems whose domains contain 0 or 1; problems that have some or all domains empty. The domain ordering on P contains all such problems. A domain function is $\sigma(x_1; x_1, x_2)$ defined as follows on the domain family $\mathcal{D}(P)$ of P: if $\boldsymbol{B} := B_1, B_2, B_3$ is in $\mathcal{D}(P)$, then $\sigma(x_1; x_1, x_2)(\boldsymbol{B})$ has domains B'_1, B'_2 and B'_3 defined as follows:

$$\begin{aligned} B'_1 &:= \Pi_1(C(x_1, x_2) \cap B_1 \times B_2), \\ B'_2 &:= B_2, \\ B'_3 &:= B_3. \end{aligned}$$

The set $\boldsymbol{B'}$ is still in the domain ordering of P, hence $\sigma(x_1; x_1, x_2)$ is a domain function. Analogously, functions of the form $\sigma(x_i; x_i, x_j)$ and $\sigma(x_j; x_i, x_j)$ can be defined for each pair of the problem variables x_i and x_j such that $i < j$. Such functions return in Section 4.2, where they are used to characterise certain algorithms for CSPs.

Constraint orderings

Algorithms for CSPs such as those in Section 4.3 (the so-called path consistency algorithms) do not modify domains, but alter constraints. These algorithms usually require to first complete the input CSP (see Definition 2.4.8).

Thus, consider a CSP $P := \langle X, \boldsymbol{D}, \boldsymbol{C} \rangle$ with n variables, and the closure $P\uparrow$ of P. Let $\mathcal{F}(P)$ be a subfamily of $P\uparrow$, and assume that all the problems in $\mathcal{F}(P)$ are k complete, for $k \leq n$, and differ at most in their constraint sets: i.e.,

$$\text{if } P' := \langle X, \boldsymbol{D'}, \boldsymbol{C'} \rangle \in \mathcal{F}(P), \text{ then } P' = \bar{P}'_k \text{ and } \boldsymbol{D'} = \boldsymbol{D}. \tag{2.4}$$

Then, if $\bar{P}_k$ belongs to $\mathcal{F}(P)$, the structure

$$\langle \mathcal{F}(P), \sqsubseteq, \bar{P}_k \rangle \tag{2.5}$$

is *a k-constraint ordering* over P. Notice that $\bar{P}_k$ is the bottom of such orderings.

EXAMPLE **2.5.6.** Consider the problem in Example 2.5.4. All 2-constraint orderings will contain $\bar{P}_2$, that is the problem that has the same scheme, domain and constraints as P, plus the additional constraint $C(x_1, x_3)$ equal to the whole set $\{0, 1\}^2$. Those constraint orderings will differ for the binary constraints, that must be subsets of those of $\bar{P}_2$. An example is the 2-constraint ordering that only contain $\bar{P}_2$ and the problem whose constraints are all equal to $\{(0, 1)\}$.

Suppose that all the problems in a subfamily $\mathcal{F}(P)$ of $P\!\uparrow$ are strongly k complete and differ at most in their constraints: i.e.,

$$\text{if } P' := \langle X, \boldsymbol{D'}, \boldsymbol{C'}\rangle \in \mathcal{F}(P), \text{ then } P' = \bar{P}'^{s}_{k} \text{ and } \boldsymbol{D'} = \boldsymbol{D}. \tag{2.6}$$

If the strong k completion of P, namely $\bar{P}^s_k$, belongs to $\mathcal{F}(P)$, then the structure

$$\langle \mathcal{F}(P), \sqsubseteq, \bar{P}^s_k\rangle \tag{2.7}$$

is *a strong k-constraint ordering* over P. Clearly, P^s_k is the bottom of such orderings.

EXAMPLE **2.5.7.** Let us consider Example 2.5.6. Every strong 2-constraint ordering will contain $\bar{P}^s_2$, namely the problem that is as $\bar{P}_2$ in Example 2.5.6, and has in addition the unary constraints on x_1, x_2 and x_3, all equal to $\{0, 1\}$. An instance of a strong 2-constraint ordering is given by the family of problems $\bar{P}^s_2$ and P', which is defined as follows: P' only differs from $\bar{P}^s_2$ in its unary constraints, which are all equal to the empty set.

Suppose that a family $\mathcal{F}(P)$ contains all the k complete problems that satisfy the property (2.4). In this case, $\langle \mathcal{F}(P), \sqsubseteq, \bar{P}_k\rangle$ is called *the k-constraint ordering* on P. If $\mathcal{F}(P)$ contains all the strongly k complete problems that satisfy (2.6), then we refer to the structure $\langle \mathcal{F}(P), \sqsubseteq, \bar{P}^s_k\rangle$ as *the strong k-constraint ordering* on P.

Domain orderings are useful to characterise some functions for certain CSP algorithms, and so are constraint orderings. Thus, given a constraint ordering (2.5) or (2.7), we name a function of the form

$$f \,:\, \mathcal{F}(P) \mapsto \mathcal{F}(P) \tag{2.8}$$

a *constraint function.* Since all the problems in a constraint ordering (2.5) or (2.7) differ at most in their constraints, we can redefine a constraint function as a function on the constraint set $\boldsymbol{C}$ of the constraint ordering (2.5) or (2.7). In fact, if we introduce the family of constraint sets

$$\mathcal{C}(P) := \{\boldsymbol{C'} \,:\, \text{ there exists } P' \in \mathcal{F}(P) \text{ such that } P' := \langle X, \boldsymbol{D}, \boldsymbol{C'}\rangle\}$$

and restrict the ordering $\sqsubseteq$ on $\mathcal{F}(P)$ to the constraint sets in $\mathcal{C}(P)$, then we can equivalently regard a function as in (2.8) as a function of the form

$$f \,:\, \mathcal{C}(P) \mapsto \mathcal{C}(P)$$

on the structure $\langle \mathcal{C}(P), \sqsubseteq, \boldsymbol{C}\rangle$. Notice that $\boldsymbol{C}$ is the bottom of such orderings.

EXAMPLE **2.5.8.** A constraint function on the 2-constraint ordering in Example 2.5.6 is the function $\sigma(x_1, x_2; x_3)$ defined as follows. Consider the constraint set $\boldsymbol{B} := B(x_1, x_2), B(x_2, x_3), B(x_1, x_3)$ in $\mathcal{C}(P)$; then $\sigma(x_1, x_2; x_3)(\boldsymbol{B})$ has constraints $B_1'(x_1, x_2)$, $B_2'(x_2, x_3)$ and $B_3'(x_1, x_3)$ defined as

$$\begin{aligned}
&B'(x_1, x_2) := B(x_1, x_2) \cap \Pi_{x_1,x_2}(B(x_1, x_2) \bowtie B(x_2, x_3)),\\
&B'(x_2, x_3) := B(x_2, x_3),\\
&B'(x_1, x_3) := B(x_1, x_3).
\end{aligned}$$

Thus $\sigma(x_1, x_2; x_3)(\boldsymbol{B})$ is still in the 2-constraint ordering on P, hence $\sigma(x_1, x_2; x_3)$ is a constraint function. Analogously, functions like $\sigma(x_i, x_j; x_k)$ can be defined for each pairwise distinct $i < j$ and k, from 1 to 3. These functions return in Section 4.3, where they are used to characterise certain algorithms for CSPs.

2.6 Conclusions

In this chapter, CSPs are introduced: these are shown to generalise a number of well-known problems, such as map colourability, n-SAT, temporal and spatial reasoning, scheduling and planning problems. In general, the task of finding a solution to these problems is intractable. In the CSP community, a number of approximate algorithms were devised for removing inconsistencies from the solution search space of CSPs; these algorithms are used before the search for solutions, or to this alternated.

In the following chapter, we present a simple theory to describe and analyse all those algorithms, and we put it at work in Chapters 4 and 5.

Chapter 3

A Schema of Function Iterations

3.1 Introduction

3.1.1 Motivations

In the remainder of the first part of our thesis, we shall gradually zoom on constraint propagation. Under this name gathers a number of mainly polynomial-time algorithms; each of these iteratively remove certain inconsistencies from CSPs, thereby attempting to limit the combinatorial explosion of the solution search space. More interestingly, all these algorithms avoid backtracking: at each iteration, a constraint propagation algorithm may remove values from CSPs, but never add them back in subsequent iterations.

Constraint propagation algorithms are known in the literature under other various names: filtering, narrowing, local consistency (which is, for some authors, a more specific notion), constraint enforcing, constraint inference, Waltz algorithms, incomplete constraint solvers, reasoners. However, here and in the remainder of this thesis, we adopt the most popular name, and always refer to them as constraint propagation algorithms.

In [Apt00a], the author states that "the attempts of finding general principles behind the constraint propagation algorithms repeatedly reoccur in the literature on constraint satisfaction problems spanning the last twenty years".

On a larger scale, the search for general principles is a common drive, shared by theoretical scientists of diverse disciplines: a series of methods to solve certain problems are devised; in turn, at a certain stage, this process calls for a uniform and general view if a common pattern can be envisaged. For instance, think of polynomial equations. Until the fifteenth century, algebra was a mere collection of stratagems for solving only numerical equations; these were expressed in words, and the account of the various solving methods was, sometimes, pure literature*. It was Viète in his *Opera Arithmetica* (1646) to introduce the use of vowels

*Cf. "La 'grande arte': l'algebra nel Rinascimento", U. Bottazzini, in *Storia della scienza* for unknown values; this simplified notation paved the way to a general theory of polynomial equations and solving methods, no more restricted to numerical equations.

In this chapter, we propose a simple theory for describing constraint propagation algorithms by means of *function iterations*; the aim is to give a more general view on constraint propagation, based on functions and their iterations. It is well known that partial functions can be used for the semantics of deterministic programs; for instance, see [Jon97, LP81, Pap94]. The primary objective of our theorisation thus becomes that of tackling the following issues:

- abstracting *which* functions perform the task of pruning or propagation of inconsistencies in constraint propagation algorithms,

- *describing* and *analysing how* the pruning and propagation process is carried through by constraint propagation algorithms.

In this chapter, we mainly focus on the latter item, that is on how functions remove certain inconsistencies from CSPs and propagate the effects of this pruning. The basic theory, proposed in this chapter, will provide a uniform reading of a number of constraint propagation algorithms. Then, in Chapter 6, only after describing and analysing those algorithms in Chapters 4 and 5 via that theory, we specify which functions are traced in their study.

3.1.2 Outline

The topic of this chapter is a basic theory of iterations of functions for constraint propagation algorithms.

We first characterise iterations of functions (see Section 3.2) and then introduce the basic algorithm schema that iterates them by following a certain strategy (see Section 3.3). Thus, in the remainder of this chapter, we investigate some properties of the proposed algorithm schema by studying those of the iterated functions and the iterations themselves, see Section 3.4. For example, idempotency of functions will be related to fruitless loops, in terms of pruning, that can be thereby cut off. In turn, this property of functions will be traced in some specific constraint propagation algorithms in which it is used to avoid redundant computations, see Chapter 4.

On the one hand, the proposed algorithm schema is sufficient for *describing* many constraint propagation algorithms in terms of functions on a generic set, see Subsection 3.3.1, or on an equivalent set, see Subsection 3.4.3. On the other hand, a partial order on the function domain provides a sharper tool for *analysing* and studying the behaviour of these algorithms, loosely speaking. More precisely, a partial order on the function domain gives us a means to partially order the

---

moderna e contemporanea, vol. I, TEA.

possible computations of algorithms, see Subsection 3.3.2. Thereby, by means of the domain order, we can pose and answer the following sort of questions about the behaviour of constraint propagation algorithms.

- Can the order of constraint propagation affect the result?
- Or is the output problem independent of the specific order in which constraint propagation is performed (see Theorem 3.3.8)?
- Do constraint propagation algorithms always terminate?
- Or what is sufficient to guarantee their termination (see Theorem 3.3.9 and Corollary 3.3.10)?

In all the analysed cases in Chapter 4, functions for constraint propagation algorithms turn out to be inflationary with respect to a suitable partial order on their domain, see p. 32. This property of functions also accounts for the absence of backtracking in constraint propagation algorithms: pruning of values is never resumed, since every execution of a constraint propagation algorithm always proceeds along an order.

Other properties of functions, related to the order, can be further used to prune branches from the algorithm search tree; we shall study this issue in Subsection 3.4.2. For instance, a property that we call stationarity will be introduced as a stronger form of idempotency; hence functions that enjoy it need to occur at most once in any execution of the algorithm schema.

3.1.3 Structure

First, we introduce iterations of functions in Section 3.2, and the basic schema to iterate them in Section 3.3. Variations of the basic schema, along with the related properties of functions, are treated in details in Section 3.4. Finally, we summarise and discuss the results of this chapter in Section 3.5.

3.2 Iterations of Functions

Given a finite set F of functions $f : O \mapsto O$ over a set O, we define a sequence $\langle o_n : n \in \mathbb{N} \rangle$ with values in O as follows:

1. $o_0 := \perp$, where $\perp$ is a selected element of O;
2. $o_{n+1} := f(o_n)$, for some $f \in F$.

Each sequence $\langle o_n : n \in \mathbb{N} \rangle$ is called an *iteration of* F functions (*based on* $\perp$). An iteration of F functions $\langle o_n : n \in \mathbb{N} \rangle$ *stabilises* at o_n if $o_{n+k} = o_n$ for every $k \geq 0$.

In this chapter, we shall mainly be concerned with iterations of F functions that stabilise at some specific points: in fact, we shall be interested in iterations that stabilise at a *common fixpoint* of all the functions: namely, an element $o \in O$ such that

$$f(o) = o \text{ for all } f \in F.$$

Indeed, it is not sufficient for an iteration to stabilise at o for this to be a common fixpoint of all the F functions, as the following simple example illustrates.

EXAMPLE **3.2.1.** Consider $O := \{0, 1, 2\}$ and the set F with the following two functions:

$$f(0) := 0,\ f(1) := 2 \text{ and } f(2) := 2,$$
$$g(0) := 1,\ g(1) := 1 \text{ and } g(2) := 2.$$

Now, consider the iteration $\langle o_n : n \in \mathbb{N} \rangle$ based on 0 such that $o_{n+1} := f(o_n)$ for every $n \in \mathbb{N}$. Indeed, the iteration stabilises at 0; but this is not a fixpoint of g since $g(0) \neq 0$.

In the above Example 3.2.1, the function g is never selected. Would it be sufficient to choose g after f to guarantee that o is a common fixpoint of the F functions? Certainly not: define first $o_1 := f(o_0) = 0$, then $o_{j+1} := g(o_j)$ for $j > 0$. This iteration stabilises at 1 and not at 2, which is the only common fixpoint of the two functions f and g. We can repeat the above trick infinitely many times, one for every $k > 0$: in fact, it is sufficient to set $o_{i+1} = f(o_i)$ for $0 \leq i < k$, and $o_{j+1} := g(o_j)$ for $j \geq k$; still the resulting iteration stabilises at 1. How can we remedy this? The answer is given below, by the algorithm schema in Section 3.3: this is designed to compute a common fixpoint of finitely many functions.

3.3 The Basic Iteration Schema

The Structured Generic Iteration algorithm, briefly `SGI`, is a slightly more general version of the Generic Iteration algorithm of [Apt00a]. Both of them aim at computing a common fixpoint of finitely many functions, simply by iterating them until such a fixpoint is computed. The `SGI` algorithm is more general in that its first execution can start with a subset of all the given functions; then these are introduced, *only if necessary*, in subsequent iterations. So `SGI` covers more algorithms than the Generic Iteration algorithm does.

The `SGI` algorithm is displayed as Algorithm 3.3.1. Its parameters are characterised as follows.

CONVENTION **3.3.1** (`SGI`).

- F is a finite set of functions, all defined on the same set O;

- $\perp$ is an element of O;
- $F_\perp$ is a subset of F that enjoys the following property: every F function f such that $f(\perp) \neq \perp$ belongs to $F_\perp$;
- G is a subset of F functions;
- *update* instantiates G to a subset of F.

Algorithm 3.3.1: SGI($\perp, F_\perp, F$)

$o := \perp$;
$G := F_\perp$;
while $G \neq \emptyset$ **do**
  choose $g \in G$;
  $G := G - \{g\}$;
  $o' := g(o)$;
  if $o' \neq o$ **then**
    $G := G \cup update(G, F, g, o)$;
    $o := o'$

As we shall see below, the *update* operator returns a subset of F according to the functions in G, the current O value o and F function g; its computation can be expensive, unless some information on the chosen function g and input value o is provided that can help to compute the F functions returned by *update*, as we shall see in Chapter 4. Besides, in the `SGI` schema below, the function g is chosen non deterministically; no strategy for choosing g is imposed in this schema; but this is done on purpose, since `SGI` aims at being a general template for a number of CSP algorithms. Indeed, the complexity of the algorithm will vary according to the way in which the *update* operator will be specified and the function g chosen.

3.3.1 The basic theory of `SGI`

The `SGI` algorithm is devised to compute *a common fixpoint of the F functions*: i.e., an element $o \in O$ such that $f(o) = o$ for every $f \in F$. Suppose that the following predicate

$$\forall f \in F - G \; f(o) = o \qquad (Inv)$$

is an invariant of the **while** loop of the `SGI` algorithm. If o is the last input of a terminating execution of `SGI`, then G is the empty set and the predicate *Inv* above implies that o is a common fixpoint of all the F functions. We restate this as the following fact, which is used over and over in the remainder of this chapter.

FACT **3.3.1** (COMMON FIXPOINT). *Suppose that the above predicate Inv is an invariant of the* **while** *loop of* `SGI`*. If o is the last input of a terminating execution of* `SGI`*, then o is a common fixpoint of the F functions.* □

Common fixpoint

The Common Fixpoint Fact 3.3.1 above suggests a simple, yet sufficient condition for `SGI` to compute a common fixpoint of the F functions: after an iteration of the **while** loop, we only need to keep, in G, the functions for which the input value of the **while** loop is not a fixpoint. As for this, it is sufficient that the *update* operator in `SGI` satisfies the following axiom.

AXIOM **3.3.1** (COMMON FIXPOINT). Let $o' := g(o)$ for $g \in F$, and $Id(g, o') := \{g\}$ if $g(o') \neq o'$, else $Id(g, o')$ is the empty set. If $o' \neq o$, then

$$update(G, F, g, o) \supseteq \{f \in (F - G) \, : \, f(o) = o \text{ and } f(o') \neq o'\} \cup Id(g, o');$$

otherwise $update(G, F, g, o)$ is the empty set.

In other words, the *update* operator adds to G a least all the F functions, not already in G, for which o is a fixpoint and the new value o' is not; besides, g has to be added back to G if $g(o') \neq o'$.

LEMMA **3.3.2** (INVARIANCE). *Let us assume the Common Fixpoint Axiom 3.3.1. Then the Inv predicate on p. 31 is an invariant of the* **while** *loop of* `SGI`*.*

PROOF. The base step follows from the definition of $F_\perp$ (see Convention 3.3.1 above), and the induction step is easily proved by means of the Common Fixpoint Axiom 3.3.1. □

The above Common Fixpoint Fact 3.3.1 and Invariance Lemma 3.3.2 immediately imply the following theorem.

THEOREM **3.3.3** (COMMON FIXPOINT). *Let us assume the Common Fixpoint Axiom 3.3.1. If o is the last input of a terminating execution of the* `SGI` *algorithm, then o is a common fixpoint of all the F functions.* □

EXAMPLE **3.3.4.** Let us consider Example 3.2.1 as input to `SGI` so that $\perp := 0$ and $F_\perp := \{g\}$. In the first **while** loop, only g can be chosen and applied; so, after the loop, v is set equal to 1. In the same loop, *update* adds f to G and leaves g in G. So, at the end of the first loop, $G = \{f, g\}$ and $o = 1$. Then, if f is chosen and applied to 1, o is set equal to 2 and G to the empty set at the end of the fourth loop. So `SGI` terminates by computing 2, a common fixpoint of the two functions. Instead, if g is chosen again in the second loop, then it is also removed and only f can be chosen in the third loop; so the computation of `SGI` terminates with 2 at the end of the fourth loop. Notice that, given Axiom 3.3.1, there are three different executions of the `SGI` algorithm with f and g.

`SGI` iterations

We started this chapter with generic iterations of functions, and provided a schema that computes a common fixpoint of finitely many functions. Hereby we show how function iterations and the `SGI` schema are related. First of all, let us denote by id the identity function on the domain of the iterated functions of F; indeed all the common fixpoints of the F functions are, trivially, fixpoint of id. Then every execution of `SGI` gives rise to an iteration of the $F \cup \{id\}$ functions. To explain how, we first introduce traces of `SGI` executions.

Consider an execution of the `SGI` algorithm — see Algorithm 3.3.1. The `SGI` *trace* $\langle (o_n, G_n) \,:\, n \in \mathbb{N} \rangle$ of the execution is defined as follows:

- $o_0 := \bot$, $G_0 := F_\bot$;
- suppose that o_n and G_n are the input of the n-th **while** loop of `SGI`. If G_n is the empty set, then $o_{n+1} := id(o_n)$ and $G_{n+1} := \emptyset$. Otherwise, let g be the chosen function and set o_{n+1} equal to $g(o_n)$. Then we define G_{n+1} as the set $G_n \cup update(G_n, F, o, o_n)$ if $o_{n+1} \neq o_n$, otherwise as the set $G_n - \{g\}$.

Then $\langle o_n \,:\, n \in \mathbb{N} \rangle$ is an `SGI` *iteration* of the F functions.

Example **3.3.5.** Let us revisit Example 3.3.4. There we have the following three `SGI` traces:

1. $\langle (0, \{g\}), (1, \{f, g\}), (2, \{f, g\}), (2, \{f\}), (2, \emptyset), \ldots \rangle$;
2. $\langle (0, \{g\}), (1, \{f, g\}), (2, \{f, g\}), (2, \{g\}), (2, \emptyset), \ldots \rangle$;
3. $\langle (0, \{g\}), (1, \{f, g\}), (1, \{f\}), (2, \{f\}), (2, \emptyset), \ldots \rangle$.

The first two traces give rise to the same `SGI` iteration $\langle 0, 1, 2, \ldots \rangle$; whereas the `SGI` iteration in the third item is $\langle 0, 1, 1, 2, \ldots \rangle$.

Traces provide another tool to formulate and study properties of `SGI`, like termination. We shall say that the *trace* $\langle (o_n, G_n) \,:\, n \in \mathbb{N} \rangle$ *stabilises at* o_k iff the iteration $\langle o_n \,:\, n \in \mathbb{N} \rangle$ does so and $G_k = \emptyset$. Now, the termination condition for the **while** loop in `SGI` is that G must be empty; the last input o of a terminating execution of `SGI` is the *value computed by* `SGI`. Hence it is easy to check that the following statements are equivalent:

1. the `SGI` algorithm terminates by computing o;

2. the `SGI` trace stabilises at $o_k = o$.

We reformulate this equivalence as the following fact, as it will allow us to switch from executions of `SGI` to traces, and vice versa, when dealing with the termination of `SGI`.

FACT **3.3.6.** *An execution of the* `SGI` *algorithm terminates by computing o iff the associated trace stabilises at o.* □

3.3.2 Ordering Iterations

Suppose that we can define a partial order $\sqsubseteq$ over the set O. Then this can be used to order iterations.

Least common fixpoint

Suppose that the F *functions* are *monotone* with respect to a partial order $\sqsubseteq$ on O; namely, for every $f \in F$,

$$o \sqsubseteq o' \text{ implies } f(o) \sqsubseteq f(o').$$

Then we can prove that the *common fixpoints of the F functions*, as computed by `SGI`, coincide with the least fixpoint of the F functions. So let us assume the following statement.

AXIOM **3.3.2** (LEAST FIXPOINT).
(i). The structure $\langle O, \sqsubseteq, \bot\rangle$ is a partial ordering with bottom $\bot \in O$.
(ii). The F functions are all monotone with respect to $\sqsubseteq$.

Given the above axiom, we have the following lemma as in [Apt00a].

LEMMA **3.3.7** (STABILISATION). *Assume the Common Fixpoint Axiom 3.3.1 and the Least Fixpoint Axiom 3.3.2. Consider a fixpoint w of the F functions. Let $\langle o_i : i \in \mathbb{N}\rangle$ be a generic iteration of F that satisfies the following properties:*

- $o_0 := \bot$*;*
- $o_{i+1} := g(o_i)$*, for some* $g \in F$*.*

Then $o_i \sqsubseteq w$, for every $f \in F$ and o_i in the iteration $\langle o_i : i \in \mathbb{N}\rangle$.

PROOF. The proof is by induction on $i \in \mathbb{N}$. The base case is trivial since $\bot$ is the bottom of the ordering. As for the induction step, let us assume that $o_i \sqsubseteq w$. Thus, we invoke monotonicity (see Axiom 3.3.2) and obtain $o_{i+1} := g(o_i) \sqsubseteq w$. □

The above lemma shows how a partial ordering can be used to compare computations of `SGI` with functions on the ordering, and highlights the role of monotonicity in the following result, which follows from the lemma and Theorem 3.3.3.

Theorem **3.3.8** (least fixpoint). *Let F be a finite set of functions over O, and assume the Common Fixpoint Axiom 3.3.1 and the Least Fixpoint Axiom 3.3.2. Then all the terminating executions of* `SGI` *compute the same value: i.e., the least common fixpoint of all the F functions with respect to the partial order on O.* $\square$

Termination

From this point onwards, let us write $o \sqsubset o'$ whenever $o \sqsubseteq o'$ and $o \neq o'$. A $\sqsubset$-*chain in* O is any subset of O, that is totally ordered by $\sqsubset$.

In order to ensure the termination of `SGI`, for any input, we must ascertain that every `SGI` iteration stabilises. If all the F functions are computable, every `SGI` iteration is totally ordered by $\sqsubseteq$, and all $\sqsubset$-chains are finite, then we can guarantee termination. The following axiom formalises these ideas.

Axiom **3.3.3** (termination).
- Each $f \in F$ is a computable function over a partial ordering with bottom $\mathcal{O} := \langle O, \sqsubseteq, \perp \rangle$.
- The F functions are *inflationary* with respect to the partial order: namely, $o \sqsubseteq f(o)$ for every $o \in O$ and $f \in F$.
- The ordering $\langle O, \sqsubseteq \rangle$ satisfies the *ascending chain condition* (ACC), i.e. each $\sqsubset$-chain in O is finite.

Now, given the above axiom, we can prove the following termination result.

Theorem **3.3.9** (termination 1). *Assume the Common Fixpoint Axiom 3.3.1 and the Termination Axiom 3.3.3. Then* `SGI` *always terminates, by computing a common fixpoint of the F function.*

Proof. At each iteration of the **while** loop, either $o \sqsubset o'$ — due to inflationarity, see Axiom 3.3.3 — or g is removed from G. Axiom 3.3.3 yields that all $\sqsubset$-chains are finite; since $G \subseteq F$ is finite too, the algorithm terminates. $\square$

Notice that every finite partial ordering satisfies the ACC in Axiom 3.3.3; moreover, every function on a finite set is computable. Thus we draw the following conclusion as a corollary of Theorem 3.3.9.

Corollary **3.3.10** (termination 2). *Let us assume the Common Fixpoint Axiom 3.3.1 and that the F functions are defined on a finite partial ordering with bottom $\langle O, \sqsubseteq, \perp \rangle$. Suppose that the F functions are inflationary with respect to $\sqsubseteq$. Then every execution of* `SGI` *terminates.* $\square$

Note **3.3.11.** Many algorithms for CSPs deal with finite domains. Whenever those algorithms are instances of `SGI`, the above Corollary 3.3.10 will ensure that a simple condition on the iterated functions is sufficient to guarantee the termination of the instance algorithms. However, functions for non-standard CSPs as in Chapter 5, often, have infinite domains; then Corollary 3.3.10 will not be of help, and we shall need to resort to the above Theorem 3.3.9.

3.3.3 Finale

The main results concerning the basic `SGI` algorithm schema are collected in the following corollary; this gathers Theorems 3.3.3, 3.3.8, 3.3.9 and Corollary 3.3.10. Figure 3.3.3 depicts a search tree of `SGI`, under the assumptions of either one of the statements in the following corollary.

Corollary **3.3.12.**
(i). Assume the Common Fixpoint Axiom 3.3.1, the Least Fixpoint Axiom 3.3.2 and the Termination Axiom 3.3.3. Then every execution of `SGI` *terminates by computing the least common fixpoint o of the iterated functions.*
(ii). Assume the Common Fixpoint Axiom 3.3.1 and that the F functions are defined on a finite partial ordering with bottom $\langle O, \sqsubseteq, \bot \rangle$. Suppose that the F functions are all monotone and inflationary with respect to $\sqsubseteq$. Then every execution of `SGI` *terminates by computing the least common fixpoint o of the iterated functions.* □

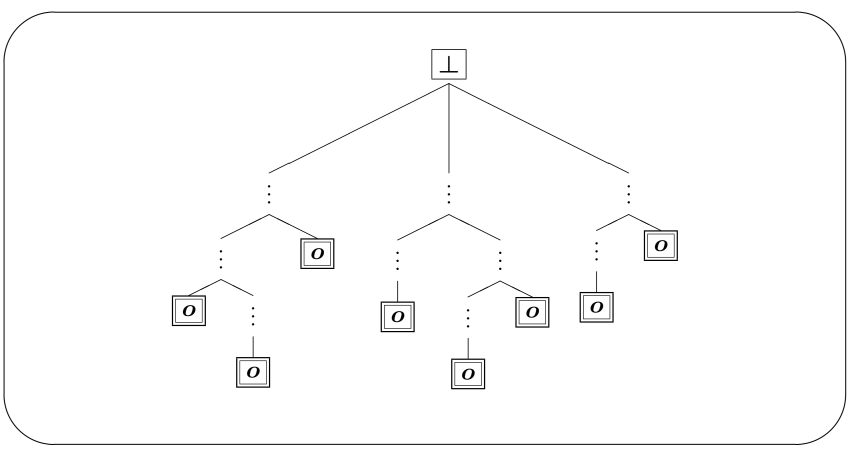


Figure 3.1: `SGI` search tree.

3.4 Variations of the Basic Schema

3.4.1 The Generic Iteration Schema

We started Section 3.3 by claiming that `SGI` is a slightly more general version of the Generic Iteration (`GI`) algorithm of [Apt00a]. We shall also need to refer to the latter schema in Chapter 4, and hence we explain it in more details as below.

The difference between the two basic schemas is that the set of functions G in `GI` is initialised to the whole set of functions F: i.e., $F_\perp$ is the whole set F. Therefore, we have the following results for `GI` as a consequence of Corollary 3.3.12.

Theorem **3.4.1.**
(i). Assume the Common Fixpoint Axiom 3.3.1, the Least Fixpoint Axiom 3.3.2 and the Termination Axiom 3.3.3. Then `GI` *always terminates by computing the least common fixpoint of the iterated functions.*
(ii). Assume the Common Fixpoint Axiom 3.3.1 on update and that the F functions are defined on a finite partial ordering with bottom $\langle O, \sqsubseteq, \perp \rangle$. Suppose that the F functions are monotone and inflationary with respect to $\sqsubseteq$. Then `GI` *always terminates, computing the least common fixpoint of the iterated functions.* $\Box$

3.4.2 Iterations Modulo Function Properties

The `GI` algorithm is a variation of `SGI` in that G is differently initialised. Other variations of the basic schema `SGI` are obtained by optimising the instantiation of G in the **while** loop via *update*: in `SGI`, all the functions for which the new computed value $o' = g(o)$ is not a fixpoint are added to G. Indeed, more functions are added to G, more executions of the **while** loop are needed. In the following, we study some properties of functions that allow us to reduce the number of executions of the **while** loop by an efficient instantiation of G via *update*. Each property is studied separately and gives rise to a different version of the `SGI` schema; all these or their combinations will be used in Chapter 4.

Idempotent functions

Notice that the chosen function g is removed from the set G of iterated functions in `SGI` if the test $gg(o) = g(o)$ returns true. This is always true, independently of the specific value o, if g is idempotent, as specified below.

Definition **3.4.2.** A function $g : O \mapsto O$ is idempotent if $gg(o) = g(o)$ for every $o \in O$.

As suggested above, an idempotent function can always be removed after being chosen. The following diagram shows what happens otherwise.

$$\begin{tikzcd} {} \arrow[r, dotted] & o \arrow[r] & \boldsymbol{g}(o) \arrow[r] & \boldsymbol{g}g(o) \arrow[l, bend right, Rightarrow, "="'] \arrow[r, dotted, dash] & {} \end{tikzcd}$$

So any iteration as above can be equivalently reduced to one in which the second application of g is removed, if this function is idempotent; i.e., $Id(g, o')$, as in the Common Fixpoint Algorithm refaxiom:sgi:1, is always set to the empty set for g idempotent. The following lemma is an immediate consequence of that axiom and Definition 3.4.2 above.

Lemma **3.4.3** (idempotency). *Consider a finite set F of idempotent functions on O. Then*

$$update(G, F, g, o) \supseteq G \cup \{f \in F - G \,:\, f(o) = o \text{ and } f(o') \neq o'\}$$

satisfies the Common Fixpoint Axiom 3.3.1. □

Let us call `SGII` the version of `SGI` with the *update* operator as in Lemma 3.4.3, where the second `I` stands in for Idempotent. Then the following theorem is a trivial consequence of that lemma and Corollary 3.3.12.

Theorem **3.4.4.**
(i). Assume the Least Fixpoint Axiom 3.3.2 and the Termination Axiom 3.3.3. Then every execution of `SGII` *terminates by computing the least common fixpoint of the iterated functions, if these are all idempotent.*
(ii). Assume that the F functions are defined on a finite partial ordering with bottom $\langle O, \sqsubseteq, \perp\rangle$. Suppose that the F functions are all monotone and inflationary with respect to $\sqsubseteq$. Then every execution of `SGII` *terminates, computing the least common fixpoint of the iterated functions, if these are all idempotent.* □

Commutative functions

Commutativity of an operation is a useful property in computations: it means that the operation provides the same result regardless of permutations of the combined elements. Function composition is not, in general, a commutative operation. However there are classes of functions on which composition is commutative; thereby the order in which these functions are composed is irrelevant. The following definition aims at capturing precisely this, and it is a special case of the notion of centraliser of an element with respect to a given operation, like in group theory; see for instance [Her75].

DEFINITION **3.4.5.** Let F be a set of functions over a set O, consider a function $g : O \mapsto O$, and the subset $Comm(F, g)$ of F of all functions f such that

$$fg(o) = gf(o), \text{ for all } o \in O.$$

Then the set $Comm(F, g)$ is the set of F functions that *commute* with g.

As stated and proved in [Apt00a], commutativity can be exploited to reduce executions in the `GI` algorithm. This carries over to the `SGI` schema in the same manner.

LEMMA **3.4.6.** *If the update operator satisfies the Common Fixpoint Axiom 3.3.1, then so does* $update(G, F, g, o) - Comm(F, g)$*.*

PROOF. Suppose that $fg(o) = gf(o)$; then $f(o) = o$ implies $gf(o) = g(o)$; thus $update - Comm$ satisfies the Common Fixpoint Axiom 3.3.1 if *update* does. □

Let us rename `SGI` with Commutativity, briefly `SGIC`, the `SGI` algorithm with $update - Comm$ in place of *update*. The above Lemma 3.4.6 allows us to transfer, to `SGIC`, all the results concerning `SGI` as summarised in Corollary 3.3.12.

THEOREM **3.4.7.**
(i). Assume Assume the Common Fixpoint Axiom 3.3.1, the Least Fixpoint Axiom 3.3.2 and the Termination Axiom 3.3.3. Then `SGIC` *always terminates by computing the least common fixpoint of the iterated functions.*
(ii). Assume the Common Fixpoint Axiom 3.3.1 on update and that the F *functions are defined on a finite partial ordering with bottom* $\langle O, \sqsubseteq, \perp \rangle$*. Suppose that the* F *functions are all monotone and inflationary with respect to* $\sqsubseteq$*. Then* `SGIC` *always terminates, computing the least common fixpoint of the iterated functions.*
□

Stationary functions

While the properties of idempotency and commutativity do not rely on any partial order on the given set O, the following property does.

DEFINITION **3.4.8.** Let f be an inflationary function, defined over a partial ordering $\langle O, \sqsubseteq \rangle$. Then the function f is *stationary from* $o \in O$ *and* o' if it enjoys the following property:

$$\text{if } f(o) \neq o,\ o \sqsubseteq o' \text{ and } f(o') \sqsubset o'' \text{ then } f(o'') = o''.$$

More in general, an inflationary function f is *stationary* if there exist o, $o' \in O$ such that f is stationary from them.

In other words: consider a totally ordered iteration and suppose that a stationary function f is known to affect a value o in it; after the first application of f to o or a subsequent value o' in the iteration, f does not change any value that follows in the iteration. The following diagram shows schematically what happens whenever a stationary function f is applied again, after f modifies a value in the iteration.

$$
\begin{tikzcd}
 & f(o) \arrow[d, Rightarrow, bend right, "\neq"'] & & & & & \\
\arrow[r, dotted] & o \arrow[r, dotted] \arrow[r, Rightarrow, bend right, "\sqsubseteq"'] & o' \arrow[r] \arrow[r, Rightarrow, bend right, "\sqsubseteq"'] & \boldsymbol{f}(o') \arrow[r, dotted] \arrow[r, Rightarrow, bend right, "\sqsubset"'] & o'' \arrow[r] & \boldsymbol{f}(o'') \arrow[l, Rightarrow, bend right, "="'] \arrow[r, dotted] &
\end{tikzcd}
$$

In brief, the iteration in the above diagram can equivalently be reduced to one in which the second application of f is removed, if this function is stationary. The below lemma states precisely that stationary functions can be added at most once to G, namely the set of functions to iterate.

In order to formulate the lemma properly, we resort to traces and state the following axiom.

Axiom **3.4.1** (stationarity).
(i). *The F functions are all stationary on $\langle O, \sqsubseteq, \bot\rangle$ and, if $f \in F_\bot$ and $f(\bot) = \bot$, then f is the identity on O.*
(ii). *If $\langle (o_n, G_n) : n \in \mathbb{N}\rangle$ is the trace of an execution of* `SGI`*, G_n denotes the set of G functions at the n-th* **while** *loop of* `SGI`*; then put*

$$
update(G_n, F, g, o_n) := \left\{ f \in F - \bigcup_{k \le n} G_k : f(o_n) = o_n,\ f(o_{n+1}) \neq o_{n+1} \right\};
$$

this for every n.

Now we can formulate the following Stationarity Lemma: there we assume that the F functions are idempotent, since this simplifies the proof, even tough the extension to the non-idempotent case is possible.

Lemma **3.4.9** (stationarity). *Assume the Stationarity Axiom 3.4.1 and that all the F functions are idempotent. Then update satisfies the Common Fixpoint Axiom 3.3.1.*

Proof. We only need to prove that, if $f \in \bigcup_{k<n} G_k$ and $f \notin G_n$, then $f(o_{n+1}) = o_{n+1}$. If $f \in F_\bot$ and $f(\bot) = \bot$, then $f(o_{n+1}) = o_{n+1}$ by Axiom 3.4.1. Else, there must be o_i in the iteration such that $i < n$ and $f(o_i) \neq o_i$, due to Axiom 3.4.1 again. Since $f \notin G_n$, then there exists $i \le j < n$ and o_j in the iteration such that $o_{j+1} = f(o_j)$. Therefore, inflationarity yields the following:

$$
o_i \sqsubseteq o_j \text{ and } f(o_j) = o_{j+1} \sqsubseteq o_n \sqsubset o_{n+1}.
$$

Thus we can conclude that $o_i \neq f(o_i)$, $o_i \sqsubseteq o_j$ and $f(o_j) \sqsubset o_{n+1}$ hold. Then Definition 3.4.8 yields $f(o_{n+1}) = o_{n+1}$. $\square$

In Chapter 4, we make an extensive use of the variation of the `SGI` algorithm with stationary and idempotent functions. So we rewrite `SGI` with Stationary and Idempotent functions as the `SGIIS` Algorithm 3.4.1.

Algorithm 3.4.1: SGIIS($\perp, F_\perp, F$)

$v := \perp$;
$G := \emptyset$;
while $F_\perp \neq \emptyset$ **do**
  choose $g \in F_\perp$;
  $F_\perp := F_\perp - \{g\}$;
  $o' := g(o)$;
  if $o' \neq o$ **then**
    $G := G \cup update(G, F, g, o)$;
    $F := F - update(G, F, g, o)$;
    $o := o'$;
while $G \neq \emptyset$ **do**
  choose $g \in G$;
  $G := G - \{g\}$;
  $o' := g(o)$;
  if $o' \neq o$ **then**
    $G := G \cup update(G, F, g, o)$;
    $F := F - update(G, F, g, o)$;
    $o := o'$;

The Stationarity Lemma 3.4.9 and the Idempotency Lemma 3.4.3 allow us to transfer, to `SGIIS`, all the results concerning `SGI` as summarised in Corollary 3.3.12.

THEOREM **3.4.10.**
(i). Assume the Stationarity Axiom 3.4.1, the Least Fixpoint Axiom 3.3.2 and the Termination Axiom 3.3.3. Then every execution of `SGIS` *terminates by computing the least common fixpoint of the iterated functions.*
(ii). Assume the Stationarity Axiom 3.4.1, the Least Fixpoint Axiom 3.3.2, and that the F functions are defined on a finite partial ordering with bottom $\langle O, \sqsubseteq, \perp\rangle$. Then every execution of `SGIS` *terminates by computing the least common fixpoint of the iterated functions.* $\square$

3.4.3 Iterations Modulo Equivalence

The SGI algorithm with Equivalence (SGIE) is SGI with functions defined on an equivalence structure $\langle V, \equiv \rangle$, and such that the **if** conditional depends on the *equivalence* of the input and output values, and not necessarily their identity; see Algorithm 3.4.2. Like for SGI, the *update* operator selects and returns functions of F. Thus this algorithm iterates functions from a set F until a value v is found for which $g^V(v) \equiv v$. Indeed, if the equivalence relation $\equiv$ collapses into the identity, we have the SGI algorithm back.

Algorithm 3.4.2: SGIE($\perp, \equiv, F_\perp, F$)

$v := \perp^V$;
$G^V := F^V_{\perp^V}$;
while $G^V \neq \emptyset$ **do**
  choose $g^V \in G^V$;
  $G^V := G^V - \{g^V\}$;
  $v' := g^V(v)$;
  if $v' \not\equiv v$ **then** $G^V := G^V \cup \mathit{update}(F^V, g^V, v)$;
  $v := v'$;

NOTE **3.4.11.** We let $\mathit{update}(F^V, g^V, v)$ be an unspecified subset of G. In fact, in the case of the SGIE algorithm schema, the *update* operator varies according to the instance CSP algorithms. However, as for the results in this part, we do not need to assume anything more of *update* that it is returns a subset of F functions.

SGIE iterations

As in the case of SGI, we associate SGIE traces with executions of an SGIE algorithm. Again, notice that the identity function id^V on V does not affect any value in any computation of SGIE; i.e., $id^V(v) \equiv v$ for every $v \in V$.

The SGIE *traces* $\langle (v_n, G^V_n) \ : \ n \in \mathbb{N} \rangle$ of executions of the SGIE algorithm are defined like the SGI traces:

- $v_0 := \perp_V$, $G^V_0 := F^V_\perp$;
- suppose that v_n and G^V_n are the input of the n-th **while** loop of SGIE. If G^V_n is the empty set, then $v_{n+1} := v_n$ and $G^V_{n+1} := \emptyset$. Otherwise, let g^V be the chosen function and set v_{n+1} equal to $g^V(v_n)$. Then the set G^V_{n+1} is defined as $G^V_n - \{g^V\} \cup \mathit{update}(F^V, G^V_n, g^V, v_n)$ if $v_{n+1} \not\equiv v_n$, otherwise as the set $G^V_n - \{g^V\}$.

The iteration $\langle v_n : n \in \mathbb{N}\rangle$ is called an `SGIE` *iteration* of `SGIE`.

As in the case of `SGI`, traces are useful to formulate and study termination conditions on `SGIE`. So we shall say that the `SGIE` *trace stabilises at* v_n if the iteration $\langle v_n : n \in \mathbb{N}\rangle$ does so and $G^V_n = \emptyset$. The following equivalence will be useful in the below part.

FACT **3.4.12.** *An iteration of an* `SGIE` *algorithm terminates by computing* v *iff the associated trace stabilises at* v. □

The least ≡-class and termination

Suppose that we can devise a partial order $\sqsubseteq$ on a quotient set O isomorphic to $V/\equiv$, such that $\langle O, \sqsubseteq, \bot^O\rangle$ turns out to be a partial ordering with bottom. Then we can try to transfer the analysis and results concerning `SGI`, over the partial ordering $\langle O, \sqsubseteq, \bot^O\rangle$, to `SGIE` over the equivalence structure $\langle V, \equiv\rangle$.

Let F^V and F^O be, respectively, a finite set of functions over V and O. Consider a bijective map $\mathtt{f} : F^O \mapsto F^V$ that maps the identity of F^O to the identity function of F^V. Let us denote

$$f^V := \mathtt{f}(f^O),$$

for each $f^O \in F^O$. Now, suppose that an `SGIE` trace $\langle (v_n, G^V_n) : n \in \mathbb{N}\rangle$ of F^V functions can be associated with an `SGI` trace $\langle (o_n, G^O_n) : n \in \mathbb{N}\rangle$ of F^O functions via `f`, and that such traces enjoy the following property:

1. $v_0 \in o_0$;
2. for every $n \geq 0$, if $v_{n+1} = f^V(v_n)$ for $f^V \in G^V_n$ then $o_{n+1} = f^O(o_n)$ for $f^O \in G^O_n$, and the following property holds:

$$\text{there exists } m \geq n+1 \ \text{ such that } v_m \in o_m.$$

Then the two traces $\langle (v_n, G^V_n) : n \in \mathbb{N}\rangle$ and $\langle (o_n, G^O_n) : n \in \mathbb{N}\rangle$ are called ≡-*equivalent* via `f`.

The characterisation of ≡-equivalence, via a function `f`, is sufficient to obtain the following result.

LEMMA **3.4.13.** *Consider an* `SGI` *trace* $\mathtt{o} := \langle o_n : n \in \mathbb{N}\rangle$ *and an ≡-equivalent* `SGIE` *trace* $\mathtt{v} := \langle v_n : n \in \mathbb{N}\rangle$.

- *The trace* `v` *stabilises at a value* $v \in V$ *if the trace* `o` *does so at a value* $o \in O$;
- *furthermore, the value* $o \in O$ *(where* O *is isomorphic to* $V/\equiv$*) corresponds to the ≡-class of* v.

PROOF. Suppose that $\langle o_n : n \in \mathbb{N}\rangle$ stabilises at o_n. Then $G^O_n = \emptyset$, hence G^V_n is empty due to the definition of equivalent traces above. Then $v_n \in o_n$ follows from the above definition. Therefore $\langle (v_n, G^V_n) : n \in \mathbb{N}\rangle$ stabilises at $v_n \in o_n$. $\square$

The following definition extends the notion of $\equiv$-equivalence between traces to an analogous between algorithm executions.

DEFINITION **3.4.14.** If there exists a map `f` such that every `SGIE` trace with F^V is $\equiv$-equivalent to an `SGI` trace with F^O via `f`, then `SGIE` with F^V is called $\equiv$-*equivalent* to `SGI` with F^O.

The following theorem is a consequence of Facts 3.3.6 and 3.4.12, and the above Lemma 3.4.13.

THEOREM **3.4.15.** *Suppose that every execution of* `SGI` *terminates by computing the least common fixpoint o of the F_O functions. If* `SGIE` *with F^V functions over $\langle V, \equiv\rangle$ is $\equiv$-equivalent to* `SGI`*, then every execution of* `SGIE` *terminates by computing $\equiv$-equivalent values; i.e. values v in the $\equiv$-class that corresponds to o.* $\square$

Theorem 3.4.15 above implies that we can *study* instances of the `SGIE` schema — that takes in input a set V with an equivalence relation — if we can provide, for them, equivalent instances of the `SGI` algorithm schema:

1. we devise a partial ordering with bottom on a set O, isomorphic to the quotient set $V/\equiv$;
2. then we check whether the given instance of `SGIE` on V is $\equiv$-equivalent to an instance of `SGI` on O, with suitable functions on O;
3. if this equivalence holds, then Theorem 3.4.15 implies that, if `SGI` terminates by always computing the same value, then `SGIE` terminates by always computing values which belong to the same equivalence class.

These transfer results, summarised as in the below corollary, are consequences of Theorem 3.4.15, and Corollary 3.3.12 for `SGI`.

COROLLARY **3.4.16.** *Consider an instance of* `SGI` *with F^O functions on a finite partial ordering $\mathcal{O} := \langle O, \sqsubseteq, \perp^O\rangle$. Let* `SGIE` *be instantiated with F^V functions on an equivalence structure $V/\equiv$ that is isomorphic to O. Furthermore, suppose that this instance of* `SGIE` *is $\equiv$-equivalent to the instance of* `SGI` *with the F^O functions on the partial ordering $\mathcal{O}$. Thus we have the following results:*

- *if the F^O functions are inflationary, then every execution of* `SGIE` *with the F^V functions on V terminates;*

- *if the F^O functions are also monotone, then every execution of* `SGIE` *with the F^V functions and V terminates, by computing values which are all in the $\equiv$-class of the least common fixpoint of the F^O functions.* □

Variations of `SGIE`

All versions of `SGI` can be modified similarly and so generate a corresponding version of `SGIE`. However, in Chapter 4, we only deal with the following variations of `SGIE`:

- the `GI` algorithm (see Subsection 3.4.1) with equivalence, namely `GIE`;
- the `SGIIS` algorithm (see Algorithm 3.4.1) with equivalence, denoted by `SGIISE`.

All these algorithms share the same parameters, which are specified as follows:

- an equivalence structure, namely a set V and an equivalence binary relation $\equiv$ on it;
- $\perp^V$, an element of V;
- a finite set F^V of functions $f^V : V \mapsto V$;
- a subset $F^V_\perp$ of F^V that contains every F^V function f^V for which $f^V(\perp) \not\equiv \perp$;
- the *update* operator that selects and returns a subset of F functions.

The definitions and results given above for `SGIE` are easily extended to the cases of `GIE` and `SGIISE`. We leave the task to fill in the details to the reader.

3.5 Conclusions

3.5.1 Synopsis

This chapter presents a basic algorithm schema, `SGI`, and some of its variations. The `SGI` schema iteratively applies functions until a common fixpoint of theirs is found: the Common Fixpoint Axiom 3.3.1 provides a sufficient property for this, and characterises the basic strategy of `SGI`. Then Axioms 3.3.1 and 3.3.2 state sufficient properties for `SGI` to find the least common fixpoint of the functions and terminate, respectively. Notice that all those properties are encountered in most constraint propagation algorithms, see Chapter 4 below; there, the `SGI` schema or its variations are used as "templates" to explain and differentiate those algorithms.

Variations of the basic schema are thus studied in Section 3.4: they are differentiated in terms of *update* and properties of functions; these differences account for different strategies of the algorithms in Chapter 4. Besides, some of those algorithms use additional support structures, so to speak: i.e., they remove values from the given CSP domains or constraints by storing information in other structures. In those cases the `SGIE`, namely `SGI` on an set equipped with an equivalence relation, proves useful: first the algorithms are instantiated to `SGIE`; then the additional structures are "scraped away" through the adopted equivalence relation, so that `SGI` can be used to analyse those algorithms too. These instances of `SGI` iterate functions that only remove values from domains or constraints, and do it in a monotone and inflationary manner; thus we are able to transfer the results obtained for `SGI` instances to `SGIE` instances.

Some of the main variations of `SGI` and `SGIE` are summarised in the following table that contains in each cell, from left to right:

- a variation of `SGI` or `SGIE`;
- the related properties of functions;
- where the related *update* operator is characterised;
- where a variation is applied in Chapter 4, which deals with constraint propagation algorithms for CSPs — these are introduced in Chapter 2.

`SGI` & `SGIE` Variations	Properties of Functions	The *update* Operator	Where in Chapter 4
`SGIIS` `SGIISE`	idempotency, stationarity	Idempotency Lemma 3.4.3, Stationarity Lemma 3.4.9	`(H)AC-4`, `(H)AC-5`, `PC-4`
`GI`		Common Fixpoint Axiom 3.3.1	`(H)AC-1`, `PC-1`, $\mathtt{RC}_{(i,m)}$
`GIC`	commutativity	Commutativity Lemma 3.4.6	`AC-3` `PC-2`
`GIIS` `GIISE`	idempotency, stationarity	Idempotency Lemma 3.4.3, Stationarity Lemma 3.4.9	`KC`

Chapter 5 concerns itself with non-standard CSPs that allow to obtain optimal partial solutions, according to certain criteria: the original algorithm schema for constraint propagation is extended via `SGI`. In Chapter 2 and Sections 5.4, 5.5 of Chapter 5, we also apply the results of the present chapter as displayed in the following table: this shows how properties of functions or *update* are correlated to properties of algorithms in both Chapters 2 and 5.

Properties of *update* or Functions	Properties of Algorithms	Where in Chapter 4	Where in Chapter 5
Common Fixpoint Axiom 3.3.1	partial correctness	Corollaries 4.2.4, 4.2.6, 4.2.12, 4.2.15, 4.2.16, 4.3.5, 4.3.7, 4.3.12, 4.4.7, 4.5.4	Corollary 5.4.4
Monotonicity Axiom 3.3.2	confluence	as above	Corollaries 5.4.5 and 5.4.6
Inflationarity Axiom 3.3.2	termination	as above	Corollaries 5.4.7, 5.4.14 and 5.4.15

3.5.2 Discussion

Using a single framework for presenting constraint propagation algorithms makes it easier to verify and compare these algorithms. Again from a theoretical viewpoint, this approach allows us to separate the properties that concur in the definition of a constraint propagation algorithm: e.g., inflationarity is related to termination and absence of backtracking; monotonicity to confluence; stationarity, commutativity and idempotence explain optimised strategies for various constraint propagation algorithms. Preserving equivalence is another important property of those algorithms: in such a general setting, we cannot tackle it, since we study functions on "generic" sets, i.e., not on CSPs. Nonetheless, in Chapter 4, it is always easy to prove, by means of the adopted functions, that constraint propagation algorithms maintain equivalence.

From an applicative viewpoint, this approach allows us to parallelise constraint propagation algorithms in a simple and uniform way and result in a general framework for distributed constraint propagation algorithms; see [Mon00]. This shows that constraint propagation can be viewed as the coordination of cooperative agents. Additionally, such a general framework facilitates the combination of these algorithms, a property often referred to as solver cooperation or combination. Finally, the generic iteration algorithm `SGI` and its specializations can be used as a template for deriving specific constraint propagation algorithms in which specific scheduling strategies are employed.

Chapter 4

Constraint Propagation Algorithms

4.1 Introduction

4.1.1 Motivations

In general, satisfying a constraint problem means computing a solution to it. CSPs can be so satisfied by a generate-and-test procedure: each possible combination of assignments is generated and tested against the given constraints. More refined strategies rely on the backtracking method: variables are instantiated sequentially; when a partial assignment is found inconsistent with a constraint, backtracking is performed to the last instantiated variable and, if possible, another value gets assigned to it. There are several variations of this basic form of backtracking, for instance see [Bac01, KvB97]. However, the run-time performance of backtracking is in general exponential for most CSPs, see [Kum92]. In [KvB97], the authors propose a theoretical comparison of backtracking algorithms for CSPs, based on a graph representation of these and on the number of visited nodes and edges.

What emerges from the analysis of the basic backtracking algorithm is that the reason for its poor performance is due to "trashing": search keeps failing on the same type of subspace of the solution search space. Constraint propagation algorithms attempt to tackle this problem in various ways. For instance, constraint propagation algorithms, known as arc consistency algorithms, prune a CSP's *domains* from values that are inconsistent with the binary constraints of the CSP. Others, known as path consistency algorithms, prune inconsistent values from *binary constraints* of CSPs.

4.1.2 Outline

In this chapter, we present various algorithms for constraint propagation, and make use of the `SGI` schema (see Chapter 3) to explain these algorithms: by representing them as *instances* of `SGI` or one of its variations; by *analysing* them, applying the theoretical results studied in Chapter 3.

Instantiation. Every time we need to represent an algorithm as an instance of `SGI`, we need to specify for `SGI`:

- an appropriate set,
- functions and a suitable partial order on the function domain, or on an equivalent set (see Subsection 3.4.3),
- the *update* operator, which is in charge of returning the necessary functions to iterate in `SGI`.

Analysis. Thus the general results obtained for `SGI` and its variations are used to *analyse* constraint propagation algorithms. For instance, inflationarity of functions is related to termination of algorithms and the absence of backtracking. Stationarity (see Definition 3.4.8) explain why some algorithms do not repeat the same kind of pruning. But these are still too general statements; let us try to specialise and clarify them in the context of the algorithms in this chapter.

Classification. Constraint propagation algorithms differentiate for the way and the type of pruning they perform on CSPs: i.e., in the terminology of Chapter 3, they correspond to different domain functions (see p. 23) or constraint functions (see p. 24). Thus a first broad classification separates constraint propagation algorithms according to this criterion; the sections of the present chapter correspond to the different constraint propagation classes that so emerge.

Section 4.2 is dedicated to the so-called arc and hyper-arc consistency algorithms; the `SGI` schema of Chapter 3 explains how those algorithms remove values from domains, and how the effects of the removals are propagated. In Section 4.3, we describe and study the so-called path consistency algorithms by means of `SGI`; the analysis there conducted also highlights how inconsistencies, at the level of binary constraints, are inferred and propagated along a 2-constraint ordering on the problem (see Subsection 2.5.2). Sections 4.4 and 4.5 deal with generalisations of the above algorithm classes; again, the `SGI` schema is sufficiently general to cover also these cases, and explain how pruning is performed (via functions) and propagation of inconsistencies is carried over (in particular, via the *update* operator).

Separation. A further, subtler analysis detects how each algorithm of a given class enforces and reaches its level of consistency, and so differentiates such an algorithm from the others that pertain to the same class. For instance, in Section 4.2, we describe and analyse in total four arc consistency algorithms; their differences are easily grasped via the common language and framework of `SGI` function iterations (see Chapter 3). In fact, when these algorithms are shown to be all instances of the `SGI` schema, their differences can be explained in terms of diverse function iterations: e.g., through the use of diverse functions; through the use of some properties of functions like commutativity (see Definition 3.4.5), or stationarity (see Definition 3.4.8).

4.1.3 Structure

Section 4.2 is concerned with arc consistency and its generalisation, namely hyper-arc consistency. Then path consistency is described and studied in Section 4.3. We present an algorithm for k-consistency and discuss it in Section 4.4, and finally relational consistency in Section 4.5. This chapter is concluded by a summary table, see Section 4.6.

4.2 Arc and Hyper-arc Consistency

In this section, we deal with constraint propagation algorithms that only modifies domains of CSPs. We begin by introducing these notions, and then show how various algorithms for enforcing them can be recast in the framework of `SGI`.

First of all, let us introduce the main of those CSP properties that we deal with in this section: i.e., hyper-arc consistency as defined by Mohr and Masini, see [MM88].

DEFINITION **4.2.1.** Consider a CSP $P := \langle X, D, C\rangle$ and a constraint $C(s)$ of C. Then the *constraint* $C(s)$ is *hyper-arc consistent* if the following condition is met:

$$\text{for every } x_i \in s \text{ and } a \in D_i \text{ there exists } d \in C(s) \text{ such that } a = d\,[i]\,.$$

We call the CSP P *hyper-arc consistent* iff all its constraints are hyper-arc consistent.

In other words: a CSP is hyper-arc consistent if, for each variable x_i of the problem, each value in D_i (the x_i domain) is part of each constraint of the problem that involves x_i. For instance, the MAP COLOURABILITY PROBLEM in Subsection 2.3.1 is not hyper-arc consistent: in fact, the value *cyan* for x_1 is forbidden by the constraint (arc) on x_1 and x_2.

The better known notion of *arc consistency* of [Mac97] is obtained by restricting the above definition to the case of CSPs with only binary constraints; as in the case of the aforementioned MAP COLOURABILITY PROBLEM. Hence, that notion can be recast as follows: a *binary constraint* $C(x_i, x_j)$ is *arc consistent* if

$$\begin{aligned}&\text{for every } a \in D_i \text{ there exists } b \in D_j \text{ such that } (a, b) \in C(x_i, x_j),\\ &\text{for every } b \in D_j \text{ there exists } a \in D_i \text{ such that } (a, b) \in C(x_i, x_j).\end{aligned}$$

A CSP with only binary constraints is *arc consistent* if all of its constraints are.

EXAMPLE **4.2.2.** The MAP COLOURABILITY PROBLEM $P_{\mathtt{map}}$ of Subsection 2.3.1 is not arc consistent, hence hyper-arc consistent. In fact, if we choose the variable x_1 and look up for its values in D_1, we see that *cyan* is forbidden by the constraint on x_1 and x_2; also *blue* is forbidden by the constraint on x_1 and x_3. Therefore, an arc consistency algorithm will remove the value *cyan* and *blue* from D_1, so obtaining a new CSP, with the same scheme and constraints as $P_{\mathtt{map}}$, and domain $D_1 := \{aqua\}$. No value will be removed from the domain of x_2, since *cyan* is not forbidden by any constraints that involves x_2. On the contrary, the value *cyan* for x_3 will be removed from the domain of x_3 because of the constraint on x_2 and x_3. Therefore, the obtained CSP, on x_1, x_2, x_3, has the same constraints as the MAP COLOURABILITY PROBLEM $P_{\mathtt{map}}$ but domains which are subsets of the $P_{\mathtt{map}}$ ones: i.e., $D_1 = \{aqua\}$, $D_2 = \{cyan\}$, $D_3 = \{blue\}$.

4.2.1 The Basic Arc and Hyper-arc Consistency Algorithms

In this part, we describe and study the basic arc and hyper-arc consistency algorithms. First we introduce the basic algorithm for hyper-arc consistency, namely `HAC-1`, of which the one for arc consistency, namely `AC-1`, represents a special case. Then we show how `GI` can be instantiated to `HAC-1`, hence to `AC-1`. Finally we infer some properties concerning those basic algorithms by studying their `GI` instances.

The `HAC-1` and `AC-1` algorithms

The basic hyper-arc consistency algorithm `HAC-1` enforces hyper-arc consistency by choosing a variable domain and iteratively enforcing hyper-arc consistency on this; `AC-1` is like `HAC-1` but it enforces arc consistency, i.e., only binary constraints are taken in consideration. In what follows, we first instantiate `GI` to both `HAC-1` and `AC-1`, and then analyse these via the correlated instantiations of `GI`.

Instantiation

To prove that the algorithm `HAC-1` for hyper-arc consistency is an instance of the `GI` algorithm, we need to specify, in the order, the following components:

1. a partial ordering with bottom;

2. a finite set of functions over the partial ordering;

3. the *update* operator.

Partial ordering with bottom. Consider a CSP $P := \langle X, \boldsymbol{D}, \boldsymbol{C}\rangle$. As partial ordering for `HAC-1`, we adopt the domain ordering on P, written as

$$\langle \mathcal{D}(P), \sqsubseteq, \boldsymbol{D}\rangle,$$

and partially ordered by the reverse of the subset relation, see Subsection 2.5.2. Therefore, given two domain sets $B := B_1, \ldots, B_n$ and $B' := B_1, \ldots, B_n$ of problems in the domain ordering, we have

$$B \sqsubseteq B' \text{ iff } B_i \supseteq B'_i \text{ for every } i = 1, \ldots, n.$$

Remember that the input domain $\boldsymbol{D}$ is the bottom of such a domain ordering.

Functions. We associate a function $\sigma(x_i; s)$ to each domain D_i in $\boldsymbol{D}$, and constraint $C(s)$ of the given input problem P such that $x_i \in s$. Then we define the function $\sigma(x_i; s)$ as follows over the domain ordering: if B is a domain set in $\mathcal{D}(P)$, then $B' := \sigma(x_i; s)(B)$ differs from B at most in the domain B'_i of x_i, this being

$$B'_i := \Pi_i(C(s) \cap B\,[s]).$$

So B' is always greater than B with respect to $\sqsubseteq$, the domain order. Therefore each such function $\sigma(x_i; s)$ is trivially inflationary and monotone with respect to $\sqsubseteq$. Moreover, all these functions are idempotent, since intersection and projection are.

By taking into account only the binary constraints we obtain an analogous characterisation of arc consistency. The partial ordering is still the domain ordering but now on a binary CSP; see also Subsection 2.5.2. Then we associate two functions, $\sigma(x_i; x_i, x_j)$ and $\sigma(x_j; x_i, x_j)$, with, respectively, the domain D_i and D_j, for each problem constraint $C(x_i, x_j)$ of the input problem. Thereby we define such functions over $\mathcal{D}(P)$ as follows, where B is a domain set in the domain ordering on P:

- $B' := \sigma(x_i; x_i, x_j)(B)$ differs from B at most in the domain B'_i on x_i, since

$$B'_i := \Pi_i(C(x_i, x_j) \,\cap\, B_i \times B_j);$$

- $B' := \sigma(x_j; x_i, x_j)(B)$ differs from B at most in the domain B'_j on x_j, since

$$B'_j := \Pi_j(C(x_i, x_j) \,\cap\, B_i \times B_j).$$

Update. The *update* operator for `HAC-1` is characterised as follows:

$$update(G, F, \sigma(x_i; s), B) := F - G.$$

Clearly, the above characterisation of *update* satisfies Axiom 3.3.1. Indeed, it is not an optimal instantiation of *update* in terms of space and executions of the main **while** loop of `GI`. We shall see that the `HAC-3` algorithm has a better instantiation of the *update* operator, and hence that algorithm does not suffer from these drawbacks of `HAC-1`.

A similar characterisation of *update* can be given for `AC-1`, and we leave it to the reader.

Analysis

The use of the above defined functions is clarified by the following lemma, which also sums up the relevant properties of the $\sigma(x_i; s)$ functions.

LEMMA **4.2.3.**
(i) A CSP $P := \langle X, \boldsymbol{D}, \boldsymbol{C}\rangle$ is hyper arc consistent iff $\boldsymbol{D}$ is a common fixpoint of all the functions of the form $\sigma(x_i; s)$, hence the last one with respect to the domain order on P.
(ii) Each function $\sigma(x_i; s)$ is idempotent, monotone and inflationary with respect to the order of the domain ordering on P. □

Fix now a CSP P. By instantiating the `GII` algorithm with the above defined functions $\sigma(x_i; s)$, we get the basic arc consistency algorithm `AC-1`. So we can prove that this algorithm enjoys the following properties as a consequence of the theoretical results for `SGII`, hence for `GII` — see Theorem 3.4.4.

COROLLARY **4.2.4** (`HAC-1` AND `AC-1`). *Consider a finite CSP $P := \langle X, \boldsymbol{D}, \boldsymbol{C}\rangle$.*

- *Assume that P is not binary. Then the hyper-arc consistency algorithm* `HAC-1` *on P terminates by computing the greatest hyper-arc consistent problem that is equivalent to P; that is the least common fixpoint of the $\sigma(x_i; s)$ functions, defined as above.*

- *Suppose that P is binary. Then the arc consistency algorithm* `AC-1` *on P always terminates by computing the greatest arc consistent problem that is equivalent to P; that is the least common fixpoint of the $\sigma(x_i; x_i, x_j)$ and $\sigma(x_j; x_i, x_j)$ functions, defined as above.*

PROOF. We only need to observe that each $\sigma(x_i; s)$ function preserves equivalence and then apply the results of Theorem 3.4.4. As for the equivalence, it is sufficient to notice that projection does not remove solutions from a CSP; therefore, neither its composition with intersection does. □

4.2.2 The HAC-3 and AC-3 Algorithms

In the HAC-1 algorithm, each function $\sigma(x_i; s)$ is associated with a variable domain D_i and a constraint $C(s)$ on a scheme to which the variable belong; each time $\sigma(x_i; s)$ is applied and modifies its arguments, all functions of type σ that are associated with a constraint involving the variable x_i are added to the set G of functions to iterate. In this section we show how this information about the commutativity can be exploited to add less projection functions of the form $\sigma(x_i; s)$ to the set G. What follows was devised in [Apt00a].

Recall that, in Definition 3.4.5, we introduced the notion of commutativity between two functions, f and g, on the same domain O as follows:

$$fg(o) = gf(o) \text{ for all } o \in O.$$

First, it is worthwhile to note that not all pairs of HAC-1 functions commute. In general, functions like $\sigma(x_i; x_i, x_j)$ and $\sigma(x_k; x_i, x_k)$ do not need to commute; see [Apt00a] for an example of this phenomenon.

The following lemma clarifies which of the above functions commute.

Lemma **4.2.5** (Commutativity). *Consider a CSP P and a constraint $C(s)$ of P on the scheme s.*

- *For x_i, $x_j \in s$ the functions $\sigma(x_i; s)$ and $\sigma(x_j; s)$ commute.*
- *If $C'(t)$ is a constraint of P on a scheme t and the variable x_i occurs in both schemes s and t, then $\sigma(x_i; s)$ and $\sigma(x_i; t)$ commute.*

Proof. We only sketch how the first claim is proved, and refer the reader to [Apt00a] for a full proof.

Consider $\sigma(x_i; s)$ and $\sigma(x_j; s)$, for $i \neq j$. Let B be a domain set in the domain ordering of the given problem. The former function can only modify B_i, the domain of x_i; whereas the latter function can only modify B_j, the domain of x_j. Both functions do it by looking up for d in $C(s)$ such that $d \in B\,[s]$. Whenever $d \in B\,[s]$ and $d \notin C(s)$, $\sigma(x_i; s)$ and $\sigma(x_j; s)$ remove the projections of d from B_i and B_j, respectively. Instead, if $d \in C(s)$, neither $d\,[i]$ nor $d\,[j]$ are removed from B_i and B_j by $\sigma(x_i; s)$ and $\sigma(x_j; s)$, respectively. Therefore these functions commute. □

Fix now a CSP. We derive a modification of the hyper-arc consistency algorithm HAC-1 from Subsection 4.2.1 by instantiating, this time, the GIC algorithm schema, see Subsection 3.4.1 and Theorem 3.4.7. We use the same set of functions $\sigma(x_i; s)$ as for HAC-1. Additionally we employ the functional *Comm* (see Definition 3.4.5) that, given a function $\sigma(x_i; s)$ from F, returns the set of functions that $\sigma(x_i; s)$ commute with:

$$Comm(\sigma(x_i; s), F) := \{\sigma(x_j; s),\ \sigma(x_i; t) \,:\, x_j \in s \text{ and } x_i \text{ is in } s \text{ and } t\}\,.$$

By virtue of the Commutativity Lemma 4.2.5 each set $Comm(\sigma(x_i; s), F)$ satisfies the assumptions of Lemma 3.4.6.

By limiting oneself to the set of functions $\sigma(x_i; x_i, x_j)$ and $\sigma(x_j; x_i, x_j)$ associated with the binary constraints, we obtain an analogous modification of the corresponding arc consistency algorithm. Using now the commutative version `GIIC` of `GII`, we conclude that the `AC-3` algorithm enjoys the same properties as the `AC-1` algorithm. A more general conclusion holds for `HAC-1` and the instantiation of `GIIC` with the above functions $\sigma(x_i; s)$.

We can now state that the `HAC-3` and `AC-3` algorithms enjoy the following properties, which are immediate consequences of the Commutativity Lemma 4.2.5 and the theoretical results for `SGIIC`, hence for `GIIC`: i.e., Theorems 3.4.4 and 3.4.7.

COROLLARY **4.2.6** (`HAC-3` AND `AC-3`). *Consider a finite CSP* $P := \langle X, \boldsymbol{D}, \boldsymbol{C} \rangle$.

- *Assume that P is not binary. Thus the hyper-arc consistency algorithm* `HAC-3` *on P, namely* `GIIC` *with the above defined $\sigma(x_i; s)$ functions, always terminates by computing the greatest hyper-arc consistent problem that is equivalent to P.*

- *Suppose that P is binary. The arc consistency algorithm* `AC-3` *is an instance of* `GIIC` *on P with the above defined $\sigma(x_i; x_i, x_j)$ and $\sigma(x_j; x_i, x_j)$ functions. Thus it always terminates by computing the greatest arc consistent problem that is equivalent to P.* □

The difference between (`H`)`AC-3` and (`H`)`AC-1` relates to the different specifications of the *update* operator. As a consequence of this, the former will gain in execution time.

4.2.3 The `HAC-4` and `AC-4` Algorithms

In this part, we describe and study the `HAC-4` algorithm of [MM88] via the `SGIISE` — see p. 42 — and `SGIIS` algorithms — see p. 41. First we introduce the original algorithm; then we display the necessary set and functions for the algorithm `HAC-4` to become an instance of `SGIISE`; finally, we infer properties of `HAC-4` by studying an equivalent `SGIIS` instance.

The `AC-4` algorithm is `HAC-4` for binary constraints. We limit ourselves to the description and analysis of `HAC-4`, since `AC-4` is a specialisation of `HAC-4`. For a detailed analysis of `AC-4` via iterations of functions, we invite the reader to consult [Gen00].

Notice that the `HAC-4` and `AC-4` algorithms assume the input CSP to be normalised, i.e. to have at most one constraint on each scheme; see Subsection 2.4.1. Otherwise, the CSP is first normalised, and then propagation takes place on its normalisation.

The original algorithm

The HAC-4 algorithm enforces hyper-arc consistency by first constructing a set of elements, called G, that do not participate in any consistent instantiation for any of the input problem constraints. Then the propagation phase is carried on as in Algorithm A.4, see Appendix A.

In the initial phase of HAC-4, a construction of structures and an initial pruning take place. Each $a \in D_i$, for every variable domain x_i of the problem, is checked: i.e., for each constraint problem $C(s)$ such that $x_i \in s$, only all $d \in C(s)$ for which $d\,[x_i] = a$ holds are stored in $C(x_i, a; s)$. If one of the $C(x_i, a; s)$ is empty, then a is removed from D_i and the pair (x_i, a) is added to G.

In the HAC-4 algorithm propagation phase (see Algorithm A.4, Appendix A), a pair (x_i, a) is non-deterministically chosen from G and the effects of the removal of a from D_i are propagated through all $C(x_j, b; s)$.

Instantiation

Since we want to instantiate SGISE to HAC-4, we are in need to define what follows:

- an equivalence set;
- suitable functions;
- the *update* operator.

Equivalence set. Assume that $P := \langle X, \boldsymbol{D}, \boldsymbol{C}\rangle$ is a CSP with n variables. Thus consider subsets of the form $C(x_i, a; s)$ of each input constraint $C(s)$, i.e., one for each $x_i \in s$ and $a \in D_i$. Denote with $\boldsymbol{C}_{\mathtt{HAC-4}}$ a set of relations and sequences, defined as follows: create a sequence of the form $\langle x_i, a, s\rangle$, for each x_i, $a \in D_i$ and scheme s of X; then $\boldsymbol{C}_{\mathtt{HAC-4}}$ collects all pairs of the form $\langle C(s), (x_i, a, s)\rangle$. We usually denote each such pair as $C(x_i, a; s)$.

For instance, if $C(s) = C(x_1, x_2)$ and $D_1 = \{a\}$ and $D_2 = \{a, b\}$, then we have three "copies" of $C(s)$: $C(x_1, a; s)$, $C(x_2, a; s)$, $C(x_2, b; s)$.

Given $\boldsymbol{C}_{\mathtt{HAC-4}}$, we can finally define $\mathcal{C}_{\mathtt{HAC-4}}$ as follows:

- $\boldsymbol{C}_{\mathtt{HAC-4}}$ belongs to $\mathcal{C}_{\mathtt{HAC-4}}$;
- if E belongs to $\mathcal{C}_{\mathtt{HAC-4}}$, then E collects pairs of the form $E(x_i, a; s) := \langle E(s), \langle x_i, a; s\rangle\rangle$, where $E(s) \subseteq C(s)$, and $\langle x_i, a; s\rangle$ is specified as for $C(x_i, a; s)$. There is precisely one $E(x_i, a; s)$ in E, for each $C(x_i, a; s)$ in $\boldsymbol{C}_{\mathtt{HAC-4}}$.

Given $\mathcal{C}_{\mathtt{HAC-4}}$ as above, we define the equivalence structure $\langle P_{\mathtt{HAC-4}}, \equiv_{\mathtt{HAC-4}}\rangle$ as follows:

- its elements are subsets of tuples $\langle B, E\rangle$ in which: B is a domain set in the domain ordering on P (see Subsection 2.5.2); whereas E is in $\mathcal{C}_{\mathtt{HAC-4}}$;

- the equivalence relation $\langle B, E\rangle \equiv_{\texttt{HAC-4}} \langle B', E'\rangle$ holds iff $B = B'$;
- the input element $\perp_{\texttt{HAC-4}}$ is set equal to the pair $\langle \boldsymbol{D}, C_{\texttt{HAC-4}}\rangle$.

Indeed, the binary relation $\equiv_{\texttt{HAC-4}}$ is an equivalence relation, since it is reflexive, symmetric and transitive.

We introduce the functions over $P_{\texttt{HAC-4}}$ that we shall use to instantiate `SGISE` to `HAC-4` as below.

Functions. In the following, we define two types of functions for `HAC-4`: one type is used for the first **while** loop, the other type is for the second **while** loop.

- Let us define a function $\theta(x_i, a; s)$ for each input domain problem D_i and element $a \in D_i$. Such function is the identity everywhere except, possibly, on each $E(x_i, a; s)$ and B_i:

$$\left[\begin{array}{l} E'(x_i, a; s) := sel_{i=a} C(x_i, a; s), \\ B'_i := B_i - (\{a\} - \Pi_i(E'(x_i, a; s))), \end{array}\right.$$

  where $sel_{i=a}$ selects all the tuples d in $C(x_i, a; s)$ such that $d\,[i] = a$. In words: $E(x_i, a; s)$ is mapped into its subset $E'(x_i, a; s)$ of all $d \in C(x_i, a; s)$ — that is equal to $C(s)$ — the i-th component of which is a. Then B'_i differs in a from B_i if $E'(x_i, a; s)$ turns out to be empty.

- We define a function $\phi(x_i, a; s)$, for each $C(x_i, a; s)$. If $a \in B_i$, then $\phi(x_i, a; s)$ is the identity. Otherwise it is the identity everywhere except on each current domain B_j and current $E(x_j, b; s)$, for all $x_j \in s$ different from i and $b \in D_j$, that are mapped to B'_j and $E'(x_j, b; s)$, respectively, as follows:

$$\left[\begin{array}{l} E'(x_j, b; s) := E(x_j, b; s) - E(x_i, a; s); \\ B'_j := B_j - \bigcup_{b \in D_j} \Pi_j(E(x_j, b; s) - E'(x_j, b; s)). \end{array}\right.$$

  So, whereas $\theta(x_i, a; s)$ prunes the value a from the current domain B_i if it has no supports in $C(s)$, the function $\phi(x_i, a; s)$ takes care of propagating the effects of the removal of a from its domain. So $\phi(x_i, a; s)$ visits each set $E(x_j, b; s)$, for $x_j \in s$ different from x_i and $b \in B_j$; it removes the tuples in which a occurs from the supports of b, thus determines whether b should be removed.

NOTE **4.2.7.** Observe that all the functions of type ϕ as described in the latter item are the identity on the input problem: in fact, for each $\phi = \phi(x_i, a; s)$, the element a belongs to D_i, the input domain of x_i, by definition of $\phi(x_i, a; s)$. So the only functions that can modify the input problem are the θ functions defined in the former item. Besides, the θ functions that do not modify the input problem collapse into the identity function.

The *update* operator. We characterise the *update* operator as follows.

- If $\theta(x_i, a; s)(B, E) \not\equiv \langle B, E\rangle$, then $update(G, F, \theta(x_i, a; s), P))$ is the set of functions $\phi(x_i, a; t)$ from $(F - F_\perp) - G$.
- Similarly, if $\phi(x_i, a; s)(B, E) \not\equiv \langle B, E\rangle$, then the set $update(G, F, \phi(x_i, a; s))$ only contains all the functions $\phi(x_j, b; t)$ of $(F - F_\perp) - G$ that satisfy the following conditions: $x_j \in s$, $j \neq i$, $b \in B_j - B'_j$.

The following result is trivial, hence we state it as a fact.

FACT **4.2.8.** *The* `HAC` *algorithm is an instance of the* `SGIISE` *with the above defined equivalence set and functions.* □

Analysis

Now, we also set up to *study* `SGIISE`, once this is instantiated with the above set and functions. As for that, we need some technical lemmas; they are useful to prove that each `SGIISE` trace with θ and ϕ functions over the equivalence set $P_{\texttt{HAC-4}}$ can be mapped into an $\equiv_{\texttt{HAC-4}}$-equivalent `SGIIS` trace over $P_{\texttt{HAC-4}}/\equiv_{\texttt{HAC-4}}$, and that this computes the greatest hyper-arc consistent problem that is equivalent to the input problem.

We start considering the following functions that are then iterated by `SGIIS`. These functions are defined on the domain ordering of P, see Subsection 2.5.2:

- $\Theta(x_i, a; s)$ is the identity on each B_j with $j \neq i$, whereas it maps B_i to $B'_i := B_i - (\{a\} - \Pi_i(C(s)))$;
- $\Phi(x_i, a; s)$ is the identity on B_k with $k = i$ or $k \notin s$; whereas it maps every other B_j to $B'_j := B_j - E_j$, where E_j is the set of all $b \in B_j$ that enjoy both the following properties:

$$\left[\begin{array}{l} \exists d \in C(s) \text{ such that } d\,[i] = a \text{ and } d\,[j] = b, \\ \forall d' \in B(s) \cap C(s)\ d'\,[j] \neq b. \end{array}\right.$$

The *update* operator is characterised like in Lemmas 3.4.3 and 3.4.9:

- if $\Theta(x_i, a; s)(B) = B$, then $update(G, F, \Theta(x_i, a; s), P)$ is the empty set; otherwise $update(G, F, \Theta(x_i, a, s), P))$ is the set of functions $\Phi(x_i, a; t)$ from the set $(F - F_\perp) - G$;
- similarly, if $\Phi(x_i, a; s)(B) = B$, then $update(G, F, \Phi(x_i, a; s), P)$ is the empty set. If that is not the case, then the set $update(G, F, \Phi(x_i, a; s)$ only contains all the functions $\Phi(x_j, b; t)$ of $(F - F_\perp) - G$ that satisfy the following conditions: $x_j \in s$, $j \neq i$, $b \in B_j - B'_j$.

Clearly, the following statement holds, and it is proved as Lemma 4.2.3.

LEMMA **4.2.9.**
(i) A CSP $P := \langle X, \boldsymbol{D}, \boldsymbol{C} \rangle$ is hyper-arc consistent iff $\boldsymbol{D}$ is a common fixpoint of all functions of type Θ and Φ, hence the last one with respect to the domain order on P.
(ii) Each function of type Θ and Φ is stationary, monotone and inflationary with respect to the domain order on the input problem; besides, if a Θ function is the identity on the input problem, then it is the identity function. □

At this point, we need to prove the following two lemmas, concerning executions of `SGIIS` with the ϕ and θ functions, before we can state our main equivalence results.

LEMMA **4.2.10.** *Consider a CSP P and the associated P_{HAC-4} equivalence set. Let $F_\perp$ be the set of all the θ functions as above. Suppose that B and E are the current input in an execution of the second* **while** *loop. Then we have the following:*

- *if $d \in E(x_i, a; s)$ then $d\,[i] = a$;*
- *if $\phi(x_j, b; s) \in G$ then $b \notin B_j$;*
- *if $\phi(x_i, a; s)$ is the chosen function and $d \in E(x_j, b; s) - E(x_i, a; s)$, then $d\,[i] \neq a$ or $b \notin B_j$.*

PROOF. The first item relies on the definition of $\theta(x_i, a; s)$ and *update*. Similarly, the statement in the second item follows from the definition of $\theta(x_j, b; s)$ and *update*.

As for the last item, observe that every $d \in E(x_j, b; s)$ belongs to $C(s)$. If $d\,[i] = a$ then the fact that $d \notin E(x_i, a; s)$ is not due to $\theta(x_i, a; s)$ but to some function of the form $\phi(x_k, c; s)$. If $k \neq j$, then d cannot belong to $E(x_j, b; s)$ either, due to the same function $\phi(x_k, c; s)$. Hence $d\,[i] \neq a$. Finally, if $k = j$, then $b = d\,[j] = c$, hence $\phi(x_k, c; s)$ is equal to $\phi(x_j, b; s)$. Thus the second item yields that $b \notin B_j$. □

Given the above lemma, it is easy to conclude the following one concerning the equivalence of the `SGIISE` algorithm, with θ and ϕ functions, and `SGIIS` with Θ and Φ functions.

LEMMA **4.2.11.** *The* `SGIISE` *algorithm with $P_{\mathtt{HAC-4}}$, θ and ϕ functions is $\equiv_{\mathtt{HAC-4}}$-equivalent to* `SGIIS` *with $P_{\mathtt{HAC-4}}/\equiv_{\mathtt{HAC-4}}$, Θ and Φ functions.*

Proof. Consider an execution of the `SGIISE` algorithm with input Θ, Φ functions and $\perp := D$; notice that the domain of functions is isomorphic to $P_{\texttt{HAC-4}}/\equiv_{\texttt{HAC-4}}$. Then the equivalence of the iterations with θ and Θ functions is obvious. On the other hand, the equivalence of executions with ϕ and Φ functions follows from Lemma 4.2.10 and the above definition of *update*. In fact, if $a \notin B_i$, then

$$\left[\begin{array}{l} E'(x_j, b; s) := E(x_j, b; s) - E(x_i, a; s), \\ B'_j := B_j - \bigcup_{b \in D_j} \Pi_j(E(x_j, b; s) - E'(x_j, b; s)), \end{array}\right.$$

and Lemma 4.2.10 entails $B' = \Phi(B)$. A similar argument proves the opposite implication. □

Therefore, we get the following result concerning `HAC-4` and its specialisation `AC-4` to binary CSPs.

Corollary **4.2.12** (`HAC-4` and `AC-4`).
(i). Every execution of the `HAC-4` *algorithm terminates by computing the least fixpoint of the above defined Θ and Φ functions; i.e., the greatest hyper-arc consistent problem equivalent to the input one.*
(ii). Every execution of the `AC-4` *algorithm terminates by computing the least fixpoint of the above defined Θ and Φ functions on binary constraints; i.e., the greatest arc consistent problem equivalent to the input one.*

Proof. Theorem 3.4.10 and Lemma 4.2.9 imply that every execution of the `SGIIS` algorithm terminates, by computing the least common fixpoint of the Θ and Φ functions; this is the greatest hyper-arc consistent problem that is equivalent to the input one, due to Lemma 4.2.9 again. Thus Lemma 4.2.11 and Corollary 3.4.16 yield our corollary. □

4.2.4 The `HAC-5` and `AC-5` Algorithms

The `AC-5` algorithm of [vHDT92] is itself an algorithm schema, devised to enforce arc consistency on binary CSPs. As `AC-4`, the `AC-5` algorithm is split in two main procedures: in the initial phase, a construction of structures takes place; then the real propagation part starts, and elements that do not participate in any consistent instantiation to some problem constraints are iteratively removed from their respective domain.

The original algorithm

The algorithm by [vHDT92] is split into two main steps. We describe the following version of `AC-5`, as proposed in [vHDT92].

1. In the first step, for any constraint $C(x_i, x_j)$ of the given CSP, the procedure *arc-cons* creates a subset $\Delta(i)$ of D_i, for each x_i of the problem; the set $\Delta(i)$ collects all the elements $a \in D_i$ for which no elements b exist in D_j such that $(a, b) \in C(x_i, x_j)$. Then, for each $a \in \Delta(i)$, all triples $\langle (x_k, x_i), a\rangle$ such that $C(x_i, x_j)$ is a constraint of the problem are stored for future iterations, and the elements of the set $\Delta(i)$ are deleted from D_i.

2. In the second step, a triple $\langle (x_i, x_j), b\rangle$ is non-deterministically chosen and deleted from G; if b has been removed from D_j, then *loc-arc-cons* updates the set $\Delta(i) \subseteq D_i$ by adding all elements a that are no (more) supported in $C(x_i, x_j)$ by any element of D_j (after b has been removed from D_j); then, for each $a \in \Delta(i)$, all triples $\langle (x_k, x_i), a\rangle$ such that $C(x_k, x_i)$ is a constraint of the problem are stored for future iterations, and the elements of $\Delta(i)$ are removed from the domain D_i.

As pointed out in [vHDT92], `AC-5` is a generic algorithm: in fact, it can also be instantiated to `AC-4` by slightly changing the definition of the sets $\Delta(i)$. In the latter case, the functions that we used for `AC-4` are adopted. In case the definition of $\Delta(i)$ is chosen as stated item 1 and 2 above, we need a new equivalence relation and new functions to instantiate `SGI` to this version of `AC-5`.

Instance

In the following, we define the main ingredients to instantiate `SGISE` to the aforementioned version of `AC-5`:

- an equivalence set;
- suitable functions;
- the *update* operator.

Equivalence set. Consider a CSP $P := \langle X, \boldsymbol{D}, \boldsymbol{C}\rangle$ and define $P_{\texttt{AC-5}}$ as the set of pairs $\langle B, E\rangle$, where B and E are both domain sets of the domain ordering on P, see Subsection 2.5.2. Then the binary relation $\equiv_{\texttt{AC-5}}$ over $P_{\texttt{AC-5}}$ is defined as follows:

$$\langle B, E\rangle \equiv \langle B', E'\rangle \text{ iff } B = B'.$$

The element $\perp$ is set equal to $\langle \boldsymbol{D}, \boldsymbol{D}\rangle$, where $\boldsymbol{D}$ is the input domain set. Indeed, the above defined relation is of equivalence.

Functions. The set F contains two sorts of functions that we describe as below.

(θ). The function $\theta(x_i; x_i, x_j)$ corresponds to *arc_consistency*(x_i, x_j, Δ_i): in fact, the function $\theta(x_i; x_i, x_j)$ maps $\langle B, E\rangle$ into $\langle B', E'\rangle$ so that B and B' and, respectively, E and E' differ at most in B_i and B'_i as follows:

$$\left[\begin{array}{l} E'_i := E_i - \Pi_i(C(x_i, x_j)), \\ B'_i := B_i - E'_i. \end{array}\right.$$

The function $\theta(x_j; x_i, x_j)$ is characterised in a similar way; we leave it to the reader.

(ϕ). The function $\phi(x_j, b; x_i, x_j)$ corresponds to the `AC-5` procedure that is called *local_arc_consistency*(x_i, x_j, b, Δ_i). In fact, if $b \notin B_j$, then $\phi(x_j, b; x_i, x_j)$ maps $\langle B, E\rangle$ into $\langle B', E'\rangle$ so that B and B' and, respectively, E and E' differ at most in B_i and B'_i as follows:

$$\left[\begin{array}{l} E'_i := \{a \in B_i : P1(a, x_j, b; a; x_i, x_j) \text{ and } P2(x_j, b; a; x_i, x_j) \text{ hold } \}, \\ B'_i := B_i - E'_i, \end{array}\right.$$

where $P1(x_j, b; a; x_i, x_j)$ and $P2(x_j, b; a; x_i, x_j)$ are, respectively,

$$\exists d \in C(s)\ (d\,[i] = a \wedge d\,[j] = b), \quad (P1) \qquad \forall d \in C(s) \cap B(s)\ d\,[i] \neq a, \quad (P2)$$

Otherwise, if $b \in B_j$, the function $\phi(x_j, b; x_i, x_j)$ is the identity function. The function $\phi(x_i, a; x_i, x_j)$ is characterised in a similar manner.

The set F contains all the above defined functions. The subset $F_\perp$ contains the functions of type θ, which are the only F functions that can modify the input value $\langle \boldsymbol{D}, \boldsymbol{D}\rangle$. So $F - F_\perp$ contains all the remaining functions of type ϕ.

The *update* operator. The first part of the proposed version of the algorithm `AC-5` is encoded in the actions of inspecting and deleting all functions like $\theta(x_i; x_i, x_j)$ or $\theta(x_j; x_i, x_j)$ from G in `SGIISE`; when, for instance, $\theta(x_i; x_i, x_j)$ is chosen and applied, the operator *update* propagates the effects of the eventual reduction of B_i by adding the suitable functions $\phi(x_j, b; x_i, x_j)$ to G. Besides we want to instantiate `SGIISE` to the second part of the algorithm `AC-5` by means of the functions $\phi(x_j, b; x_i, x_j)$ of F. Therefore we define *update* as follows:

- if $\theta(x_i; x_i, x_j)(B, E) \not\equiv \langle B, E\rangle$, then *update*$(G, F, \theta(x_i; x_i, x_j), \langle B, E\rangle)$ is the subset of $F - G$ functions $\phi(x_i, a; x_i, x_k)$ or $\phi(x_i, a; x_k, x_i)$ such that $a \in E'_i$; the function $\theta(x_j, (x_i, x_j))$ is characterised analogously;

- if $\phi(x_j, b; x_i, x_j)(B, E) \not\equiv \langle B, E\rangle$, then *update*$(G, F, \phi(x_j, b; x_i, x_j), \langle B, E\rangle)$ is the subset of $F - G$ functions $\phi(x_i, a; x_i, x_k)$ or $\phi(x_i, a; x_k, x_i)$ of $F - G$ such that $a \in E'_i$; otherwise it is the empty set. An analogous characterisation can be given for $\phi(x_i, a; x_i, x_j)$.

Now it is trivial to check that `AC-5` becomes an instance of `SGIISE` by means of the above θ and ϕ functions.

FACT **4.2.13.** *The* `AC-5` *algorithm is an instance of* `SGIISE`. □

Analysis

It is as well easy to check that `SGIISE` with the above θ and ϕ functions is $\equiv_{\mathtt{AC-5}}$ equivalent to `SGIS` with the functions Θ and Φ for `AC-4`, as described in Subsection 4.2.3.

LEMMA **4.2.14.** *The* `SGISE` *algorithm with the above θ and ϕ functions is $\equiv_{\mathtt{AC-5}}$-equivalent to the* `SGIS` *algorithm with the Θ and Φ functions in Subsection 4.2.3, for binary constraints, and defined on the 2 domain ordering on P.* □

We have now all we need to prove the following results concerning `AC-5`.

COROLLARY **4.2.15** (`AC-5`). *Given a finite CSP P, the* `AC-5` *algorithm always terminates by computing the greatest arc consistent problem equivalent to P; that is, the least common fixpoint of the functions of type Θ and Φ defined as above.*

PROOF. Our thesis follows from Lemma 4.2.9 and Theorem 3.4.10, concerning the Θ and Ψ functions, via Lemma 4.2.14 and Corollary 3.4.16. □

A hyper-arc consistency version of `AC-5`

By exploiting the generality of `SGIISE`, we can extend `AC-5` to an algorithm schema that enforces hyper-arc consistency like `AC-5` enforces arc consistency. Indeed, it is sufficient to recast the above functions θ and σ for `AC-5` as follows.

(θ). The function $\theta(x_i; s)$, where $x_i \in s$ and $C(s)$ is a constraint of the input problem, maps $\langle B, E\rangle$ into $\langle B', E'\rangle$ so that B and B', and respectively E and E' differ at most in their i-th components as follows:

$$\left[\begin{array}{l} E'_i := E_i - \Pi_i(C(s)), \\ B'_i := B_i - E'_i. \end{array}\right.$$

(ϕ). The function $\phi(x_j, b; s)$ maps $\langle B, E\rangle$ into $\langle B', E'\rangle$ so that B and B', and respectively E and E' differ at most in their i-th components as follows, for every $x_i \in s$ different from x_j:

$$\left[\begin{array}{l} E'_i := \{a \in B_i : P1(x_j, b; a; x_i, x_j) \text{ and } P2(x_j, b; a; x_i, x_j) \text{ hold }\}, \\ B'_i := B_i - E'_i, \end{array}\right.$$

where $P1(x_j, b; a; x_i, x_j)$ and $P2(x_j, b; a; x_i, x_j)$ are, respectively,

$$\exists d \in C(s)\ (d\,[i] = a \wedge d\,[j] = b), \quad (P1) \quad \forall d' \in B(s) \cap C(s)\ d'\,[j] \neq b, \quad (P2)$$

Otherwise, i.e. if $b \in B_j$, the function $\phi(x_j, b; s)$ behaves like the identity function.

As for `AC-5`, we can derive the following result by Theorem 3.4.10, via Lemma 4.2.9, and Corollary 3.4.16.

Corollary **4.2.16** (`HAC-5`). *Given a finite CSP P, the* `SGIIS` *algorithm with the above defined functions θ and ϕ always terminates, by computing the greatest hyper-arc consistent problem equivalent to P.* □

4.3 Path Consistency

The notion of path consistency was introduced in [Mon74]. It is defined for a special type of CSPs. For simplicity we limit ourselves to binary CSPs: i.e., their constraints are only binary.

In Subsection 2.5.1, we introduced the join operation on constraints. In case of binary relations like $R \subseteq D_i \times D_j$ and $S \subseteq D_j \times D_k$, the composition of R and S, which is defined as follows

$$R \cdot S := \{(a, b) \,:\, (a, c) \in R \text{ and } (c, b) \in S\}\,,$$

amounts to a sequential application of join and projection to R and S. Note that, if $C(x_i, x_j)$ is a constraint on the variables x_i and x_j, and $C(x_j, x_k)$ is a constraint on the variables x_j and x_k, then $\Pi_{x_i,x_j}(C(x_i, x_j) \cdot C(x_j, x_k))$ is a constraint on the variables x_i and x_k. Whereas, if $k < j$, then the composition of $C(x_i, x_j)$ with $C(x_k, x_j)$ is not defined; yet, their join is. This is due to the commutativity of join, as defined in Subsection 2.5.1.

We first introduce the standard notion of path consistency, and then see how we can recast it through the join operation. In the following definition, instead of schemes, we have sets of variables.

Definition **4.3.1.** We call a CSP $P := \langle X, \boldsymbol{D}, \boldsymbol{C} \rangle$ path consistent if its 2 completion $\bar{P}^2$ enjoys the following property: for each set of distinct variables $\{x_i, x_j, x_k\}$ of P, we have

$$C(x_i, x_k) = C(x_i, x_k) \cap (C(x_i, x_j) \cdot C(x_j, x_k)).$$

In other words, a CSP is path consistent if, for each triple of its variables, x_i, x_j and x_k, the following holds: if $(a, c) \in C(x_i, x_k)$, then there exists $b \in D_j$ such that $(a, b) \in C(x_i, x_j)$ and $(b, c) \in C(x_j, x_k)$.

We provide an alternative characterisation of path consistency. In fact, in the above definition, relations of the form $C(x, x')$ are used, for any subset $\{x, x'\}$ of the considered sequence of variables. If $\{x, x'\}$ is not a scheme of the given CSP scheme of variables, then $C(x, x')$ is a supplementary relation that is not a constraint of the original CSP. At the expense of some redundancy we can rewrite the above definition so that only the constraints of the considered CSP are involved. This is the contents of the following characterisation, whose proof follows from the definition of join, projection, intersection and composition.

FACT **4.3.2** (ALTERNATIVE PATH CONSISTENCY). *A 2 complete CSP is* path consistent *iff the three following relations hold all true*

$$\left[\begin{array}{lcl} C(x_i, x_j) & := & C(x_i, x_j) \cap \Pi_{x_i,x_j}(C(x_i, x_j) \bowtie C(x_j, x_k)), \\ C(x_i, x_k) & := & C(x_i, x_k) \cap \Pi_{x_i,x_k}(C(x_i, x_k) \bowtie C(x_j, x_k)), \\ C(x_j, x_k) & := & C(x_j, x_k) \cap \Pi_{x_j,x_k}(C(x_i, x_j) \bowtie C(x_i, x_k)), \end{array}\right.$$

for each scheme x_i, x_j, x_k of the CSP variable. □

EXAMPLE **4.3.3.** The Temporal CSP in Subsection 2.3.4 is not path consistent: in fact, the relation $follows$ in $C(x_1, x_4)$ is not consistent with $\Pi_{x_1x_4}(C(x_1, x_2) \bowtie C(x_2, x_4))$. Thereby, path consistency algorithms will remove $follow$, and so reduce $C(x_1, x_4)$ to the singleton relation *precedes*.

4.3.1 The `PC-1` Algorithm

The `PC-1` is the basic algorithm for path consistency; it is presented in Appendix A. In the present subsection, we show that `PC-1` is an instance of `GI`, and then we analyse it through `GI` iterations.

Instantiation

To instantiate the `GII` algorithm to `PC-1` we need to specify the following components:

- a partial ordering,
- finitely many function on this,
- and the *update* operator.

We do it as below.

Partial ordering. To study path consistency, given a 2 complete CSP $P := \langle X, \boldsymbol{D}, \boldsymbol{C}\rangle$, we consider the 2-constraint ordering defined in Subsection 2.5.2.

Functions. Next, given a scheme x_i, x_j, x_k of the variables of P we introduce three functions on the partial ordering on P as below. Denote an element of the partial ordering with B.

- The function $\sigma(x_i, x_k; x_j)$ only modifies the binary constraint $B(x_i, x_k)$ of B, by returning the following set $B'(x_i, x_k)$:

$$B'(x_i, x_k) := B(x_i, x_k) \cap \Pi_{x_i, x_k}(B(x_i, x_j) \bowtie B(x_j, x_k)).$$

- The function $\sigma(x_i, x_j; x_k)$ only modifies the binary constraint $B(x_i, x_j)$ of B, by returning the following set $B'(x_i, x_j)$:

$$B'(x_i, x_j) := B(x_i, x_j) \cap \Pi_{x_i, x_j}(B(x_i, x_k) \bowtie B(x_j, x_k)).$$

- The function $\sigma(x_j, x_k; x_i)$ only modifies the binary constraint $B(x_j, x_k)$ of B, by returning the following set $B'(x_j, x_k)$:

$$B'(x_j, x_k) := B(x_j, x_k) \cap \Pi_{x_j, x_k}(B(x_i, x_j) \bowtie B(x_i, x_k)).$$

In what follows, when using a function $\sigma(x_j, x_k; x_i)$, we implicitly assume that the variables x_i, x_j, x_k are pairwise different and that $j < k$.

Finally, the notion of path consistency is clearly related to the common fixpoints of the above defined functions, and these are idempotent, monotone and inflationary over the constraint ordering. We collect these properties as in the following lemma, whose proof is just a consequence of the given characterisation of the functions σ as above.

LEMMA **4.3.4.**
(i). A CSP is path consistent if it a common fixpoint of the functions $\sigma(x_i, x_k; x_j)$ defined as above.
(ii). The functions $\sigma(x_i, x_k; x_j)$ are idempotent, monotone and inflationary over the 2-constraint ordering of the given CSP.
(iii). The functions $\sigma(x_i, x_k; x_j)$ do not remove solutions from the given CSP.

The *update* operator. The *update* operator is specified as follows:

$$update(\sigma(x_i, x_k; x_j), F, G, B) := \{\sigma(x_l, x_m; x_n) \,:\, |\,\{x_i, x_k\} \cap \{x_l, x_m, x_n\}\,| \geq 2\}\,.$$

In other words: each time a function $\sigma(x_i, x_k; x_j)$ modifies B, all functions that involve at least two of the variables x_i and x_k are added to G. Indeed, this is not an optimal instantiation of *update*. We shall see how it can be optimised by resorting to commutativity again, as in the case of the `AC-3` algorithm in Subsection 4.2.2.

Analysis

Given the above lemma, it is now easy to prove the following result, as a consequence of Corollary 3.3.12; it is sufficient to proceed as for `HAC-1` and `AC-1`. The reader is invited to consult [Apt99a, Apt00a] for a more detailed analysis.

COROLLARY **4.3.5** (`PC-1`). *Consider a* 2 *complete CSP* $P := \langle X, \boldsymbol{D}, \boldsymbol{C}\rangle$*, such that all constraints in* $\boldsymbol{C}$ *are finite. Let* P *be the input of* `PC-1`*. Then every execution of* `PC-1` *terminates, by computing the greatest path consistent problem, equivalent to* P*; i.e., the least common fixpoint of all the functions* $\sigma(x_i, x_k; x_j)$ *defined above.*

4.3.2 The `PC-2` Algorithm

In Section 4.2, we illustrated how the `AC-3` constitutes an improvement of `AC-1`, by a clever instantiation of the *update* operator. The `PC-2` algorithm is an improvement of `PC-1` much in the same spirit as `AC-3` is of `AC-1`.

In the `PC-1` algorithm, each time a function $\sigma(x_i, x_k; x_j)$ is applied and modifies its arguments, all functions associated with a triplet of variables including x_i and x_k are added to the set G of functions to iterate. This is not, indeed, an optimal choice of *update*.

In [Apt00a], the author proves how fewer functions can be added via *update*, by taking into account commutativity. To this end, the following lemma is proved. Its proof is as for the case of `AC-3`, thus we invite the reader to consult the latter or *ib.*

LEMMA **4.3.6.** *Consider a* 2 *complete CSP, involving among others the variables* x_i*,* x_j*,* x_k *and* x_l*. Then the functions* $\sigma(x_i, x_k; x_j)$ *and* $\sigma(x_i, x_k; x_l)$ *commute.* □

In other words, each pair of functions of the form $\sigma(x_i, x_k; \_)$ commute; this for every variable scheme $\langle x_i, x_k\rangle$ of the problem. The functional $Comm(\sigma(x_i, x_k; x_j))$ is then defined as follows, for each variable x_j such that $j \neq i, \neq k$ — consult also Definition 3.4.5:

$$Comm(\sigma(x_i, x_k; x_j), F) = \{\sigma(x_i, x_k; x_l) \,:\, x_l \text{ is different from } x_i,\, x_k\}\,.$$

Thus, for each function of type σ, the set $Comm(\sigma, F)$ contains precisely $n - 3$ elements, where n is the number of variables of the considered CSP. This quantifies the maximal gain obtained by using the commutativity information, loosely speaking: more precisely, *update* will need less functions at each iteration of the instance of `GIIC` for `PC-2`, than in the correlated instance of `GI` for `PC-1`.

By virtue of the above lemma and Theorem 3.4.7, the following result is easily proved.

Corollary **4.3.7** (PC-2). *Consider a 2 complete CSP $P := \langle X, \boldsymbol{D}, \boldsymbol{C}\rangle$, such that all constraints in $\boldsymbol{C}$ are finite. Let P be the input of* PC-2. *Then every execution of* PC-2 *terminates, by computing the greatest path consistency problem, equivalent to P; i.e., the least common fixpoint of all the functions $\sigma(x_i, x_k; x_j)$ defined above.* □

4.3.3 The PC-4 Algorithm

The PC-3 algorithm was devised in [MH86]. However here we refer to its corrected version, named PC-4, presented in [HL88]. This algorithm enforces path consistency on binary CSPs, by exploiting additional structures in the same fashion as (H)AC-4 and AC-5. So, in the following, we shall restrict our attention to binary CSPs, and prove that PC-4 is an instance of the SGIISE algorithm schema.

Notice that we assume that the input problem P is 2 complete — see Subsection 2.4.2. This will help us to reduce the overload of notations, and it is a minor change, since the initialisation phase of PC-4 reduces the input problem to its 2 completion. Therefore, constraint propagation takes place on a 2 complete problem.

The algorithm

The PC-4 algorithm is split into two parts:

- the first part of the algorithm consists in an initialisation of structures such that, at the end of it, the following properties hold: if E is the current constraint set and C the input one, then
  1. $(x_k : d\,,\, x_l : e) \in G$ iff $(x_k : d\,,\, x_l : e) \in C(x_k, x_l) - E(x_k, x_l)$,
  2. each $C(x_k : d\,,\, x_j : c\,;\, x_l)$ is a subset of D_l, and $e \in C(x_k : d\,,\, x_j : c\,;\, x_l)$ iff $(d, e) \in C(x_k, x_l)$ and $(e, c) \in C(x_l, x_j)$;
- in the second part, the real propagation phase takes place, see Algorithm A.5 in Appendix A. In fact, every tuple $(x_k : d\,,\, x_l : e) \in G$ is chosen and removed from G, each only once; then the pairs, affected by the removal of (d, e) from $C(x_k, x_l)$, are all inspected. So, if one of them has no more supporting pairs in $E(x_k, x_l)$, it gets removed from its corresponding binary constraint; then it is added to G in order to propagate the effect of its removal.

We have slightly changed PC-4 (see Algorithm A.5 in Appendix A) for this makes it easier to instantiate SGIISE to PC-4. Notice that this new version is equivalent to the original one by [HL88] and retains its worst time and space complexity; the difference is in the use of structures of the form $C(x_k : d\,,\, x_j : c\,;\, x_l)$ instead of so-called counters.

Instantiation

Equivalence set. We can assume that the input CSP $P := \langle X, D, C\rangle$ is 2 complete, see Subsection 2.4.2; else we add the necessary constraints to it.

While arc consistency algorithms remove elements from domains, path consistency algorithms such as `PC-1` and `PC-2` (see Subsections 4.3.1 and 4.3.2, respectively) propagate constraints by modifying binary constraints; so does `PC-4`. However, contrary to `PC-2` and `PC-1`, the `PC-4` algorithm has also support structures, like `AC-4`, `AC-5` and their respective hyper-arc versions. Those structures are used to avoid checking, more than once, that a given pair is consistent with the input constraints in $P := \langle X, D, C\rangle$.

In order to define the equivalence set, for instantiating `SGIIE` to `PC-4`, we introduce supplementary structures as follows. For each scheme $\langle x_i, x_j\rangle$ of the problem, variable x_k such that $i,\ j \neq k$, we put, for each $a \in D_i$, $b \in D_j$, and $c \in D_k$:

$$
\begin{aligned}
&E(x_i : a\,,\, x_k : c\,;\, x_j) := \langle E_j, \langle i, a, k, c, j\rangle\rangle && \text{if } i < k \text{ and } E_j \subseteq D_j, && \text{(IK)}\\
&E(x_k : c\,,\, x_i : a\,;\, x_j) := \langle E_j, \langle k, c, i, a, j\rangle\rangle && \text{if } k < i \text{ and } E_j \subseteq D_j; && \text{(KI)}\\
&E(x_k : c\,,\, x_j : b\,;\, x_i) := \langle E_i, \langle k, c, j, b\rangle\rangle && \text{if } k < j \text{ and } E_i \subseteq D_i, && \text{(KJ)}\\
&E(x_j : b\,,\, x_k : c\,;\, x_i) := \langle E_i, \langle j, b, k, c, i\rangle\rangle && \text{if } j < k \text{ and } E_i \subseteq D_i. && \text{(JK)}
\end{aligned}
$$

Let E denote the set that contains all such structures: precisely, either (IK) if $i < k$ or (KI) otherwise, and (KJ) if $k < j$ or (JK) otherwise; this for each scheme $\langle x_i, x_j\rangle$, x_k such that $k \neq i$, $a \in D_i$, $b \in D_j$ and $c \in D_k$. Thus $\mathcal{C}_{\texttt{PC-4}}$ collects all sets like E, whose elements are structures as above.

Finally, we can define the equivalence set $P_{\texttt{HAC-4}}$, that contains all pairs $\langle B, E\rangle$ defined as follows:

- B belongs to the 2-constraint ordering on the problem P;
- E belongs to $\mathcal{C}_{\texttt{PC-4}}$, i.e., is a collection of structures as above.

The equivalence relation $\equiv_{\texttt{HAC-4}}$ we need to define is, clearly, the following one: $\langle B, E\rangle \equiv \langle B', E'\rangle$ iff $B = B'$.

Functions. As in the cases of `(H)AC-4` and `(H)AC-5`, there are two types of functions: the former, denoted by θ, is used in the first **while** loop of `SGIIE`; the latter, denoted by ϕ, is used in the second **while** loop of `SGIIE`. We describe them as below.

The θ functions. For each $(a, c) \in C(x_i, x_k)$ and $j = 1, \ldots, n$ different from i and k, we define a function $\theta(x_i : a\,,\, x_k : c\,;\, x_j)$ that is the identity everywhere except, possibly, on the following sets:

- the subset $E(x_i : a\,,\, x_k : c\,;\, x_j)$ of D_j is mapped into the subset $E'(x_i : a\,,\, x_k : c\,;\, x_j)$ of D_j, such that $b \in E'(x_i : a\,,\, x_k : c\,;\, x_j)$ iff both the following properties hold:

$$\left[\begin{array}{ll} (a,b) \in C(x_i,x_j) & \text{if } i<j, \text{ otherwise } (b,a) \in C(x_j,x_i), \\ (b,c) \in C(x_j,x_k) & \text{if } j<k, \text{ otherwise } (c,b) \in C(x_j,x_k); \end{array}\right.$$

- the subset $B(x_i,x_k)$ of the input $C(x_i,x_k)$ is mapped into its subset $B'(x_i,x_k)$ so that

$$B'(x_i,x_k) := \begin{cases} B(x_i,x_k) - \{(a,c)\} & \text{if } E'(x_i : a\,,\, x_k : c\,;\, x_j) = \emptyset, \\ B(x_i,x_k) & \text{else.} \end{cases}$$

The ϕ functions. For each $(a,c) \in C(x_i,x_k)$ and $j = 1,\ldots,n$ different from i and k, we define a function $\phi(x_i : a\,,\, x_k : c\,;\, x_j)$ that is the identity if $(a,c) \in B(x_i,x_k)$; else, it is the identity almost everywhere except, possibly, on the following sets — where b ranges over D_j:

- If $i<j$ each subset $E'(x_i : a\,,\, x_j : b\,;\, x_k)$ of D_k is mapped into its subset

  $$E'(x_i : a\,,\, x_j : b\,;\, x_k) := E(x_i : a\,,\, x_j : b\,;\, x_k) - \{c\}\,,$$

  else each $E'(x_j : b\,,\, x_i : a\,;\, x_k)$ of D_k is mapped into its subset

  $$E'(x_j : b\,,\, x_i : a\,;\, x_k) := E(x_j : b\,,\, x_i : a\,;\, x_k) - \{c\}\,.$$

  Similarly, if $j<k$ each $E'(x_j : b\,,\, x_k : c\,;\, x_i)$ is mapped into

  $$E'(x_j : b\,,\, x_k : c\,;\, x_i) := E(x_j : b\,,\, x_k : c\,;\, x_i) - \{a\}\,,$$

  else each $E'(x_k : c\,,\, x_j : b\,;\, x_i)$ is mapped into

  $$E'(x_k : c\,,\, x_j : b\,;\, x_i) := E(x_k : c\,,\, x_j : b\,;\, x_i) - \{a\}\,.$$

- Then the set $B(x_i,x_j)$ is mapped into

  $$B'(x_i,x_j) := \begin{cases} B(x_i,x_j) - \{(a,b)\} & \text{if } E'(x_i : a\,,\, x_j : b\,;\, x_k) = \emptyset, \\ B(x_i,x_j) & \text{else,} \end{cases}$$

  if $i<j$, else the set $B(x_j,x_i)$ is mapped into

  $$B'(x_j,x_i) := \begin{cases} B(x_j,x_i) - \{(b,a)\} & \text{if } E'(x_j : b\,,\, x_i : a\,;\, x_k) = \emptyset, \\ B(x_j,x_i) & \text{else.} \end{cases}$$

Similarly, the set $B(x_j, x_k)$ is mapped into

$$B'(x_j, x_k) := \begin{cases} B(x_j, x_k) - \{(b, c)\} & \text{if } E'(x_j : b\,,\, x_k : c\,;\, x_i) = \emptyset, \\ B(x_j, x_k) & \text{else.} \end{cases}$$

if $j < k$, else the set $B(x_k, x_j)$ is mapped into

$$B'(x_k, x_j) := \begin{cases} B(x_k, x_j) - \{(c, b)\} & \text{if } E'(x_k : c\,,\, x_j : b\,;\, x_i) = \emptyset, \\ B(x_k, x_j) & \text{else.} \end{cases}$$

The *update* **operator** Given the above functions of type θ and ϕ, we can define the update operator as follows:

(θ). if $\Pi_1\theta(x_i : a\,,\, x_k : c\,;\, x_j)(B, E) \neq B$, then *update* adds to G all the ϕ functions of the form $\phi(x_i : a\,,\, x_k : c\,;\, x_l)$, for $l \neq i,\ k,\ j$;

(ϕ). if $\Pi_1\phi(x_i : a\,,\, x_k : c\,;\, x_j)(B, E) \neq B$, then *update* adds to G all the ϕ functions of the following form: if $i < j$, all functions $\phi(x_i : a\,,\, x_j : b\,;\, x_l)$ for $(a, b) \in E(x_i, x_j) - E'(x_i, x_j)$ and $l \neq i,\ j,\ k$, else all functions $\phi(x_j : b\,,\, x_i : a\,;\, x_k)$ for $(b, a) \in E(x_j, x_i) - E'(x_j, x_i)$ and $l \neq i,\ j\ k$; if $j < k$, all functions $\phi(x_j : b\,,\, x_k : c\,;\, x_l)$ for $(b, c) \in E(x_j, x_k) - E'(x_j, x_k)$ and $l \neq i,\ j,\ k$, else all functions $\phi(x_k : c\,,\, x_j : b\,;\, x_l)$ for $(c, b) \in E(x_k, x_j) - E'(x_k, x_j)$ and $l \neq i,\ j,\ k$.

Now it is easy to check that the following statement holds.

FACT **4.3.8.** *The* `PC-4` *algorithm is an instance of* `SGIISE` *whenever this algorithm iterates the above defined functions of type θ and ψ.*

Analysis

As in the case of `(H)AC-4` and `(H)AC-5`, we need to define functions on the quotient set $P_{\equiv_{\mathtt{HAC-4}}}/\equiv_{\mathtt{HAC-4}}$ to study `PC-4` by means of `SGIIS`. Hence we define functions of two types, Θ and Φ, as below.

In order to obtain a more compact notation, we introduce the following shorthand in the remaining of this subsection.

CONVENTION **4.3.1.** For every pair of distinct $i,\ j = 1, \ldots, n$, let $\overrightarrow{B(x_i, x_j)}$ denote $B(x_i, x_j)$ if $i < j$, else $B(x_j, x_i)$.

Thus $\overrightarrow{B(x_i, x_j)} = \overrightarrow{B(x_j, x_i)}$, and this will simplify the presentation of the functions for `SGIIS` as below.

The Θ functions. For each pair $(a, c) \in C(x_i, x_k)$ in the 2 completion of P, and $j = 1, \ldots, n$ different from i and k, we define a function $\theta(x_i : a\,,\, x_k : c\,;\, x_j)$

that is the identity if (a, c) belongs to $\Pi_{x_i,x_k}(\overrightarrow{C(x_i, x_j)} \bowtie \overrightarrow{C(x_j, x_k)})$; else it is the identity almost everywhere except, possibly, on the set $B(x_i, x_k)$ that is mapped into $B'(x_i, x_k)$, defined as follows:

$$B'(x_i, x_k) := B(x_i, x_k) - \{(a, c)\}\,.$$

The Φ *functions*. For each pair $(a, c) \in C(x_i, x_k)$ in the 2 completion of P, and $j \neq i,\ k$, we define a function $\Phi(x_i : a\,,\, x_k : c\,;\, x_j)$ that is the identity if $(a, c) \in B(x_i, x_k)$; else it is the identity almost everywhere except, possibly, on the constraint on the variables x_i and x_j, and on the constraint on the variables x_j and x_k:

$$\left[\begin{array}{lcll} B'(x_i, x_j) & := & B(x_i, x_j) \cap \Pi_{x_i,x_j}(B(x_i, x_k) \bowtie_{k\neq c} \overrightarrow{B(x_k, x_j)}) & \text{if } i < j, \\ B'(x_j, x_i) & := & B(x_j, x_i) \cap \Pi_{x_j,x_i}(B(x_i, x_k) \bowtie_{k\neq c} \overrightarrow{B(x_k, x_j)}) & \text{else,} \end{array}\right.$$

where $\bowtie_{k\neq c}$ results from the composition of first $\bowtie$ and then $sel_{k\neq c}$;

$$\left[\begin{array}{lcll} B'(x_j, x_k) & := & B'(x_j, x_k) \cap \Pi_{x_j,x_k}(\overrightarrow{B(x_i, x_j)} \bowtie_{i\neq a} B(x_i, x_k)) & \text{if } j < k, \\ B'(x_k, x_j) & := & B'(x_k, x_j) \cap \Pi_{x_k,x_j}(\overrightarrow{B(x_i, x_j)} \bowtie_{i\neq a} B(x_i, x_k)) & \text{else,} \end{array}\right.$$

where $\bowtie_{i\neq a}$ results from the composition of first $\bowtie$ and then $sel_{i\neq a}$.

Consider now the 2-constraint ordering on P, see Subsection 2.5.2. We summarise the main properties of the above functions Θ and Σ as in the following lemma.

LEMMA **4.3.9.** *The above defined* Θ *and* Φ *are monotone, stationary and inflationary on the* 2*-constraint order on the input problem; besides, if a* Θ *function is the identity on the input problem, then it is the identity function.* □

Given Lemma 4.3.9 we can infer the following result as a corollary of Theorem 3.4.10.

COROLLARY **4.3.10.** *Every execution of* `SGIIS`*, with input a finite CSP* P *and the above defined functions of type* Θ *and* Φ*, always terminates by computing the greatest path consistent problem that is equivalent to* P*.* □

The following lemma can be proved by an argument similar to that used in Lemma 4.2.11 for `AC-4`.

LEMMA **4.3.11.** *Every* `SGISE` *execution with* θ *and* ϕ *functions is* $\equiv_{\texttt{PC-4}}$*-equivalent to an* `SGIS` *iteration with* Θ *and* Φ *functions.* □

Now we have all the technical results for proving the following statement.

COROLLARY **4.3.12** (`PC-4`). *The* `PC-4` *algorithm, with input a finite CSP P, always terminates by computing the least common fixpoint of the above defined Θ and Φ functions, that is the greatest path consistent problem, equivalent to the input problem P.*

PROOF. The claim follows by Lemma 4.3.11, Corollaries 3.4.16 and 4.3.10. □

NOTE **4.3.13.** Note the difference in the termination conditions for `PC-1` or `PC-2` versus `PC-4`. In the former two cases, we only need that the constraints are finite to ensure termination; hence, those algorithms can be applied and terminate in the case of the Temporal CSP in Subsection 2.3.4. However, this is not the case for `PC-4`: in fact the termination condition in Corollary 4.3.12 assumes that also domains are finite. The reason is easy to explain via the functions used for `PC-4` in `SGIIISE` or `SGIIIS`: these functions are parametrised by domain elements, hence we have a finite number of F functions only if the input CSP domains are finite.

4.4 Local Consistency

4.4.1 Local Consistency as k Consistency

In Section 4.2 and 4.3, we defined two properties of CSPs that gave rise to a number of constraint propagation algorithms: arc and path consistency. Freuder generalised both those properties to a general form of *local consistency* in [Fre78]. There are two versions of this notion, a weak and a strong one, defined as below.

Weak consistency

Consider a CSP $P := \langle X, \boldsymbol{D}, \boldsymbol{C} \rangle$ with n variables in X, and a scheme s of X of length k. An assignment d for s is k *consistent* iff it satisfies every constraint $C(s')$ of P over a scheme s' of s. So, different levels, hence notions of local consistency can be defined for the same problem $P := \langle X, \boldsymbol{D}, \boldsymbol{C} \rangle$:

- the problem P is 1 consistent, or *node consistent*, iff, for every variable x_i of P, the unary constraint $C(x_i)$ on x_i is contained in the domain D_i of x_i;
- given $1 < k \leq |X|$, P is k consistent iff every $(k-1)$ consistent assignment d for P on s can be extended to a k consistent instantiation, for every possible extension $s' = s \cup x_j$ of s.

Suppose that k is equal to 2. Then arc consistency on binary CSPs, that are node consistent, clearly coincide with 2 consistency. The same holds for path consistency: a binary CSP, node consistent, is path consistent iff it is 3 consistent. We summarise these properties as follows.

FACT **4.4.1.**
(i). *A binary CSP,* 1 *consistent, is arc consistent iff it is* 2 *consistent.*
(ii). *A binary CSP,* 1 *consistent and* 2 *complete, is path consistent iff it is* 3 *consistent.* □

Strong local consistency

A stronger notion of local consistency can be defined as follows: a problem $P := \langle X, \boldsymbol{D}, \boldsymbol{C} \rangle$ is *strongly k-consistent*, for $1 \leq k < |X|$, iff it is j-consistent for every $j \leq k$.

Notice that requiring a problem to be strongly k consistent is by far more demanding than requiring it to be k consistent. A problem can be trivially $k+1$ consistent, if there are not k consistent instantiations, which is not the case for strong k consistency. Consider the following example.

EXAMPLE **4.4.2.** Let P be the CSP on two variables, x_1 and x_2, and x_3, domains equal to $\{0, 1\}$, and constraints on each scheme of two variables, forbidding that these are equal (see Example 2.5.4). This problem is clearly arc consistent, hence 2 consistent: in fact, for each variable, there is an instantiation for it that satisfies all binary constraints of P. However, P is not strongly 2 consistent, nor consistent.

As the above toy example suggests, if a problem is "sufficiently" strong consistent, then it is also consistent. We make the above claim precise as follows. The proof is easy, and the requirement that at least one domain should not be empty is fundamental to guarantee the existence of a 1 consistent instantiation; see also [Fre78].

FACT **4.4.3.** *Consider a CSP P with $k \geq 1$ variables. If P is strongly k consistent, with at least one not-empty domain, then P is consistent.* □

Thus, a strongly n consistent CSP is globally consistent, i.e., any consistent instantiation of a scheme of the variables can be extended to a consistent instantiation of all of the variables *without backtracking*.

4.4.2 The `KS` Algorithm

The `KS` algorithm by Cooper [Coo89] is an optimisation of the synthesis algorithm by Freuder [Fre78]. Both algorithms enforce strong k consistency over a CSP; the former can enforce k consistency with a minor simplification. We shall account for the whole strong k consistency algorithm by Cooper here, and explain how the `GIISE` schema can be instantiated to it.

In the following part, we need to extend a scheme or remove a variable from it, i.e. to make use of both projection and join, see Subsection 2.5.1. Hereby, we remind the following abbreviations: if s is a scheme of variables and $x_j \notin s$, then $t := s \cup x_j$ will denote the scheme on the set of variables of s plus x_j. Similarly, if x_i is one of the variables in s, then $r := s - \{x_i\}$ will stand in for the scheme on the variables of s minus x_i. Finally, if $d \in C(s)$, the variable x_j does not occur in s and $a \in D_j$, then $e := d \bowtie a$ is the tuple of $D[s \cup \{x_j\}]$ such that $e[s] = d$ and $e[j] = a$.

The original algorithm

As in the case of `PC-4`, also the Cooper algorithm enforces propagation at the level of constraints by exploiting additional structures for efficiency reasons; namely to store already checked values.

The algorithm by Cooper is split in two main sub-programs: the initialisation process takes place in the first step; then propagation is achieved by iteratively pruning i inconsistent values, for all $i \leq k$.

However, the first sub-program of `KS` is only meant to construct structures, there is no pruning of values. There, the `KS` algorithm reduces the input problem P to a k strong complete one, for some k not greater than the number of variables in P; so to an equivalent problem that has precisely one constraint per scheme, the length of which is not greater than k, see also Subsection 2.4.2. In Appendix A, we present the initialisation and propagation phases in Algorithm A.1; these are slightly modified versions of the original, in which counters (e.g. $Counter[d, r, j]$) are used to store the number of support values in place of the support values themselves, as we instead do (e.g. via $C(d, r, j)$). In the second sub-program, a pruned tuple d is chosen and the effects of its removal from $C(t)$ are propagated in two stages. Let i be the length of t. If $i < k$, first all $(i + 1)$ consistent instantiations d' such that $d'[t] = d$ are considered; then, if $i > 1$, all $(i - 1)$ consistent instantiations d' on schemes s of t are checked for supports if $d' = d[s]$.

Therefore we shall overlook the initialisation phase of `KS` when defining functions to instantiate `SGIISE`. Nevertheless, we are able to devise functions for `SGIE` that account for the optimal behaviour of the propagation phase of `KS`, compared to the synthesis algorithm by Freuder; in fact, those functions are defined on a set that exploits further structures than constraint orderings. Hence we shall define an appropriate equivalence relation on the domain of those functions, and show that we can devise a constraint ordering on the quotient set. Then we shall study this instance of `SGIISE` by passing to the quotient set and functions on this for `SGIS`.

Instantiation

The equivalence structure. Consider a CSP $P := \langle X, \boldsymbol{D}, \boldsymbol{C} \rangle$ and assume that it is strongly k complete. Remind, from Subsection 2.4.2, that the process of strong k completion amounts to constructing an equivalent problem that has the same variable scheme and domain set as the original one, but precisely one constraint over each scheme.

Now, let $P_{\mathtt{KS}}$ be the class of pairs $\langle B, E\rangle$ in which B is a constraint set in the strong k-constraint ordering on P (see Subsection 2.5.2), and E is a family of structures, defined as follows: for each scheme s on X, $x_j \notin s$ and $d \in C(s)$, precisely one structure of the form

$$E(d, s, x_j) := \langle E_j, \langle d, s, x_j\rangle\rangle,$$

for E_j a subset of D_j, belongs to E.

The binary relation $\equiv_{\mathtt{KS}}$ on $P_{\mathtt{KS}}$ is defined as follows:

$$\langle B, E\rangle \equiv_{\mathtt{KS}} \langle B', E'\rangle \text{ iff } B = B'.$$

Indeed, $\equiv_{\mathtt{KS}}$ is an equivalence relation and the quotient set $P_{\mathtt{KS}}/{\equiv_{\mathtt{KS}}}$ is isomorphic to the universe O_k of the strong k-constraint ordering on P.

Functions. For each scheme s of X of length i and $d \in D\,[s]$, we define a function $\phi(d, s, i)$ that propagates the i inconsistent instantiation d to all the $i-1$ and $i+1$ instantiations. Namely, if $\langle B, E\rangle$ is the input to $\phi(d, s, i)$, then the output value $\langle B', E'\rangle$ differs from $\langle B, E\rangle$ at most in the following components if $d \notin B(s)$, else $\phi(d, s, i)$ is the identity everywhere:

- in case i, the length of s, is less than k, then $\phi(d, s, i)$ considers every $x_k \notin s$ and the resulting join scheme $t := s \cup \{x_k\}$, and maps each such $B(t)$ into

$$B'(t) := B(t) - \bigcup_{a \in D_k} \{d \bowtie a\}\,;$$

- if i, the length of s, is greater than 1, then $\phi(d, s, i)$ considers every $x_j \in s$, the resulting projection scheme $r := s - \{x_j\}$, and modifies the sets $E(r, e, j)$, for each $e \in D(r)$, and $B(r)$ as follows:

$$\left[\begin{array}{lcl} E'(r, e, j) & := & E(r, e, j) - \{d\,[j]\}\,, \\ B'(r) & := & B(r) - \{e \,:\, E'(r, e, j) = \emptyset\}\,. \end{array}\right.$$

The *update* operator. At this point, we are left to characterise the *update* operator: if $\Pi_1\phi(e, s, i)(B) \neq B$, then *update* adds to G all the remaining F functions $\phi(e, r, i-1)$ or $\phi(e', t, i+1)$ such that $e \in B(r) - B'(r)$ and $e' \in B(t) - B'(t)$.

Given the above definitions of the functions ϕ and *update*, we can state the following result as fact.

FACT **4.4.4.** *The propagation phase of Algorithm A.1 in Appendix A is an instance of* SGIISE*: $\perp$ is the input $P_{\mathtt{KS}}$; the set F collects all the ϕ functions as defined above; the set $F_\perp$ collects all the $\phi(d, s, i)$ functions such that $d \in D\,[s] - C(s)$; the update operator is instantiated as above.*

Analysis

Given the CSP P, we have to define functions, say Φ, over the strong k-constraint ordering on P. If we let O_k denote the universe of such constraint ordering on P, then, clearly, O_k is isomorphic to $P_{\mathtt{KS}}/\equiv_{\mathtt{KS}}$. Moreover, we want each `GIISE` iteration with the functions ϕ over $P_{\mathtt{KS}}$ to be $\equiv_{\mathtt{KS}}$-equivalent to an `GIIS` iteration with the functions Φ over O_k. Therefore, we define such functions Φ as below.

Functions over O_k. We define a set of functions Φ over the strong k-constraint ordering on P. For each scheme s of length i of X and $d \in C(s)$, let $\Phi(s, d, i)$ be the identity if $d \in B(s)$, else it modifies only the following subsets of the input B:

- if i, the length of s, is strictly less than k, then $\Phi(s, d, i)$ considers every $x_k \notin s$ and the resulting join scheme $t := s \cup \{x_k\}$, and maps $B(t)$ into
$$B'(t) := B(t) - \bigcup_{a \in D_k} \{d \bowtie a\}\,;$$
- if i, the length of s, is strictly greater than 1, then $\Phi(d, s, i)$ considers every $x_j \in s$, the resulting projection scheme $r := s - \{x_j\}$, and modifies $B(r)$ as follows:
$$B'(r) := \begin{cases} B(r) - \{d\,[r]\} & \text{if, for all } d' \in B(s),\ d'\,[r] = e \\ & \text{yields } d'\,[x_j] = d\,[x_j]\,, \\ & \\ B(r) & \text{else.} \end{cases}$$

Clearly, the Φ functions are monotone, inflationary and stationary with respect to the strong k-constraint ordering on P. The following lemma collects the main properties of these functions that will be used to study `SGIIS` for the Cooper algorithm.

Corollary **4.4.5.**
(i). *A CSP P is strongly k consistent iff it is a common fixpoint of the Φ functions, hence their least one with respect to the strong k-constraint order on P.*
(ii). *The Φ functions are monotone, stationary and inflationary with respect to the strong k-constraint order on P; besides, each $F_\perp$ function that does not modify the input problem is the identity.* □

The proof of the equivalence of the considered instantiations of `SGIISE` and `SGIIS` is analogous to that for `HAC-4`.

LEMMA **4.4.6.** SGIISE *with the* ϕ *functions over* $P_{\mathtt{KS}}$ *is* $\equiv_{\mathtt{KS}}$*-equivalent to* SGIIS *with the* Φ *functions over* $O_k \cong P_{\mathtt{KS}}/\equiv_{\mathtt{KS}}$. □

Therefore, we infer the following result from Corollary 4.4.5, Theorem 3.4.10, the above Lemma 4.4.6 and Corollary 3.4.16.

COROLLARY **4.4.7** (KS). *The* KS *algorithm over a finite problem* P *terminates by computing the least fixpoint of the functions* Φ *as above defined; i.e., the greatest strongly* k *consistent problem that is equivalent to* P. □

4.5 Relational Consistency

Whereas in k and strong k consistency, variables and their instantiations are the key notions, in the definition of relational consistency as below, relations rather than variables are under analysis. In this section we define this new notion of consistency, as in [DvB97], and prove that the basic algorithm schema for enforcing it is an instance of GI.

DEFINITION **4.5.1.** Consider a CSP with constraint set $\boldsymbol{C}$, and scheme X. Assume that $\boldsymbol{C}' := \{C(s_1), \ldots, C(s_n)\}$ is a subset of distinct constraints in $\boldsymbol{C}$, and s is the join scheme of $s_1, \ldots, s_n$.

- Let t be a scheme of s. Then $\boldsymbol{C}'$ is *relationally* m *consistent relative to* t if any t consistent instantiation can be extended to an s instantiation that satisfies $\bowtie_i^n R(s_i)$.

- The set $\boldsymbol{C}'$ is *relationally* (i, m) *consistent* if it is relationally m consistent relative to each scheme t of s, that has length i. If $\boldsymbol{C}'$ is relationally (i, m) consistent for every $i \leq m$, then it is *relationally* m *consistent.*

- A *CSP* is *relationally* (i, m) *consistent* if every subset of m constraints in $\boldsymbol{C}$ is such. The characterisation of *relational* m *consistency* is analogous.

- A *CSP* is *strongly relational* (i, m) *consistent* if it is (i, k) relational consistent for each $k \leq m$. The characterisation of *strong relationally* m *consistency* is analogous.

To illustrate the above defined notions, we copy the following example directly from [DvB97].

Example **4.5.2.** Consider the CSP over the scheme $X := x_1, x_2, x_3, x_4, x_5$, where the domains of the variables are all $D = \{a, b, c\}$ and the relations are given by,

$$C(x_2, x_3, x_4, x_5) := \{(a, a, a, a), (b, a, a, a), (a, b, a, a), (a, a, b, a), (a, a, a, b)\}$$
$$C(x_1, x_2, x_5) := \{(b, a, b), (c, b, c), (b, a, c)\}\,.$$

The constraints are not relationally 2 consistent. For example, the instantiation $x_2 = a$, $x_3 = b$, $x_4 = b$ is a consistent instantiation as it trivially satisfies all the applicable constraints. Similarly, the constraints are not relationally 3 consistent. For example, the instantiation $x_1 = c$, $x_2 = b$, $x_3 = a$, $x_4 = a$ is, trivially, a consistent instantiation, but it does not have an extension to x_5 that satisfies $C(x_2, x_3, x_4, x_5)$ and $C(x_1, x_2, x_5)$ simultaneously. If we add the constraints $C(x_2) = C(x_3) = C(x_4) = \{a\}$ and $C(x_1) = C(x_5) = \{b\}$, the set of solutions of the CSP does not change, and it can be verified that the CSP is both relationally 2 and 3 consistent.

As the authors of [DvB97] remark, when all the problem constraints are binary, relational m consistency is identical (up to minor preprocessing) to variable-based m consistency. The virtue in their notion of relational m consistency is that it can be embedded, naturally, into algorithms for enforcing desired levels of relational m consistency, and it allows a simple generalisation of k consistency.

However, as for k consistency and hyper-arc consistency, verifying relational m consistency can be exponential even for relational 2 consistency, if the arity of the constraints is not bound.

4.5.1 The $\mathtt{RC}_{(i,m)}$ Algorithm

The original algorithm for relational m consistency, called $\mathtt{RC}_m$, is, in the authors' words, "a brute-force algorithm for enforcing strong relational m consistency on a CSP". In the remainder of the present subsection, we use `GI` to enforce relational (i, m) consistency in the spirit of $\mathtt{RC}_m$ (cf. [DvB97]), and then analyse this is instance of `GI`.

Instantiation

Consider a CSP $P := \langle X, \boldsymbol{D}, \boldsymbol{C}\rangle$. To instantiate the `GII` algorithm to $\mathtt{RC}_{(i,m)}$ on P, we need to provide the following components:

1. a partial ordering on P;
2. functions on the chosen partial ordering;
3. the specification of the *update* operator.

Partial ordering. To enforce relational (i, m) consistency, we employ the i constraint ordering on P as partial ordering. Thus we assume $P := \langle X, \boldsymbol{D}, \boldsymbol{C} \rangle$ to be i complete; else we complete it as in Subsection 2.5.2.

Functions. Consider B in the i constraint ordering on P. Assume a scheme s, and a scheme t of s such that the length of t is i. Thus consider m constraints $B(s_1), \ldots, B(s_m)$ of B such that $s = \bigcup_i^m s_i$, and denote

$$\mathtt{s} := \langle s_1, \ldots, s_m \rangle .$$

Finally, define the function $\sigma(t, s, \mathtt{s})$ as follows on B: if $B' := \sigma(t, s, \mathtt{s})(B)$, then B' differs from B for the constraint on t, this being

$$B'(t) := B(t) \cap \Pi_t(\bowtie_{s_i \in \mathtt{s}} B(s_i)).$$

If $B'(t)$ is the empty set, then $\sigma(t, s, \mathtt{s})$ sets the whole B to the empty set. Else all the other constraints in B are unaffected by $\sigma(t, s, \mathtt{s})$.

The *update* operator. The *update* operator returns the empty set if $B'(t)$ is empty. Else, it adds all the constraints of the problem to G for further inspection. Clearly, this choice of the *update* operator could be optimised in a number of ways; for instance, by requiring that *update* should only add the relations which are affected by the change of $B(t)$.

Analysis

At this point, it is routine to check that the functions $\sigma(t, s, \mathtt{s})$ are idempotent, monotone and inflationary over the i constraint ordering.

Lemma **4.5.3.**
(i) A CSP $P := \langle X, \boldsymbol{D}, \boldsymbol{C} \rangle$ is relational (i, m) consistent iff $\boldsymbol{C}$ is a common fixpoint of all the functions of the form $\sigma(t, s, \mathtt{s})$, hence the last one with respect to the i constraint ordering on P.
(ii) Each function $\sigma(t, s, \mathtt{s})$ is idempotent, monotone and inflationary with respect to the i constraint ordering on P. $\square$

Fix now a CSP P. By instantiating the `GII` algorithm with the above defined functions $\sigma(t, s, \mathtt{s})$, we get the algorithm $\mathtt{RC}_{(i,m)}$. Thus we can prove that this algorithm enjoys the following properties as a consequence of Theorem 3.4.4.

Corollary **4.5.4** ($\mathtt{RC}_{(i,m)}$). *Consider a CSP $P := \langle X, \boldsymbol{D}, \boldsymbol{C} \rangle$ with finite constraints. Suppose that P is binary. Then $\mathtt{RC}_{(i,m)}$ on P always terminates by computing the greatest relational (i, m) consistent problem that is equivalent to P; that is the least common fixpoint of the $\sigma(t, s, \mathtt{s})$ functions, defined as above.*
$\square$

The above result completes our analysis of quite a number of constraint propagation algorithms. In the following section, we summarise what we have learnt from this analysis, and so conclude the present chapter.

4.6 Conclusions

4.6.1 Synopsis

In this chapter, we describe and analyse a series of constraint propagation algorithms through the unifying framework of SGI iterations, see Chapter 3. Properties of the algorithms are interpreted as properties of functions, so that the verification of the algorithms becomes a straightforward application of the theoretical results obtained for SGI function iterations.

Thus these algorithms are separated into classes, according to the space the correlated functions prune of inconsistencies: see the rightmost column in Table 4.1. Then a more refined analysis differentiates between the algorithms that pertain to the same class. In Table 4.1, such differences are expressed in the following terms: functions, whether these are related to sets (i.e., domains or constraints) or points (i.e., values in domains or tuples in constraints); properties of functions, (i.e. commutativity, inflationarity, stationarity and idempotency) used to avoid fruitless **while** loops, via an efficient instantiation of *update*.

Algorithms	Functions	*update*	Search Space
AC-1	set	Axiom 3.3.1	domain ordering
AC-3	set	commutativity and idempotency	domain ordering
AC-4	point	stationarity, inflationarity idempotency	domain ordering
PC-1	set	Axiom 3.3.1	2-constraint ordering
PC-2	set	commutativity and idempotency	2-constraint ordering
PC-4	point	stationarity, inflationarity and idempotency	2-constraint ordering
KC	point	stationarity, inflationarity and idempotency	(strong) k-constraint ordering
$\mathtt{RC}_{(i,m)}$	set	idempotency	i-constraint ordering

Table 4.1: Constraint propagation algorithms and their comparison through SGI.

4.6.2 Discussion

The framework of SGI iterations allows us to verify various constraint propagation algorithms in a systematic and uniform way, and compare them as above. The more traditional ways of verifying the correctness of those algorithms are complicated by the diversity of structures or representation of CSPs adopted in them; cf. the algorithms in Appendix A. Besides, this heterogeneity complicates their direct comparison, which is usually done only by examples.

In the cases of (H)AC-4, (H)AC-5, PC-4 and KS, CSP constraints and domains are pruned of inconsistencies by means of additional support structures, used for time efficiency reason; these have to be "scraped away" through an equivalence relation (i.e., passing from SGIISE to SGIIS) to obtain functions that modify domains or constraints in a monotonic manner. This complicates the verification proofs and the simplicity of GI gets slightly lost. Nevertheless, at the level of domains and constraints, also those algorithms work in a monotone and inflationary manner: i.e., they all iterate functions that satisfy our Axioms 3.3.1 and 3.3.2. This results from the analysis conducted in this chapter.

This same analysis can be extended to algorithms like bound-consistency (see [Apt99a]) that iterates functions defined on specific domains: intervals of real numbers. Other algorithms for arc consistency such as AC-6 can be studied through SGI as well.

The AC-6 algorithm exploits the same intuitions behind AC-4: functions are set based, in that each function f is associated with a specific value c in a domain, and searches whether c is consistent with a given constraint of the problem. The main difference is that AC-6 functions searches for only a *single* value b such that the pair of b and c belongs to the given constraint; whereas AC-4 searches for *all* such b. Therefore, functions for AC-6 do not perform a *universal selection* but a sort of *existential selection*: i.e., they stop their search when a value that satisfies certain properties is found. In AC-6, this is achieved by imposing a total order on each domain and selecting always the minimum value in the ordering that satisfies the required properties.

We shall clarify this difference, between the so-called universal and existential selection in Chapter 6. First, we enrich our class of constraint propagation algorithms by studying non-standard constraints. The new algorithms are briefly presented in the following chapter, where they are still described and analysed via SGI iterations.

Chapter 5

Soft Constraint Propagation

5.1 Introduction

5.1.1 Motivations

The standard approach to constraint programming, as in Chapters 2 and 4, assumes as constraints only the so-called crisp, hard or classical constraints. As stated in Chapter 2 on p. 11, in the classical constraint programming paradigm a constraint can only assign one out of two values to an instantiation: either $true$, the instantiation is consistent with the constraint, or $false$, it is inconsistent. While this approach leads to efficient constraint solving and propagation algorithms, in some real life situations, it can be too restrictive, and a more expressive framework is regarded as more adequate and natural.

For instance, suppose a user is uncertain whether a constraint should either allow or forbid a certain instantiation; namely the user cannot decide, on the evidence of the available information, whether a certain set of values is consistent or inconsistent with the constraint. Then the user has only two alternatives, either yes or no, in a crisp constraint setting. Instead, in a soft constraint environment the user would be able to choose to what extent an instantiation can satisfy a constraint. This is the case of systems for user interface applications, based on constraint hierarchies (see [BFBW92, WB93, SMFBB93, BAFB96, Jam96, BMSX97, Bar97a, Bar97b, Hos98, Bar98, BFB98]): a user can set preferences on geometrical constraints, for example, between the cursor and the lines that the user wants to draw on a computer screen; some constraints will have a higher priority over the others, so that any optimal partial solution to the problem has always to satisfy them.

In general, soft constraints allow users to model, naturally, those real life problems which possess features like preferences, uncertainties, costs, levels of importance or absence of solutions.

5.1.2 Outline

There are several formalisations of soft constraint problems. In this thesis, we consider the one based on semirings [BMR97]. A semiring structure provides:

- a non-empty set of elements that represent the desired features, like uncertainties, preferences or others;
- a partial order to compare those and thus constraints;
- two operations for combining features and hence constraints: one returns the least upper bound with respect to the semiring order; whereas the other returns a lower bound with respect to the same order.

This formalism is a generalisation of existing approaches to soft CSPs. For instance, the semiring based framework is a generalisation of the following others: crisp CSPs, see Chapter 2; weighted or optimisation CSPs, see [DKL01]; valued CSPs, see [SFV95, dGVS97, Sch00]; some fuzzy ([DFP93, Rut94]) and probabilistic ([FL93]) CSPs.

In the literature, some of the standard constraint propagation algorithms for crisp CSPs were successfully extended, and adapted, to semiring based CSPs. This has lead to an algorithm schema, based on rules, for soft constraint propagation. At each step of that schema, a subproblem of the original input problem is solved, see [BMR97]. In this chapter, we briefly recall this rule based schema and show how it can be recast through iterations of functions, see Section 5.4 below.

Constraint propagation over crisp constraints was studied in depth in Chapter 4 by means of the `SGI` schema presented in Chapter 3. In this chapter, we prove that the `SGI` schema can also be instantiated to a number of soft constraint propagation algorithms: all those that are instances of the rule based schema, since this is generalised by the `SGI` schema, see Section 5.4; a series of algorithms which the original rule based schema cannot account for, see Section 5.5. In order to prove this, it is sufficient to define appropriate partial orders between soft CSPs, see Section 5.3.

Moreover, by analysing the types of functions that `SGI` iterates, we shall prove in Section 5.4 that soft constraint propagation can be enforced by means of functions which are not necessarily idempotent, as instead originally demanded by the rule based schema. Thus ours is a double generalisation: in fact, we neither require that functions for soft constraint propagation should solve a subproblem; nor that they should be idempotent, see Section 5.4. The relaxation of these assumptions allows us to account for further notions of soft constraint propagation that are not expressible through the rule based schema, see Section 5.5.

Therefore, the `SGI` algorithm provides a general schema for soft constraint propagation as well. Properties of the schema are applicable to all its instances.

In particular in Section 5.4 we apply `SGI` to soft CSPs and attack not so easy tasks, like the following, in the context of soft constraint propagation:

- Is it true that each execution of a soft constraint propagation algorithm always return the same result? When does it happen (see Subsection 5.4.3)?
- Under which conditions does every execution of a soft constraint propagation algorithm terminate (see Subsection 5.4.4)?

In Chapter 4, we prove that the `SGI` algorithm always terminates whenever it iterates inflationary functions over finite domains or constraints; this finiteness hypothesis turn out to be common to all the analysed constraint propagation algorithms in Chapter 4. However, when we deal with semiring based constraints the set of preferences (i.e., the semiring universe) can be infinite, although the domain and constraint set are finite: for example, probabilistic constraints can be defined through a semiring that contains all the real numbers in the ordered interval $[0, 1]$. Furthermore, the semiring structure for weighted constraints contains either all real or all natural numbers. Thus it is not realistic to assume that semiring based CSPs have finite constraints, hence we need to find another property than finiteness to guarantee the termination of the `SGI` schema.

Our first termination result is mainly concerned with the partial order over soft constraint satisfaction problems. It can be used to prove the termination of soft constraint propagation algorithms over weighted constraints, if preferences range over natural numbers and suitable functions are used for soft propagation. Our second result for the termination of `SGI` is related to the two semiring operations and its order. In turn, the latter result can be used to guarantee the termination of soft constraint propagation algorithms over probabilistic constraints.

Both the aforementioned results guarantee termination in a number of cases, however their hypotheses may be difficult to check. Nevertheless, when the second semiring operation coincides with the greatest lower bound operation, the semiring is also a distributive lattice; then all finitely generated sets, via the semiring operations, are finite; thereby, functions for constraint propagation, expressed through the semiring operations, can only return finitely many values. For instance, this is the case of crisp constraints, and also that of fuzzy constraints, e.g. functions for constraint propagation are combinations of max and min on the interval $[0, 1]$ of real numbers. Thus in such cases we obtain a third termination result, whose range of applicability is the most restricted, but whose hypotheses are the easiest to check.

5.1.3 Structure

The chapter is organised as follows. First, Section 5.2 introduces semirings, the semiring based formalism for soft constraints, and its basic operations on constraints. Section 5.3 treats some orders among semirings, constraints, and problems, necessary for defining soft constraint propagation via rules and `SGI`. So, in

Section 5.4, the SGI algorithm schema is extended to soft CSPs, and is proved to encompass the rule based schema for soft constraint propagation. Subsection 5.4.4 is concerned with the termination of the SGI schema. Finally, in Section 5.5 we display some arc constraint propagation algorithms and study them via SGI, and in Section 5.6 we discuss some limitations of our approaches and possible future directions.

5.2 Soft Constraints

In the semiring based formalism of [BMR97], a soft constraint is like a classical constraint, namely a relation, such that each of its tuples gets assigned a preference value. So, if we recast relations through their respective characteristic functions, passing from crisp to soft constraints means allowing the characteristic function of a constraint to range over more values than just $\top$ (true) and $\perp$ (false). Once additional values are provided for constraints, suitable operations for their combination and comparison have to be provided as well. Semiring structures, as characterised below, give us all those ingredients.

5.2.1 Constraint Semirings

Constraint semirings and lattices

A *constraint semiring*, briefly *c-semiring*, is a structure $\mathcal{S} := \langle S, <_S, \times, \perp, \top \rangle$ that enjoys the following properties:

- $\langle S, <_S, \perp, \top \rangle$ is a complete lattice, with bottom $\perp$, and top $\top$;
- $\times$ is a binary operation on S, which is commutative, associative, has $\perp$ as absorbing element and $\top$ as unit one. Moreover, $\times$ distributes over the least upper bound operation, denoted with $\vee$.

Notice that the above definition is not the original one that is adopted in *ib.*, nevertheless ours is equivalent to the latter. In this thesis, we prefer the lattice characterisation of c-semirings because it directly highlights the partial order relation, which plays a central role in the study of termination for soft constraint propagation.

Some useful properties of c-semirings

The greatest lower bound operation and $\times$ are related. In fact, if $\wedge$ denotes the greatest lower bound operation of $\mathcal{S}$, then

$$a \times b \leq_S a \wedge b,$$

for every pair of elements a and b in S. The above relation holds iff both the following ones do:

$$a \times b \leq_S a \text{ and } a \times b \leq_S b.$$

To prove the above relations, it is sufficient to ascertain that $\times$ enjoys the following property:

$$a \times c \leq_S a \tag{5.1}$$

for every pair of elements a and c in S; this amounts to saying that $\times$ is inflationary with respect to the reverse $\geq_S$ order relation. Now, the relation (5.1) holds iff $\times$ is monotone with respect to $\leq_S$. This can be proved to hold iff the following relation holds:

$$a \times c \leq_S a \times b \text{ if } c \leq_S b. \tag{5.2}$$

Then we exploit the fact that the top element $\top$ is the unit of $\times$ and infer (5.1). So we are left to prove the monotonicity relation as in (5.2). But this is easy, because the hypothesis $c \vee b = b$ yields the equality

$$a \times (c \vee b) = a \times b,$$

and because $\times$ distributes over $\vee$.

We collect all the above results and some well-known facts concerning the least upper bound operation $\vee$ in the following lemma, as they will be used over and over in this chapter; see also *ib*.

Lemma **5.2.1.**

- *For every pair of elements a and b in S, we have that $a \times b \leq_S a \wedge b$.*
- *The $\times$ operation is* inflationary *with respect to $\geq_S$, whereas $\vee$ is inflationary with respect to $\leq_S$: i.e., $a \leq_S a \vee c$ and $a \times c \leq_S c$, for every a and c in S.*
- *Both $\times$ and $\vee$ are* monotone *with respect to $\leq_S$: i.e., $a \times b \leq_S c \times d$, $a \vee b \leq_S c \vee d$ whenever $a \leq_S c$ and $b \leq_S d$.*
- *The $\vee$ operation is* idempotent*: i.e., $a \vee a = a$, for every $a \in S$. If $\times$ is idempotent as well, then this coincides with $\wedge$ and the c-semiring is a complete distributive lattice.*

Proof. We are only left to prove the second part of the last item. So, let us assume that $\times$ is idempotent. Then the relations

$$\begin{cases} a \wedge b \leq_S a, \\ a \wedge b \leq_S b, \end{cases}$$

and the monotonicity of $\times$ with respect to $\leq_S$ entail the relation

$$(a \wedge b) \times (a \wedge b) \ \leq_S \ a \times b. \tag{5.3}$$

By hypothesis, $\times$ is idempotent, hence $(a \wedge b) \times (a \wedge b)$ is equal to $a \wedge b$. This and (5.3) yield the relation

$$a \wedge b \ \leq_S \ a \times b.$$

Thus the equality $a \wedge b = a \times b$ follows now from the last relation above and its reverse as in the first item. □

5.2.2 Soft Constraints

Given a c-semiring $\mathcal{S} := \langle S, <_S, \times, \bot, \top \rangle$, a domain scheme X and a domain D over X (see Chapter 2), we can now define an $\mathcal{S}$ *soft constraint* $C(s)$, over a scheme s of X and the domain D on X, as a function

$$C(s) \, : \, D\,[s] \mapsto S.$$

Intuitively, $C(s)$ provides each tuple $d \in D\,[s]$ with a preference value in the c-semiring universe S.

A *soft constraint satisfaction problem* (SCSP) *on* $\mathcal{S}$ is a structure $P = \langle X, \boldsymbol{D}, \boldsymbol{C} \rangle$ that is defined as follows:

- X is a variable scheme and $\boldsymbol{D}$ is a domain set over X,
- $\boldsymbol{C}$ is a set of $\mathcal{S}$ soft constraints over schemes in X.

Whenever the semiring $\mathcal{S}$ to which we refer is clear from the context, we shall not mention it; hence we shall usually write soft constraints and SCSP.

5.2.3 Examples

Crisp CSPs

Crisp CSPs are SCSPs for which the chosen c-semiring is the Boolean algebra

$$Bool = \langle \{\top, \bot\}, \leq, \wedge, \top, \bot \rangle,$$

in which $\leq$ is defined via the Boolean disjunction: $a \leq b$ iff $a \vee b = b$.

By means of $Bool$ we can associate a Boolean value, either $\bot$ (false) or $\top$ (true), with each tuple of elements in $\boldsymbol{D}$. So here a constraint over $Bool$ corresponds to the characteristic function of a crisp constraint as in Chapter 2.

Weighted CSPs

Weighted or optimisation CSPs over natural numbers (see [DKL01]) can be defined by means of the c-semiring

$$Weight = \langle \mathbb{N}, \leq_{\mathbb{N}}, +, 0, +\infty \rangle,$$

in which $\leq_{\mathbb{N}}$ is the standard total order over the set of natural numbers.

EXAMPLE **5.2.2.** Reconsider the MAP COLOURABILITY PROBLEM as in Subsection 2.3.1 and suppose now to have exactly two colours, *aqua* and *blue*, for every variable x_1, x_2 and x_3 of the problem. Clearly, the problem does not admit any solution if expressed as a crisp CSP. But suppose now that we want to find an optimal solution; in that, we want to *maximise* the *number* of satisfied constraints. Then it is sufficient to recast the same constraints we had in Subsection 2.3.1 as weighted constraints. Each of them will assign 0 to the pairs $(aqua, aqua)$ and $(blue, blue)$, and 1 to all the remaining pairs. In this case, given a total assignment, we shall count the number of constraints it satisfies (via $\sum$), and if this this number is the maximum (with respect to $\leq$) we shall regard this assignment as a "solution"' to this weighted MAP COLOURABILITY PROBLEM.

Probabilistic and fuzzy CSPs

Probabilistic CSPs with minimum and maximum (see [FL93]) can be defined via the c-semiring

$$Prob(min,\, max) = \langle [0,1]\,, \geq_{\mathbb{R}}, \max, 0, 1\rangle,$$

in which $\geq_{\mathbb{R}}$ is the standard linear order over real numbers and max the corresponding maximum operator.

On the other hand, fuzzy CSPs with maximum and minimum (see [DFP93, Rut94]) can be defined via the c-semiring

$$Fuzzy(max,\, min) = \langle [0,1]\,, \leq_{\mathbb{R}}, \min, 0, 1\rangle,$$

in which $\leq_{\mathbb{R}}$ is the standard linear order over real numbers and min the corresponding minimum operator.

EXAMPLE **5.2.3.** Let us consider the MAP COLOURABILITY PROBLEM as in Example 5.2.2 again, and suppose that now we have a slight preference towards any assignment that gives x_3 the colour *aqua*; that is, we are not happy with maximising the number of satisfied constraints, but we have a preference towards a specific colour for a specific country. Then we could express this preference by recasting $C(x_1, x_3)$ and $C(x_2, x_3)$ as fuzzy constraints, so that these assign a greater value in the interval $[0, 1]$ to the tuple $(blue, aqua)$ than to $(aqua, blue)$. A "solution" is such that the returned value for the assignment *aqua* to x_3 is the maximum over all the minimum values for the assignments in which the colour *aqua* is given to x_3.

5.2.4 Basic Operations on Soft Constraints

Having defined soft constraints, we can now extend the basic operations for crisp constraints, namely projection and join, to analogous operations for soft ones. These are easily provided by means of the c-semiring operation $\times$ and the least upper bound one; see [BMR97], where join is called combination.

Join

Given two constraints, $C_1 := C_1(s_1)$ and $C_2 := C_2(s_2)$, their *join* $C_1 \bowtie C_2$ is the constraint over the scheme $t := s_1 \cup s_2$ such that

$$C_1 \bowtie C_2\,(d) := C_1(d\,[s_1]) \times C_2(d\,[s_2]),$$

for every $d \in D\,[t]$ (remember that $d\,[s_i]$ is the projection of the tuple d over s_i).

In other words, the join of two constraints assigns, to each tuple, a value that is the product (via the c-semiring $\times$) of the respective values returned by the two joint constraints.

The join operation is associative, since $\times$ is. Therefore $\bowtie$, which is defined as a binary operation, is easily extended to an operation over any finite number of constraints.

Example **5.2.4.** In Example 5.2.3 the join of the two fuzzy constraints $C_1 := C(x_1, x_3)$ and $C_2 := C(x_2, x_3)$ is a constraint on $C(x_1, x_2, x_3)$ that assigns, for instance, to $(aqua, blue, aqua)$ the minimum between the values $C_1(aqua, aqua)$ and $C_2(blue, aqua)$. While, in the case of Example 5.2.2, the join of the constraints C_1 and C_2, now interpreted over the weighted c-semiring with natural numbers, is the sum of $C_1(aqua, aqua)$ and $C_2(blue, aqua)$.

Projection

Given a constraint $C := C(s)$ and a scheme t of s, the *projection* of C over t, denoted by $\Pi_t(C)$, is the unique constraint over t such that

$$\Pi_t(C)(d) := \bigvee \{C(e) \,:\, e \in D\,[s] \text{ and } e\,[t] = d\}\,.$$

So, the projection of a constraint over s assigns, to each tuple d, the least upper bound of the values assigned, by the original constraint, to all the tuples that are equal to d when projected over s.

Example **5.2.5.** Let us consider Example 5.2.3 and assume that $C := C(x_1, x_3)$ assigns $1/2$ to $(aqua, blue)$, then 1 to the most preferred assignment $(blue, aqua)$, and 0 to all the other ones. Then the projection of C over x_3 is the constraint $C(x_3)$ on x_3 that assigns 1 to *aqua*, and $1/2$ to *blue*. If C assigns $+\infty$ to $(blue, aqua)$, 1 to $(aqua, blue)$, 0 to the remaining pairs, and is hence recast as a weighted constraint as in Example 5.2.2, then the projection of C over x_3 will assign $+\infty$ to *aqua* and 1 to *blue*.

5.2.5 Solutions and Equivalent Problems

Solutions

In case of a crisp CSP $P := \langle X, \boldsymbol{D}, \boldsymbol{C}\rangle$, a solution to P is a tuple of D that belongs to each constraint in $\boldsymbol{C}$, see Chapter 2. In other words, a solution is a tuple to which every P constraint assigns the value *true*, namely $\top$, in the Boolean structure *Bool*, see Subsection 5.2.3.

We repeat the same procedure for computing soft solutions. First, we need to *complete* the soft problem P and obtain its completion $\bar{P}$, like in Subsection 2.4.2 in case of crisp CSPs:

- if there is more than one constraint on a scheme s, say $C_1, \ldots, C_k$, then we replace them with the constraint on s equal to $\bowtie_{i=1}^{k} C_i$ — that corresponds to intersection in the crisp case;

- if there are no constraints on a scheme s in P, then we create a new constraint $C := C(s)$ that assigns the top value to every tuple in $D\,[s]$, namely $C(d) := \top$, for every $d \in D\,[s]$.

Consider the completion $\bar{P} = \langle X, \boldsymbol{D}, \boldsymbol{C} \rangle$ of an SCSP P. Then the *solution* to P is the constraint

$$Sol(P) = \bowtie_{C \in \bar{P}} C;$$

that is the constraint which assigns, to each tuple, the product via $\times$ of the values assigned by the constraints in the completed problem.

If we are only interested in a solution over a scheme s of X, then we obtain a *solution to P over s* by projecting $Sol(P)$ over s; so we define it as

$$Sol(P, s) = \Pi_s Sol(P).$$

Clearly, $Sol(P) = Sol(P, X)$.

Equivalence

In the case of crisp constraints, two CSPs on the same scheme are equivalent if they have the same set of solutions; see Definition 2.4.1. A similar characterisation of equivalence among problems is found in the semiring based case, with the requirement that the problems share the same c-semiring and domain set.

Two SCSPs on the same c-semiring $\mathcal{S}$, scheme X and domain set $\boldsymbol{D}$, say $P_1 := \langle X, \boldsymbol{D}, \boldsymbol{C}_1 \rangle$ and $P_2 := \langle X, \boldsymbol{D}, \boldsymbol{C}_2 \rangle$, are *equivalent* if $Sol(P_1) = Sol(P_2)$. In this case, we write $P_1 \equiv P_2$.

Since two equivalent SCSPs are defined on the same c-semiring and domain set, they also have equal solutions on each scheme of X: i.e., their equivalence yields

$$Sol(P_1, s) := Sol(P_2, s) \tag{5.4}$$

for every scheme s of X. Notice that, if the two domains were different, there would be no guarantee for (5.4) to hold. Indeed, the concept of equivalence and, more specifically, that of solution in case of SCSPs are interesting and subtle issues, see [BCR00, Gen01a, Gen01b].

EXAMPLE **5.2.6.** The solution to the max-min fuzzy CSP P of Example 5.2.3 assigns, to each state x_i and colour for x_i, a value in $[0, 1]$. For instance, let us consider the tuple of colours $(aqua, blue, aqua)$ for x_1, x_2, x_3. If $C(x_1, x_2)$ assigns $1/2$ to $(aqua, blue)$, $C(x_2, x_3)$ assigns 1 to $(blue, aqua)$ and $C(x_1, x_3)$ assigns $1/3$ to $(aqua, aqua)$, then $Sol(P)$ will assign the minimum of all these values, i.e. $1/3$, to $(aqua, blue, aqua)$. Suppose that $Sol(P)$ assigns $1/4$ to $(aqua, blue, blue)$. Then $Sol(P, \langle x_1, x_2\rangle)$ will assign $1/3$, i.e. the maximum between $1/3$ and $1/4$, to $(aqua, blue)$.

5.3 Soft Orders

As in the case of crisp CSPs, we need partial orders on SCSPs, to compare both them and the iterations of functions defined on them. In the crisp case, partial orders are based on the subset relation, see Subsection 2.5.2. For instance, in the crisp case, we can compare the constraint $C_1(s)$ and $C_2(s)$ if $C_1(s) \subseteq C_2(s)$. If we recast the subset relation in terms of characteristic functions, we can rewrite it as follows:

$$\chi_{C_1}(d) \leq \chi_{C_2}(d), \tag{5.5}$$

for every d in the domain on s, where $\leq$ can only compare $\top$ and $\bot$.

Now, there is an obvious candidate for $\leq$ in (5.5) when we generalise crisp to semiring based constraints: that is, the partial order relation $\leq_S$ of the c-semiring.

Following this idea, we first introduce a partial order relation among constraints via $\leq_S$, see Subsection 5.3.1; then we lift such order to a partial order on constraint sets in Subsection 5.3.2, and finally to problems, see Subsection 5.3.3.

In order to simplify the discussion, we adopt a common convention when dealing with soft constraints: we always assume that SCSPs are complete. If necessary, incomplete SCSPs can first be completed.

CONVENTION **5.3.1.** In the remainder of this chapter, we always assume that every SCSP P is complete.

The above assumption, strictly speaking, is not necessary for the results in the remainder of this chapter. However, it simplifies our discussion and notation.

5.3.1 Constraint Order

Given the partial ordering $\leq_S$ of a c-semiring $\mathcal{S}$, we can define a new partial order relation among constraints, as follows.

DEFINITION **5.3.1.** Let $\mathcal{S} := \langle S, <_S, \times, \bot, \top\rangle$ be a c-semiring, X a scheme and $\boldsymbol{D}$ a domain set over X. Then consider two constraints $C_1 := C_1(s)$ and $C_2 := C_2(s)$ over the scheme s of X. We write $C_1 \sqsubset_S C_2$ if the following conditions are both satisfied:

1. for all the tuples $d \in D[s]$, $C_2(d) \leq_S C_1(d)$;

2. there exists a tuple $d \in D[s]$ for which $C_2(d) <_S C_1(d)$.

We write $C_1 \sqsubseteq_S C_2$ in case only the first relation holds.

In other words, the constraint C_1 is smaller than or equal to C_2 in the order $\sqsubseteq_S$ if the former constraint assigns, to each tuple, either the same value as C_2, or a greater value with respect to $<_S$. Loosely speaking, C_2 is preferred to C_1 if the former constraint is possibly more restrictive than the latter, loosely speaking.

THEOREM **5.3.2.** *The relation $\sqsubseteq_S$ is a partial order among constraints defined on the same c-semiring $\mathcal{S} := \langle S, <_S, \times, \bot, \top \rangle$, scheme X and domain set $\boldsymbol{D}$.*

PROOF. We need to prove that $\sqsubseteq_S$ is a reflexive, antisymmetric and transitive relation. Reflexivity holds trivially. To prove antisymmetry, suppose that $C_1 \sqsubseteq_S C_2$ and $C_2 \sqsubseteq_S C_1$; this yields that both constraints share the same scheme, say s. Now, for all tuples $d \in D[s]$, we have both $C_1(d) \leq_S C_2(d)$ and $C_2(d) \leq_S C_1(d)$, hence $C_1(d) = C_2(d)$. Therefore, $C_1 = C_2$. The transitivity of $\sqsubseteq_S$ follows from the transitivity of $\leq_S$. □

5.3.2 Constraint Set Order

We can now easily extend the order $\sqsubseteq_S$ over constraints to a new order between *constraint sets* as follows.

DEFINITION **5.3.3.** Let $\mathcal{S} := \langle S, <_S, \times, \bot, \top \rangle$ be a c-semiring, X a scheme and $\boldsymbol{D}$ a domain set over X. Consider two sets of constraints, $\boldsymbol{C}_1$ and $\boldsymbol{C}_2$, over X, $\boldsymbol{D}$ and $\mathcal{S}$. We write $\boldsymbol{C}_1 \sqsubseteq_C \boldsymbol{C}_2$ if the following two properties both hold:

1. there exist precisely one constraint $C_1(s)$ in $\boldsymbol{C}_1$ on s, and precisely one constraint $C_2(s)$ in $\boldsymbol{C}_2$ on s, for each scheme s of X;

2. for each scheme s on X, we have that the relation $C_1(s) \sqsubseteq_S C_2(s)$ holds between the constraints of $\boldsymbol{C}_1$ and $\boldsymbol{C}_2$ on s.

The intuitive reading of $\boldsymbol{C}_1 \sqsubseteq_C \boldsymbol{C}_2$ is that the problems that $\boldsymbol{C}_2$ yield are at least as constraining as those of $\boldsymbol{C}_1$, because $\boldsymbol{C}_2$ has at least as many more or equally restrictive constraints as $\boldsymbol{C}_1$ has, loosely speaking.

THEOREM **5.3.4.** *Consider a collection $\mathcal{C}$ of constraint sets on the same c-semiring $\mathcal{S} := \langle S, <_S, \times, \bot, \top \rangle$, scheme X and domain set $\boldsymbol{D}$. Then the relation $\sqsubseteq_C$ is a partial order between constraint sets in $\mathcal{C}$.*

Proof. Reflexivity trivially holds. As for antisymmetry, suppose that both $\boldsymbol{C}_1 \sqsubseteq_C \boldsymbol{C}_2$ and $\boldsymbol{C}_2 \sqsubseteq_C \boldsymbol{C}_1$ hold. Thus both the following relations hold for each scheme s on X: $C_1(s) \sqsubseteq_S C_2(s)$ and $C_1(s) \sqsubseteq_S C_2(s)$, for $C_1 \in \boldsymbol{C}_1$ and $C_2 \in \boldsymbol{C}_2$. Hence $C_1(s) = C_2(s)$ for each scheme s on X, because $\sqsubseteq_S$ is a partial order relation, see Theorem 5.3.2. Transitivity follows similarly, by exploiting the transitivity of $\sqsubseteq_S$. □

5.3.3 Problem Orderings

At this point, we know two partial orders for SCPS: a partial order among constraints ($\sqsubseteq_S$); this lifted to constraint sets ($\sqsubseteq_C$). However, constraint propagation algorithms have SCSPs in input; therefore, we need a partial order relation among SCSPs to enforce soft constraint propagation by means of the `SGI` algorithm.

Definition **5.3.5.** Given a c-semiring $\mathcal{S} := \langle S, <_S, \times, \bot, \top \rangle$, consider two problems over it, with the same scheme X and domain set $\boldsymbol{D}$; say $P_1 = \langle X, \boldsymbol{D}, \boldsymbol{C}_1 \rangle$ and $P_2 = \langle X, \boldsymbol{D}, \boldsymbol{C}_2 \rangle$. Then we write $P_1 \sqsubseteq_P P_2$ if $\boldsymbol{C}_1 \sqsubseteq_C \boldsymbol{C}_2$.

Note **5.3.6.** Notice how the above relation does not involve the domain set $\boldsymbol{D}$ at all; the comparison of semiring-based problems takes place at the level of constraint sets. Compare it to the definition of partial orders on CSPs, see Subsection 2.5.2. There are at least two reasons for such a difference, the latter following from the former:

1. the definition of equivalence among SCSPs is meaningful only if the problems share the same variable domains, see Subsection 5.2.5;

2. arc consistency for SCSPs modifies unary constraints instead of variable domains; in general, constraint propagation algorithms for SCSPs do not alter the domains of variables.

We are in the position to define a partially ordered structure that contains all of the SCSPs that constraint propagation algorithms step-by-step compute, starting from their input SCSP. The following definition is a generalisation of Definition 2.5.2 to the case of semiring based soft constraints.

Definition **5.3.7.** Consider a c-semiring $\mathcal{S} := \langle S, <_S, \times, \bot, \top \rangle$ and an SCSP $P := \langle X, \boldsymbol{D}, \boldsymbol{C} \rangle$ over it. Then the *soft closure* of P, denoted by $P\uparrow$, is the class of all problems P' on $\mathcal{S}$, X and $\boldsymbol{D}$ such that the relation $P \sqsubseteq_P P'$ holds.

Theorem **5.3.8.** *Consider a c-semiring $\mathcal{S} := \langle S, <_S, \times, \bot, \top \rangle$, and an SCSP $P := \langle X, \boldsymbol{D}, \boldsymbol{C} \rangle$ over it. Then the following statements hold:*

- $\langle P\uparrow, \sqsubseteq_P \rangle$ *is a partial ordering;*

- *the bottom of* $\langle P\!\uparrow, \sqsubseteq_P\rangle$ *is* P *itself.*

Proof. We prove the first claim, the other one is immediate from the definition of $P\!\uparrow$. As usual, we only prove that the relation is antisymmetric, because transitivity can be proved similarly and reflexivity is trivial. Hence, suppose that both $P_1 \sqsubseteq_P P_2$ and $P_2 \sqsubseteq_P P_1$ hold between two SCSPs in $P\!\uparrow$. So both those SCSPs are defined on the same scheme X and domain set $\boldsymbol{D}$. Moreover, $\boldsymbol{C}_1 \sqsubseteq_C \boldsymbol{C}_2$ and $\boldsymbol{C}_2 \sqsubseteq_C \boldsymbol{C}_1$. From the two last relations and Theorem 5.3.4, we infer the identity $\boldsymbol{C}_1 = \boldsymbol{C}_2$. Thus $P_1 = P_2$. □

Again, we have an analogous result for crisp and soft CSPs.

Fact **5.3.9.** *Consider an SCSP* P *over a c-semiring* $\mathcal{S}$*, and its soft closure* $P\!\uparrow$*. Then the following statements hold:*

1. *if* $P_1 \sqsubseteq_P P_2$ *and* $P_1 \in P\!\uparrow$*, then* $P_2 \in P\!\uparrow$*;*
2. *if* $P_1 \sqsubseteq_P P_2$*, then* $P_2\!\uparrow \subseteq P_1\!\uparrow$*.* □

The above statement has a counterpart in the crisp case: Fact 2.5.3.

We have now obtained a partial ordering $\langle P\!\uparrow, \sqsubseteq, P\rangle$, for every problem P. The problems in $P\!\uparrow$ differ at most in their constraint components. However, such a partial ordering on a SCSP P may contain too many problems with respect to those that specific soft constraint propagation algorithms step-by-step compute, starting from the given SCSP P.

We faced a similar situation with crisp CSPs: as our analysis in Chapter 4 clarifies, the closure of a problem carries over too many subproblems, that are not iterated by certain constraint propagation algorithms. So, in Subsection 2.5.2, we carved out those orderings — each based on a different subfamily of $P\!\uparrow$ — that are of interest in the analysis of constraint propagation algorithms as in Chapter 4.

In the case of crisp CSPs, we distinguished between domain orderings (see p. 24) and constraint orderings (see p. 25) on CSPs. We do not have this distinction for SCSPs, since soft problem orderings are based on the comparison of constraint sets only; see also Note 5.3.6. So all we shall need in the analysis of soft constraint propagation is the following refinement of the problem ordering on SCSPs.

Definition **5.3.10.** Let P be a problem over a c-semiring $\mathcal{S}$, and consider its soft closure $P\!\uparrow$. Assume that $\mathcal{F}(P)$ is a family of problems $P' \in P\!\uparrow$. If P belongs to $\mathcal{F}(P)$, then the structure

$$\langle \mathcal{F}(P), \sqsubseteq, P\rangle$$

is called a *problem ordering on* P.

Given the above definition, any subset of $P\!\uparrow$, to which the complete problem P belongs, gives rise to a problem ordering on P. This generality will prove helpful when studying termination conditions, see Subsection 5.4.4 below.

5.4 Soft Constraint Propagation via SGI

Our goal in this section is to analyse soft constraint propagation. First we define a schema for constraint propagation which is based on functions that are called "rules", as originally defined in [BMR97], see Subsection 5.4.1; each of those rules is limited to a combination of projection and join, in that they solve a subproblem of the input problem. Then we extend soft constraint propagation based on rule iterations by means of SGI iterations — see also Subsection 5.4.2 — and conclude by tackling the issue of termination for soft constraint propagation algorithms in Subsection 5.4.4.

5.4.1 Soft Constraint Propagation via Rules

Several traditional constraint propagation algorithms for crisp CSPs can be extended to SCSPs. For this purpouse, the notion of *constraint propagation rule* was introduced in *ib*.

Rule iterations

Let us consider a c-semiring $\mathcal{S} := \langle S, <_S, \times, \bot, \top \rangle$, a domain D over X and a constraint $C'(s)$ on a scheme s of X and the domain D. Assume that $P := \langle X, \boldsymbol{D}, \boldsymbol{C} \rangle$ is a problem over $\mathcal{S}$, and $C(s)$ is the constraint on s of P. Then $P\,[C(s)/C'(s)]$ denotes the problem that differs from P only in the constraint on s, which is set equal to $C'(s)$.

Let P be an SCSP as above; then consider a scheme s of X and a scheme t of s itself. A *constraint propagation rule* r^s_t *for* P, *on* s *and* t, is a function on $P\!\uparrow$ that is defined as follows:

$$r^s_t(P') := P'\,[C(t)/\Pi_t Sol(P, s)]\,,$$

for any $P' \in P\!\uparrow$.

In other words, the application of r^s_t to $P' \in P\!\uparrow$ adds the constraint $\Pi_t Sol(P', s)$ over the scheme t to P'. That constraint is obtained by combining all the constraints of P' on s, and then projecting the resulting constraint over t.

Once a c-semiring $\mathcal{S} := \langle S, <_S, \times, \bot, \top \rangle$ and an SCSP $P := \langle X, \boldsymbol{D}, \boldsymbol{C} \rangle$ are given, a *rule based on* X is any rule r^s_t for P, on two schemes, s of X, and t of s.

The application of a sequence of rules to a problem is obtained by composing the rules in their order of occurrence in the sequence. An infinite sequence of rules from a finite set R is *fair* if each rule of R occurs in the sequence infinitely often.

So, given a problem $P := \langle X, \boldsymbol{D}, \boldsymbol{C} \rangle$ and a finite set of rules R on X, *rule-based constraint propagation* R on X, *starting from* P, is defined as a fair iteration of rules from R so that the first rule in the sequence is applied to P. The following statement concerns itself with some properties of constraint propagation via rules.

THEOREM **5.4.1** ([BMR97]). *Consider a rule-based constraint propagation* `R`*, starting from an SCSP P. If* `R` *stabilises at P', then P' is a common fixpoint of all the rules from R. This common fixpoint P' is both unique and equivalent to P, provided that $\times$ is idempotent.* □

Notice that fairness is assumed to ensure that the problem at which the iteration stabilises is a common fixpoint of all the rules; and the idempotency of $\times$ is used to ensure both the uniqueness of the result and the equivalence.

However, the additional hypothesis of fairness is redundant for the `SGI` algorithm schema, as its task is accomplished by the *update* operator, see Subsection 5.4.2. So we shall drop the assumption of fairness and only restrict our attention to bare iterations of rules. Still, we shall be able to prove that the stabilisation point is always unique due to the monotonicity of rules, see Subsection 5.4.3. In the following part, we list the main properties of rules — as functions — as they are used in Subsection 5.4.3.

Order-related Properties of Soft Constraint Propagation Rules

We recall, from Subsection 5.2.5, that two problems P_1 and P_2, defined on the same scheme and domain, are equivalent if they have the same solution constraint; if this is the case, we write $P_1 \equiv P_2$.

The following lemma lists the main properties of soft constraint propagation rules that are related to the SCSP order, see Definition 5.3.5.

LEMMA **5.4.2.** *Consider a problem $P := \langle X, \boldsymbol{D}, \boldsymbol{C} \rangle$ on a c-semiring $\mathcal{S}$, and a rule r based on X.*

- *If $P' \in P\!\uparrow$, then $r(P')$ is still in $P\!\uparrow$. So r, defined on X, is a function on $P\!\uparrow$. Moreover, $P' \equiv r(P')$ if $\times$ is idempotent.*
- *Furthermore, $P' \sqsubseteq_P r(P')$, for every $P' \in P\!\uparrow$.*
- *Consider two SCSPs, P_1 and P_2 in $P\!\uparrow$. If $P_1 \sqsubseteq_P P_2$, then $r(P_1) \sqsubseteq_P r(P_2)$.*

PROOF. The second property states that rules are inflationary functions over any constraint ordering on P. This is based on the inflationarity of $\times$ and $\vee$, as in Lemma 5.2.1, and the definition of SCSP order. Whereas the last property is concerned with the monotonicity of rules, and again follows from Lemma 5.2.1 and the definition of SCSP order. □

5.4.2 Soft Constraint Propagation via the SGI Schema

The functions that are iterated in a rule based constraint propagation are of a specific form: they are obtained from join and projection, so that each of them solves a subproblem of the input problem. Whereas the `SGI` algorithm does not impose any specific request on the iterated functions. Functions can be of any sort in the `SGI` schema; so we are able to generalise rule-based constraint propagation as follows.

DEFINITION **5.4.3.** Consider an SCSP problem P over a c-semiring S. A *soft constraint function* for P is a function $f : P{\uparrow} \mapsto P{\uparrow}$.

Notice that a function f on a family $\mathcal{F}(P)$ of $P{\uparrow}$, i.e.,

$$f : \mathcal{F}(P) \mapsto \mathcal{F}(P), \tag{5.6}$$

can be uniquely extended to a function on the whole set $P{\uparrow}$: that is to a constraint function as in Definition 5.4.3. We only need to define it equal to the identity on every problem that does not belong to $\mathcal{F}(P)$. Vice versa, if f is a constraint function and $\mathcal{F}(P)$ is closed for it as in (5.6), then f can be restricted to a function over $\mathcal{F}(P)$. The distinction is only notational in this context, therefore we shall usually ignore it, and freely refer to any function for which (5.6) holds as a *constraint function over* $\mathcal{F}(P)$.

Functions versus rules. The characterisation of soft constraint function given in Definition 5.4.3 abandons a number of assumptions and attributes of rules and rule-based constraint propagations (see Subsection 5.4.1):

- soft constraint functions do not necessarily compute solutions to subproblems;
- soft constraint functions are neither assumed to be monotone and inflationary, nor idempotent if $\times$ is;
- the fairness assumption on iterations of soft constraint functions is not required.

The first generalisation is the most rewarding one, in that a number of constraint propagation algorithms do not exactly solve subproblems, but compute an approximation of its solutions; see, for example, the basic algorithm for *bound-consistency* in [MS98], or generalised arc consistency for SCSPs as in Section 5.5 below. Thus `SGI` for SCSPs allows us to instantiate more algorithms than the original rule based schema.

As for the fairness hypothesis, there is no loss in abandoning it, as *update* takes its role. In fact, as far as the *update* operator satisfies Axiom 3.3.1, see Chapter 3, the SGI schema computes *a common fixpoint* of all the iterated functions. We rewrite that axiom and the SGI algorithm below for the reader's convenience, and specialise them to SCSPs. Remember that F_P is the F subset that collects all the F functions which could alter the input problem P.

AXIOM **5.4.1** (COMMON FIXPOINT). Let F be a set of functions on $P\uparrow$. Suppose that $g(P') \neq P'$, for $g \in F$ and $P' \in P$. Then the *update* operator adds to G all the $F - G$ functions f such that $f(P') = P'$ while $fg(P') \neq g(P')$.

Algorithm 5.4.1: SGI(P, F_P, F)

$o := P$ `% complete input problem`;
$G := F_P$;
while $G \neq \emptyset$ **do**
  choose $g \in G$;
  $G := G - \{g\}$;
  $o' := g(o)$;
  if $o' \neq o$ **then**
    $G := G \cup \mathit{update}(G, F, g, o)$;
    $o := o'$

The following result is a corollary of Theorem 3.3.3 in Chapter 3, and it states the every execution of the SGI schema on SCSPs computes a common fixpoint of the iterated functions.

COROLLARY **5.4.4.** *Consider an SCSP P on $\mathcal{S} := \langle S, <_S, \times, \bot, \top\rangle$ and a finite set F of functions on a family $\mathcal{F}(P)$ of $P\uparrow$ problems, to which P belongs. Then every terminating execution of the* SGI *algorithm that satisfies the Common Fixpoint Axiom 5.4.1, with input P and F, computes a common fixpoint of all the functions from F.* □

5.4.3 The Role of Monotonicity

As in the case of crisp constraints, if an instance of SGI for soft SCSPs iterates functions which are monotone, with respect to a certain problem order, than that instance of SGI *always* returns the same common fixpoint of the iterated functions; i.e., *the least* problem, with respect to the assigned problem order, that is a common fixpoint of all the iterated functions. So let us first recast Axiom 3.3.2 for generic problem orderings — as for these, see Definition 5.3.10.

AXIOM **5.4.2** (LEAST FIXPOINT). The finite set F only contains monotone constraint functions over a problem ordering $\langle \mathcal{F}(P), \sqsubseteq, P\rangle$.

Given the above axiom, we infer the following corollary of Theorem 3.3.8.

COROLLARY **5.4.5.** *Consider an SCSP P on $\mathcal{S} := \langle S, <_S, \times, \bot, \top \rangle$, a finite set F of functions on a problem ordering $\langle \mathcal{F}(P), \sqsubseteq_P, P \rangle$, and assume the Common Fixpoint Axiom 5.4.1 and the Least Fixpoint Axiom 5.4.2. Then every terminating execution of* `SGI`*, with input P and F, computes the same problem: i.e., the least common fixpoint of all the F functions with respect to the problem ordering.* $\Box$

The above corollary is interesting in that it highlights the role of monotonicity in the confluence of `SGI` on SCSPs. This property for rule based constraint propagation was originally restricted in [BMR97] to the case of c-semirings with an idempotent $\times$ operation, see Theorem 5.4.1. Instead, Corollary 5.4.5 allows us to extend this to all executions of `SGI` with rules, independently of the idempotency of $\times$, or the fairness hypothesis.

COROLLARY **5.4.6.** *Let R be a finite set of rules for an SCSP $P := \langle X, \boldsymbol{D}, \boldsymbol{C} \rangle$. Then all the terminating executions of* `SGI`*, with R rules and P as input, compute the same problem. This is the least common fixpoint of all the rules of F with respect to the problem ordering on P.*

PROOF. Lemma 5.4.2 yields that any rule based on X is an inflationary and monotone soft constraint propagation function on $\langle P\!\uparrow, \sqsubseteq_P, P \rangle$. Corollary 5.4.5 implies now our claim. $\Box$

We now turn our efforts towards the issue of termination of `SGI` with SCSPs and soft constraint functions. This is not an easy task in case of SCSPs; in fact, even if SCSP domains are finite, the c-semiring may be infinite, which is obviously a source of possible non-termination. So, whereas for crisp CSPs inflationarity of functions and finite domains are feasible and sufficient assumptions to ensure the termination of `SGI` (see Corollary 3.3.10), we shall see, in the following subsection, that this is not the case for all SCSPs.

5.4.4 Termination

As remarked in the preceding subsection, the presence of an infinite c-semiring universe may lead to a constraint propagation algorithm which does not always terminate. In this subsection we focus on the issue of termination and investigate under which conditions the `SGI` schema *always* terminates. We obtain general termination conditions, that yield the termination of every soft constraint propagation algorithm that is an instance of `SGI`.

Problem orderings and termination

Our first termination result below, concerns itself with the problem order (see Definition 5.3.5): instead of demanding the finiteness of the ordering, we assume that its ascending chains have finite length. Then Theorem 3.3.9 guarantees the termination of the `SGI` schema, in case of soft constraint propagation functions which are inflationary and computable. In other words, if Axiom 3.3.3 is satisfied. We rewrite that axiom as follows, and specialise it to the case of SCSPs and soft constraint propagation functions.

Axiom **5.4.3** (termination).

- Each soft constraint function $f \in F$ is computable over a problem ordering $\langle \mathcal{F}(P), \sqsubseteq_P, P \rangle$.
- The F functions are inflationary with respect to the assigned problem order.
- The ordering $\langle \mathcal{F}(P), \sqsubseteq_P, P \rangle$ satisfies the *ascending chain condition* (ACC): i.e., each $\sqsubset_P$-chain in $P\!\uparrow$ is finite.

The following statement is an immediate consequence of Theorem 3.3.9.

Corollary **5.4.7** (termination 1). *Let us consider an SCSP problem P on a c-semiring $\mathcal{S}$ and a problem ordering $\mathcal{FP} := \langle \mathcal{F}(P), \sqsubseteq_P, P \rangle$ on P. Let us instantiate* `SGI` *with P and a finite set F of functions on $\mathcal{FP}$. Furthermore, let us assume the Common Fixpoint Axiom 5.4.3. Then every execution of this instance of* `SGI` *terminates, computing a common fixpoint of the F functions.* □

The above corollary can be used to prove termination in some cases, like the ones discussed in Section 5.5. However, it is a highly general result, so its hypothesis might be sometimes difficult to check. In turn, in some circumstances, specific properties of the adopted functions or c-semiring, easier to verify, can imply the assumptions of the above corollary, and thus immediately ensure the termination of `SGI`. This is the case of problem orderings step-by-step computed from the c-semiring operations, as made precise in the remainder of this section.

Semiring based functions and termination

The termination result in Corollary 5.4.7 refers to any inflationary functions on a problem ordering. In the following part, we explore and characterise some problem orderings and functions that satisfy Axiom 5.4.3.

The main motivation for relaxing the assumptions of Axiom 5.4.3 to a generic family of problems in $P\!\uparrow$ is that, in general, the whole family $P\!\uparrow$ contains too many problems with respect to those generated by specific constraint propagation algorithms. Above all, this is the case if we analyse constraint propagation algorithms to establish whether and under which conditions they termination. The following simple example, that pertains to a type of SCSPs of primary interest, explains our concern.

EXAMPLE **5.4.8.** We consider the fuzzy c-semiring $\langle [0,1], \leq, \min, 0, 1\rangle$, and a fuzzy CSP P over it. The problem P has variable domains equal to $\{a\}$ for both of its variables, x and y; P has just the trivial constraint $c = \langle \mathbf{1}, \{x, y\}\rangle$, where $\mathbf{1}$ is the constant function that assigns the value 1 to each possible instantiation of x and y in D. Then $P\uparrow$ is the class of *all* problems on x and y. It is evident that the problem order on $P\uparrow$ cannot satisfy the ACC.

Consider, for instance, the set of problems P_n in $P\uparrow$, the constraints of which are defined as follows:

- both unary constraints on x and y assign 1 to a, their only possible instantiation;
- the constraint of P_n on x and y assigns the value $1/n + 1$ to the pair (a, a).

The relation $P_n \sqsubset P_{n+1}$ is of strict order for every $n \in \mathbb{N}$, since $\langle 1/n : n \in \mathbb{N}\rangle$ is an infinite $<$-descending chain. So we have an infinite $\sqsubset$-chain in $P\uparrow$.

A similar example applies to probabilistic CSPs with max instead of min.

However, if we restrict our attention to soft constraint functions and families of problems that are, in some sense, finitely generated via those functions, we may avoid the flaws of the above example. First we characterise such families and then study `SGI` with them.

DEFINITION **5.4.9.** Consider an SCSP $P := \langle X, \boldsymbol{D}, \boldsymbol{C}\rangle$ on a c-semiring $\mathcal{S}$, and a finite set F of constraint functions on $P\uparrow$. Define $F(P)$ to be the following subset of $P\uparrow$:

- P belongs to $F(P)$;
- if P' belongs to $F(P)$, then there are finitely many F functions, say $g_1, \ldots, g_k$, such that P' is equal to $g_1 \cdots g_k(P)$;
- nothing else belongs to $F(P)$.

Then $\langle F(P), \sqsubseteq, P\rangle$ is the problem ordering that is *finitely generated* by F on P.

This is a typical construction in set theory, see [DP90].

The following statement is only Corollary 5.4.7 rephrased for orderings generated by F functions; there is nothing really new about it but the choice of orderings. However, we shall use it as follows in the remainder of this section.

Lemma **5.4.10.** *Given a c-semiring $\mathcal{S}$ and an SCSP problem P on it, instantiate the* SGI *algorithm with a finite set F of constraint functions and P. Suppose that Axioms 5.4.1 and 5.4.3 hold for $\langle F(P), \sqsubseteq_P, P\rangle$ and the F functions. Then every execution of the* SGI *algorithm terminates, computing a common fixpoint of the F functions.* □

In case of crisp constraints, such lemma could be applied to arc consistency problems and the ordering $F(P)$ would be contained in the domain ordering over P, see Section 4.2. However, in the soft case it may not be trivial to envisage $F(P)$, or a family that contains it and for which Axiom 5.4.3 holds. However, we do have the c-semiring ordering and operations, and so the question is whether these can be used to construct $F(P)$ as in the above lemma.

In the following, we focus on the c-semirings operations and partial order, and introduce the notion of semiring closure of a soft constraint problem; this mirrors the notion of ordering generated by functions.

Definition **5.4.11.** Consider an SCSP $P := \langle X, \boldsymbol{D}, \boldsymbol{C}\rangle$ over a c-semiring $\mathcal{S} := \langle S, <_S, \times, \bot, \top\rangle$, and the set $\boldsymbol{C}(P)$ of c-semiring values that occur in P: i.e.,

$$\boldsymbol{C}(P) := \bigcup_{C \in \boldsymbol{C}} \{C(d) \,:\, d \in D\,[s]\}\,.$$

Then the *semiring closure* of P, denoted by $\bar{\boldsymbol{C}}(P)$, is the smallest (with respect to set inclusion) of all sets B that enjoy the following properties:

1. $\boldsymbol{C}(P) \subseteq B \subseteq S$;

2. B is closed under $\vee$ and $\times$.

Notice that the previous definition is meaningful, since there always is a set that satisfies the two properties as above; that is, the c-semiring universe S itself.

What we need now is to single out those constraint functions that, applied to a problem P, compute values that are in the semiring closure of P. Such functions, intuitively, are defined via the two semiring operations, $\vee$ and $\times$. So, given an SCSP P, those functions will return values that are either in $\boldsymbol{C}(P)$ or obtained from elements of $\boldsymbol{C}(P)$ by means of $\vee$, $\times$, or their composition.

Definition **5.4.12.** Consider an SCSP P over $\mathcal{S}$, and a soft constraint function $g : P{\uparrow} \mapsto P{\uparrow}$. Then g is a *semiring based function* if, for every $P' \in P{\uparrow}$, it enjoys the following property:

$$\bar{\boldsymbol{C}}(g(P')) \subseteq \bar{\boldsymbol{C}}(P').$$

In other words, the c-semiring values that a semiring based function associates with a problem P' can all be found in the semiring closure of P'.

The line of our argument should be clear by now: since semiring based functions can only return values in the semiring closure of their input problem, the termination of `SGI` with such functions can be established by studying semiring closures.

So, let us turn our attention to semiring closures and semiring based functions as in the following lemma.

Lemma **5.4.13.** *Consider an SCSP P on $\mathcal{S}$ and a finite set of semiring based functions F. Assume also that the c-semiring order $<_S$ satisfies the Descending Chain Condition (DCC), when restricted to $\bar{\boldsymbol{C}}(P)$: i.e., there are no infinite descending $<_S$-chains of $\bar{\boldsymbol{C}}(P)$ elements. Then the problem ordering $F(P)$ satisfies the ACC.*

Proof. Suppose that the thesis of this lemma does not hold. That is, suppose that there is an infinite chain of $F(P)$ problems,

$$P = P_0 \sqsubset_P \cdots \sqsubset_P P_n \sqsubset_P P_{n+1} \sqsubset_P \cdots \tag{5.7}$$

where, for each $n \geq 1$, $P_n := g_1 \cdots g_k\,(P)$, for some $g_1, \ldots, g_k \in F$. Since all the F functions g are semiring based, we also have

$$\bar{\boldsymbol{C}}(P_n) \subseteq \bar{\boldsymbol{C}}(P), \tag{5.8}$$

for every $n \in \mathbb{N}$. We now aim at proving that an infinite descending $<_S$-chain can be extracted from (5.7), thereby contradicting our assumption on $<_S$.

In fact, (5.7) yields the existence of a constraint $C_n \in P_n$ that is strictly greater than $C_{n+1} \in P_{n+1}$; that is for which $C_n \sqsubset_S C_{n+1}$ holds. This is true for every $n \in \mathbb{N}$. As the chain in (5.7) is infinite, while the scheme X is finite, we can extract (at least) an infinite chain of constraints, all on the same scheme s, from the chain (5.7):

$$\cdots \sqsubset_S C_m(s) \sqsubset_S C_{m+1}(s) \sqsubset_S \cdots \tag{5.9}$$

Now, $C_m(s) \sqsubset_S C_{m+1}(s)$ means that there exists a tuple $d \in D\,[s]$ for which the following relation holds:

$$C_m(s)(d) >_S C_{m+1}(s)(d).$$

As the variable domain set $\boldsymbol{D}$ is finite while the chain in Equation (5.9) is infinite, we can extract an infinite chain of semiring elements from the chain (5.9) of the following form:

$$\cdots >_S C_k(d') >_S C_{k+1}(d') >_S \cdots \tag{5.10}$$

All the c-semiring elements that occur in (5.10) belong to $\bar{\boldsymbol{C}}(P)$ due to (5.8). Therefore, the restriction of the order $<_S$ to the set $\bar{\boldsymbol{C}}(P)$ does not satisfy the DCC, which contradicts our hypothesis. □

The following corollary follows now from Lemma 5.4.10 (which is Corollary 5.4.7, specialised to finitely generated orderings) via Lemma 5.4.13.

Corollary **5.4.14** (termination 2). *Consider an SCSP P over $\mathcal{S}$ and a finite set F of semiring based functions. Assume the Common Fixpoint Axiom 5.4.1. Suppose that the F functions are computable and inflationary over a problem ordering of the form $\langle F(P), \sqsubseteq_P, P\rangle$. Assume, also, that the c-semiring order $<_S$ satisfies the Descending Chain Condition (DCC) when restricted to $\bar{\boldsymbol{C}}(P)$; namely, there are no infinite descending $<_S$-chains of $\bar{\boldsymbol{C}}(P)$ elements. Then every execution of the* SGI *algorithm terminates, computing a common fixpoint of the F functions.* □

Nevertheless, even the assumptions of Corollary 5.4.14 may be difficult to check. In fact, it might not always be trivial to determine the semiring closure of a given SCSP P; furthermore, we should also check that the restriction of the semiring order to the closure satisfies the DCC. Nevertheless, if the multiplicative operation of the semiring is idempotent, then the semiring closure of any given problem is always finite, and hence satisfies the DCC.

Corollary **5.4.15** (termination 3). *Consider an SCSP P on $\mathcal{S}$ and a finite set of F of semiring based functions. Assume the Common Fixpoint Axiom 5.4.1. Suppose that these are inflationary on a problem ordering defined on $F(P)$. Assume, also, that the $\times$ operation of $\mathcal{S}$ is idempotent.*

- *Then the semiring closure of P is finite.*
- *Thus every execution of the* SGI *algorithm terminates, computing a common fixpoint of the F functions.*

Proof. We only need to prove the first item, and the second follows then from Corollary 5.4.14. First, let us recall that a c-semiring with an idempotent $\times$ operation is a distributive complete lattice, see Lemma 5.2.1. Furthermore, every finitely generated sublattice of a distributive lattice is finite, see [DP90]. Thus every finitely generated sublattice of a c-semiring with idempotent $\times$ operation is finite.

Now, the set $\boldsymbol{C}(P)$ of all semiring elements in P is finite, since every SCSP has finitely many constraints, finite domains and finitely many variables. So the semiring closure of P is finitely generated. Therefore the semiring closure of P is finite, due the aforementioned result of lattice theory. □

Notice that the above two results concerning terminations are the analogues of Theorem 4.14 in [BMR97]. However there the set $\bar{\boldsymbol{C}}(s)$ is *assumed* to be finite in order to guarantee the termination of a constraint propagation algorithm. This hypothesis is much more restrictive, and implies ours in Corollary 5.4.14. Thus Theorem 4.14 in *ib.* is a special case of our Corollaries 5.4.14 and 5.4.15.

Finale

Our results concerning termination can suggest various strategies for proving the termination of `SGI` instances, like the following ones.

- If the constraint functions are semiring based and the multiplicative operation $\times$ of the c-semiring is idempotent, then we resort to Corollary 5.4.15.
- If $\times$ is not idempotent, but the soft constraint functions are semiring based, we can check whether the restriction of the semiring order to the semiring closure of P satisfies the DCC and appeal to Corollary 5.4.14.
- If the restriction of the semiring order to the semiring closure of P does not satisfy the DCC or the soft constraint functions are not all semiring based, then we can try to prove that the problem order on $P\uparrow$, or a smaller problem ordering on P satisfies the ACC. If the F functions are inflationary with respect to that ordering, then `SGI` terminates by Corollary 5.4.7.

In Section 5.5 below, we investigate some examples of constraint propagation algorithms for SCSPs, and briefly analyse them by means of the theoretical results of the present section.

5.5 Soft Constraint Propagation Algorithms

In this section, we briefly survey several soft constraint propagation algorithms and show how the general results concerning `SGI` over soft CSPs can be used to both describe and analyse those specific algorithms. We restrict our attention to arc consistency algorithms over SCSPs; for these are the most used constraint propagation algorithms over SCSPs and easier to explain.

Our account of constraint propagation algorithms in this section has no pretence of completeness. However, it proves that a number of constraint propagation algorithms for SCSPs are extensions of their corresponding crisp counterparts, since all of them are instances of `SGI`. By these claims, we mean that the basic process of "prune-and-propagate" is carried over, and only slightly modified by passing from crisp to soft CSPs:

- pruning of domain or constraint values is transformed as reduction to more restrictive constraint problems, so to speak;

- whereas the propagation phase is characterised as in the crisp case, because *update* is not substantially different.

We start our survey and analysis with the fuzzy and probabilistic cases, based on the $\leq_R$ total order over real numbers, and the max or min operator. Then we deal with a form of generalised arc consistency for soft constraint problems, that is defined through functions which are not finitely generated by means of the two c-semiring operations. Finally, we briefly present the partial arc consistency counter algorithm by Freuder and Wallace, and show that this simple constraint propagation algorithm is an instance of `SGI` too.

5.5.1 Probabilistic and Fuzzy Arc Consistency Algorithms

Let us consider, as in Subsection 5.2.3, a fuzzy CSP $P := \langle X, \boldsymbol{D}, \boldsymbol{C} \rangle$ based on the c-semiring $\langle [0,1], \leq_{\mathbb{R}}, \min, 0, 1 \rangle$. Notice that $\times$ is the minimum operator, so it is idempotent and the underlying c-semiring is a distributive lattice.

Then, given a unary constraint C_i, we define, for each x_j such that $i \neq j$, a function of the form $f(x_i, s)$; the scheme s is equal either to $\langle x_i, x_j \rangle$ if $i < j$, or to $\langle x_j, x_i \rangle$ if $j < i$. Then, $f(x_i, s)$, if applied to P, returns a problem P' that differs from P at most in its unary constraint $C'(x_i)$, that is defined as follows:

$$\begin{aligned} E(s)(d) &:= \min \{C(x_i)(d[x_i]),\, C(s)(d),\, C(x_j)(d[x_j])\}, \text{ for every } d \in D[s], \\ C'(x_i)(a) &:= \max \{E(s)(d) \,:\, d \in D[s] \text{ and } d[x_i] = a\}, \end{aligned} \tag{5.11}$$

in which $C(x_i)$, $C(x_j)$ and $C(s)$ are all P constraints. Each $f(x_i, s)$ function is a soft constraint function on $F(P)$, that collects all problems P' such that $P \sqsubseteq P'$, and P differs from P' at most in the unary constraints.

Thus, Corollary 5.4.15 can be directly employed to prove that any constraint propagation algorithm with $f(x_i, s)$ functions such as (5.11) always terminates, whenever it is an instance of `SGI`. The returned problem is a common fixpoint of the iterated functions as in (5.11).

Since those functions are also monotone with respect to the problem order $\sqsubseteq$ on $F(P)$, then all the executions of the `SGI` algorithm with them computes the same problem; namely, the least common fixpoint with respect to the problem order, that is more constraining than P. This is due to Corollary 5.4.5.

Notice that each function $f := f(x_i, s)$ such as (5.11) preserves equivalence:

$$Sol(P) = Sol(f(P)).$$

It is not a difficult exercise to verify that the above relation holds, since max and min are both idempotent. We refer the reader to [Gen01a, Gen01b] for a deeper analysis on this topic.

Finally, let us consider a probabilistic CSP $P := \langle X, \boldsymbol{D}, \boldsymbol{C} \rangle$ based on the c-semiring $\langle [0,1], \geq_{\mathbb{R}}, \max, 0, 1 \rangle$. Notice that $\times$ is the maximum operator, so it is idempotent and the underlying c-semiring is a distributive lattice.

As in the fuzzy case, arc consistency is enforced by means of functions of the form $f(x_i, s)$. When this function is applied to P, it returns a problem P' that differs from P at most in its unary constraint $C'(x_i)$, that is defined as follows:

$$E(s)(d) := \max\{C(x_i)(d[i]),\, C(s)(d),\, C(x_i)(d[j])\}, \text{ for every } d \in D[s],$$
$$C'(x_i)(a) := \min\{E(s)(d) : d \in D[s] \text{ and } d[x_i] = a\}, \tag{5.12}$$

in which $C(x_i)$, $C(x_j)$ and $C(s)$ are all P constraints. As in the case of fuzzy CSPs, we conclude that any constraint propagation algorithm with $f(x_i, s)$ functions as in (5.12) always terminates, whenever it is an instance of SGI (by Corollary 5.4.15). The returned problem is the least common fixpoint of the iterated functions as in (5.12), with respect to the problem order (by Corollary 5.4.5).

5.5.2 Generalised Arc Consistency Algorithms

The above functions in Equations (5.11) and (5.12) are semiring based functions, as characterised in Definition 5.4.12. However, the user may profit from having at her/his disposal further functions than those provided by the chosen c-semiring.

To our knowledge, it was first Schiex to underline the faults of soft arc consistency for weighted CSPs and propose a solution to it. In fact, as shown for instance in [Sch00], if only addition is used to enforce soft constraint propagation, equivalence gets lost. So, in [Sch00, CS01], arc consistency is refined by introducing a sort of inverse operation to addition; this new operation is not — in our terminology — semiring based.

Algorithm 5.5.1: `AC-proj` and `AC-join`

procedure `AC-proj`$(C(s), x_i, a)$
  $\beta := \min\{C(s)(t \cup \{a\}) : t \in D[s - \{x_i\}]\}$;
  $C'(x_i)(a) := \min\{C(x_i)(a), \beta\}$;
  if $C'(x_i)(a) \neq C(x_i)(a)$ **then**
    $C(x_i)(a) := C'(x_i)(a)$;
    for $d \in D[s]$ **do** $C(s)(d, a) := C(s)(d, a) - \beta$;

procedure `AC-join`$(C(s), x_i, a)$
  for $d \in D[s - \{x_i\}]$ **do**
    $\beta := \sum_{x_j \in s} C(x_j)((d \cup \{a\})[x_j])$;
    $\gamma := \max\left\{C(s)(d \cup \{a\}), \sum_{x_j \in s} C(x_j)((d \cup \{a\})[x_j])\right\}$;

The resulting algorithm in [CS01] iterates two sorts of functions, illustrated as in Algorithms 5.5.1. The algorithm of [CS01] can be easily turned into an instance of SGI. Hence this explains how the algorithm computes a common fixpoint of all the iterated functions. The algorithm of [CS01] preserves equivalence, but neither its confluence nor its termination can be ensured in the general case; in fact, functions for this are neither inflationary nor monotone over the soft constraint orderings.

5.5.3 Maximal CSPs

An interesting case of weighted CSPs, see Subsection 5.2.3, is constituted by maximal CSPs, briefly max-CSPs. Given an over constrained CSP, namely a CSP that does not admit solutions, Freuder et al. (see [FW92]) devised a series of algorithms to *maximise* the *number* of satisfied constraints, regardless of their importance. In other words, each constraint can assign one out of two levels of preference to each constraints; either 0 (yes), or 1 (no). Then, at each step of the basic algorithm in [FW92], the number of constraints, unsatisfied by the extension d' of the current assignment d, gets computed; then this number is compared with the stored number of unsatisfied constraints by a previously computed total assignment — the initial stored number being 0 or another lower bound, chosen by the programmer. If the stored number is less than the new computed value, then search is abandoned along the path of d'; else it continues to extend it to another variable. This basic branch-and-bound algorithm terminates when the number of violated constraints is minimised, and a total assignment that does so is returned.

In [FW92], Freuder and Wallace devise also a form of arc consistency for max-CSPs, named partial arc consistency. Their partial arc consistency counter (`PACC`) algorithm computes the number of satisfied constraints for each variable instantiation. The `PACC` algorithm is extended, by means of `SGI`, to its hyper arc version, namely the Partial Generalised Arc Consistency Counter (`PGACC`) Algorithm 5.5.2 displayed as below. This is a naive algorithm, and it could be improved in a number of way, for instance by ordering variables. However, we only aim at showing that `PACC` is an instance of the more general schema, `SGI`, for soft constraint propagation.

First we regard the given max-CSP as a weighted CSP, so that its constraints can only assign either 0 or 1 to their tuples. Therefore our extension of the partial arc consistency counter algorithm becomes straightforward. In fact, instead of assuming only binary constraints, we check that a variable instantiation is consistent with constraints of any arity. As for this, we introduce new unary constraints in the original problem; namely we define a "counter constraint" $CC(i)$ for each variable x_i of the problem. So each $CC(i)$ assigns a natural number to each value of the original CSP domain D_i; whereas all the original problem constraints can only assign either 0 or 1 to their tuples. In other words, $CC(i)$ is assumed to count the number of satisfied constraints for each assignment a of the variable x_i. If no prior knowledge is assumed, each $CC(i)$ of the extended input problem, denoted by P, assigns 0 to each value in D_i.

The functions, iterated by the PGACC algorithm, are defined as follows: for each value a in D_i, the input problem domain, we define a function $f(x_i, a)$ of the form $\min\{\_, \prod \sum \_\}(a)$. So, given a problem $P_i \supseteq P$, the computed problem $P_{\mathtt{k+1}} := f(x_i, a)(P_{\mathtt{k}})$ differs from $P_{\mathtt{k}}$ at most in the added constraint $CC_{\mathtt{k+1}}(x_i)$ that is characterised as follows:

$$CC_{\mathtt{k+1}}(x_i)(a) := \min\left\{C_{\mathtt{k}}(x_i), \prod_i \sum \{C_{\mathtt{k}}(s) \,:\, x_i \in s,\ s \neq x_i\}\right\}(a).$$

Notice that each selected function is removed after being applied in PGACC, and that *update* is always empty; in fact the adopted functions $f(i, a)$ are all idempotent and commutative. Therefore, there is no propagation as, commonly, in the case of classical crisp constraints.

Algorithm 5.5.2: PGACC(P, F)

$P :=$ given problem, extended with GC;
$G := F$;
while $G \neq \emptyset$ **do**
  choose $f(x_i, a) \in G$;
  $P' := f(x_i, a)(P)$;
  $G := G - \{f(x_i, a)\}$

The above algorithm can only count the number of constraint violations for each assignment. It could be modified so as to incorporate a propagation phase, as suggested in see [FW92].

The fact that PGACC terminates is obviously true, since there is no propagation phase; i.e., *update* is empty. The termination of the PGACC algorithm is also a trivial consequence of Corollary 5.4.7, since the generated problem order on P is always finite, and hence it satisfies the ACC.

5.6 Conclusions

5.6.1 Synopsis

In the present chapter, constraints with preference values, i.e., soft constraints, are introduced and a number of constraint propagation algorithms for these are studied. So, the SGI schema is used to *represent* and *analyse* semiring-based constraint propagation algorithms, as we did in Chapter 4 for classical constraints. Therefore, the SGI algorithm provides a general schema for soft constraint propagation as well. Again, the results obtained for the schema are applicable to all its soft instances as straightforward corollaries.

In particular, termination is not an easy issue in the context of soft constraint propagation algorithms: yet our analysis stresses the role that, in this, inflationarity of soft constraint functions and well-founded orders play. This results in three general termination properties that can be applied in different soft frameworks, as we summarise at the end of Subsection 5.4.4.

5.6.2 Discussion

The `SGI` algorithm schema iterates functions according to a certain strategy; however these are not required to have special properties for the algorithm to compute a common fixpoint of theirs. Due to this generality, we are able to instantiate this schema to a larger class of soft constraint propagation algorithms than the rule based schema of [BMR97] can account for, see Section 5.5.

Also, properties of functions, i.e. monotonicity and inflationarity, are studied as separate issues in Chapter 3. Thus their respective roles in connection with certain behaviours of soft constraint propagation algorithms are differentiated. This is already an achievement: in the soft constraint literature, often, the two properties are studied together and the role of each in the analsys of soft constraint propagation thus gets lost. This distinction can help in the design of new algorithms: e.g., it is often the case that in planning we do not want an algorithm to terminate, and an optimal solution is the one that maximises the number of satisfied constraints; hence inflationarity is a property that should be overlooked in the design of algorithms for such problems. On the other hand, the capability to predict the outcome of the algorithms' computations could be essential; thus monotonicity becomes an important requirement.

In the remainder of this part, we conclude our theoretical analysis of constraint propagation algorithms: in the following chapter, we summarise and specify all the functions used to characterise them.

Chapter 6

Constraint Propagation Functions

6.1 Introduction

6.1.1 Motivations

The relational model is one of the best-known database models, see [Ull80]: the primitive entities are relations, represented as tables, and operations on these. The relational model has at least two advantages: it is easy to grasp; it supports a high-level programming language called SQL (Structured Query Language) that allows the user to query the database, update and retrieve data stored in tables through a number of functions.

In Chapters 4 and 5 we described and analysed, via iterations of certain functions, a number of constraint propagation algorithms for classical and soft constraints, respectively. At this point, we collect all those functions in a homogeneous setting, thereby putting forward the resemblances and differences of those constraint propagation algorithms and the query programs in the relational database model.

6.1.2 Outline and Structure

In the introduction to Chapter 3, we claim that our theorisation of constraint propagation algorithms has two aims:

- one of describing and analysing constraint propagation algorithms in terms of function iterations; this is accomplished in Chapter 3;
- the other of abstracting *which* functions perform the task of pruning or propagation of inconsistencies in constraint propagation algorithms, in both the crisp (see Chapter 4) and soft (see Chapter 5) cases.

In Chapter 3 we do not specify which functions are used for constraint propagation — there we only study properties of functions as traced in constraint propagation algorithms. We do so in this chapter; thus we complete our theoretical work and tackle the task in the latter item.

Section 6.2 characterises the basic and derived operations that are traced in the representation and analysis of classical constraint propagation algorithms in Chapter 4. Finally, in Section 6.3, we define functions useful in the description and analysis of soft constraint propagation algorithms as in Section 5.5.

6.2 Functions for CSPs

In what follows, we usually need to fix a scheme X and a domain set $\boldsymbol{D}$ on X. Since structures of the form

$$CS := \langle X, \boldsymbol{D} \rangle$$

will often recur in the remainder of this chapter, we name them *constraint systems*. We follow the conventions of Subsection 2.2.1, and let D denote the domain $D_1 \times \cdots \times D_n$. Then for a *constraint* $C(s)$ *of* CS we mean a constraint on a scheme s of X, that is a subset of $D\,[s]$. In the limit, $D\,[s]$ is a CS constraint as well; i.e., the universal constraint on s.

6.2.1 Atomic Formulas

The user of a database will often query the database to select information that matches some criteria: for instance, the database could store bibliographical data, and the user, a librarian, could be interested in selecting all books in a category that were published before or after a certain year. Such criteria are usually expressed through numerical formulas such as $2 < 3$, meaning that every value in the second column must be less than the one in the third column that is in the same row.

We use something similar to formulas like $2 < 3$ for defining functions for constraint propagation in Chapter 4: for instance, functions for `HAC-4` (see Subsection 4.2.3) are defined through a selection operation such as

$$sel_{x_i=a} R(s)$$

that extracts all tuples d from $R(s)$ for which the equality $d\,[x_i] = a$ holds. The following definition then aims at characterising the basic formulas, such as $x_i = a$, that are used to express criteria for selecting data from CSP domains.

Definition **6.2.1.**
(i). Consider a constraint system $CS := \langle X, \boldsymbol{D} \rangle$ with variable scheme $X := \langle x_1, \ldots, x_n \rangle$ and domain $D := D_1 \times \cdots \times D_n$. The set of CS *atomic formulas* is defined as follows:

- $t = d$ and $t \neq d$ are atomic formulas for each scheme, scheme t of X and tuple $d \in D\,[t]$;

- $t \in S[t]$ is an atomic formula for each scheme t of X and constraint $S(t)$ of CS;

- nothing else is an atomic formula.

(ii). Consider a scheme s of X, a scheme t of s, and assume that $C(s)$ is a constraint of CS on the scheme s. Given a tuple $e \in C(s)$ and an atomic formula of the form $t = d$, we say that $C(s)$ *satisfies* $t = d$ in d if $e[t] = d$; similarly, $C(s)$ satisfies $t \neq d$ in e if $e[t] \neq d$; it satisfies $t \in S$ in e if $e[t] \in S$.
(iii). A *CS formula* $\psi(t)$ is an atomic formula or a finite conjunction of CS formulas: i.e.,

$$\psi(t) := \bigwedge_{i=1}^{n} \psi_i(t_i)$$

where each $\psi_i(t_i)$ is an atomic formula and t is the join $\bigcup_{i=1}^{n} t_i$. If $C(s)$ is a constraint of CS and t is a scheme of s, then $C(s)$ *satisfies* $\psi(t)$ in $e \in C(s)$ if it satisfies each atomic subformula $\psi_i(t_i)$ of $\psi(t)$ in e.

The definition of conjunctive formula is not, loosely speaking, necessary if we do not mind the order in which elements are selected: in this case, its effect can be obtained by composing a finite number of basic functions, as defined in Subsection 6.2.2 below. However, conjunctions of atomic formulas are useful to introduce derived formulas such as the following, which is itself a convenient shorthand: given a scheme s of X, a finite number of schemes $t_1, \ldots, t_k$ of s, and t equal to the join of these, put

$$t \subseteq D[s] := t_1 \in D[s] \wedge \cdots \wedge t_k \in D[s].$$

Notice also that, if S is a finite constraint over s, i.e.,

$$S = \{d_1, \ldots, d_n\},$$

then the formula $t \in S[t]$ is equivalently rewritten as the conjunctive formula

$$t = d_1[t] \wedge \cdots \wedge t = d_n[t].$$

Hence, if S is as above, we can also introduce the following shorthand

$$t \notin S[t]$$

to denote the finite conjunction $t \neq d_1[t] \wedge \cdots \wedge t \neq d_n[t]$.

6.2.2 Basic Functions

In what follows, we assume that a constraint system $CS := \langle X, \boldsymbol{D} \rangle$ is given. As usual, D denotes the domain $D_1 \times \cdots \times D_n$ and $D[s]$ the domain over the scheme s. Constraints and domains over CS are denoted by the letters R or S, with additionally superscripts. At this point, we can distinguish a number of effectively computable *basic functions* over CS as follows.

Union. Set-union of R and S over the same scheme s.

Join. Let R be over the scheme s and R' over the scheme t. The join of R and S, denoted by $R \bowtie S$, is a relation over the scheme $s \cup t$ defined as follows: $e \in R \bowtie S$ iff there exists $d \in R$ and $d' \in R'$ such that $d(s) = e(s)$ and $d'(t) = e(t)$.

Projection. Given R over s and a subsequence t of s, the projection $\Pi_t(R)$ of R over t is the set of tuples d for which there exists $e \in R$ and $e[t] = d$.

Universal selection. Consider a CS constraint $R(s)$ and a CS formula $\psi(t)$, for t a subscheme of s. Then $\forall\_sel_\psi R$ is the subset of all R tuples d such that $d[t]$ satisfy $\psi(t)$.

Existential selection. Consider a CS constraint $R(s)$ and let $\psi(t)$ be a CS formula, for t subscheme of s. Then $\exists\_sel_\psi R$ is either a singleton $\{d\}$ for $d \in R$ such that $d[t]$ satisfies $\psi(t)$, or the empty set if no tuple in R satisfies $\psi(t)$.

6.2.3 Constraint Propagation Functions

The set of *constraint propagation functions* over CS is the smallest inductive set of functions that contains the basic functions and is closed under composition.

Simple examples are the difference function and the Cartesian product.

Example **6.2.2.**

Intersection. Set-intersection of R and S, over the same scheme s, is defined by means of the join operation: $R \cap S := R \bowtie S$.

Difference. Consider two finite constraints R and S over the same scheme s. Then the difference of R and S, denoted by $R - S$, is $\forall\_sel_{s \notin S} R$.

Cartesian product. Let R be over s and S over s' such that s and s' are disjoint schemes. Then $R \times S$ is the join of R and S: i.e., $R \bowtie S$.

The two relations in Example 6.2.2, finite difference and Cartesian product, are obtained, by means of composition, from basic constraint propagation functions. The following example proposes a constraint propagation function that is instead generated from a derived constraint propagation function, i.e., the Cartesian product.

EXAMPLE **6.2.3.**

Cartesian product operator. Given two constraint propagation functions f and g, let $f{\times}g$ be the function defined as

$$f{\times}g(R,S) := f(R) \times g(S)$$

for every R and S. Then $f{\times}g$ is the Cartesian product of f and g.

6.3 Functions for SCSPs

In Chapter 5, we surveyed some constraint propagation algorithms for soft constraints. The definition of basic functions for those algorithms is complicated by the presence of preference structures as c-semirings.

A *c-semiring constraint system* CS carries over the c-semiring structure $\mathcal{S} := \langle S, <_S, \times, \perp, \top \rangle$ used to define soft constraints, see also [BMR97]:

$$CS := \langle X, \boldsymbol{D}, \mathcal{S} \rangle,$$

where X and $\boldsymbol{D}$ are as in the classical case. As for the rest, we follow the conventions introduced in Section 5.2, and define a *soft constraint of* CS as a soft constraint on a scheme s of X, that maps $D\,[s]$ into the c-semiring universe S.

In Section 5.5, it is shown how a number of soft constraint propagation algorithms employ different functions from, in our terminology, the semiring based functions (see Definition 5.4.12). Those functions are strictly dependent on the specific c-semiring structure adopted, or depend on some extensions of it, see Subsection 5.5.2. Therefore, it is rather difficult to provide a sufficiently general definition, unless the c-semiring framework is replaced by a more general structure; such as universal algebras, with a lattice structure on the underlying universes, [Gen01a].

6.3.1 Soft Constraint Propagation Functions

Here we limit ourselves to summarise the basic functions used for soft constraint propagation, without any pretence of completeness.

Union. Given two constraints R and S over the same scheme s, the union of R and S is the constraint on s defined as follows: $R{\cup}S(d) = R(d) \vee S(d)$, for each $d \in D\,[s]$.

Join. Let R be over the scheme s and R' over the scheme t. The join of R and S, denoted by $R \bowtie S$, is a relation over the scheme $r := s \cup t$ defined as follows:

$$R \bowtie S(e) = R(e) \times S(e),$$

for every $e \in D\,[r]$; see p. 92.

Projection. Given R over s and a subsequence t of s, the projection $\Pi_t(R)$ of R over t is the function from $D\,[s]$ to the c-semiring universe S that maps each $d \in D\,[s]$ to the c-semiring value $\bigvee \{e \in D\,[s] \,:\, e\,[t] = d\}$; see p. 92.

It is also possible to generalise, to some extent, the operations of selection from the classical to the soft case. We do it as below and then explain how these could be useful in the design of constraint propagation or solving algorithms for SCSPs.

First of all, notice that universal selection amounts to assigning $\top$ to all the tuples for which a certain formula holds true; and implicitly assigning $\perp$ to all the other ones. On the other hand, existential selection returns $\top$ only to the first tuple that is found to satisfy a certain formula; $\perp$ to all tuples if none satisfies the formula. However, a c-semiring offers us more than two values. Thus, we modify the definition of CS formulas in Definition 6.2.1 as follows.

Definition **6.3.1.**
(i). Consider a constraint system $CS := \langle X, \boldsymbol{D}, \mathcal{S}\rangle$ with variable scheme $X := \langle x_1, \ldots, x_n\rangle$ domain $D := D_1 \times \cdots \times D_n$, and c-semiring $\mathcal{S}$. The set of CS *atomic formulas* is defined as follows:

- $t = a$ and $t \leq a$ are atomic formulas for each scheme t of X, tuple $d \in D\,[t]$ and c-semiring value a;
- $t \neq a$ and $t \geq a$ are atomic formulas for each scheme t of X, tuple $d \in D\,[t]$ and c-semiring value a;
- nothing else is an atomic formula.

(ii). Consider a scheme s of X, a subsequence t of s, and assume that $C := C(s)$ is a soft constraint of CS on the scheme s. Given a tuple $e \in D(s)$ and an atomic formula of the form $t = a$, we say that C *satisfies* $t = a$ in e if $\Pi_t(C(e)) = a$; similarly, C satisfies $t \neq a$ in e if $\Pi_t(C(e)) \neq a$; it satisfies $t \leq a$ in e if $\Pi_t(C(e)) \leq a$, and $t \geq a$ in e if $\Pi_t(C(e)) \geq a$.
(iii). A *CS formula* $\psi(t)$ is an atomic formula or a finite conjunction of CS formulas: i.e.,

$$\psi(t) := \bigwedge_{i=1}^{n} \psi_i(t_i)$$

where each $\psi_i(t_i)$ is an atomic formula and t is the join $\bigcup_{i=1}^{n} t_i$. If $C(s)$ is a constraint of CS and t is a subsequence of s, then $C(s)$ *satisfies* $\psi(t)$ in $e \in C(s)$ if it satisfies each atomic subformula $\psi_i(t_i)$ of $\psi(t)$ in e.

Note that we choose a local definition of satisfiability: we evaluate $\psi(t)$ true in $C(s)$ *and* a tuple $e \in D(s)$. Other choices are possible: for instance, we could define satisfiability with respect to $C(s)$ and the maximum value returned by $C(s)$ on $D[s]$, i.e., by using the projection function Π_t on the whole domain $D[s]$. However, this local definition of satisfiability is sufficient to characterise a universal selection function for soft constraints as a generalisation of the corresponding function for classical constraints.

Universal selection. Consider a relation $R(s)$ and a CS formula $\psi(t)$, for t a subscheme of s. Then $\forall\_sel_\psi R$ is defined as follows: for each $d \in D[s]$,

$$sel_\psi R(d) := \begin{cases} R(d) & \text{if } R \text{ satisfies } \psi \text{ in } d, \\ \bot & \text{otherwise.} \end{cases}$$

The definition of selection could be more general: for each c-semiring value a, we could have a selection function $\forall\_sel_{\psi,a}$ that maps each tuple d that does not satisfy ψ to a.

6.3.2 On Optimal Strategies

At this point, let us revisit the definition of solution constraint for an SCSP $P := \langle X, \boldsymbol{D}, \boldsymbol{C} \rangle$, cf. p. 92:

$$Sol(P, s) := \Pi_s \bowtie_{C \in \bar{P}} C,$$

Suppose that the user queries a soft constraint system and only wants to retrieve *all* those tuples $d \in D[s]$ from the system for which $Sol(P, s) \geq a$, where a is a certain preference value in the c-semiring universe, that the user chooses. Then $\forall\_sel_{s \geq a}$, applied to $Sol(P, s)$, will return the user only those tuples to which $Sol(P, s)$ assigns a value greater than a.

Incidentally, notice that the above method for computing solutions does not seem highly efficient: in fact, first we have to compute $\bowtie$, then Π and, only afterwards, select those tuples. A better choice would be to apply a selection function as soon as possible, that is before applying $\bowtie$. But then $\bowtie$ would return a constraint on X, the scheme of the CSP P, that assigns $\bot$ to all the tuples in the domain D. The situation can be remedied by adopting the more general definition of selection so that this assigns to each tuple, that do not satisfy a formula, the maximum value $\top$ of the c-semiring.

It would be interesting to see whether this abstract view on constraint propagation and solving functions could be useful to tackle optimisation tasks, as in the above case:

- which function is it better to apply first to satisfy the user's query in an "economic" manner from the viewpoint of computations?

Finally, a soft generalisation of existential selection can be given as in the case of the universal selection function; we let the reader spell out the details. This function could be applied, for instance, whenever a user wants to retrieve only *a* solution that satisfies a given formula, and not all of them.

6.4 Conclusions

6.4.1 Synopsis

In Chapter 3, a template for constraint propagation algorithms is proposed: that is the `SGI` algorithm schema, which iterates functions according to a certain strategy. The present chapter collects the functions used for constraint propagation, in both the crisp and soft case, in a homogeneous setting; the class of functions in `SGI` iterations is thus restricted to those that are actually traced in constraint propagation algorithms in Chapters 4 and 5.

Also, this digest highlights that a number of operations are common to constraint propagation algorithms and to languages for manipulating data in relational database systems. We briefly elaborate on this issue as below.

6.4.2 Discussion

The theoretical analysis of constraint propagation algorithms, proposed in this chapter and in Chapter 3, can immediately be used for optimisation tasks. Query optimisation is already a well explored topic in the database community. Most optimisation strategies involve transforming algebraic expressions; for instance, if two operations commute, then the order in which these are applied is not relevant; therefore, the optimal sequence of applications is obtained by applying first the least expensive operation. We encounter something similar in Chapter 3: a number of properties of functions, such as stationarity or commutativity, are proved to optimise the performance of the basic `SGI` schema; these properties are traced in a number of constraint propagation algorithms in Chapter 4.

Therefore, a closer investigation of the similarities and differences between the two worlds, that of constraint algorithms and that of database query languages, appears to be promising for optimisation tasks. The fact that there is a common language, that of operations on relations as explained in this chapter, helps to clarify further the connections between database theory and CSP algorithms, and should help in transferring the acquired knowledge and the developed strategies from one field to the other.

In [Var00], the author showed how certain classes of finite CSPs can be reduced, in polynomial time, to Datalog programs or view-based query answering, and vice versa; notice that, there, the aim is to identify tractable *classes* of CSPs; not to analyse the behaviour of each single *algorithm*, as instead we do. Also, the analyses in [GLS99] have the same objective as those of [Var00]. However, the theoretical analysis of constraint propagation algorithms, proposed in this chapter and in Chapter 3, could also help in this respect.

In the following part, relations continue to be the protagonists of this thesis. Relations and relational structures are in fact at the basis of modal logics; these make use of restricted formal languages for describing relations and relational structures: loosely speaking, properties of these are expressed as theorems of modal logics. Thus we shall also see how constraint programming, which manipulates relations, can be used for automated theorem proving in modal logics.

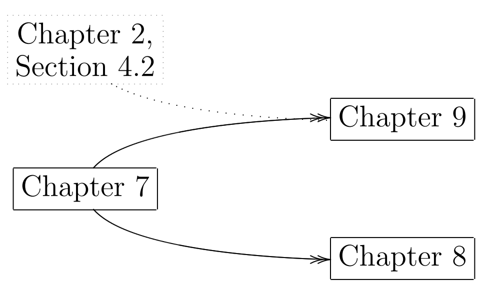
Part II

Diamond Satisfaction

"Niente affatto", disse il secondo, "la virtù sta nel mezzo". ("Not at all", said the second, "virtue is most often found in the middle of the road".)
G. Rodari, Vecchi Proverbi, from *Favole al Telefono*, Einaudi, 1971.

After five chapters devoted to CSPs and efficient reasoning on them, we change tack quite drastically. This part of the thesis is concerned with automated theorem proving in modal logics. The standard relational translation of modal languages into first-order languages is introduced in **Chapter 7**. Then its refinement, called the "layered translation", is presented in **Chapter 8**, and proved to preserve satisfiability in the case of basic modal logics. In **Chapter 9**, we see how the same semantic intuitions which motivate the layered translation give rise to constraint satisfaction solvers for basic modal logics. So in the end we return to constraints and CSPs after all.

Chapter 2 and the introduction to Section 4.2 are needed for the comprehension of Chapter 9. The following diagram summarises the dependencies of the chapters or sections in this second part, and their dependencies on some material in Part I (see the dotted box).



Chapter 7

Modal Logics

7.1 Introduction

7.1.1 Motivations

Modal logic [BdRV01] was originally conceived as the logic of necessity and possibility. Indeed, for many years modal logic was viewed as an extension of propositional logic by the addition of the modal operators $\Diamond$ (possibly) and $\Box$ (necessarily). To some extent this picture is still valid and useful. For instance, in the branch of modal logic that is known as *epistemic logic*, the modal language is used to reason about the knowledge of an agent. Under this reading, $\Box\phi$ stands for "the agent knows that ϕ".

But over the past decade or so, the picture has changed, or rather, *broadened* considerably: modal logic has developed into a powerful discipline on the interface of computer science and mathematics that deals with *restricted* logical languages for talking about various kinds of relational structures (see [Are00]). Let us elaborate on this. First, relational structures (that is, sets equipped with relations on them) are to be found just about everywhere. For example, in computer science, we use labelled transition systems (LTSs) to model program executions, but an LTS is just a set (the states) together with a collection of binary relations (the transition relations) that model the behaviour of programs [HR00]. Second, modal languages are restricted languages because they talk about relational structures in a special way: modal formulas are evaluated locally, *at a particular state*, and only the states that are linked to the current state through a relation may be explored. Because of such restrictions, many modal logics end up being fragments of first-order logic. Moreover, they often end up being *decidable* fragments of first-order logic. The decidability of many important modal systems stems from the step-by-step way that modal formulas are evaluated. More generally, the latter helps to explain why many modal logics enjoy the so-called *tree model property*: if a formula has a model, it has a model that looks like a tree. The tree model property has become a key tool in establishing decidability and complexity results for modal languages [Grä99, Var97]. And as we shall see below, a very strong form of the tree model property can be used to devise practical algorithms for modal languages — but this is running ahead of things.

We have to address another issue first: what do constraints and constraint propagation have to do with modal logic? Sitting, as it does, between first-order and propositional logic, two natural strategies suggest themselves for reasoning with modal logic:

- constrain first-order methods so that they become decision procedures for modal logics,
- boost propositional reasoning methods so that they fit modal languages.

In this part of the thesis, we follow both strategies. More specifically, in Chapter 8, we follow the first strategy when we exploit the stepwise way of evaluating modal formulas, and use it to devise a new translation from the modal language into a highly constrained fragment of first-order logic. We provide ample experimental evidence to show that this translation into a fragment of first-order logic yields significant improvements in processing times.

Then, in Chapter 9, we follow the other strategy: boosting propositional methods to make them work for modal languages. Various computational problems have been solved by reformulating them as propositional satisfiability (SAT) problems. Even problems for higher complexity classes than SAT can be efficiently solved by reformulating them as (a sequence of) SAT problems. This is what we do: to solve the modal satisfiability problem we reformulate it as a sequence of SAT problems, and each of those SAT problems is then reformulated as a constraint satisfaction problem — see also Subsection 2.3.2.

7.1.2 Outline and Structure

The present chapter introduces the non-expert reader to the basics of modal logics. In Section 7.2, we briefly touch on modal languages, the basic modal logics and their semantics, as needed for the comprehension of the remaining two chapters of this thesis. We only assume from the reader a working knowledge of propositional logic, so to speak. Section 7.3 treats the standard relational translation from modal to first-order languages; as for the latter languages, it is sufficient to know them as extensions of propositional languages through variables and quantifiers, relation and function symbols.

7.2 Background

7.2.1 Modal Languages

Formulas of a *unimodal language* are built up from proposition letters p, using the propositional operators $\neg$. $\vee$, $\wedge$ and the modal operators $\Diamond$ and $\Box$. Formally, let P be a set of proposition letters, that we usually denote as p or q, or possibly these with indices. So, consider all sets B of finite strings of modal operators, proposition letters and operators, that enjoy the following properties:

1. $P \subseteq B$;

2. if ϕ belongs to B, then so does $\neg\phi$;

3. if ϕ and ψ belong to B, then so do $\psi \wedge \phi$ and $\psi \vee \phi$;

4. if ϕ belongs to B, then so does $\Box\phi$.

Then the *unimodal language* $\mathcal{ML}(P)$ is the smallest (with respect to subset inclusion) of such sets. We denote formulas of $\mathcal{ML}(P)$ by means of Greek alphabet letters, usually ϕ and ψ.

The dual of the box operator $\Box$, namely the diamond operator $\Diamond$, is introduced as an abbreviation: i.e., $\Diamond\phi$ stands for $\neg\Box\neg\phi$, for every ϕ in $\mathcal{ML}(P)$.

Let $Index$ be some index set. Formulas of the *multimodal language* denoted by $\mathcal{MML}(Index, P)$ are built up, as in the unimodal case, from proposition letters, by using $\vee$, $\wedge$ as above, and modal operators $\Box_i$, for $i \in Index$.

Since the extension to the multimodal language is often straightforward, we usually state definitions and results for the unimodal language, and only sketch the corresponding ones for the multimodal case.

7.2.2 Modal Models

Models for $\mathcal{ML}(P)$ are structures of the form $\mathcal{M} = \langle M, R, V \rangle$, where:

- W is a non-empty domain,

- R is a binary relation on W,

- V is a function from P into the power set $\wp(W)$.

The elements of W are often referred to as *states* or *worlds*. They are supposed to represent the states/worlds in which a proposition p holds true.

In fact, truth is defined relative to a state in a model, following the classical interpretation of propositional operators. The important case is given by formulas with modal operators. Formally, consider a model $\mathcal{M}$ and a world $w \in W$. Then we write $M, w \models \phi$, and read it as $\mathcal{M}$ *satisfies* ϕ *at* w, iff the following is true:

1. in case ϕ is $p \in P$, $w \in V(p)$ holds;
2. in case ϕ is $\neg\psi$, $\mathcal{M}, w \models \psi$ holds;
3. in case ϕ is $\psi_1 \vee \psi_2$, $\mathcal{M}, w \models \psi_1$ or $\mathcal{M}, w \models \psi_2$ hold;
4. in case ϕ is $\psi_1 \wedge \psi_2$, both $\mathcal{M}, w \models \psi_1$ and $\mathcal{M}, w \models \psi_2$ hold;
5. in case ϕ is $\Box\psi$, for any $v \in W$, we have that either Rwv does not hold or $\mathcal{M}, v \models \phi$ does.

Formulas of the form $\Diamond\psi$ are interpreted dually:

$$\mathcal{M}, w \models \Diamond\psi \text{ iff there exists } v \in M \text{ such that } \mathcal{M}, v \models \phi \text{ and } Rwv.$$

Models for $\mathcal{MML}(Index, P)$ are structures of the form

$$\langle W, \{R_i : i \in Index\}, V\rangle,$$

on which modal operators $\Box_i$, for $i \in Index$, are interpreted using the associated binary relation R_i, marked by the same index $i \in Index$.

A natural generalisation of the above definition is that of satisfiability of ϕ in a model $\mathcal{M}$ for the language of ϕ: the modal formula ϕ is *satisfiable in the model* $\mathcal{M}$, or the *model satisfies* ϕ, if ϕ holds true at some world of $\mathcal{M}$. The formula ϕ is *satisfiable* if there exists a model, for the language of ϕ, that satisfies ϕ.

The notion of *unsatisfiability* is then derived in the obvious manner.

7.2.3 Basic Modal Logics

At this point, we have seen both modal languages (see Subsection 7.2.1) and structures for interpreting those languages, namely models (see Subsection 7.2.2). The next step consists in defining logics in those languages, and see if and how each is the perfect counterpart of some class of models: i.e., if soundness or completeness holds for the logic with respect to a certain class of models. In the remainder of this subsection, we introduce the non-expert reader to basic modal logics, and state their soundness and completeness with respect to the class of all models; we refer those interested in a complete introduction to modal logics to [BdRV01]. In what follows, we only assume some basic knowledge of logics like propositional logic: i.e., the notion of tautology, axiom and inference rule.

We start with the basic unimodal logic, as in the following definition.

Definition **7.2.1.** Given a unimodal language $\mathcal{ML} := \mathcal{ML}(P)$, the *basic modal logic* $\boldsymbol{K}$ in $\mathcal{ML}$ has the following set of axioms, where ψ, ϕ and θ range over all the modal formulas in $\mathcal{ML}$:

(P1). $\phi \rightarrow (\psi \rightarrow \phi)$;

(P2). $(\phi \to \psi) \to (\neg\psi \to \neg\phi)$;

(P3). $(\phi \to (\psi \to \theta)) \to ((\psi \to \phi) \to (\psi \to \theta))$;

(K1). $\Box(\phi \to \psi) \to (\Box\phi \to \Box\psi)$.

The inference rules of $\boldsymbol{K}$ are as follows:

(MP). $\phi,\, \phi \to \psi \,/\, \psi$;

(NEC). $\phi \,/ \Box\phi$.

An $\mathcal{ML}$ formula ϕ is a $\boldsymbol{K}$ *theorem* if it is either an axiom of the above form, or is obtained by applying one of the rules MP or NEC to $\boldsymbol{K}$ theorems.

If ϕ is a $\boldsymbol{K}$ theorem, then we write $\vdash_{\boldsymbol{K}} \phi$ or simply $\vdash \phi$.

The definition for $\boldsymbol{K}(Index)$ is analogous to the above one, the only difference being in the language: all the $\boldsymbol{K}$ axioms and rules come with indices.

As stated at the opening of the present part, we are interested in automated theorem proving. In this setting, a strategy to prove that a formula is a theorem of a logic appeals to the related semantics: the prover has to derive that the *negation of the formula is unsatisfiable.* Hence, if the logic is sound and complete with respect to its semantics, this strategy allows us to test whether a formula is or is not a theorem of a logic. As far as basic modal logics are concerned, soundness and completeness hold as follows.

Theorem **7.2.2.**
i. *A unimodal formula is a theorem of* $\boldsymbol{K}$ *iff its negation is unsatisfiable.*
ii. *A multimodal formula is a theorem of* $\boldsymbol{K}(Index)$ *iff its negation is unsatisfiable.* $\Box$

For a proof of Theorem 7.2.2, the reader is invited to consult [BdRV01].

7.2.4 Examples

Now that we have introduced some basic formal machinery for modal logic, let us return to the informal discussion in Section 7.1, and provide some examples to complement it with.

Recall that in *epistemic logic*, $\Box\phi$ is read as "the agent knows that ϕ" and for that reason one often writes $K\phi$ in stead of $\Box\phi$. Given that we are talking about knowledge in epistemic logic (as opposed to, say, belief or rumour), it seems natural to view all instances of $\Box\phi \to \phi$ as true: if the agent really knows that ϕ, then ϕ must hold. On the other hand (assuming that the agent is not omniscient) we would regard $\phi \to K\phi$ as false.

We pointed out, in Section 7.1, that labelled transition systems are an especially important kind of relational structures, and, hence, of modal models. They are typically described using *temporal logic*, where one has modal operators $[G]$ and $[H]$. The intended interpretation of a formula $[G]\phi$ is "ϕ is always **g**oing to be the case", and the intended interpretation of $[H]\phi$ is "ϕ **h**as always been the case". We can express many interesting assertions involving time with this language; for instance, we could use $request_i \rightarrow \neg[G]ignored_i$ to say that, whenever resource i is requested, it is eventually granted.

Researchers developing formalisms for reasoning about graphs have sometimes come up with notational variants of modal logic. For example, computational linguists use *Attribute-Value Matrices* (AVMs) for describing *feature structures* (directed acyclic graphs that encode linguistic information). Here is a fairly typical AVM:

$$\begin{bmatrix} \textsc{agreement} & \begin{bmatrix} \textsc{person} & \mathit{1st} \\ \textsc{number} & \mathit{plural} \end{bmatrix} \\ \textsc{case} & \mathit{dative} \end{bmatrix}$$

But this is just a two dimensional notation for the following modal formula:

$$\langle\textsc{agreement}\rangle(\langle\textsc{person}\rangle\mathit{1st} \wedge \langle\textsc{number}\rangle\mathit{plural}) \wedge \langle\textsc{case}\rangle\mathit{dative}.$$

Similarly, researchers in artificial intelligence needing a notation for describing and reasoning about ontologies developed *description logic*. For example, the concept of "being a free-lance musician" is true of any individual who is a musician and is employed by someone who organizes a birthday party. In description logic we can define the latter concept as follows:

$$\texttt{musician} \sqcap \exists\texttt{employer}.\texttt{organizer}.$$

But this is simply the following modal formula lightly disguised:

$$\texttt{musician} \wedge \langle\texttt{employer}\rangle\texttt{organizer}.$$

The links between modal logic on the one hand, and feature and description logic on the other, are far more interesting than these rather simple examples might suggest; see [BdRV01] for details and references on these connections.

7.3 The Standard Translation

As we pointed out in the introduction to the present chapter, modal languages can often be mapped into fragments of suitable first-oder languages. To make this annotation precise, we need some basic definitions.

The begin with, the vocabulary of the first-order language $\mathcal{FO}(P)$ has a unary predicate symbol P for each proposition letter p in P, and a single binary relation symbol R. Instead of a single binary relation symbol, the vocabulary of the first-order language $\mathcal{FO}(Index, P)$ has a binary relation symbols R_i for each $i \in Index$.

Definition **7.3.1.** [vB83] The *standard relational translation* $ST(\phi)$ of unimodal formulas into first-order formulas of $\mathcal{FO}(P)$ is defined as below. In what follows, let x and y be distinct individual variables:

$$
\begin{aligned}
ST_x(p) &:= P(x), && (7.1)\\
ST_x(\neg\phi) &:= \neg ST_x(\phi), && (7.2)\\
ST_x(\phi \wedge \psi) &:= ST_x(\phi) \wedge ST_x(\psi), &&\\
ST_x(\phi \vee \psi) &:= ST_x(\phi) \vee ST_x(\psi), &&\\
ST_x(\Diamond\phi) &:= \exists y\,(Rxy \wedge ST_y(\phi)), && (7.3)\\
ST_x(\Box\phi) &:= \forall y\,(\neg Rxy \vee ST_y(\phi)). && (7.4)
\end{aligned}
$$

The translation ST is defined to be ST_x for a generic individual variable x.

The above is easily extended to a translation taking multimodal formulas into $\mathcal{FO}(Index, P)$, by means of the relation symbol R_i instead of R in the translation of the operators $\Diamond_i$ and $\Box_i$, for $i \in Index$.

Note **7.3.2.** In (7.1) and (7.2), P is the unary predicate symbol corresponding to the proposition letter p. Observe how (7.3) and (7.4) reflect the truth definitions of the modal operators.

As a consequence of Note 7.3.2, models for the unimodal language $\mathcal{ML}(P)$ and the multimodal language $\mathcal{MML}(Index, P)$ can be recast as structures for the corresponding first-order languages $\mathcal{FO}(P)$ and $\mathcal{FO}(Index, P)$, respectively. To interpret the unary predicate symbols, we look up the values of the corresponding proposition letters in the valuation.

Example **7.3.3.**

- The unimodal formula $\Box(\neg p \vee \Diamond p)$ translates into the first-order formula $\forall y\,(\neg Rxy \vee (\neg Py \vee \exists z\,(Ryz \wedge Pz)))$.
- The multimodal formula $\Box_i(\neg p \vee \Diamond_k p)$ translates into the first-order formula $\forall y\,(\neg R_i xy \vee (\neg Py \vee \exists z\,(R_k yz \wedge Pz)))$.

Theorem **7.3.4** ([vB83]). *A modal formula is satisfiable iff its standard relational translation is.*

The above result effectively embeds the modal languages considered here into first-order languages, and paves the way to deciding modal satisfiability by first-order means, as explained in Subsection 8.2.2.

7.4 Conclusions

The standard translation from modal to first-order languages, devised in [vB83], is at the base of *correspondence theory*: in this setting, the translation is conceived as a first step in the study of the expressivity of modal languages for describing both models and the relational structures models are based on, namely *frames*. Thus the standard translation is devised to preserve satisfiability, as quoted above, and also satisfiability at every world of certain relational structures, that is *validity* with respect to classes of frames.

As we prove in Chapter 8, our translation preserves satisfiability, but not validity. Our aim is to make use of *automated theorem provers* to decide whether a modal formula is a theorem of a certain modal logic or not. For this task, preserving satisfiability is sufficient, due to the soundness and completeness of the logics we are interested in, see Subsection 7.2.3. Given this, the goal of our translation becomes to convey information that *boosts automated theorem proving*. Therefore, our translation aims at preserving the structure of the original modal formula, as much as possible, and encoding semantic properties of basic modal logics that are computationally relevant, loosely speaking. Chapter 8 explains how this is achieved.

Chapter 8

The Layered Translation

8.1 Introduction

8.1.1 Motivations

The need for efficient automated reasoning methods for modal logics is increasingly being felt in areas such as knowledge representation, reasoning about programs, and reasoning systems for autonomous agents [Are00, Was00]. We can identify at least four general strategies for modal theorem proving:

1. develop purpose-built calculi and tools, like tableaux systems;

2. translate modal problems into automata-theoretic problems, and then adopt automata-theoretic methods to obtain answers;

3. translate modal problems into first-order problems, and use general first-order tools;

4. build dedicated solvers for modally quantified formulas on top of solvers for propositional formulas; for instance, in [Seb97, GS00], a tableux-based procedure for modal logic is built on top of the Davis-Logemann-Loveland procedure for the propositional component, known as `DPLL` or `DP` in the SAT community — where the letter `P` stays for Putnam.

The advantage of indirect methods such as (2), (3) and (4) is that they allow us to re-use well-developed and well-supported tools instead of having to develop new ones from scratch.

In this chapter, we focus on the third option: translation-based theorem proving for modal logics, where modal formulas are translated into first-order formulas and reasoning problems are to be fed to first-order provers. Our starting point is the standard relational translation introduced in Section 7.3. First-order theorem provers perform poorly on the standard outputs of this translation [ONdRG01, HS97]. To overcome this, very sophisticated decision procedures have been developed [dNdR02] together with alternative translations [ONdRG01].

Our proposal in this chapter consists in a simple refinement of the standard relational translation that allows us to encode additional modal information. In fact, this new translation centres around a strong form of the tree modal property, which is often identified, nowadays, as one of the main reasons for the good computational behaviour of those modal logics that enjoy it (see [Grä99, Var97] and also Section 8.3 below): a modal formula is satisfiable (or more precisely: $\mathbf{K}(Index)$-satisfiable) if and only if it is satisfiable at the root of a model based on a tree.

8.1.2 Outline

We divide the material of this chapter in two main parts. First, we propose our new translation of modal formulas. Our translation results from the composition of the standard relational translation and a translation that maps modal formulas into an intermediate multimodal language. It is in this intermediate multimodal language that we first encode the semantic property known as the tree model property — this fact is also used in Chapter 9. This semantic information is then partially encoded by the relational translation into the layered fragment: i.e., the first-order fragment that is carved out by the new translation. This fragment is contained in the one identified by the standard translation; more precisely, the former fragment is strictly contained in the latter in the case of multimodal languages.

In the second part of the present chapter, we show how to use a first-order theorem prover on the first-order fragments identified by the standard translation and the new one, respectively. The theorem prover Spass is used to perform the experimental comparison between the outcome of the two translations: hence the analysis of the comparison results, that we illustrate in Section 8.5, highlights that Spass performs up to several orders of magnitude better on the outcome of the new translation than on the standard one, in terms of both memory space and execution times.

The encoding of formula layers, carried over by our translation, is the key factor behind the improvement in performance: it is this encoding that allows us to partially exploit the tree-model property of $\mathbf{K}(Index)$ at the purely syntactic level of the theorem proving process.

8.1.3 Structure

The chapter is organised as follows. In Section 8.2, we provide the base inference rules behind first-order theorem proving, as this is used then in Section 8.5:

i.e, propositional and first-order resolution. Then Section 8.3 is devoted to the so-called tree model property of basic modal logics. That property is used in Section 8.4 to define our refinement of the standard translation from modal into first-order logics. In that section, we also introduce the layered fragment of first-order logic, as carved out by the new translation. We then show, in Section 8.5, how the theorem prover Spass performs better on that fragment than on the full first-order fragment carved out by the standard translation. We conclude this chapter with Section 8.6.

8.2 Modal Theorem Proving via the Standard Translation and Resolution

Resolution is at the core of most automated theorem provers for first-order logic. It is a refutation procedure, whose goal is to derive a logical contradiction from a given formula. The basic rule applies to conjunctions of disjunctions, and is essentially based on the following propositional tautology:

$$M \wedge \left(\bigvee L \vee \neg M\right) \rightarrow \bigvee L.$$

In other words: the occurrence of both M and $\neg M$ as in the antecedent of the above formula is irrelevant with respect to the truth value of the overall formula. Thus M and $\neg M$ can be safely removed.

In what follows, we give a precise content to this brief introduction, as much as space allows. We only assume the reader to have an idea of what substitutions and variable renamings are. For a complete introduction to the topic, there are a number of good texts in the literature: we refer the interested reader to [RV01] for a comprehensive overview of automated reasoning methods, to [Lov78] for an introduction to them, based on logics; to [Doe94] for a more logic-programming approach to resolution, and to [Apt97] as its natural companion for the logic programming language Prolog.

8.2.1 Propositional Resolution

Before passing a propositional formula to a theorem prover based on resolution, the formula has to be in "conjunctive normal form". This is essentially a conjunction of disjunctions, where negations are pushed inwards. We provide a bit of terminology below, as it will be used over and over in the remainder of the thesis.

A propositional *literal* is either an atom, like p, or a negation of an atom, like $\neg p$. We shall mainly consider disjunctions of literals in the remainder of this chapter: for instance, formulas like $p \vee \neg q$. Literals of the form L and $\neg L$ are *complementary*.

The following definition is used over and over in the remainder of this thesis, thus we highlight it as follows.

DEFINITION **8.2.1.** A propositional formula ϕ is in *conjunctive normal form* (CNF) if it is a conjunction of disjunctions of literals.

For instance: $\neg p \wedge (\neg q \vee p)$ is in conjunctive normal form, whereas $\neg(p \vee q) \wedge p$ is not. There is a standard procedure, based on the famous De Morgan tautologies of classical logic

$$\neg\phi \wedge \neg\psi \leftrightarrow \neg(\phi \vee \psi), \quad \neg\phi \vee \neg\psi \leftrightarrow \neg(\phi \wedge \psi),$$

that reduces any given formula to its conjunctive normal form. In the above example, the formula $\neg(p \vee q) \wedge p$ is reduced to the equivalent formula $\neg p \wedge \neg q \wedge p$.

Literal disjunctions are transformed into the set of their literals, called *clauses* (this is the *clausification* process); for instance, $p \vee \neg q$ is reduced to $\{p, \neg q\}$. Here and in the following, we adopt the standard convention of representing a clause without parentheses; for instance, $\{p, \neg q\}$ will be rewritten as p, q. Then a conjunction of literal disjunctions is represented as a set of clauses. These are usually represented as list. For instance, the formula $\neg p \wedge (p \vee q)$ corresponds to the clause set represented as the following clause list:

1. $\neg p$,

2. p, q.

Thus the binary *ground resolution rule* can be applied to a clause set and return a clause set as displayed in the following.

$$\frac{\boldsymbol{L}, L_1, \ldots, L_n \quad \neg\boldsymbol{L}, L'_1, \ldots, L'_m}{L_1, \ldots, L'_n, L'_1, \ldots L'_m} \tag{Res}$$

A *derivation via* (Res) of the empty clause from a given clause set $\mathcal{C}$ is a sequence of clause sets, each of which is either $\mathcal{C}$ or obtained from antecedent clause sets in the sequence via (Res).

This simple rule is sufficient for determining whether a formula is a classical tautology, due to the following result.

THEOREM **8.2.2.** *A propositional formula is unsatisfiable iff the empty clause can be derived from it by means of (Res).*

A proof of the above statement can be found, for instance, in [Lov78].

8.2.2 First-order Resolution

The first-order case is more complicated than the propositional one due to the presence in the language of variables, function symbols and quantifiers.

Let us assume that the only logical symbols that occur in the formula ϕ are conjunction, disjunction and negation; this is not a restricted assumption, since the other logical symbols can be defined in terms of this.

A first-order formula like ϕ is first reduced by pushing all negation symbols inwards, and so obtaining its *negated normal form.* This is done by using the following classical equivalences:

$$\begin{array}{cc} \forall x\phi \leftrightarrow \neg\exists\neg\phi, & \exists x\phi \leftrightarrow \neg\forall\neg\phi, \\ \neg\phi \wedge \neg\psi \leftrightarrow \neg(\phi \vee \psi), & \neg\phi \vee \neg\psi \leftrightarrow \neg(\phi \wedge \psi), \end{array}$$

Then the resulting formula is reduced to its *Skolem form* (there are different and equivalent versions of this, see [dN94]). This amounts to substituting variables, bound by only existential quantifiers, by new different constant symbols (i.e., not occurring elsewhere in the formula). Moreover, each occurrence of an existential quantifier, within the scope of $n+1$ occurrences of universal quantifiers of the form $\forall x_0 \cdots \forall x_n$, results in the removal of the variables bound by the existential quantifier, and their substitutions with new different function symbols, applied to the variables bound by the universal quantifiers; e.g., $f(x_1, \ldots, x_n)$. The resulting formula is satisfiable iff the original formula is.

As soon as a formula is in Skolem form, all universal quantifiers are moved leftwards, renaming variables if needed; then all these universal quantifiers are removed. For instance, $\forall x(Rx \wedge \forall x Sx)$ is equivalently transformed into $\forall x \forall y(Rx \wedge Sy)$, and then into $Rx \wedge Ry$, where we implicitly read all variables as being universally quantified over.

Finally, the resulting formula is reduced to its conjunctive normal form (see Subsection 8.2.1), and this into a clause set as in Subsection 8.2.1. In the first-order case, literals are atomic formulas or their negations. For instance, Rx, $\neg Ry$ and Rx are first-order clauses; the set that contains both of them is a clause set.

The resulting clause sets can be passed to the resolution rule for first-order logic. However, the presence of universally quantified variables forces us to "unify" variables in the resolution inference. For instance, Rx and $\neg Rc$ do not contradict each others propositionally. But remind that Rx is implicitly universally quantified over; thus Rx stands also for Rc, so to speak; therefore the two clauses Rc and $\neg Rx$ constitute a contradiction in first-order logic. Roughly speaking, the way we can put forward this contradiction is by interleaving propositional resolution steps and substitutions. In our example, first Rx would be instantiated to Rc, and afterwards a propositional resolution step would infer a contradiction, by generating the empty clauses. Put more precisely, we need to incorporate unification, as defined below, in the first-order resolution inferences.

The *unification* procedure by Martelli and Montanari, as quoted in [Apt97], returns the *most general unifier* θ of a set of terms. Initially, θ is instantiated to the set containing $t = s$, where t and s are the given two terms to unify. Non-deterministically, an equation $t = s$ is chosen from θ, and the associated action is performed:

1. in case $t = s$ is of the form $ft_1 \cdots t_n = gs_1 \cdots s_m$, then the procedure halts and returns `failure` if f and g are two different function symbols; else $m = n$ and the procedure restarts with θ equal to the union of $(\theta - \{t = s\})$ and the set $\{x_1 = y_1, \ldots, x_n = y_n\}$;

2. in case $t = s$ is $x = s$ and x occurs elsewhere in θ, then each occurrence of x in $\theta' := \theta - \{s = t\}$ is simultaneously substituted by s, and θ is set equal to $\theta' \cup \{x = s\}$;

3. in case $t = s$ is $x = s$ and x occurs in s, then the procedure halts and returns `failure`;

4. in case $t = s$ is $t = x$, then E is set to $(E - \{s = t\}) \cup \{x = t\}$;

5. in case $t = s$ is $x = x$, then remove it from θ.

The same procedure is applicable with atoms in place of terms. In both cases, it terminates (see [Apt97]) by producing either failure, or a most general unifier only if it exists; this is unique modulo variable renamings.

Resolution calculus, in the first-order case, can be cast in terms of two rules: *resolution and factorisation.* These embed unification and, if applied to a clause set, each returns a clause set as displayed in the following:

$$\frac{\boldsymbol{M}, L_1, \ldots, L_n \quad \boldsymbol{M}', L_1', \ldots, L_m'}{L_1\mu, \ldots, L_n'\mu, L_1'\mu, \ldots L_m'\mu} \quad \text{(Res')} \qquad \frac{\boldsymbol{N}, L_1, \ldots, \boldsymbol{N}', \ldots, L_n}{\boldsymbol{N}\mu, L_1\mu, \ldots, L_n\mu}, \quad \text{(Fact)}$$

where μ is the most general unifier of the literals, respectively, in $\{\boldsymbol{M}, \neg\boldsymbol{M}\}$ and $\{\boldsymbol{N}, \boldsymbol{N}'\}$. The (Res') rule is applied to two clauses that have no variables in common. This requirement of variable disjointness can be easily met by renaming variables, if necessary.

A *derivation via* (Res') and (Fact) of the empty clause from a given clause set $\mathcal{C}$ is a sequence of clause sets, each of which is either $\mathcal{C}$ or obtained from antecedent clause sets in the sequence via (Res').

Again, we have a result similar to the one in Theorem 8.2.2; see [Lov78] for a proof.

Theorem **8.2.3.** *A first-order formula is unsatisfiable iff the empty clause can be derived from it by means of (Res') and (Fact).*

8.2.3 Challenging Cases

Consider the formula $\Box(\neg p \vee \Diamond p)$ of Example 7.3.3 again; that formula is clearly satisfiable, for instance, on a model with only one world and no relations. Proving this in first-order logic, by means of resolution, amounts to showing that the set with the following clauses is satisfiable:

1. $\neg R(a, y),\ \neg P(y),\ R(y, f(y))$,
2. $\neg R(a, z),\ \neg P(z),\ P(f(z))$.

Observe now that the above clauses have two resolvents:

3. $\neg R(a, a),\ \neg P(a),\ \neg P(f(a)),\ P(f(f(a)))$
4. $\neg R(a, f(z)),\ R(f(z), f(f(z))),\ \neg R(a, z),\ \neg P(z)$.

Clauses 2 and 4 resolve to produce the following new clause:

5. $\neg R(a, f(f(z))), R(f(f(z)), f(f(f(z)))), \neg R(a, f(z)), \neg R(a, z), \neg P(z)$.

Clauses 2 and 5 resolve again to produce an analogue of 5, with even higher term-complexity etc. None of the clauses is redundant and can be deleted; in the limit our input set has infinitely many resolvents. This shows that standard resolution may not terminate in the case of clauses that result from the standard translation of satisfiable modal formulas, even though the satisfiability problem for basic modal logics is decidable — in non-deterministic space.

An obvious question suggests itself: What went wrong in the above example? More precisely: Which features of the original modal formula get lost when clauses are generated from the first-order formulas returned by the standard translation, that is instead needed by the above resolution based method to terminate? How can we recover that information?

Observe that, to obtain the resolvent in line 4, the unary P literals were resolved upon; these literals (or rather the modal operators in which scope the literals are) occur at different modal depths in the original formula $\Box(p \to \Diamond p)$. Thus this resolution step is pointless, from the perspective of modal logics like $\boldsymbol{K}$: the negative P literal derives from the $\Box$-operator, so this literal occurs at modal depth 1; whereas the positive P literal is also bound by the $\Diamond$-operator, hence this literal occurs at modal depth 2. Unless we stipulate so, by means of additional axioms, distinct modal depths are independent. A similar comment pertains to the resolvent obtained in line 3, where again we resolved upon binary R literals that correspond to modal operators occurring in the formula at different modal depths.

Similar examples as the above one, and the questions they pose triggered our refinement of the standard relational translation. In [AGHdR00], the latter was refined by marking literals, with distinct modal depths, by means of syntactically distinct indices. The mathematical justification is provided by a strong form of the tree model property, as we explain below, in Section 8.3.

8.3 The Importance of Having Layers

The example in Section 8.2.3 is interesting in many ways. Above all, it highlights how the structure of the original modal formula gets lost in the standard translation process from modal to first-order formulas in clausal form, and naturally suggests how this information could help to avoid the flaws of first-order resolution in deciding the satisfiability of modal formulas. In fact, in our remark following the quoted example, we propose to consider "layers" of modal formulas as key information to be retrieved and passed to the theorem prover. We explain precisely what we mean by layers and their use with respect to first-order theorem proving in the remainder of the present section.

8.3.1 Trees and Layers

In what follows, S^+ and S^* denote the transitive and reflexive, transitive closure of the relation S, respectively.

Definition **8.3.1.** A *rooted tree*, or simply a *tree* is a relational structure of the form $\mathcal{T} := \langle T, S\rangle$ that enjoys the following properties:

1. T, the set of nodes, contains a special node $r \in T$, called the *root*;
2. the root r is the only node in T such that $\forall t \in T\,(S^*rt)$;
3. every element of T, distinct from r, has a single S predecessor: that is, $\forall t \in T\,(\exists s \in T \wedge Sst \wedge \forall s' \in T\,(Ss't \rightarrow s' = t))$; the root has no S predecessors;
4. S^+ is acyclic: i.e., $\forall t \in T\,(\neg S^+tt)$.

A *path* in a tree $\mathcal{T}$ is a finite sequence of T nodes of the form $\mathtt{s} := \langle t_i : i \leq n\rangle$ such that St_it_{i+1} holds for every two adjacent nodes t_i and t_{i+1} in the sequence and t_0 is the root r of $\mathcal{T}$. The *length* of the path $\mathtt{s}$ is the number n of nodes in $\mathtt{s}$ minus 1.

The above properties are quite intuitive if we keep in mind the image of a tree. The first property states that a tree cannot be empty, at least its root must belong to it. The second property qualifies the root as the only node from which all the other nodes can be reached, via a finite number of S transitions. Then the third property requires that any node, different from the root, should have precisely one predecessor via an S transition; moreover, the root cannot be reached via any S transition. The last property imposes a tree to be free of loops: i.e., there cannot be a finite number of S transitions starting and finishing at the same node.

DEFINITION **8.3.2.**

- A *tree model* for the unimodal language $\mathcal{ML}(P)$ is a model $\mathcal{M} = (W, R, V)$ such that the relational structure $\langle W, R\rangle$ is a tree.
- A *tree-like model* for the multimodal language $\mathcal{MML}(Index, P)$ is a model $\langle W, \{R_i : i \in Index\}, V\rangle$ such that $\langle W, \bigcup_i R_i\rangle$ is a tree.
- A logic **L** has the *tree model property* if every **L**-satisfiable formula is satisfiable at the root of a tree or tree-like model for **L**.

The advantage of dealing with a tree-like structure $\mathcal{T}$ is that every world of $\mathcal{T}$ can be reached through a *unique* path of $\mathcal{T}$'s relations starting from w. We state this well-known property of trees as a fact (for instance, see [Wil96]), and it is immediate to prove given our Definition 8.3.1.

FACT **8.3.3.** *There is precisely one path terminating at each node of a tree-like structure.* □

The above fact is used over and over in the remaining proofs of this chapter, to well define valuations in models via paths of trees or tree-like structures.

8.3.2 Modal Depth and Layers

The notion of layering for basic modal logics emerges at both the semantic level, via tree models, and the syntactic level of modal formulas. In fact, tree or tree-like models as introduced above come with a layering induced by paths. Likewise, the parse tree of a modal formula induces a natural formula layering, where new layers begin at nodes labelled by modal operators. For instance, in $\Box(\neg p \vee \Diamond p)$, the operator $\Box$ occurs in layer 1, while the operator $\Diamond$ and its argument occur in layer 2. The following definition captures precisely this sort of syntactical layering.

DEFINITION **8.3.4.** Let ϕ be a modal formula. The *modal depth* $\mathrm{mdepth}(\phi)$ of ϕ is defined as:

$$\begin{array}{rcccl} \mathrm{mdepth}(p) & = & \mathrm{mdepth}(\neg p) & = & 0 \\ \mathrm{mdepth}(\psi \wedge \chi) & = & \mathrm{mdepth}(\psi \vee \chi) & = & \max\{\mathrm{mdepth}(\psi), \mathrm{mdepth}(\chi)\} \\ \mathrm{mdepth}(\Diamond\psi) & = & \mathrm{mdepth}(\Box\psi) & = & 1 + \mathrm{mdepth}(\psi). \end{array}$$

There is a direct correlation between formula layers and layers in a tree or tree-like models; we state and prove it in the following section. As a consequence of the results below, literals occurring in distinct formula layers will not be resolved upon, and not be combined; in this manner, we avoid the problems encountered in the example discussed in Subsection 8.2.3.

8.3.3 The Tree Model Property: Layers at Work

Since we are only concerned with satisfiability in a world, the theorem below allows us to restrict our attention to a tree model and its root. Furthermore, the result below highlights the link between the modal depth of a formula, on which our translation is based (see Definitions 8.4.1 and 8.4.7 below), and the layering that comes with a tree-like model, as remarked after Definition 8.3.4.

The proof of the following statement can be found in books of modal logic like [dR93, BdRV01].

THEOREM **8.3.5** (TREE MODEL PROPERTY). *Let $\mathcal{L}$ be any multimodal language, ϕ an $\mathcal{L}$ formula, and $\mathcal{M}$ an $\mathcal{L}$ model. Then there exists an $\mathcal{L}$ tree-like model $\mathcal{T}$ that enjoys the following properties:*

- *ϕ is satisfiable at the root of $\mathcal{T}$ iff it is satisfiable in $\mathcal{M}$;*
- *consider a natural number i such that $0 \leq i \leq \mathrm{mdepth}(\phi)$, and assume that ψ is a subformula at modal depth i in ϕ; then the satisfiability of ψ can be tested in a $\mathcal{T}$ world t such that, if k is the length of the $\mathcal{T}$ path to t, then $i \leq k \leq \mathrm{mdepth}(\phi)$.* □

Figure 8.1 below illustrates Theorem 8.3.5 for the simple case of $\phi := \Diamond\Diamond p \vee \Diamond q$: to test the satisfiability of ϕ and $\Diamond\Diamond p$ we need to walk, from the root, along paths of length at most 2; the satisfiability of both $\Diamond p$ and $\Diamond q$ can be tested starting from layer 1, and reaching at most layer 2; finally, the satisfiability of p and q can be tested in the layer 2.

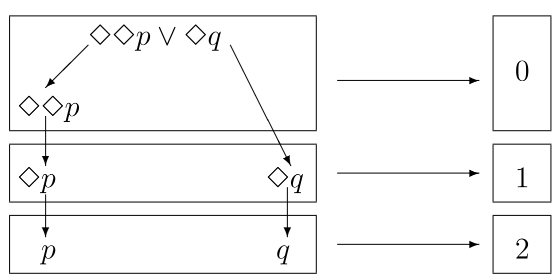


Figure 8.1: The Tree Model Property.

Observe that the tree model property and the finite model property are independent: in fact, there are modal logics for which the former fails but the latter holds, and vice versa. We refer the reader to any introduction to modal logic for these; see [BdRV01], for instance.

8.4 Layer by Layer

In this section we exploit the tree model property to devise our refinement of the standard translation of modal formulas into first-order formulas.

The new translation proceeds in two steps. First, modal formulas are translated into formulas of an intermediate modal language. It is in this intermediate step that the layering induced by trees (see Theorem 8.3.2) is made explicit at the syntactical level: the modal depths of formulas (see Definition 8.3.4), which are related to the layers of tree-like structures as in Theorem 8.3.5, are encoded as indices. In turn, these intermediate formulas with layers as indices are passed to the standard translation (see Definition 7.3.1), and thus transformed into formulas of a first-order language.

All in all, the new translation will mark relations and propositions according to the *number* of modal operators in whose scope a given modal subformula occurs; i.e., the modal depth at which the subformula occur. For instance, the modal formula

$$\Diamond\Diamond p$$

is translated into a multimodal formula with diamonds and proposition letters labelled according to the modal depth at which these occur in the above formula:

$$\Diamond_1\Diamond_2 p_2.$$

The standard relational translation into first-order logic transforms the latter into the following formula:

$$\exists y\,(R_1xy \,\wedge\, \exists z(R_2yz \wedge P_2z)).$$

Similarly, $\Box(p \to \Diamond p)$ becomes first $\Box_1(p_1 \to \Diamond_2 p_2)$; then the standard translation generates the first-order formula

$$\forall y\,(R_1xy \to (P_1(y) \to \exists x(R_2yx \to P_2x))).$$

In the remainder of the present section, we focus on the unimodal case (see Subsection 8.4.1 below), and briefly touch on the multimodal one (see Subsection 8.4.2 below), since this constitutes a trivial extension of the former.

8.4.1 The Unimodal Case

As the above example illustrates, our final relational translation is reached via an intermediate step through an intermediate multimodal language. This collects the modal operator and the proposition letters of the given unimodal language, and mark them with natural numbers, as formalised in the following definition.

DEFINITION **8.4.1.** Consider a modal language $\mathcal{ML} := \mathcal{ML}(Index, P)$, and a multimodal language $\mathcal{IML} := \mathcal{IML}(P)$ with set of propositions equal to $\{p_n : p \in P\}$, and modal operators in $\{\Box_i, \Diamond_i : i \geq 0\}$.

- Suppose that ϕ is a modal formula of $\mathcal{ML}$. Let n be a natural number. The translation $Tr(\phi, n)$ of ϕ into the intermediate multimodal language $\mathcal{IML}$ is defined as follows:

$$\begin{array}{rlrl} Tr(p,n) &:= p_n, & Tr(\neg\psi,n) &:= \neg\, Tr(\psi,n), \\ Tr(\psi\wedge\chi,n) &:= Tr(\psi,n)\wedge Tr(\chi,n), & Tr(\psi\vee\chi,n) &:= Tr(\psi,n)\vee Tr(\chi,n), \\ Tr(\Diamond\psi,n) &:= \Diamond_{n+1} Tr(\psi,n+1), & Tr(\Box\psi,n) &:= \Box_{n+1} Tr(\psi,n+1). \end{array}$$

- Denote by LT_x the composition of $Tr(\_,0)$ and ST_x: i.e., for every modal formula ϕ,

$$LT_x(\phi) := ST_x \circ Tr(\phi, 0).$$

The *layered relational translation* LT is LT_x, for a generic first-order variable x.

In the two lemmas below, we adopt the following notational convention.

Convention **8.4.2.** If $\mathcal{T}$ is a tree or tree-like structure with root r, then let $path(t)$ denote the length of the $\mathcal{T}$ path to t.

Notice that this path is unique in virtue of Fact 8.3.3.

Lemma **8.4.3.** *Let ϕ be a unimodal formula and $\mathcal{T}$ a tree-like model in the language of ϕ. If the intermediate multimodal formula $Tr(\phi, n)$ is satisfiable at a world t in the model $\mathcal{T}$ such that $path(t) = n$, then the unimodal formula ϕ is satisfiable as well.*

Proof. Let $\mathcal{T}$ be a tree model as in the above statement, with universe T, a finite number of labelled relations R_i and valuation V.

Now, let us construct a unimodal model $\mathcal{N}$ on T whose relation R is defined as follows:

$$R := \{(t, u) \in T \times T \,:\, R_j tu \text{ for some } R_j \text{ of } \mathcal{T}\}\,. \tag{Rel}$$

The valuation V' of the model $\mathcal{N}$ is defined as follows: for every proposition letter p and for every t such that $path(t) = n$, $t \in V'(p)$ iff $t \in V(Tr(\phi, n))$.

Given this model $\mathcal{N}$ and our tree-like model $\mathcal{T}$, we can prove the following stronger claim, from which follows our lemma:

$$\mathcal{T}, t \models Tr(\phi, n) \text{ iff } \mathcal{N}, t \models \phi,$$

for every $t \in T$ such that n is equal to $path(t)$.

We prove the above claim by structural induction on ϕ. The atomic and Boolean cases are easy to spell out, since both immediately follow from the above choice of V' and the fact that the intermediate translation *Tr* is a homomorphism on Boolean formulas, see Definition 8.4.1. Next, assume that ϕ is a formula of the form $\Diamond\psi$. In this case,

$$Tr(\phi, n) = \Diamond_{n+1} Tr(\psi, n+1).$$

Assume that t is a node of T, and the length of the $\mathcal{T}$ path to t is n; i.e., $path(t) = n$. We have that $\mathcal{T}, t \models Tr(\phi, n)$ iff there exist u and R_j in $\mathcal{T}$ such that $R_j tu$, and $\mathcal{T}, u \models Tr(\psi, n)$. Since $path(u)$ is equal to the length of the path to t plus 1 (i.e., the length of the R_j transition from t to u), by induction hypothesis we know that $\mathcal{T}, u \models Tr(\psi, n+1)$ is equivalent to $\mathcal{N}, u \models \psi$. Therefore, this and (Rel) yield that $\mathcal{T}, t \models Tr(\phi, n)$ iff $\mathcal{N}, t \models \phi$. □

In the following lemma, we prove the reverse implication of the above lemma.

Lemma **8.4.4.** *Let ϕ be a unimodal formula and $\mathcal{T}$ a tree model in the language of ϕ. If ϕ is satisfiable at the world t in the model $\mathcal{T}$ such that $path(t) = n$, then the intermediate multimodal translation $Tr(\phi, n)$ of ϕ is satisfiable as well.*

Proof. We define a model $\mathcal{N} := \langle T, \{R_{n+1} : n \geq 0\}, V' \rangle$ that has the same universe T as $\mathcal{T}$. The relations of $\mathcal{N}$ are defined by stipulating the following:

$$R_{n+1}uv \text{ holds iff both } path(u) = n \text{ and } Ruv \text{ hold.}$$

We complete the characterisation of $\mathcal{N}$ by defining its valuation V' as follows: for every proposition letter p and every world $t \in T$ such that $path(t) = n$, we stipulate that $t \in V'(Tr(p, n))$ holds iff $t \in V(p)$.

The following intermediate claim follows easily now by structural induction on ϕ, like in Lemma 8.4.3: for every unimodal formula ϕ, every world t and n such that $path(t) = n$, we have

$$\mathcal{T}, t \models \phi \text{ iff } \mathcal{N}, t \models Tr(\phi, n) \text{ holds.}$$

Our lemma is clearly an immediate consequence of the above claim. □

We now combine the above lemmas and prove that the intermediate translation *Tr* preserves satisfiability.

Theorem **8.4.5.** *Assume a modal formula ϕ. Thus the following holds true:*

- *ϕ is satisfiable iff its intermediate modal translation $Tr(\phi, n)$ is satisfiable;*
- *ϕ is satisfiable iff its intermediate modal translation $Tr(\phi, 0)$ is satisfiable.*

Proof. The first item is an immediate consequence of Lemmas 8.4.4 and 8.4.3, via Theorem 8.3.5, and it yields the second item. $\Box$

Finally, the combination of Theorems 8.4.5 and 7.3.4 yields that the layered translation, being the composition of Tr and the standard translation, preserves satisfiability too.

Theorem **8.4.6.** *Let ϕ be a modal formula. Then ϕ is satisfiable iff $LT_x(\phi)$ is so, for any first-order variable x.*

Proof. The statement follows from Theorems 8.4.5 and 7.3.4, since LT_x results from the composition of the standard translation ST_x and $Tr(\_, 0)$, see Definition 8.4.1. $\Box$

In the subsection below, we reformulate some of the above definitions and statements for the case of the multimodal logics $\boldsymbol{K}(Index)$.

8.4.2 The Multimodal Case

The layered relational translation is easily extended to the multimodal language $\mathcal{MML}(Index, P)$ by means of a slightly more complex encoding. We need strings of labels instead of natural numbers to capture the different relations involved. The result is an analogous of Definition 8.4.1.

The set of operators of the intermediate language is now labelled by sequences, whose values are the indices of the modal operators of the original language; so is its set of proposition letters, as we specify below.

Definition **8.4.7.** Consider a multimodal language $\mathcal{MML} := \mathcal{MML}(Index, P)$, with modal operators in $\{\Box_a,\ \Diamond_a : a \in Index\}$. Then the multimodal language $\mathcal{IMML} := \mathcal{IMML}(Index, P)$ is the multimodal language with set of propositions equal to $\{p_s : p \in P \text{ and } s \in Index^*\}$, and modal operators in $\{\Box_s,\ \Diamond_s : s \in Index^*\}$.

- Suppose that ϕ is a multimodal formula in $\mathcal{MML}$. Let $s \in Index^*$. The intermediated translation $Tr_m(\phi, s)$ of ϕ into the intermediate multimodal language $\mathcal{IMML}$, for a string s in $Index^*$, is defined as follows:

$$\begin{aligned}
Tr_m(p, n) &:= p_n, \\
Tr_m(\neg\psi, n) &:= \neg\, Tr_m(\psi, n), \\
Tr_m(\psi \wedge \chi, n) &:= Tr_m(\psi, n) \wedge Tr_m(\chi, n), \\
Tr_m(\psi \vee \chi, n) &:= Tr_m(\psi, n) \vee Tr_m(\chi, n), \\
Tr_m(\Diamond_a\psi, s) &:= \Diamond_{s*\langle a\rangle}\, Tr_m(\psi, s*\langle a\rangle), \\
Tr_m(\Box_a\psi, s) &:= \Box_{s*\langle a\rangle}\, Tr_m(\psi, s*\langle a\rangle).
\end{aligned}$$

- MLT_x denotes the composition of the translation $Tr_m(\_, \epsilon)$ above and ST_x: i.e., for every multimodal formula ϕ of $\mathcal{MML}$,

$$MLT_x(\phi) := ST_x \circ Tr_m(\phi, \epsilon),$$

where ϵ is the empty sequence. The *multimodal layered relational translation* MLT is MLT_x for some first-order variable x.

- The *layered fragment* of first-order logic is the range of the multimodal layered translation.

The following result is proved as in the unimodal case. We let the reader spell out the details or check the proof in [AGHdR00].

THEOREM **8.4.8.** *Let ϕ be a multimodal formula. Then ϕ is satisfiable iff $MLT_x(\phi)$ is so, for any first-order variable x.* □

8.4.3 Finale

The layered translation constitutes a new way of turning modal problems into first-order problems. The new translation, and the intermediate translation into multimodal languages are both *conservative*, in the sense that they can work *on top* of existing strategies for first-order and modal logics, respectively. We discuss the latter fact in Chapter 9, and put at work the former in Section 8.5 below.

In particular, the layered translation is a refinement of the standard translation; hence the layered fragment is contained in the fragment identified by the standard translation and in its generalisation, i.e., the guarded fragment. Thus we can use *any decision procedure* and strategy tuned for the latter, see [dNdR02]. We state this precisely as follows.

THEOREM **8.4.9.** *Let $\mathcal{R}_{ST}(\phi)$ and $\mathcal{R}_{LT}(\phi)$ denote the sets of clauses derivable by means of resolution and factoring from $ST(\phi)$ and $LT(\phi)$ respectively. Then $|\mathcal{R}_{ST}(\phi)| \leq |\mathcal{R}_{LT}(\phi)|$. The same result holds with LT replaced by MLT.* □

The above result yields that first-order theorem provers will perform at least as well on the layered translations as on the standard one. In the section below, we report our experimental comparison between the two translations. This witnesses the improvements — up to orders of magnitude better — that the theorem prover SPASS gains by means of the new translation LT.

8.5 Experimental Comparisons

In this section, we compare the two translations, the standard versus the layered one, by running some experimental tests. First we briefly introduce and comment on the problem set and prover used in our experiments, then we display and explain the results.

8.5.1 The Problem Set

Our tree-based heuristics was evaluated by running a series of tests on a number of problem sets. Our main focus was on the *modal QBF benchmark*. This benchmark is the basic yardstick for the Tableaux Non-Classical Systems Comparisons (`TANCS`) competition on theorem proving and satisfiability testing for non-classical logics, see [TAN00]. It is a random problem generator that has been designed to evaluate solvers of either satisfiable or unsatisfiable problems of the modal logic **K**.

The modal formulas of this benchmark are generated by means of quantified Boolean formulas. For the generation, first a quantified Boolean formula is generated with C clauses, quantifier alternation depth equal to D, and maximum number of variables V for each alternation. Then the resulting quantified Boolean formula is translated into modal logic via an encoding that was originally proposed by Halpern, see [Hal95]. See [HdR01] for a detailed analysis of the QBF test set.

The output of the QBF generator is is a file named p-qbf-cnf-K4-Cn-Vm-Dl, in which the numerical parameters are explained as follows: n is the number of clauses; m the number of variables; D the quantifier depth.

8.5.2 The Theorem Prover

Tests were performed on a Sun ULTRA II (300 MHz) with 1Gb RAM, under Solaris 5.2.5, with the automated theorem prover Spass version 1.0.3. This is an automated theorem prover for full sorted first-order logic with equality that extends superposition by sorts and a splitting rule for case analysis; it has been in development at the Max-Planck-Institut für Informatik for a number of years, see [SPA00]. Spass was invoked with the automode switched on; no sort constraints were built, and both optimized and strong Skolemization were disabled.

8.5.3 Experimental Comparisons

The modal QBF benchmark

To explore the behaviour of our heuristics in a large portion of the landscape of the **K**-satisfiability problem, we randomly generated sets of 10 problems by means

C/V/D	ST Average Time	LT Average Time	M
5/2/1	9.6222	0.53469	1
10/2/1	3.9909	0.41734	1
15/2/1	0.13172	0.10859	0
5/2/2	450.44	0.66141	3
10/2/2	370.09	0.78297	3
15/2/2	147.38	0.75656	2
5/2/3	N/A	36.048	N/A
10/2/3	N/A	58.886	N/A
15/2/3	2094.4	94.192	1
5/2/4	N/A	20.362	N/A
10/2/4	N/A	33.084	N/A
15/2/4	2094.4	35.068	1
5/2/5	N/A	1136.1	N/A
10/2/5	N/A	2896	N/A
15/2/5	N/A	3758.2	N/A
5/3/1	7.1862	2047.9	2
10/3/1	9.752	2324.2	2
15/3/1	14.066	1506.8	2
5/3/2	N/A	7.0931	N/A
10/3/2	N/A	8.3192	N/A
15/3/2	N/A	9.3902	N/A
5/3/3	N/A	1445.2	N/A
10/3/3	N/A	4045.1	N/A
15/3/3	N/A	4865.4	N/A

Table 8.1: Comparison by average time.

of the modal QBF generator for different sets of parameters. Table 8.1 compares the average time in CPU seconds, while Table 8.2 compares the average number of clauses for two methods: layered (our improved translation, see Definition 8.4.1) and standard (the relational method, see Definition 7.3.1). The shorthand C/V/D in the first column denotes the number of *clauses*, the number of *variables*, and the *depth* used in the generation. Columns labelled by M show the magnitude of the difference between the preceding two columns, i.e., round($\log N/N'$). We used a time out of 3 hours on a shared machine; N/A indicates that a value is not available due to a time out.

As can easily be seen from Tables 8.1 and 8.2, our improved translation method outperformed the standard translation in every case, both in computing time (CPU time) and number of clauses generated. This is not only an average behaviour, but it was observed in each instance. For some configurations the drop

C/V/D	*ST* Average Clause Number	*LT* Average Clause Number	M
5/2/1	5695	726	1
10/2/1	2367	546	1
15/2/1	10	10	0
5/2/2	27209	437	2
10/2/2	22306	500	2
15/2/2	11368	473	1
5/2/3	N/A	10714	N/A
10/2/3	N/A	15395	N/A
15/2/3	45789	20786	1
5/2/4	N/A	3121	N/A
10/2/4	N/A	4971	N/A
15/2/4	N/A	5358	N/A
5/2/5	N/A	48546	N/A
10/2/5	N/A	91767	N/A
15/2/5	N/A	106870	N/A
5/3/1	105960	4372	1
10/3/1	108110	5390	1
15/3/1	72605	6687	1
5/3/2	N/A	1804	N/A
10/3/2	N/A	2221	N/A
15/3/2	N/A	2687	N/A
5/3/3	N/A	52153	N/A
10/3/3	N/A	107800	N/A
15/3/3	N/A	119150	N/A

Table 8.2: Comparison by average number of generated clauses.

in computing time is as much as three orders of magnitude or two. This is always the case when the depth of the formula increases, as our translation cleverly exploits the modal depth information. The average number of clauses generated was nearly always smaller by one order of magnitude.

In Figure 8.2 we display a sample from our experimental results: 64 instances of the 10/3/1 configuration. The top curve indicates the CPU time needed by the standard relational translation, and the bottom one the CPU time needed by the layered translation. Note that the standard translation can be very sensitive to certain hard problems, which results in significant differences between easy and hard instances; the layered method responds in a much more controlled way to hard problems. Interestingly, the curves follow each other, even at many orders of magnitude of difference. This shows that our heuristics does not change the

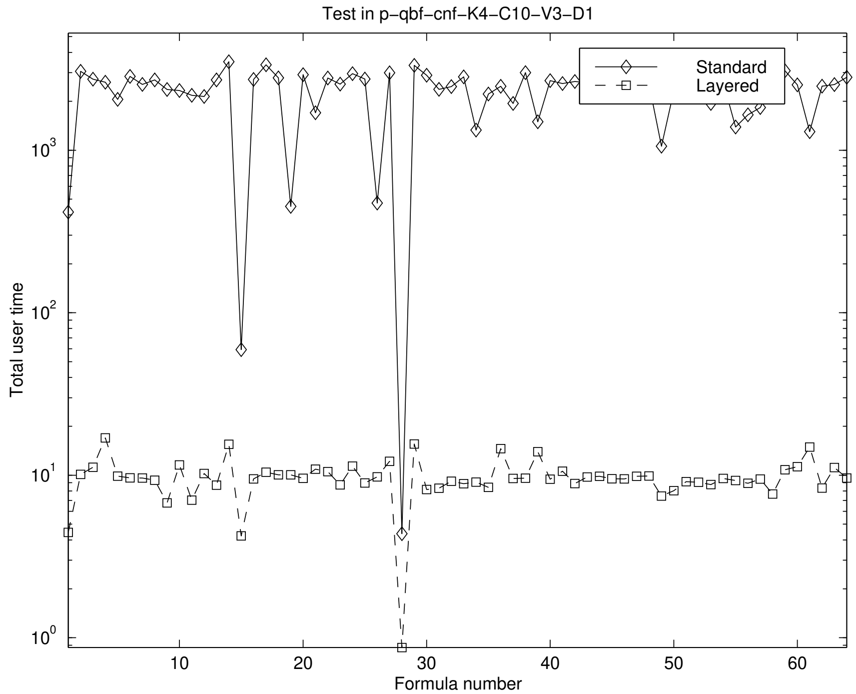


Figure 8.2: A sample from the tests.

nature of the problem: it simply makes it much easier for the resolution prover.

The latter phenomenon can also be observed more globally. The plots in Figure 8.3 were obtained with the following settings: V = D = 2, while C ranged from 2 to 40. Figures 8.3 (a) and (b) show the number of clauses generated and the CPU time needed, respectively, for the standard and layered method, while 8.3 (c) plots the proportion of satisfiable instances as C increases. The curves for the standard and layered methods are very similar, with the layered method lacking the sharp lows and highs that seem to be characteristic for the relational method. Both display a clear easy-hard-easy behaviour, but the layered translation is better by several orders of magnitude.

Note that the biggest improvements are achieved in the satisfiable region, i.e., for C < 26. Once we were confident that the layered method consistently displayed a good behaviour and a significant improvement over the standard translation, we ran the standardized tests provided by TANCS (64 instances randomly generated with the 20-clauses/2-variables/2-depth parameters); see Figure 8.4 for the outcomes.

Finally, to obtain the results in Figure 8.5 we generated 64 instances of problems for 2 and 3 variables with depths ranging from 1 to 6, again with a time out of 3 hours. The figure shows the average values we obtained. We ran the

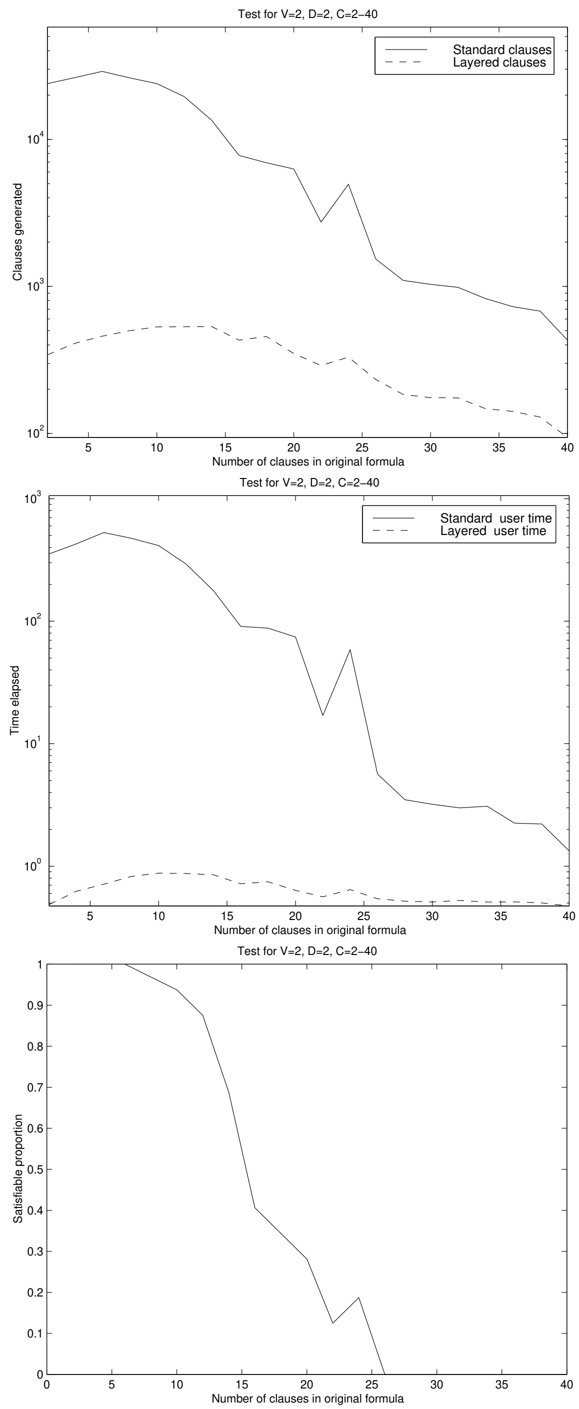


Figure 8.3: Easy-hard-easy.

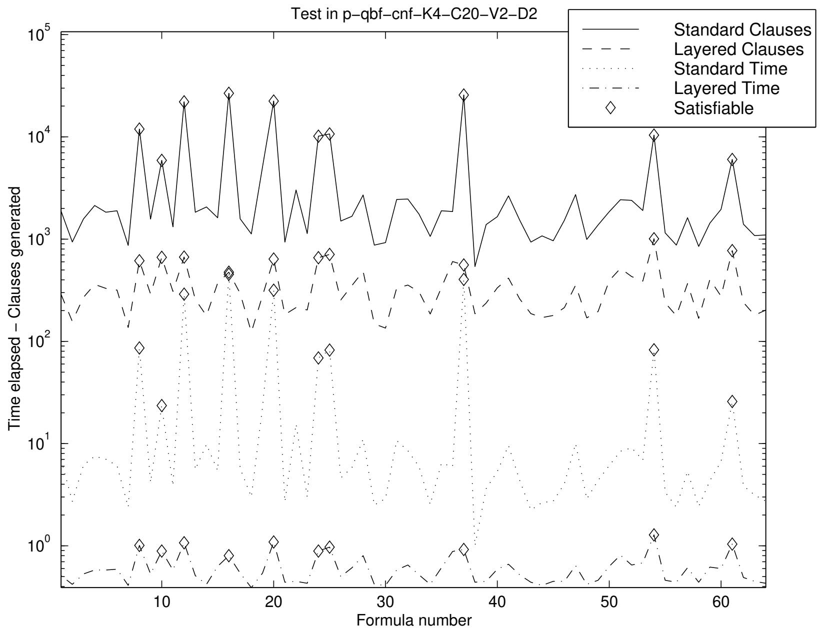


Figure 8.4: Standard TANCS Test 20/2/2.

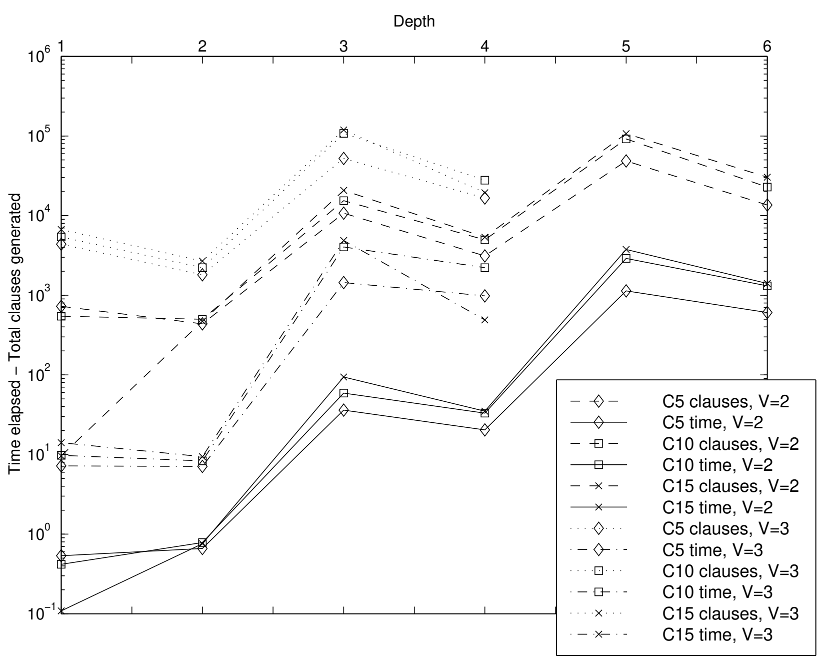


Figure 8.5: *LT* Tests on 64 Problem Instances.

same tests with the standard instead of the layered translation, but even for moderate depths the computing time and number of clauses exceeded the available resources.

Additional Tests

Given that the problems returned by the QBF generator were generally too hard for the prover using the standard translation, we also performed tests with a number of easier problem sets — so to speak — that include the one proposed by Heuerding and Schwendimann in [HS96], which were used in, for example, Tableaux'98. Invariably, the layered translation outperformed the standard one; it was able to solve substantially harder instances in all categories.

8.6 Conclusions

8.6.1 Synopsis

In this chapter, a new relational translation of modal formulas into first-order formulas is described. The key idea underlying this refinement is to encode a very strong form of the tree model property in an intermediate translation into multi-modal languages, and hence in a translation into first-order languages — see Definition 8.4.1.

Using our tree-based heuristics, we have consistently observed improvements, both in terms of the number of clauses generated and in terms of CPU time used.

8.6.2 Discussion

So the methodology used pays off: instead of modifying theorem provers, or develop new ones from scratch, we reuse existing ones and optimise their behaviour by refining the encoding of the modal problems. In the future, it could be interesting to explore the behaviour of our heuristics in larger parts of the problem space.

It could also be interesting to see how to encode weaker forms of the tree model property to boost the performance of resolution provers on input from different modal logics, such as K4, S4, and temporal logic.

In Chapter 9, we appeal to the same semantic intuitions underpinning the results of this chapter. As noticed above, the intermediate translation is already based on these: it makes explicit, at the syntactic level, that the satisfiability of a modal formula can be tested propositionally, proceeding layer by layer (i.e., index by index) in the modal formula. It is precisely this semantic property which is used in procedures for modal logics that are based on propositional solvers, as explained in the following chapter.

Chapter 9

Diamonds and Constraints

9.1 Introduction

9.1.1 Motivations

In Chapter 8 we base the satisfiability procedure for $\boldsymbol{K}$ and $\boldsymbol{K}(Index)$ on a translation from modal logic into the layered fragment of first-order logic. This translation goes through an intermediate translation which maps modal formulas into formulas of an intermediate multimodal language. This intermediate translation itself possesses interesting features:

- it encodes a very strong form of the tree model property,
- it preserves satisfiability,
- and, as a consequence, it could be directly employed to check the satisfiability of the original modal formulas.

The work of [GS00] exploits similar semantic intuitions but without appealing to the tree model property directly. In fact, Giunchiglia and Sebastiani prove that a refinement of a SAT solver, called $\boldsymbol{K}$-SAT, can be used for $\boldsymbol{K}(Index)$ theorem proving: this decision procedure for $\boldsymbol{K}(Index)$ logics is built on top of the Davis-Logemann-Loveland (`DP`) procedure for propositional logics. The SAT solver `DP` is called on a modal formula. The propositional satisfiability of the input formula is first checked. If the result is positive, search is not abandoned, but the modal components of the formula are examined: one by one, each subformula with a diamond as its main operator is checked against all the subformulas that have a box as their main operator; this is done by means of `DP` again.

In this chapter, we follow a similar approach: however, instead of using `DP`, we want to use constraint propagation (see Chapter 4) and solving algorithms. At this early stage of our work, we are not aiming to be competitive with today's high-performance modal provers, such as `DLP` [PS02], `FaCT` [Hor02] and `RACER` [HM02].

Our aim in this chapter is to explore to what extent existing constraint satisfaction techniques, developed for propositional satisfiability, can be used in automated theorem proving for modal logics.

9.1.2 Outline and Structure

In this chapter, we propose to use constraint solving and propagation algorithms to determine the satisfiability of modal logics. These procedures are all based on constraint algorithms for propositional formulas, as the aforementioned $\boldsymbol{K}$-SAT algorithm is based on `DP`.

Our work in this chapter consists of two stages. The modal formula is first transformed into a CSP, see Subsection 9.4.2. Then a preliminary constraint based procedure for this encoding is proposed in Subsection 9.4.3. Search for solutions is alternated with a hyper-arc consistency algorithm, as described in Chapter 4. We sketch a proof of the correctness and completeness of the procedure for $\mathbf{K}$ formulas. A refinement of this procedure and alternatives to it, still based on CSP solvers, are proposed in Section 9.6.

In the literature, a number of constraint propagation and solving algorithms have been studied for reasoning about satisfiability problems. Thus, our work constitutes a first attempt to exploit well-known and corroborated techniques of constraint propagation and satisfaction for these problems to tackle modal satisfiability. We also report on preliminary experimental work aimed at testing the procedures proposed in this chapter on benchmark modal formulas.

9.2 The SAT Based Approach

In the remainder of this chapter, we implicitly assume that we are dealing with modal languages in which every occurrence of the $\Diamond$ operator has been replaced by $\neg\Box$.

CONVENTION **9.2.1.** We assume that, in the modal language $\mathcal{ML}(P)$, each occurrence of all modal operators $\Diamond$ is replaced by the equivalent $\neg\Box\neg$.

This will avoid that the solver treats modal formulas such as $\Diamond\neg p$ and $\Box p$ in a different way.

The following further convention on propositional formulas is consistent with the actual implementations of the `DP` procedure as in [Seb97], and does not constitute a theoretical limitation; see also *ib.*

CONVENTION **9.2.2.** The set of atoms in propositions or clause sets are totally ordered.

Convention 9.2.2 avoids that theorem provers based on `DP` treat formulas such as $p \vee q$ and $q \vee p$ as different. The original `DP` procedure receives as input a CNF formula and determines whether the formula is satisfiable or not. The basic **K**-SAT algorithm, displayed as Algorithm 9.2.1, applies a modified version of the `DP` procedure, recursively, on sequences of modal formulas.

We interpret what the **K**-SAT procedure does by resorting to the tree model property, see Definition 8.3.2. We recall that the intermediate translation (see Definition 8.4.1) is already based on this property: i.e., a multimodal $\boldsymbol{K}(Index)$ formula ψ is satisfiable iff it is so at the root of a tree-like model. This means that subformulas of ψ can be evaluated layer by layer: first subformulas in layer 0 are evaluated; if these are found consistent, subformulas at deeper layers are evaluated, in a top-down manner. This is also what Algorithm 9.2.1 does in tree-like terms.

Algorithm 9.2.1: $\boldsymbol{K}$-SAT(ψ)

procedure $\boldsymbol{K}$-SAT(ψ)
 return $\boldsymbol{K}$-SAT$_W(\psi, true)$;

procedure $\boldsymbol{K}$-SAT$_W(\psi, \mu)$
 if $\mu = false$ **then return** $false$; `% backtrack`
 if $\mu = true$ **then return** $\boldsymbol{K}$SAT$_A(\mu)$;
 if a unit disjunct L is in ψ **then**
   return $\boldsymbol{K}$-SAT$_W$(`Unit-propagate`$(L, \psi), \mu \wedge L)$;
 $L :=$ `Select-branch-variable`(ψ);
 return $\boldsymbol{K}$-SAT$_W$(`Unit-propagate`$(\psi, L), \nu \wedge L)$ **or**
        $\boldsymbol{K}$-SAT$_W$(`Unit-propagate`$(\psi, \neg L), \nu \wedge \neg L)$

procedure $\boldsymbol{K}$-SAT$_A(\mu)$
 $\Phi := \bigwedge \{\phi \,:\, \Box\phi$ occurs in $\mu\}$;
 for each $\Box\theta$ such that $\neg\Box\theta$ occurs in μ **do**
   $\theta := \neg\theta$;
   if not $\boldsymbol{K}$SAT$(\theta \wedge \Phi)$ **then return** $false$; `% backtrack`
 return $true$;

Algorithm 9.2.1 presents only the basic version for $\boldsymbol{K}$, restricted to CNF formulas. Thus, given a CNF formula ϕ and a literal L of ϕ, `Unite-propagate` performs unit propagation, which is explained as below.

Definition **9.2.3.** Given a CNF formula ϕ and a literal L in ϕ, *unit propagation* of L in ϕ consists of the following procedures:

- *unit resolution*: replace each disjunct of the form $\neg L \vee \psi$ by ψ in ϕ;

- *unit substitution*: remove each disjunct of the form $L \vee \psi$ from ϕ.

The original algorithm $\boldsymbol{K}$-SAT can deal with formulas of any format by refining `Unite-propagate`, see [GS00]. The correctness of Algorithm 9.2.1 is based on the result below. To formulate it, we need to briefly explain how a propositional assignment can generate a formula.

Definition **9.2.4.** Consider a finite propositional language and an assignment μ of truth values to the set of its proposition letters P: i.e., $\mu : P \mapsto \{0, 1\}$. Then the *formula generated* by μ is

$$\bigwedge \{p_i \,:\, \mu(p_i) = 1\} \wedge \bigwedge \{\neg p_j \,:\, \mu(p_i) = 0\}\,.$$

We denote the formula generated by a propositional assignment μ by $P(\mu)$. The above definition is essential for the formulation of the following result; for a proof, see [Seb97].

Theorem **9.2.5.** *Consider a modal language $\mathcal{ML}$ and an $\mathcal{ML}$ formula ϕ of the form*

$$\underbrace{\bigvee_{j}^{r_i} \Box\phi^i_j \vee \bigvee_{j'}^{r'_{i'}} \neg\Box\theta^i_{j'}}_{\textit{modal part}} \quad\vee\quad \underbrace{\bigvee_{l}^{m_i} L^i_l}_{\textit{proposition}}.$$

Then ϕ is satisfiable in an $\mathcal{ML}$ model iff there exists a truth-value assignment μ to the propositional variables in the propositional language

$$\mathcal{P} := \left\{p \,:\, p \textit{ occurs in some } L^i_l \textit{ in } \phi\right\} \cup \left\{\Box\phi^i_j,\ \Box\theta^i_{j'} \in \phi\right\}$$

such that the $\mathcal{P}$ formula $P(\mu)$ generated by μ is $\mathcal{ML}$ satisfiable, and μ satisfies the $\mathcal{P}$ formula ϕ.

9.3 Constraint Satisfaction and SAT Formulas

There is one obvious way of reformulating a SAT problem as a CSP. It requires a proposition to be reduced to its CNF; hence each resulting conjunct is regarded as a constraint. For instance, the disjunction

$$\neg x \vee y \vee z \tag{9.1}$$

is regarded as the constraint $C(x, y, z)$, the explicit description of which only rules out the triple $(1, 0, 0)$ from the interpretation domains of the variables x, y and z. In this formulation, arc consistency or hyper-arc consistency can take the place of unit propagation as proved in [Apt00b].

In the remainder of this section, we focus on the aforementioned encoding of formulas as (9.1) into constraints, and analyse a constraint propagation and solving algorithm for it. In Section 9.6, we suggest possible improvements to this, and how a different encoding of formulas as CSPs could be used for performing modal automated reasoning.

At this point, we fix the type of CSPs we deal with in the remainder of the present chapter.

DEFINITION **9.3.1.** A *Boolean* CSP $P := \langle X, \boldsymbol{D}, \boldsymbol{C} \rangle$ has domains that only contain 0 or 1.

Hence, Boolean constraints have the same domains and only differ in the chosen representation of constraints in this chapter; see [Wal00] for three other different encodings of propositional formulas as CSPs.

9.3.1 Mapping CNF Formulas into Constraints

Given Definition 9.3.1, it is not difficult to map CNF formulas into equivalent Boolean constraints. In this subsection, a constraint corresponds to a disjunction of literals — these are atoms or their negation, see Subsection 8.2.1. An explicit representation of such a constraint gives just the truth assignments that satisfy the clause. We provide the translation below.

DEFINITION **9.3.2.** Consider a CNF formula ψ. First, remove all propositional tautologies from ψ and let ψ' be the resulting CNF formula. Denote by At the ordered set of atoms that occur in ψ' (see Convention 9.2.2). Then apply the following procedure on all disjuncts ϕ in ψ':

- if ϕ is a unit disjunct L, then: if L is a propositional atom p_i, set $D_i = \{1\}$; if L is the negation of a propositional atom p_i, then set $D_i := \{0\}$;
- else, create a constraint $C(\phi)$ on the ordered set of variables $\{p_{i_1}, \ldots, p_{i_m}\}$ that occur in ϕ: a tuple d of 0's and 1's belongs to $C(\phi)$ iff the set of truth assignments
$$\mu := \left\{p_{i_j} \mapsto d\,[i_j] \;:\; j = 1, \ldots, m\right\}$$
satisfies the formula ϕ;
- associate the domain $\{0, 1\}$ with all the atoms that do not occur in unit clauses in ψ'.

Denote by $T^{CSP}_{SAT}(\psi)$ the resulting Boolean CSP.

The following result clearly holds.

FACT **9.3.3.** *An assignment μ satisfies a CNF formula ψ iff its restriction to the variables in $T^{CSP}_{SAT}(\psi)$ satisfies ψ viewed as a Boolean CSP.* □

9.3.2 Constraint Solving Algorithms

In Subsection 4.1.1, we briefly discussed constraint solving algorithms, by mentioning two well-known schemas: generate and test (`GT`) and backtracking (`BT`). We now provide some more details on them. In `GT`, all variables are instantiated (generation), and the resulting total assignment is tested against the problem constraint. In the `BT` schema, variables are instantiated sequentially; as soon as all the variables relevant to a constraint of the problem are instantiated, the resulting partial assignment is checked against the constraint. Whenever a partial assignment violates any of the constraints, `BT` backtracks to the last instantiated variable, whose current domain is non-empty. Thus `BT` searches through the space for solutions in a depth-first manner, see [Kum92].

In this subsection, we discuss a constraint solving methodology based on `BT`: this amounts to embedding one of the constraint propagation algorithms that we explained in Chapter 2 in `BT`. We explain this below, following the presentation of [Kum92].

The generic `BT` schema with some form of constraint propagation receives a CSP as input and computes a new CSP at each step. For instance, suppose that constraint propagation amounts to enforcing hyper-arc consistency in `BT`; then hyper-arc consistency is performed on each CSP that is step-by-step computed in the `BT` algorithm. If the current CSP has singleton domains (i.e., variable domains have precisely 1 element) and is hyper-arc consistent, then the problem is solved: i.e, the solution consists in assigning to the variables the unique value found in their respective domains. If during constraint propagation the domain of any variable becomes empty, then this CSP is removed from the search space. Otherwise, one of the variables, whose current domain has more than one element, is selected and a new CSP is computed, for each possible assignment of this variable. A `BT` algorithm checks the consistency of these computed CSPs in a depth-first manner until a solution is generated.

This brief outline is just one of many algorithms based on the schema "`BT` + constraint propagation". Those algorithms differ in the backtracking method used and, more interestingly for us, in the specific constraint propagation algorithm used. Moreover, there is not just the issue of *which* constraint propagation algorithm should be employed, we also have to decide *when* to use the algorithm, and till what point in the search space. In what follows, we shall not concern ourselves with this last aspect of backtracking schemas: a good introduction to this is still, to our knowledge, [Kum92]. Instead, in Subsection 9.3.3 below, we shall focus our attention on a specific algorithm schema that fits in the "`BT`+constraint propagation" methodology: forward checking.

9.3.3 The Forward Checking Algorithm Schema

Forward Checking (FC) is an algorithm schema that works like BT, except that domains of still unassigned variables change dynamically: when a variable x_i is assigned a value, the algorithm performs hyper-arc consistency checks on domains of variables that are still unassigned, by inspecting constraints on those variables and x_i. When hyper-arc inconsistency is detected, backtracking to the last assignment for x_i occurs; another value is assigned and, if this is inconsistent with a constraint on it, backtracking resorts to the variable instantiated before x_i. Nowadays, there are a number of variants of this basic schema; they mainly differ in the choice of domains and constraints on which hyper-arc consistency is performed, see [BMFL02]. The algorithm schema FC is displayed as Algorithm 9.3.1 below. Variables are partitioned in two sets, $\mathcal{A}$ and $\mathcal{U}$:

- $\mathcal{A}$ stores the set of variables that occur in the current assignment;
- $\mathcal{U}$ stores the remaining set of variables.

The algorithm HAC is called when the currently inspected variable x_i is assigned a value $a \in D_i$, and there are constraints in the current CSP $P := \langle X, \boldsymbol{D}, \boldsymbol{C} \rangle$ that involve x_i. Thus HAC performs a limited form of hyper-arc consistency on the problem which has constraints $C(s)$ such that x_i is in s, and domains of the form D_j such that x_j is in s. The specific choice of which and how many variables x_j, from $\mathcal{U}$ or $\mathcal{A}$, can occur in the constraints $C(s)$ of this problem varies according to the specific FC algorithm chosen.

Algorithm 9.3.1: FC($\mathcal{A}, \mathcal{U}, \mu, \boldsymbol{D}$)

if $F = \emptyset$ **then return** μ;
choose x_i from $\mathcal{U}$;
$stop := false$;
while $D_i \neq \emptyset$ **and not** $stop$ **do**
  choose $a \in D_i$;
  $Temp\mu := \mu \cup \{x_i := a\}$;
  $Temp\boldsymbol{D} :=$ HAC($x_i, a, \mathcal{U} - \{x_i\}, \boldsymbol{D} - \{D_i\}$);
  if not $Temp\boldsymbol{D} = \emptyset$ **then**
    return FC($\mathcal{A} \cup \{x_i\}, \mathcal{U} - \{x_i\}, Temp\mu, Temp\boldsymbol{D}$);
  else $stop = true$; % backtrack

Walsh [Wal00] provides a theoretical analysis of CSP-based approaches to SAT, and shows that a version of FC, called nFC1, outperforms the basic DP procedure on the encoding in Definition 9.3.2. The correctness and completeness of nFC1 is proved in [BMFL02]. To state the result, we need the following terminology, that also explains the name nFC1:

- denote by $\boldsymbol{C}^n_{c1}$ the set of constraints on a scheme s in which the current variable x_i and exactly one variable from $\mathcal{U}$ occur;

- denote by $\boldsymbol{CP}^n_{c1}$ the set of constraint projections on a scheme t in which the current variable x_i and exactly one variable from $\mathcal{U}$ occur.

- If x_i is the current variable passed to `HAC`, then perform hyper-arc consistency on the problem with domain $D_i = a$, constraints in $\boldsymbol{C}' := \boldsymbol{C}^n_{c1} \cup \boldsymbol{CP}^n_{c1}$, and domains of variables, different from x_i, that occur in the schemes of $\boldsymbol{C}'$. If an empty domain is generated, then return an empty domain set, else return the hyper-arc consistent domain set so generated.

Hence n`FC1` is the resulting version of `FC`. The name n`FC1` is motivated by the choice of constraints: n means that hyper-arc consistency is performed, and this involves constraints of any arity n; instead, 1 refers to the fact that exactly one uninstantiated variable is chosen to enforce hyper-arc consistency.

THEOREM **9.3.4** ([BMFL02]). *If the given CSP is consistent, then* n`FC`*1 returns a consistent assignment for it; else it reports that the problem is inconsistent.* $\Box$

9.4 The *K*`CSP` Algorithm

In the remainder of this chapter, we focus on a procedure for satisfying modal formulas via constraint propagation and satisfaction, that is based on the semantic intuitions underlying the intermediate translation in Definition 9.3.2. We suggest a number of other similar procedures in Section 9.6 below.

9.4.1 Examples

Before explaining the procedure ***K***`CSP` in Subsection 9.4.2 below, we start by considering an example formula in CNF, its encodings as CSP and how the procedure ***K***`CSP` works on it. Let us consider the following modal formula:

$$\psi := \Box(p \vee q) \wedge \neg\Box(p \wedge q) \wedge p.$$

After the minimum layer of the formula is computed, i.e., 0, the following CSP is returned — with propositional formulas as variables:

1. three propositional variables: $\Box(p \vee q)$; $\Box(p \wedge q)$; p;

2. the variable domains of $\Box(p \wedge q)$ and p are set to $\{1\}$; the variable domain of $\Box(p \wedge q)$ is set to $\{0\}$;

3. no constraints.

Then the CSP is passed to the CSP propositional solver that returns the only possible assignment μ (unique for this formula; in general, split may be needed to choose among alternative assignments): μ maps the three variables, in the order given above, to the triple $(1, 1, 0)$.

The modal procedure, invoked on μ, does the following: selects all formulas, within the scope of a $\Box$ operator, and join them in a conjunction Φ:

$$\Phi := p \vee q.$$

This is the *universal theory*: in model-theoretic terms, this is the formula that is to be satisfied by each modal successor at level 1 of a state satisfying $\Box(p \vee q)$ at level 0. Then, each formula that occurs in the scope of a negative occurrence of a $\Box$ operator is negated, hence transformed in CNF: in this case, the result is a formula Θ defined as

$$\Theta := \neg p \vee \neg q.$$

There may of course be multiple such existential theories Θ, which have to be satisfied at level 1, not necessarily at the same state. The conjunction $\Phi \wedge \Theta$ is passed to the propositional procedure; in this case, the conjunctive formula that is passed on is

$$(p \vee q) \wedge (\neg p \vee \neg q).$$

The formula is translated into a new CSP and its consistency is checked; this results into two possible assignments, hence the procedure halts returning that the original formula is satisfiable.

9.4.2 Mapping Modal Formulas into CSPs

As the example in Subsection 9.4.1 illustrates, proposition letters and modally quantified formulas are both treated as propositional variables; namely variables with only two possible values, 0 or 1.

Consider a unimodal language $\mathcal{ML} := \mathcal{ML}(P)$, and an $\mathcal{ML}$ conjunction ϕ of n disjunctions of the form

$$\underbrace{\bigvee_{j}^{r_i} \Box\phi_j^i \vee \bigvee_{j'}^{r'_{i'}} \neg\Box\theta_{j'}^i}_{\text{modal part}} \quad \vee \quad \underbrace{\bigvee_{l}^{m_i} L_l^i}_{\text{proposition}} \tag{9.2}$$

for $i = 1, \ldots, n$, where each L_l^i is either a proposition letter or its negation. The disjunction is then encoded as a constraint by treating all formulas of the form $\Box\phi_j^i$ or $\neg\Box\theta_{j'}^i$ as propositional literals, and applying the encoding T_{SAT}^{CSP} as in Definition 9.3.2. Formally, we have the following definition.

DEFINITION **9.4.1.** Consider a modal formula ϕ in CNF. Suppose that ϕ is a conjunction of n formulas such as (9.2). Let $\mathcal{L}(\phi)$ be the set of distinct propositions that occur in the set of literals L_l^i in ϕ. Thus consider the propositional language whose set of proposition variables is

$$Prop := \mathcal{L}(\phi) \cup \left\{\Box\phi_j^i,\, \Box\theta_j^i \;:\; i = 1, \ldots, n\right\},$$

and consider ϕ as a proposition in this language; call it ϕ^{Prop}. Then the *CSP translation* of the modal formula ψ into CSP form is

$$CSP(\phi) := T_{SAT}^{CSP}(\phi^{Prop}).$$

Using Fact 9.3.3, we obtain an analogue of Theorem 9.2.5.

COROLLARY **9.4.2.** *Consider a modal language $\mathcal{ML}$ and a formula ψ in CNF. The formula ψ is satisfiable in an $\mathcal{ML}$ model iff there exists a truth-value assignment μ that satisfies $CSP(\psi)$, and such that the propositional formula $P(\mu)$ generated by μ is $\mathcal{ML}$ satisfiable.*

PROOF. Assume that ϕ is a conjunction of disjunctions (9.2), and that all tautological formulas such as $p \vee \neg p$ have been removed from ϕ. Let $\mathcal{L}(\phi)$ be the set of distinct propositions that occur in the set of literals L_l^i in ϕ. Denote by $\mathcal{P}$ the propositional language whose letters are the $\mathcal{L}(\phi)$ propositions and all the ϕ disjuncts $\Box\psi_j^i$ and $\Box\theta_{j'}^i$.

Suppose that μ is an assignment that satisfies $CSP(\phi)$. Then it satisfies ϕ as a $\mathcal{P}$ formula, by Fact 9.3.3. Assume that $P(\mu)$ is satisfiable in an $\mathcal{ML}$ model $\mathcal{M}$ and world w. This implies that, if μ assigns 1 to a $\mathcal{P}$ letter in ϕ, then this holds true at w in $\mathcal{M}$; else it holds false. If μ does not assign any value to an $\mathcal{ML}$ proposition letter p, than we extend μ to p by means of the valuation in $\mathcal{M}$. Thus $\mathcal{M}$ satisfies ϕ at w.

Vice versa, suppose that $\mathcal{M}, w \models \phi$. Then define the $\mathcal{P}$ assignment μ as

$$\mu(\phi^i) = 1 \text{ iff } \mathcal{M}, w \models \phi^i,$$

for each proposition letter ϕ^i of $\mathcal{P}$. It is not difficult to prove by structural induction that μ makes ϕ true as a $\mathcal{P}$ formula and that $\mathcal{M}, w \models P(\mu)$. The result now follows from Fact 9.3.3. □

9.4.3 Mapping Modal Inferences into CSP Inferences

In Algorithm 9.4.1 below, `FC` returns an assignment for the input CSP. During search, an empty assignment ($false$) is generated iff the current CSP is detected to be inconsistent; i.e., the corresponding formula is unsatisfiable. Then backtracking takes place and another value for the last instantiated variable is checked etc. If no assignment can be found, the `FC` algorithm concludes that the input CSP is unsatisfiable. Thus the propositional search space is explored by `FC`, interleaving backtracking with hyper-arc consistency, in a depth-first manner.

The subprocedure ***K***`FC` has to handle modal satisfiability: more precisely, ***K***`FC` determines modal satisfiability by calling constraint satisfaction procedures over Boolean CSPs.

We only sketch a proof of the correctness and completeness of Algorithm 9.4.1.

THEOREM **9.4.3.** *The **K**`CSP` procedure returns a non-empty assignment iff the input modal formula is satisfiable.*

PROOF. The procedure `BoolCSP` transforms the given formula into as CSP as in Definition 9.4.1. This is proved consistent by `FC` iff the formula is satisfiable as a propositional formula, see Theorem 9.3.4 and Fact 9.3.3. Our theorem now follows from Corollary 9.4.2. $\Box$

Algorithm 9.4.1: *K*CSP

procedure `BoolCSP`(ψ)
  return `BoolFC`($CSP(\psi)$);

procedure `BoolFC`($CSP(\psi)$)
  $\mu :=$ `FC`($CSP(\psi)$);
  if $\mu = false$ **then return** $false$; `% backtrack`
  else return ***K***`FC`(μ);

procedure ***K***`FC`(μ)
  $\Phi := \bigwedge\{\phi : \Box\phi$ is assigned 1 in $\mu\}$;
  for each $\Box\theta$ in μ that is assigned 0 **do**
    $\theta :=$ `CNF`($\neg\theta$);
    if not `BoolCSP`($\theta \wedge \Phi$) **then return** $false$; `% backtrack`
  return $true$;

Notice that the ***K***`CSP` algorithm interleaves steps in which modal information is "hidden" — within the scope of modal operators — so as to get a propositional problem, with steps in which modal information is "unpacked" — i.e., the ***K***`FC` subprocedure is called on the modally quantified formulas.

9.5 Experimental Assessment

This section contains a brief experimental discussion of $\boldsymbol{K}$`CSP`. In order to test our preliminary procedure, we considered two test sets:

- manually coded formulas: these were devised following the criteria proposed in [HS96] for the creation of benchmark formulas;
- several formulas from the problem sets proposed by Heuerding and Schwendimann in [HS96], which were used in, for example, Tableaux'98 — see also p. 155, where this set is used to compare the output of the layered translation to the one of the standard translation.

The $\boldsymbol{K}$`CSP` algorithm was implemented in ECL^{i}PS^e, version 5.4, by Sebastian Brand; the implementation is called `mc.pl`. A translator from the [HS96] format into the format of `mc.pl` was provided by Juan Heguiabehere. We ran our experiments on an AMD Athlon Processor (1.1 GHz), with 512 MB RAM, under Red Hat Linux 7.1.

The program `mc.pl` returns a full search tree for the input formula. This is not an efficient choice, and in future experiments we shall also take this feature into account. Yet, this choice gave us a better idea of the behaviour of the algorithm, and this is our major concern at this stage of the work. Several formulas used in the experiments are available at `http://www.cwi.nl/~gennari/thesis/kcsp.html`.

By running the tests in [HS96], we noticed a clear difference between the behaviour of `mc.pl` on unsatisfiable and satisfiable formulas. So our discussion is divided in two subsections as below: results with unsatisfiable formulas; experiments with satisfiable formulas.

The current implementation of $\boldsymbol{K}$`CSP` is still a prototype, and it cannot compete with highly optimised theorem provers for modal logics as those discussed in Section 8.1. In the final Subsection 9.5.3, we elaborate on this issue and some possible improvements to the basic procedure $\boldsymbol{K}$`CSP`, that are triggered by the experimental work presented as below.

9.5.1 The Unsatisfiable Case

Our experiments with manually coded formulas were rather promising: we passed to `mc.pl` a series of modal formulas with at most 18 distinct proposition letters, modal depth between 0 and 3, and at most 18 disjuncts; no redundant propositional tautologies occur in those formulas. On each instance, the program answered correctly within 0.36 seconds; a number of these formulas, and the related search trees explored by `mc.pl` are on the aforementioned web page.

Once we were confident that our algorithm consistently displayed a good behaviour on manually coded formulas, we considered a number of $\boldsymbol{K}$ theorems, as provided in [HS96]: there, theorems are partitioned into sets, and formulas in a set usually differ in the number of propositional letters (V) and the modal depth (D). To run our experiments, first those $\boldsymbol{K}$ theorems were negated, then the obtained unsatisfiable formulas were transformed in CNF — see Definition 8.2.1.

Figure 9.1 below displays some of the tests that we ran: each label on the horizontal axis corresponds to a different test set; thus the results for the first two formulas from the considered test set are compared in terms of CPU seconds, as displayed along the vertical axis.

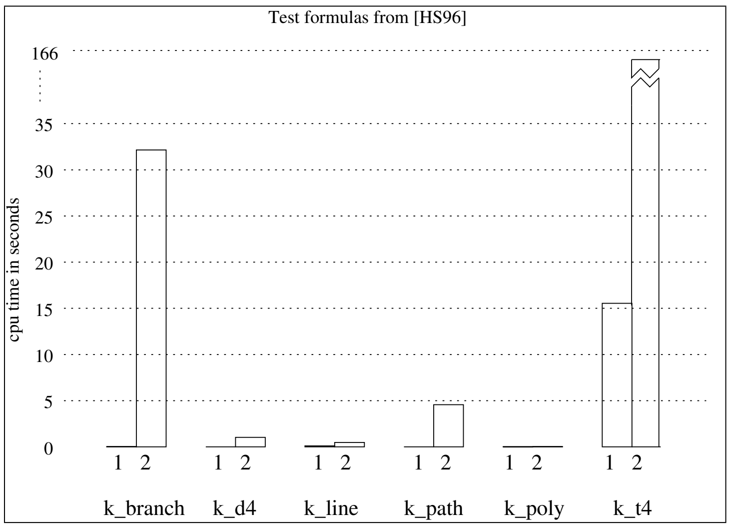


Figure 9.1: Comparison of the first two formulas from test sets in [HS96].

Table 9.1, displayed below, reports more formulas than those compared in Figure 9.1; the parameter C stands for the maximum number of distinct atoms (i.e., variables) in clauses (i.e., constraints) of the tested CNF formula. As in Section 8.5, we gave the theorem prover a time out of an hour; thus the abbreviation NA means that we did not obtain an answer within this time bound. As it is clear from Figure 9.1 and Table 9.1, tests become harder for the $\boldsymbol{K}$`CSP` algorithm when C$\geq$ 3, and D$\geq$ 4 or V$\geq$ 4; in fact, such values can result in more calls to the $\boldsymbol{K}$`CSP` procedure, and more backtracking points.

9.5.2 The Satisfiable Case

The best performances of the $\boldsymbol{K}$`CSP` algorithm was achieved in the unsatisfiable case, and the algorithm does not seem to be efficient in the satisfiable case yet. We believe that this behaviour of the algorithm does not depend on its current implementation, but on the design of the algorithm itself. For instance, the $\boldsymbol{K}$`CSP` algorithm does not fully exploit the fact that the input formulas are disjunctions;

Test formula	**V**	**D**	**C**	**CPU seconds**
k_branch_p1	5	2	3	0.03
k_branch_p2	7	3	3	32.15
k_d4_p1	1	4	2	0.00
k_d4_p2	1	5	2	1.03
k_dum_p1	1	4	4	8.55
k_dum_p2	1	5	4	NA
k_lin_p1	2	2	4	0.11
k_lin_p2	3	3	4	0.47
k_path_p1	6	1	1	0.00
k_path_p2	6	2	2	0.47
k_ph_p1	3	1	2	0.00
k_ph_p2	6	6	3	NA
k_poly_p1	7	3	2	0.01
k_poly_p2	15	6	2	0.02
k_t4_p1	4	6	3	15.54
k_t4_p2	4	7	3	165.69

Table 9.1: Test $\boldsymbol{K}$ theorems from [HS96]

i.e., it tries to return a total assignment, even when a partial assignment to its variables could be sufficient for determining satisfiability. As for this, let us consider a conjunction of $2n \geq 2$ formulas of the form

$$p \vee q_1^i \vee \cdots \vee q_n^i,$$
$$p \vee \neg q_1^i \vee \cdots \vee \neg q_n^i,$$

with $1 \leq i \leq n$. Such a formula is clearly satisfied by assigning 1 to p, and any other value to the remaining propositional letters. Still the Boolean forward checking procedure in $\boldsymbol{K}$`CSP` will search for a consistent total assignment, and uselessly enforce hyper-arc consistency. The situation becomes even worse when each q_j^i is substituted by a modally quantified formula; in fact this results in useless calls to the $\boldsymbol{K}$`FC` modal procedure.

9.5.3 Finale

As remarked in Subsection 9.5.2 above, the $\boldsymbol{K}$`CSP` algorithm still explores fruitless branches of the search tree. Such useless explorations depend on the basic `FC` algorithm schema: this is a complete solver that returns a total assignment to the input problem. In contrast, the `DP` procedure does not suffer from this drawback, thanks to unit substitution: unit substitution removes, from the search tree, those disjunctions in which the current variable that is assigned the value either *true* or *false* occurs positively or negatively, respectively. Therefore, a natural optimisation of the basic $\boldsymbol{K}$CSP algorithm will consist in embedding some form of unit substitution into $\boldsymbol{K}$CSP; then it will be meaningful and interesting to have both a theoretical and an experimental comparison of such a variation of the basic $\boldsymbol{K}$CSP procedure and the $\boldsymbol{K}$-SAT algorithm.

In the following section, we propose other possible variations of the basic $\boldsymbol{K}$CSP module: there, either hyper-arc consistency (see Subsection 9.6.1) or forward checking (see Subsection 9.6.2) are replaced by different constraint based algorithms.

9.6 Variations of $\boldsymbol{K}$CSP

In this section, we briefly propose two other possible approaches to determining modal satisfiability via constraint propagation and solving algorithms.

9.6.1 Boolean CSPs and Constraint Propagation

There is at least one other main reformulation of a SAT problem as a CSP. This makes use of so-called Boolean constraints. These are represented implicitly as Boolean formulas and equality constraints of the following forms, where x, y and z stand in for generic CSP variables:

$$x = y \text{ is an } EQ \text{ constraint;} \tag{EQ}$$

$$\neg x = y \text{ is a } NOT \text{ constraint;} \tag{NOT}$$

$$x \wedge y = z \text{ is an } AND \text{ constraint;} \tag{AND}$$

$$x \vee y = z \text{ is an } OR \text{ constraint.} \tag{OR}$$

The equality symbol in the constraints above is interpreted as $\leftrightarrow$, the logical connective of bi-implication. We can now provide a formal definition for Boolean CSPs with the above types of constraints.

Definition **9.6.1.** A Boolean CSP $P := \langle X, \boldsymbol{D}, \boldsymbol{C} \rangle$ is a *BOOL CSP* if its constraints are of the form (EQ), (NOT), (AND) or (OR) as above, that we call *BOOL constraints*.

To reason about such constraints, rules such as the following one are employed:

$$x \vee y = z,\ x = 0,\ z = 1 \quad \vdash \quad y = 1, \tag{9.3}$$

the intuitive reading of which is: if x or y have the same truth values as z, x is false and z is true, then y must be true. A set of independent rules, called $BOOL$, is provided in [Apt00b]. As the author proves, applying the set of rules $BOOL$ is "equivalent" to performing unit propagation. To formulate this equivalence, first a mapping of $BOOL$ constraints into clause sets is provided (by interpreting the equality symbol $=$ in the $BOOL$ constraints as the logical connective $\leftrightarrow$); hence by providing the opposite translation. Thus the author can prove, through those translations, that unit propagation and the $BOOL$ rule system can simulate each others in constant time, but "the simulation of the unit propagation by means of the Boolean constraint propagation [i.e., the $BOOL$ rules] leads to a generation of redundant constraints", due to the introduction of redundant variables. However, what is more interesting to us is the following equivalence.

THEOREM **9.6.2** ([APT00B]). *A CSP, without empty domains, is hyper-arc consistent iff it is closed under the applications of the rules of the BOOL system.*

Hence, the $BOOL$ system could be used instead of hyper-arc consistency in $\boldsymbol{K}$CSP to enforce modal consistency.

9.6.2 When a Bit of Cross-eye Helps

A third interesting approach is that of incorporating look-back enhancements of `BT`, like *Conflict Direct Backjumping* (`CDB`), into the basic `DP` procedure; see [BS97]. Look-back algorithms go in the opposite direction with respect to forward checking: more precisely stated, look-back algorithms exploit information about search that has already taken place, i.e., the set of the assigned variables $\mathcal{A}$ in the terminology of `FC`. The advantage of an algorithm like `DP` with `CDB` with respect to `FC` is that, like `DP`, the first does not need to return a total assignment. In fact, unit propagation is present in this improved version of `DP`; it is implemented by maintaining a pointer to the constraint, in the input CSP, that causes the exclusion of a specific assignment. We refer the reader to [BS97] for an accurate description of it and the related experimental work.

Given that also this improvement of `DP` is complete and correct, it could be interesting to see how it could improve the basic $\boldsymbol{K}$-SAT algorithm for modal logics. Not only could this be worth more exploration, but also a combination of look-ahead techniques, like `FC`, and look-back techniques, like `CDB`, already developed in the CSP community: in fact, as the experimental work in [BS97] highlights, "the dramatic performance improvements resulting from the incorporation of look-back is in fact due to a synergy between the look-ahead techniques and look-back techniques applied".

9.7 Conclusions

9.7.1 Synopsis

In Chapter 8 we partially encode the tree model property in a translation from modal to first-order logics; as shown in that chapter, this does pay off, and it allows us to reuse existing, sophisticated theorem provers for modal logics, without modifying them.

In the present chapter, the tree model property is still behind a series of procedures for modal logics. In fact, all these “reason” propositionally, layer by layer, on the input modal formula. They differentiate according to the basic propositional solver adopted.

Again, we try to reuse the “existing technology” for CSPs, and make it work for modal reasoning: this time, the encoding is not into first-order formulas but into sequences of CSPs, so to speak. These are passed to a propositional solver, that is slightly modified so as to “open” boxes and diamonds, and enforce satisfiability within them too, in a top-down manner.

9.7.2 Discussion

As shown in the first part of this thesis, a number of constraint techniques have been developed in the literature for tackling constraint satisfiability and, in particular, propositional satisfiability in an efficient manner. This chapter constitutes a first attempt to use constraint satisfaction algorithms for propositional satisfiability in the field of automated theorem proving for modal logics. Besides, we spend an entire section, namely Section 9.6 above, on possible variations of the solving schema proposed in this chapter, and possible optimisations to the latter.

It would also be interesting to study whether soft constraint propagation and solving algorithms could be used for testing a limited form of modal satisfiability. In some cases, we would like to set preferences on properties of the system: so that, even when not all properties can be satisfied, an optimal solution, with respect to our preferences, could still be returned. This could also constitute an interesting test-bed for soft constraint solving and propagation algorithms.

Part III

Finale

"Gravissimo errore", esclamò il terzo "il dolce è in fondo". ("Terrible mistake", exclaimed the third, "the sweet comes at the end".)
G. Rodari, Vecchi Proverbi, from *Favole al Telefono*, Einaudi, 1971.

In this part we formulate the conclusions to the thesis and discuss some remaining questions.

Chapter 10

Conclusions and Questions

Looking Backwards

The main themes shared by the two parts of this thesis are *knowledge representation*, and *efficient automated reasoning* on the chosen representation: Part I is concerned with a theoretical analysis of CSP algorithms, described through function iterations; Part II deals with efficient reasoning in the context of modal logics, by refining the way in which modal formulas are represented and passed to theorem provers. These parts are thus closer in rationale and methodology rather than in their contents. Below, we further motivate this claim by outlining the results in this thesis.

Back to Part I. This part describes and analyses a number of efficient algorithms for CSPs: the constraint propagation algorithms. Our aim, there, is *purely theoretical*: a unifying theory underpinning these algorithms, capable also to differentiate between them. Thus, in Chapter 3, we propose an algorithm schema, `SGI`, for constraint propagation algorithms. A simple theory for `SGI` is there developed. One of the primary objectives of our theorisation is declared to be the following (p. 28):

> using `SGI` or some of its variations for *describing* and *analysing how* the prune-and-propagate process is carried through by constraint propagation algorithms.

Hence, in Chapter 4, different domains of functions (e.g., domain orderings) are related to different classes of constraint propagation algorithms (e.g., arc consistency algorithms); thus each class of constraint propagation algorithms is associated with a type of function domains, and so separated from the others. Then we analyse each such class: we distinguish functions on the same domains for their different ways of performing pruning (point or set based), and consequently differentiate between algorithms of the same class (e.g., `AC-1` and `AC-3` versus `AC-4` or `AC-5`).

Besides and foremost, we also correlate properties of functions (e.g., commutativity or stationarity) to different strategies of propagation in constraint algorithms (see, for instance, `AC-1` versus `AC-3`), and suggest that these can be used for optimisation tasks.

In Chapter 5 the `SGI` schema is applied to soft CSP algorithms, thereby clarifying some of the similarities and differences between the crisp and soft constraint propagation algorithms. Also, properties of functions, e.g., monotonicity and inflationarity, are studied as separate issues in Chapter 3. Thus their respective roles in connection with certain behaviours of soft constraint propagation algorithms are differentiated. This is an achievement *per se*: in the soft constraint literature, often, the two properties are studied together and the role of each in the analysis of soft constraint propagation thus gets lost. Furthermore, we also obtain three new general conditions for the termination of semiring-based constraint propagation algorithms via this abstract approach.

Therefore, the adopted schema proves to be suitable for *verifying* constraint propagation algorithms, *classifying* them, *comparing* them, *explaining* and *separating* their properties. All these is done through the unifying framework of `SGI` function iterations. See also Table 4.1, p. 82.

Finally we characterise all the functions used for constraint propagation; in fact the other goal of our theorisation is (see Chapter 3):

> abstracting *which* functions, iterated as in `SGI` or its variations, perform the task of pruning or propagation of inconsistencies in constraint propagation algorithms.

We accomplish this in Chapter 6, restricting the field of domain or constraint functions to those that are actually traced in the surveyed constraint propagation algorithms.

Back to Part II. In this part we shift perspective and approach, even though this part is concerned with relations and relational structures too, but in the context of modal logics. While the aim in the first part of this thesis is purely theoretical, in Part II our task is described as follows, see Subsection 7.1.1:

> determining the satisfiability of modal formulas in an efficient manner.

In Chapter 8, we focus on one way of doing this: we refine the standard translation as the layered translation, and use existing theorem provers for first-order logic on the output of this refined translation. We provide ample experimental evidence on the improvements in performances that are gained through the refinement, see Section 8.5.

The refinement of the standard translation has strong semantic motivations: a strong form of the tree modal property. This property is also used in the basic algorithm schema in Chapter 9. In fact, that property is behind the proposed algorithms based on constraint satisfaction methods. First modal formulas are encoded into propositional formulas and these into CSPs. The chosen constraint solver thus proceeds "layer by layer" in the encoding of the modal formula and in its candidate models, by applying a CSP solver for propositional satisfiability at each layer.

Chapter 9 brings us back to constraint algorithms, and apply them to modal reasoning problem. It constitutes a first attempt to tackle modal satisfiability by means of constraint propagation algorithms, as explained in Chapter 4, or various refinements of the basic backtracking schema for constraint satisfaction.

With Chapter 9, we wish to draw the attention of constraint programmers to modal logics, and of modal logicians to CSPs. Modal logics themselves express interesting problems in terms of relations and unary predicates, like temporal reasoning tasks (see also Subsection 7.2.4). On the other hand, constraint algorithms manipulate relations in the form of constraints, and unary predicates in the form of domains or unary constraints, see Chapter 6. Thus the question of how efficiently those algorithms can be applied to modal reasoning problems seems quite natural and challenging.

Back to the origins. The general approach to constraint propagation via function iterations, presented in Part I, was first devised by Apt [Apt99a, Apt00a]. We extended it in [Gen00] to explain the `AC-4` algorithm of Mohr and Henderson [1986], the `AC-5` algorithm of Van Hentenryck et al. [1992], and the `HAC-4` algorithm of Mohr and Masini [1988]. In [Gen02], we showed how another modification of the original iteration schema can be used to explain the `PC-4` path consistency algorithm of Han and Lee [1988] and the `KS` algorithm of Cooper [1989], that can achieve either k-consistency or strong k-consistency. We also extended the theory of function iterations to soft constraints in the joint papers [BGR00, BGR02]. The work in Part I of this thesis presents a unifying framework that explains all those algorithms, and the basic strong relational consistency algorithm of Dechter and van Beek [1997].

In Part II, the work in Chapter 9 is based on a joint paper [AGHdR00] with Areces, Heguiabehere and de Rijke. This is based on the standard translation of van Benthem [1983]. Our refinement there is semantic driven. This same semantic intuition triggers the work in Chapter 9: the design of a constraint solver for modal logics, based on the propositional solver of [GS00].

Looking Ahead

As the above section highlights, a number of questions follow up from this thesis. We briefly discuss some of them as below, starting with those related to CSPs and then passing to modal logics.

Constraint Propagation

In the above section, we conclude that the `SGI` schema appears to be sufficiently general to abstract the common features of constraint propagation algorithms. Besides and foremost, the `SGI` schema is sufficiently expressive and ductile to allow us to distinguish each of those algorithms according to its specific strategy in iterating functions, so to speak. Certainly, a proof theoretic view of constraint propagation algorithms would be more general than the `SGI` based one. It seems thus natural to investigate whether such a proof theoretic view can be as expressive as the `SGI` based approach.

In the conclusions to Chapter 6, we suggest that it would be interesting to compare the given characterisation of constraint algorithms in terms of functions to the database relational model: in particular, this could be useful for optimisation tasks as our discussion on p. 122 pinpoints. We also suggest how this view can be extended to soft CSPs; we spotlight the problems that we face in this extension, and how these can be tackled.

In the conclusions to Chapters 4 and 5 we also suggest how the approach via functions, adopted in this thesis, could be useful in devising new algorithms, or new notions of constraint propagation; the basic algorithm schema can be `SGI` or a variation of it, the iterated functions should be polynomial-time computable. In what follows, we touch on this issue in the context of modal logics.

Diamonds

A question emerges from the second part of this thesis: how efficiently constraint based algorithms can be applied to modal reasoning problems. The question can be investigated by working further on both the following issues:

- mapping modal formulas into CSPs,
- mapping inferences on modal formulas into constraint propagation and satisfaction steps.

For instance, efficient reasoning in this setting can be achieved by refining the encodings of modal formulas into CSPs, as proposed in Chapter 9; just like we do in Chapter 8 where we refine the standard translation of modal formulas into first-order formulas. In turn, these refined encodings could provide more information to constraint-based propagation and satisfaction algorithms, and this information could result in more efficient constraint-based algorithms.

The area of automated theorem proving for modal logics could also constitute a test-bed for soft constraint algorithms, as suggested in Section 9.7. In real time systems, modal formulas are used to express properties of the systems; in this context, the user is often interested in retrieving an optimal solution, that prioritises certain properties. Hence soft CSPs and algorithms may be employed in such situations, or existing algorithms may be optimised for these tasks.

Appendix A

Original Algorithms

Algorithm A.1: $\texttt{KS}(X, \boldsymbol{D}, \boldsymbol{C})$

$G := \emptyset$;
$F := \{(d, s) \,:\, d \in D\,[s]\,,\, s \text{ a scheme on } X\}\,$;
for each $i := 1, \ldots, n$ and $(d, s) \in F$ s.t. the length of s is i **do**
  for each $x_j \notin s$ **do** $C(d, s, x_j) := D_j$;
  if $(d, s) \notin C(s)$ for some constraint on s **then**
    $G := G \cup \{(d, s, i)\}\,$;
    $F := F - \{(d, s)\}\,$; % to inspect it at most once
while $G \neq \emptyset$ **do**
  choose $(s, d, i) \in G$;
  $G := G - \{(d, s, i)\}\,$;
  if $i < k$ **then**
    for each scheme $t = s \cup \{x_k\}$ s.t. $x_k \notin s$ and $a \in D_k$ **do**
      $e = d \bowtie a$;
      if $(t, e) \in F$ **then**
        $G := G \cup \{(t, e, i+1)\}\,$;
        $F := F - \{(e, t)\}\,$;% to inspect it at most once
        $C(t) := C(t) - \{e\}\,$;
  if $i > 1$ **then**
    for each $j = 1 \ldots i$ **do**
      $r := s - \{x_j\}\,$;
      $e := d\,[r]\,$;
      $C(e, r, j) := C(e, r, j) - \{d\,[x_j]\}\,$;
      if $C(e, r, j) = \emptyset$ and $(e, r) \in F$ **then**
        $G := G \cup \{(e, r, i-1)\}\,$;
        $F := F - \{(e, r)\}\,$;% to inspect it at most once
        $C(r) := C(r) - \{e\}\,$;

Algorithm A.2: `HAC-1`$(X, \boldsymbol{D}, \boldsymbol{C})$

$S := \{C(s) \,:\, C(s)$ is a constraint of the problem constraint set $\boldsymbol{C}\}$;
$\boldsymbol{D}' := \boldsymbol{D}$;
while $S \neq \emptyset$ **do**
  choose $C(s) \in S$;
  $S := S - \{C(s)\}$;
  for each $x_i \in s$
    $D_i' := \{a \in D_i, : \exists d \in C(s) \cap D\,[s]$ s.t. $d\,[x_i] = a\}$;
    if $D_i' \neq D_i$ **then**
      $S := S \cup \{C(t) \,:\, \exists j \in s$ s.t. $j \in t\}$;
      $D_i := D_i'$;

Algorithm A.3: `AC-3`$(X, \boldsymbol{D}, \boldsymbol{C})$

$S := \{C(x_i, x_j), C(x_j, x_i) \,:\, C(x_i, x_j)$ is in the problem constraint set $\boldsymbol{C}\}$;
$\boldsymbol{D}' := \boldsymbol{D}$;
while $S \neq \emptyset$ **do**
  choose $C(x_k, x_l) \in S$;
  $S := S - \{C(x_k, x_l)\}$;
  $D_k' := \{a \in D_k, : \exists (a, b) \in C(x_k, x_l) \cap D\,[x_k, x_l]$ s.t. $d\,[x_k] = a\}$;
  if $D_k' \neq D_k$ **then**
    $S := S \cup \{C(x_k, x_i),\, C(x_i, x_k) : \, k \neq i,\, l\}$;
    $D_k := D_k'$;

Algorithm A.4: `HAC-4`$(X, \boldsymbol{D}, \boldsymbol{C}, G)$

while $G \neq \emptyset$ **do**
  choose $(x_i, a) \in G$;
  $G := G - \{(x_i, a)\}$;
  for each $C(x_i, a; s)$ **do**
      for each $x_j \in s - \{i\}$ **do**
          for each $b \in D_j$ **do**
            $C(x_j, b; s) := C(x_j, b; s) - \{d\}$;
            if $C(x_j, b; s) = \emptyset$ and (x_j, b) **then**
              $G := G \cup \{(x_j, b)\}$;
              $F := F - \{(x_j, b)\}$; % to choose it only once
              $D\,[j] := D\,[j] - \{b\}$;

Algorithm A.5: `PC-4`$(X, \boldsymbol{D}, \boldsymbol{C})$

while $G \neq \emptyset$ **do**
  choose $(x_k : d \; x_l : e) \in G$;
  $G := G - \{(x_k : d\,,\, x_l : e)\}\,;$
  for each $j = 1, \ldots, n$ and $c \in D_j$ **do**
    if $k < j$ **then**
      $C(x_k : d\,,\, x_j : c\,;\, x_l) := C(x_k : d\,,\, x_j : c\,;\, x_l) - \{e\}\,;$
      if $C(x_k : d\,,\, x_j : c\,;\, x_l) = \emptyset$ and $(x_k : d\,,\, x_j : c) \in F$ **then**
        $G := G \cup \{(x_k : d\,,\, x_j : c)\}\,;$
        $F := F - \{(x_k : d\,,\, x_j : c)\}\,;$% to choose it once
        $C(x_k, x_j) := C(x_k, x_j) - \{(d, c)\}\,;$
        else
        $C(x_j : c\,,\, x_k : d\,;\, x_l) := C(x_j : c\,,\, x_k : d\,;\, x_l) - \{e\}\,;$
        if $C(x_k : d\,,\, x_j : c\,;\, x_l) = \emptyset$ and $(x_j : c\,,\, x_k : d) \in F$
          $G := G \cup \{(x_j : c\,,\, x_k : d)\}\,;$
          $F := F - \{(x_j : c\,,\, x_k : d)\}\,;$% to choose it once
          $C(x_j, x_k) := C(x_j, x_k) - \{(c, d)\}\,;$
    if $j < l$ **then**
      $C(x_j : c\,,\, x_l : e\,;\, x_k) := C(j : c\,,\, l : e\,;\, k) - \{d\}\,;$
      if $C(x_j : c\,,\, x_l : e\,;\, x_k) = \emptyset$ and $(x_j : c\,,\, x_l : e) \in F$ **then**
        $G := G \cup \{(x_j : c\,,\, x_l : e)\}\,;$
        $F := F - \{(x_j : c\,,\, x_l : e)\}\,;$ % to choose it once
        $C(x_j, x_l) := C(x_j, x_l) - \{(c, e)\}\,;$
        else
        $C(x_l : e\,,\, x_j : c\,;\, x_k) := C(x_l : e\,,\, x_j : c\,;\, x_k) - \{d\}\,;$
        if $C(x_l : e\,,\, x_j : c\,;\, x_k) = \emptyset$ and $(x_l : e\,,\, x_j : c) \in F$ **then**
          $G := G \cup \{(x_l : e\,,\, x_j : c)\}\,;$
          $F := F - \{(x_l : e\,,\, x_j : c)\}\,;$ % to choose it once
          $C(x_l, x_j) := C(x_l, x_j) - \{(e, c)\}\,;$

Bibliography

[ADH$^+$87] A. Aggoun, M. Dincbas, A. Herold, H. Simonis, and P. van Hentenryck. The CHIP System. Technical Report TR-LP-24, European Computer Industry Research Centre, Munich, June 1987.

[Agg95] A. Aggoun, et al. *ECL^iPS^e User Manual*. Munich, Germany, 1995.

[AGHdR00] C. Areces, R. Gennari, J. Heguiabehere, and M. de Rijke. Tree-based Heuristics in Modal Theorem Proving. In *Proc. of the 14th European Conference on Artificial Intelligence 2000*, pages 199–203. IOS Press, 2000.

[AHV95] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley Publishing Company, 1995.

[Aie02] M. Aiello. *Spatial Reasoning. Theory and Practice*. PhD thesis, ILLC, Universiteit van Amsterdam, February 2002.

[Apt97] K.R. Apt. *From Logic Programming to Prolog*. Prentice Hall, London, U.K., 1997.

[Apt99a] K.R. Apt. The Essence of Constraint Propagation. *Theoretical Computer Science*, 221(1-2):179–210, 1999.

[Apt99b] K.R. Apt. The Rough Guide to Constraint Propagation. In *Proc. of the 5th International Conference on Principles and Practice of Constraint Programming (CP'99)*, Lecture Notes in Computer Science, pages 1–23. Springer-Verlag, 1999.

[Apt00a] K.R. Apt. The Role of Commutativity in Constraint Propagation Algorithms. *ACM TOPLAS*, 22(6):1002–1036, 2000.

[Apt00b] K.R. Apt. Some Remarks on Boolean Constraint Propagation. In *New Trends in Constraints*, volume 1865 of *Lecture Notes in Artificial Intelligence*, pages 91–107, 2000.

[Are00] C.E. Areces. *Logic Engineering. The Case of Hybrid and Description Logics*. PhD thesis, ILLC, Universiteit van Amsterdam, May 2000.

[Bac01] F. Bacchus. A Uniform View of Backtracking. Accessed via `http://www.cs.toronto.edu/~fbacchus/on-line`, December 2001.

[BAFB96] A. Borning, R. Anderson, and B. Freeman-Benson. Indigo: A local Propagation Algorithm for Inequality Constraints. In *Proc. of the 10th Annual ACM Symposium on User Interface Software and Technology*, pages 129–136, 1996.

[Bar97a] R. Barták. A Generalized Algorithm for Solving Constraint Hierarchies. Technical Report 97/1, Department of Theoretical Computer Science, Charles University, Prague, 1997.

[Bar97b] R. Barták. A Plug-in Architecture of Constraint Hierarchy Solvers. In *Proc. of the 3rd International Conference on the Pratical Application of Constraint Technology (PACT'97)*, pages 359–371, 1997.

[Bar98] R. Barták. Constraint Hierarchy Networks. In *Proceedings of the 3rd ERCIM/Compulog Workshop on Constraints*, Lecture Notes in Computer Science. Springer, 1998.

[BCR00] S. Bistarelli, P. Codognet, and F. Rossi. An Abstraction Framework for Soft Constraints and its Relationship with Constraint Propagation. In *Proc. of the Symposium on Abstraction, Reformulation, and Approximation (SARA 2000)*, volume 1864 of *Lecture Notes in Artificial Intelligence*, pages 71–86. Springer-Verlag, 2000.

[BdRV01] P. Blackburn, M. de Rijke, and Y. Venema. *Modal Logic*. Cambridge University Press, 2001.

[BFB98] A. Borning and B. Freeman-Benson. Ultraviolet: A Constraint Satisfaction Algorithm for Interactive Graphics. *CONSTRAINTS*, 3(1):9–32, April 1998.

[BFBW92] A. Borning, B. Freeman-Benson, and M. Wilson. Constraint Hierarchies. *Lisp and Symbolic Computation*, 5(3):223–270, 1992.

[BGR00] S. Bistarelli, R. Gennari, and F. Rossi. Constraint Propagation for Soft Constraint Satisfaction Problems: Generalization and Termination Conditions. In *Proc. of the 6th International Conference on*

Principles and Practice of Constraint Programming, volume 1894 of *Lecture Notes in Computer Science*, pages 83–97. Springer, 2000.

[BGR02] S. Bistarelli, R. Gennari, and F. Rossi. General Properties and Termination Conditions for Soft Constraint Propagation. *CONSTRAINTS*, 2002. In press.

[BMFL02] C. Bessière, P. Meseguer, E.C. Freuder, and J. Larrosa. On Forward Checking for Non Binary Constraint Satisfaction. *Artificial Intelligence*, 2002. To appear, available via `http://www.lirmm.fr/~bessiere`.

[BMR97] S. Bistarelli, U. Montanari, and F. Rossi. Semiring-based Constraint Satisfaction and Optimization. *Journal of ACM*, 44(2):201–236, 1997.

[BMSX97] A. Borning, K. Marriott, P. Stuckey, and Y. Xiao. Solving Linear Arithmetic Constraints for User Interface Applications. In *Proc. of the 11th Annual ACM Symposium on User Interface Software and Technology*, pages 87–96, October 1997.

[BS97] R.J.Jr. Bayardo and R.C. Schrag. Using CSP Look-back Techniques to Solve Real-world SAT Instances. In *Proc. of the Fourteenth National Conference on Artificial Intelligence*, pages 203–208, 1997.

[Coo89] M.C. Cooper. An Optimal k-Consistency Algorithm. *Artificial Intelligence*, 41:89–95, 1989.

[CS01] M.C. Cooper and T. Schiex. Arc Consistency for Soft Constraints. `arXiv:cs:AI/0111038 v3`, November 30 2001.

[DAA95] T. Dean, J. Allen, and Y. Aloimonos. *Artificial Intelligence. Theory and Practice*. Addison-Wesley Publishing Company, 1995.

[DFP93] D. Dubois, H. Fargier, and H. Prade. The Calculus of Fuzzy Restrictions as a Basis for Flexible Constraint Satisfaction. In *Proc. of the 2nd IEEE International Conference on Fuzzy Systems (IEEE)*, pages 1131–1136, 1993.

[dGVS97] S. de Givry, G. Verfaillie, and T. Schiex. Bounding the Optimum of Constraint Optimization Problems. In *Proc. of the 3rd International Conference on Principles and Practice of Constraint Programming (CP97)*, volume 1330 of *Lecture Notes in Computer Science*. Springer, 1997.

[DKL01] R. Dechter, K. Kask, and J. Larrosa. A General Scheme for Multiple Lower Bound Computation in Constraint Optimization. In T. Walsh, editor, *Proc. of the 7th International Conference of Principle and Practice of Constraint Programming (CP 2001)*, volume 2239 of *Lecture Notes in Computer Science*, pages 346–359. Springer, 2001.

[dN94] H. de Nivelle. *Ordering Refinements of Resolution*. PhD thesis, Technische Universiteit Delft, Delft, October 1994.

[dNdR02] H. de Nivelle and M. de Rijke. Deciding the Guarded Fragments by Resolution. *Journal of Symbolic Computation*, 2002.

[Doe94] H.C. Doets. *From Logic to Logic Programming*. The MIT Press, Cambridge, MA, 1994.

[DP90] B.A. Davey and H.A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 1990.

[dR93] M. de Rijke. *Extending Modal Logic*. PhD thesis, ILLC, Universiteit van Amsterdam, December 1993.

[DvB97] R. Dechter and P. van Beek. Local and Global Relational Consistency. *Theoretical Computer Science*, 173:283–308, 1997.

[FL93] H. Fargier and J. Lang. Uncertainty in Constraint Satisfaction Problems: a Probabilistic Approach. In *Proc. of European Conference on Symbolic and Qualitative Approaches to Reasoning and Uncertainty (ECSQARU)*, volume 747 of *Lecture Notes in Computer Science*, pages 97–104. Springer, 1993.

[Fre78] E.C. Freuder. Synthesizing Constraint Expressions. *Communication of ACM*, 21:958–966, 1978.

[Frü98] T. Frühwirth. Theory and Practice of Constraint Handing Rules. *Journal of Logic Programming*, 31(1-3):95–138, 1998.

[FW92] E.C. Freuder and R.J. Wallace. Partial Constraint Satisfaction. *Artificial Intelligence*, 58:21–70, 1992.

[Gen98] R. Gennari. Temporal Constraint Programming: a Survey. *CWI Quarterly*, 11(2-3), 1998.

[Gen00] R. Gennari. Arc Consistency via Subsumed Functions. In Lloyd et al., editor, *Proc. of Computational Logic 2000 (CL 2000)*, volume 1861 of *Lecture Notes in Artificial Intelligence*, pages 358–372. Springer, 2000.

[Gen01a] R. Gennari. Solution Preserving Translations for Comparing Soft Frameworks. Available via `http://www.wins.uva.nl/∼rgennari`, 2001.

[Gen01b] R. Gennari. Translations for Comparing Soft Frameworks. In T. Walsh, editor, *Proc. of the 7th International Conference on Principle and Practice of Constraint Programming (CP 2001)*, volume 2239 of *Lecture Notes in Computer Science*, page 764. Springer, 2001.

[Gen02] R. Gennari. The GIF Algorithm. A General Schema for Constraint Propagation. *Joint Bulletin of of the Novosibirsk Computing Center and Institute of Informatics Systems*, 2002. In press.

[GJ79] M.R. Garey and D.S. Johnson. *Computers and Intractability. A Guide to the Theory of NP-completeness*. Freeman, 1979.

[GLS99] G. Gottlob, N. Leone, and F. Scarcello. Hypertree Decomposition and Tractable Queries. In *Proc. of the 18th ACM Symposium on Principles of Database Systems*, pages 21–32, 1999.

[Grä99] E. Grädel. Why Are Modal Logics so Robustly Decidable? In *Bulletin EATCS*, volume 68, pages 90–103. 1999.

[GS00] F. Giunchiglia and R. Sebastiani. Building Decision Procedures for Modal Logics from Propositional Decision Procedures. The Case Study of Modal $\mathbf{K}(m)$. *Information and Computation*, 162(1–2):158–178, 2000.

[Hal95] J.Y. Halpern. The Effect of Bounding the Number of Primitive Propositions and the Depth of Nesting on the Complexity of Modal Logics. *Artificial Intelligence*, 1995.

[HdR01] J. Heguiabehere and M. de Rijke. The Random Modal QBF Test Set. In *Proc. of IJCAR Workshop on Issues in the Design and Experimental Evaluation of Systems for Modal and Temporal Logics*, 2001.

[Her75] I.N. Herstein. *Topics in Algebra*. Xeros Corporation, 1975.

[HL88] C.-C. Han and C.-H. Lee. Comments on Mohr and Henderson's Path Consistency Algorithms. *Artificial Intelligence*, 36:125–130, 1988.

[HM02] V. Haarslev and R. Möller. RACER. Accessed via `http://kogs-www.informatik.uni-hamburg.de/~race/`, September 2002.

[Hor02] I. Horrocks. FaCT. Accessed via `http://www.cs.man.ac.uk/~horrocks/FaCT/`, September 2002.

[Hos98] H. Hosobe. *Theoretical Properties and Efficient Satisfaction of Hierarchical Constraint Systems*. PhD thesis, Department of Information Science, University of Tokyo, March 1998.

[HR00] M.R.A. Huth and M.D. Ryan. *Logic in Compuer Science*. Cambridge University press, 2000.

[HS96] A. Heuerding and S. Schwendimann. A Benchmark Method for the Propositional Modal Logics **k**, **kt**, **s4**. Technical Report IAM-96-015, University of Bern, 1996.

[HS97] U. Husdtadt and R.A. Schmidt. On Evaluating Decision Procedures for Modal Logics. In M. Pollack, editor, *Proceedings of the International Joint Conference of Artificial Intelligence (IJCAI97)*, pages 202–207, 1997.

[Jam96] M. Jampel. *Over-constrained Systems in CLP and CSP*. PhD thesis, Department of Computer Science, City University, September 1996.

[Jon97] N.D. Jones. *Computability and Complexity. From a Programming Perspective*. MIT Press, 1997.

[Kum92] V. Kumar. Algorithms for Constraint Sastisfaction Problems: a Survey. *AI Magazine*, 13(1):32–44, 1992.

[KvB97] G. Kondrak and P. van Beek. A Theoretical Evaluation of Selected Backtracking Algorithms. *Artificial Intelligence*, 89:365–387, 1997.

[Lov78] D.W. Loveland. *Automated Theorem Proving: A Logical Basis*, volume 6 of *Fundamental Studies in Computer Science*. North-Holland, 1978.

[LP81] H. Lewis and C.H. Papadimitriou. *Elements of the Theoy of Computation*. Prentice-Hall, 1981.

[Mac97] A. Mackworth. Consistency in Network of Relations. *Artificial Intelligence*, 8(1):99–118, 1997.

[MH86] R. Mohr and T.C. Henderson. Arc and Path Consistency Revisited. *Artificial Intelligence*, 28:225–233, 1986.

[MM88] R. Mohr and G. Masini. Good Old Discrete Relaxation. In *Proc. of the 8th European Conference on Artificial Intelligence (ECAI'88)*, pages 651–656. Pitman Publisher, 1988.

[Mon74] U. Montanari. Networks of Constraints: Fundamental Properties and Applications to Picture Processing. *Information Science*, 7(2):95–132, 1974.

[Mon00] E. Monfroy. A coordination-based chaotic iteration algorithm for constraint propagation. In *Proc. of the 2000 ACM Symposium on Applied Computing (SAC'2000)*, pages 262–270. ACM, 2000.

[MS98] K. Marriott and P. Stuckey. *Programming with Constraints*. MIT Press, 1998.

[ONdRG01] H.J. Ohlbach, A. Nonnegart, M. de Rijke, and D.M. Gabbay. Encoding Two-valued Non-classical Logics in Classical Logic. In J. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*. Elsevier, 2001.

[Pap94] C.H. Papadimitriou. *Computational Complexity*. Addison-Wesley Publishing Company, 1994.

[PS02] P.F. Patel-Schneider. DLP. Accessed via `http://www.bell-labs.com.user/pfps/dlp/`, September 2002.

[Rut94] Zs. Ruttkay. Fuzzy Constraint Satisfaction. In *Proc. of the 3rd IEEE International Conference on Fuzzy Systems*, pages 1263–1268, 1994.

[RV01] J. Robinson and A. Voronkov, editors. *Handbook of Automated Reasoning*. Elsevier, 2001.

[Sch00] T. Schiex. Arc Consistency for Soft Constraints. In R. Dechter, editor, *Proc. of the 6th International Conference on Principles and Practice of Constraint Programming (CP 2000)*, volume 1894 of *Lecture Notes in Computer Science 1894*, pages 411–424. Springer, 2000.

[Seb97] R. Sebastiani. *Una Nuova Classe di Procedure di Decisione per le Logiche Modali e Terminologiche. Teoria, Implementazione e Testing*. PhD thesis, Facolt'a di Ingegneria, Università di Genova, 1997.

[SFV95] T. Schiex, H. Fargier, and G. Verfaillie. Valued Constraint Satisfaction Problems: Hard and Easy Problems. In *Proc. of the 14th International Joint Conference on Artificial Intelligence*, pages 631–637. Morgan Kaufmann, 1995.

[SMFBB93] M. Sannella, J. Maloney, B. Freeman-Benson, and A. 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.

[Smo95] G. Smolka. The Oz Programming Model. In J. van Leeuwen, editor, *Computer Science Today*, volume 1000 of *Lecture Notes in Computer Science*, pages 324–343. Springer, 1995.

[SPA00] SPASS 1.0.3. Accessed via `http://www.spass.mpi-sb.mpg.de/`, May 2000.

[TAN00] TANCS: Tableaux Non Classical Systems Comparison. Accessed via `http://www.dis.uniroma1.it/~tancs`, January 2000.

[Tsa93] E.P.K. Tsang. *Foundations of Constraint Satisfaction*. Academic Press, 1993.

[Ull80] J.D. Ullman. *Principles of Database Systems*. Pitman, Computer Science Press, 1980.

[UW97] J.D. Ullman and J. Widom. *A First Course in Database Systems*. Pitman, Prentice-Hall, 1997.

[Var97] M.Y. Vardi. Why is modal logic so robustly decidable? *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 31:149–184, 1997.

[Var00] M.Y. Vardi. Constraint Satisfaction and Database Theory: a Tutorial. In *Proc. of the 19th International Conference on Management Data ACM SIGMOD-SIGACT-SIGART and Symposium on Principles of Database Systems*, pages 76–85. ACM Press, 2000.

[vB83] J. van Benthem. *Modal Logic and Classical Logic*. Bibliopolis, 1983.

[vHDT92] P. van Hentenryck, Y. Deville, and C.-M. Teng. A Generic Arc-consistency Algorithm and its Specializations. *Artificial Intelligence*, 57:291–321, 1992.

[Wal75] D. Waltz. Understanding Line Drawings of Scenes with Shadows. In P.H. Winston, editor, *The Psychology of Computer Vision*, pages 19–91. McGraw-Hill, 1975.

[Wal00] T. Walsh. SAT v CSP. In R. Dechter, editor, *Proc. of the 6th International Conference on Principles and Practice of Constraint Programming*, volume 1894, pages 441–456, 2000.

[Was00] R. Wassermann. *Resource-bounded Belief Revision*. PhD thesis, ILLC, Universiteit van Amsterdam, January 2000.

[WB93] M. Wilson and A. Borning. Hierarchical Constraint Logic Programming. *Journal of Logic Programming*, 16(3–4):227–318, July, August 1993.

[Wil96] R. Wilson. *Introduction to Graph Theory*. Longman, 1996.

Samenvatting

Dit proefschrift gaat over *efficient automatisch redeneren*. Het bestaat uit drie delen.

Deel I behandelt zogeheten *constraint propagation* algoritmes. Dit zijn efficiënte algoritmes voor het oplossen van *constraint satisfaction problems*. In Hoofdstuk 2 geven we een inleiding tot dergelijke algoritmes: constraints worden gekarakteriseerd als relaties op domeinen waarin variabelen hun waarden nemen. De doelstellingen in Deel I zijn van een theoretische aard: een algemene theorie waarin de voornoemde algoritmes geformuleerd en begrepen kunnen worden. Om precies te zijn, de theorie die we in Deel I ontwikkelen, beoogt nieuwe inzichten te verschaffen omtrent de volgende vragen:

1. Volgen *constraint propagation* algoritmes een gezamenlijk strategie?
2. Wat zijn de *verschillen* en *overeenkomsten* tussen die algoritmes?

Om de eerste vraag te beantwoorden wordt in Hoofdstuk 3 een algemene algoritme-schema `SGI` voorgesteld: binnen dit schema wordt, volgens een bepaalde strategie, een verzameling functies herhaald met als doel het vinden van een gezamenlijk dekpunt. Het algemene algoritme-schema, en een aantal varianten op dit schema, worden in Hoofdstuk 3 bestudeerd. De varianten zijn essentieel bij het beantwoorden van de tweede vraag die we hierboven noemden.

De theorie van Hoofdstuk 3 wordt in Hoofdstuk 4 toegepast om klassieke constraint propagation algoritmes door middel van *iteraties van functies* te *beschrijven* en *analyseren*. Soft constraints zijn expressiever dan klassieke constraints, en ze worden voornamelijk gebruikt om onzekerheid te modelleren. Desondanks is de theorie van Hoofdstuk 3 algemeen en expressief genoeg om ook soft constraint propagation algoritmes te beschrijven en analyseren.

Nadat we hebben laten zien dat constraint propagation algoritmes instanties zijn van ons algemen `SGI` schema, doet zich de volgende vraag voor:

- welke functies verwijderen inconsistenties in constraint propagation algoritmes?

Om deze vraag te beantwoorden geven we in Hoofdstuk 6 een karakterisering van functies voor zowel klassieke als soft constraint propagation algoritmes. Hiermee besluiten we Deel I.

In **Deel II** werken we opnieuw met relaties. Dit deel betreft de modale logica, waartoe we in Hoofdstuk 7 een inleiding verschaffen. In dit tweede deel van het proefschrift twee manieren voor om de volgende vraag te antwoorden:

- hoe kan de vervulbaarheid van modale formulas op een efficiënte manier bepaald worden?

In Hoofdstuk 8 beantwoorden we deze vraag door modale logica in eerste-orde logica te vertalen: we verfijnen de standaard vertaling van modale naar eerste-orde logica, en tonen aan hoe de nieuwe vertaling de het gedrag van op resolutiemethoden gebaseerde stellingbewijzers op modale formules verbetert.

Hoofdstuk 9 exploreert een interessant nieuw onderzoekthema: hoe efficiënt zijn constraint propagation en satisfaction algoritmes voor modale logica? In dit hoofdstuk reduceren we de vervulbaarheid van algemene modale logica tot vervulbaarheid van propositielogica, waarna we constraint algoritmes gebruiken het laatste probleem op te lossen, in plaats van meer gangbare Davis-Putnam algoritmes.

Deel III geeft een samenvatting van de inhoud van het proefschrift, en besluit met een discussie over een aantal open vragen.

Titles in the ILLC Dissertation Series:

ILLC DS-1999-01: **Jelle Gerbrandy**
Bisimulations on Planet Kripke

ILLC DS-1999-02: **Khalil Sima'an**
Learning efficient disambiguation

ILLC DS-1999-03: **Jaap Maat**
Philosophical Languages in the Seventeenth Century: Dalgarno, Wilkins, Leibniz

ILLC DS-1999-04: **Barbara Terhal**
Quantum Algorithms and Quantum Entanglement

ILLC DS-2000-01: **Renata Wassermann**
Resource Bounded Belief Revision

ILLC DS-2000-02: **Jaap Kamps**
A Logical Approach to Computational Theory Building (with applications to sociology)

ILLC DS-2000-03: **Marco Vervoort**
Games, Walks and Grammars: Problems I've Worked On

ILLC DS-2000-04: **Paul van Ulsen**
E.W. Beth als logicus

ILLC DS-2000-05: **Carlos Areces**
Logic Engineering. The Case of Description and Hybrid Logics

ILLC DS-2000-06: **Hans van Ditmarsch**
Knowledge Games

ILLC DS-2000-07: **Egbert L.J. Fortuin**
Polysemy or monosemy: Interpretation of the imperative and the dative-infinitive construction in Russian

ILLC DS-2001-01: **Maria Aloni**
Quantification under Conceptual Covers

ILLC DS-2001-02: **Alexander van den Bosch**
Rationality in Discovery - a study of Logic, Cognition, Computation and Neuropharmacology

ILLC DS-2001-03: **Erik de Haas**
Logics For OO Information Systems: a Semantic Study of Object Orientation from a Categorial Substructural Perspective

ILLC DS-2001-04: **Rosalie Iemhoff**
Provability Logic and Admissible Rules

ILLC DS-2001-05: **Eva Hoogland**
Definability and Interpolation: Model-theoretic investigations

ILLC DS-2001-06: **Ronald de Wolf**
Quantum Computing and Communication Complexity

ILLC DS-2001-07: **Katsumi Sasaki**
Logics and Provability

ILLC DS-2001-08: **Allard Tamminga**
Belief Dynamics. (Epistemo)logical Investigations

ILLC DS-2001-09: **Gwen Kerdiles**
Saying It with Pictures: a Logical Landscape of Conceptual Graphs

ILLC DS-2001-10: **Marc Pauly**
Logic for Social Software

ILLC DS-2002-01: **Nikos Massios**
Decision-Theoretic Robotic Surveillance

ILLC DS-2002-02: **Marco Aiello**
Spatial Reasoning: Theory and Practice

ILLC DS-2002-03: **Yuri Engelhardt**
The Language of Graphics

ILLC DS-2002-04: **Willem Klaas van Dam**
On Quantum Computation Theory