Constraint Propagation

Models, Techniques, Implementation

Guido Tack

Dissertation

zur Erlangung des Grades
des Doktors der Ingenieurwissenschaften
der Naturwissenschaftlich-Technischen Fakultäten
der Universität des Saarlandes

Saarbrücken, 2009

Dissertation zur Erlangung des Grades
des Doktors der Ingenieurwissenschaften
der Naturwissenschaftlich-Technischen Fakultäten
der Universität des Saarlandes,

eingereicht am 5. Dezember 2008 von
Dipl.-Inform. Guido Tack,
geboren am 6. Oktober 1978 in Lippstadt.

Berichterstatter:
Prof. Dr. Gert Smolka
Prof. Dr. Christian Schulte
Prof. Dr. Frédéric Benhamou

Dekan:
Prof. Dr. Joachim Weickert

Prüfungsausschuss:
Prof. Dr. Holger Hermanns
Prof. Dr. Gert Smolka
Prof. Dr. Christian Schulte
Dr. Alexander Koller

Tag des Kolloquiums:
29. Januar 2009

Textfassung vom 29. Januar 2009

Abstract

This dissertation presents the design of a propagation-based constraint solver. The design is based on models that span several levels of abstraction, ranging from a mathematical foundation, to a high-level implementation architecture, to concrete data structures and algorithms. This principled design approach results in a well-understood, correct, modular, and efficient implementation.

The core of the developed architecture is the *propagation kernel*. It provides the propagation infrastructure and is thus crucial for correctness and efficiency of the solver. Based on a mathematical model as well as a careful design of the employed algorithms and data structures, the presented architecture results in an efficient and domain-independent kernel. Constraints are realized by propagators, and implementing a propagator is a challenging, error-prone, and time-consuming task. A practically useful solver must however provide a comprehensive propagator library. This dissertation introduces two techniques for *automatically deriving* correct and efficient propagators. *Views* generalize variables and are used to derive propagators from existing propagators. For constraints over set variables, propagators are derived from formal *constraint specifications*.

The presented techniques are the basis of Gecode, a production-quality, highly efficient, and widely deployed constraint solver. Gecode is the empirical evidence for success and relevance of the principled design approach of this dissertation.

Kurzzusammenfassung

Diese Dissertation entwirft einen propagierungsbasierten Constraintlöser auf unterschiedlichen Abstraktionsebenen, von einem mathematischen Fundament über eine Implementierungsarchitektur bis hin zu konkreten Datenstrukturen und Algorithmen. Dieser Ansatz führt zu einer gut verstandenen, korrekten, modularen und effizienten Implementierung.

Das Kernstück der vorgestellten Architektur ist der *Propagierungskernel*. Er stellt die Infrastruktur für Constraintpropagierung zur Verfügung und ist daher von großer Bedeutung für Korrektheit und Effizienz. Diese Dissertation entwickelt die Architektur für einen effizienten, domänenunabhängigen Kernel, basierend auf einem mathematischen Modell und einem sorgfältigen Entwurf der verwendeten Algorithmen und Datenstrukturen. Constraints werden durch Propagierer realisiert, und das Implementieren eines Propagierers ist fehleranfällig und zeitaufwändig. Andererseits muss eine ausreichend große Propagiererbibliothek bereitstehen. Diese Dissertation untersucht zwei Techniken, um korrekte und effiziente Propagierer automatisch abzuleiten. *Views* verallgemeinern Variablen und dienen dazu, Propagierer von vorhandenen Propagierern abzuleiten. Für Mengenconstraints werden Propagierer von *formalen Spezifikationen* der Constraints abgeleitet.

Die vorgestellten Techniken bilden die Grundlage für Gecode, einen umfangreichen, hocheffizienten und weit verbreiteten Constraintlöser. Gecode liefert den empirischen Nachweis dafür, dass die in dieser Dissertation gewählte Vorgehensweise relevant und in der Praxis erfolgreich ist.

Acknowledgements

The research that I report on in this dissertation is the result of several years of work—but not only my own. I take this opportunity to express my deep gratitude towards my advisors, colleagues, friends, and family.

I consider myself extraordinarily fortunate, being able to work with my two advisors, Gert Smolka and Christian Schulte. Gert let me choose my topic freely, and then supported me in numberless discussions with a fresh, outside look on my research and his enthusiasm for the perfect, elegant formalism. Christian offered me to join the Gecode project, and I quickly found out that that offer included the best supervision one can hope for as a doctoral student. To him I owe nearly everything I know about software design and implementation, about writing papers and writing reviews, about organizing my work and my thoughts. Mikael Z. Lagerkvist was my fellow doctoral student on the Gecode project, and I very much enjoyed working with him, sharing thoughts and discussing ideas. Furthermore, I thank Frédéric Benhamou for giving his expert opinion on this dissertation, and Holger Hermanns and Alexander Koller for serving on my examination committee.

The Programming Systems Lab at Saarland University has been an inspiring and gratifying working environment for the past seven years. I especially want to thank Marco Kuhlmann, my long-term office-mate, for his great friendship and his trust in me. For interesting discussions, their willingness to share their knowledge, and help whenever needed, I thank all my current and previous colleagues: Gert, Marco, Chad E. Brown, Mark Kaminski, Mathias Möhl, Sandra Neumann, and Jan Schwinghammer; Christian, Ondřej Bojar, Thorsten Brunklaus, Ralph Debusmann, Denys Duchier, Leif Kornstaedt, Didier Le Botlan, Joachim Niehren, Tim Priesnitz, Andreas Rossberg, Lutz Straßburger, Gábor Szokoli, and Ann van de Veire. Thanks to our student assistants Christophe Boutter, Robert Künnemann, and Hannes von Haugwitz, who made system administration much less of a hassle.

Being employed by the university of course involved teaching duties. However, the kind of teaching I was allowed to do made this an opportunity rather than a burden. I thank all my students, and I want to mention in particular those who worked with me on Gecode, Niko Paltzer, Patrick Pekczynski, and Raphael Reischuk. I hope they learned as much from me as I learned from them.

For the last year, I have had the freedom to work exclusively on my dissertation, which was partly made possible by the support from the Saarbrücken Graduate School of Computer Science. Scientific work requires meeting people in person from time to time. Both my doctoral project as well as Gecode have benefited from the DAAD travel grant that allowed me to visit Christian and Mikael in Stockholm twice a year. Peter J. Stuckey invited me to visit the G12 project at Melbourne University

and NICTA for six weeks in 2007. I am grateful for having had this opportunity, and I thank the whole G12 crew for the great time. For further cooperations and discussions, I thank Martin Mann and Sebastian Will; Mats Carlsson, Pierre Flener, and Magnus Ågren; Yves Deville, Grégoire Dooms, and Stéphane Zampelli; as well as Claude-Guy Quimper, Andrea Rendl, and Peter Tiedemann.

My friends and my family have been a constant source of support. Thank you, Paul, Achim, and especially Monika. Finally, I want to dedicate this dissertation to my mother, Ulla, whom we all sorely miss.

Saarbrücken, January 2009 *Guido Tack*

My freedom will be so much the greater and more meaningful the more narrowly I limit my field of action and the more I surround myself with obstacles. Whatever diminishes constraint diminishes strength. The more constraints one imposes, the more one frees one's self of the chains that shackle the spirit.

Igor Stravinsky, Poetics of Music

Contents

1 Introduction

This dissertation presents the design of a propagation-based constraint solver. The design is based on models that span several levels of abstraction, ranging from a mathematical foundation, to a high-level implementation architecture, to concrete data structures and algorithms. This principled design approach results in a well-understood, correct, modular, and hence efficient implementation. The presented models and techniques are the basis of the Gecode constraint solving library.

This first chapter briefly explains the context, and then lays out the motivations and contributions of this dissertation.

1.1 Constraint Programming

Constraint programming is a powerful method for solving combinatorial (optimization) problems, which has proven effective and efficient in a wide range of application areas.

CSPs. A combinatorial problem is modeled as a set of variables, representing the objects the problem deals with, and a set of constraints, representing the relationships among the objects. Such a combinatorial problem is called a *Constraint Satisfaction Problem* (CSP). The common case where the variables can only take values from a finite universe is called a *finite domain* constraint satisfaction problem. A constraint programming system implements variables and constraints and provides a *solution procedure* for CSPs, which tries to find an assignment to the variables that satisfies all of the constraints. Clearly, solving CSPs is NP-hard in general, as the satisfiability of Boolean formulas (SAT) is one instance.

Application areas. Many hard, real-world combinatorial problems lend themselves to modeling as constraint satisfaction or optimization problems. The Handbook of Constraint Programming (Rossi et al., 2006) lists example applications in the areas of scheduling and planning, vehicle routing, configuration, networks (such as power or pipeline networks), and bioinformatics. Further application areas include computational linguistics (for example Duchier, 1999), as well as verification (Yuan et al., 2006) and optimization (van Beek and Wilken, 2001) of computer programs.

Constraint solvers. The success of constraint programming as a field is due to the availability of effective and efficient solution procedures that can solve these practical problems. This dissertation concentrates on *finite-domain constraint programming*, implemented in a *propagation-based* constraint solver, based on *exhaustive search*. This class of solvers has been successful because of its best-of-several-worlds approach. They combine classic AI search methods with advanced implementation techniques from the Programming Languages community and efficient algorithms from Operations Research. Furthermore, the Constraint Programming community has identified *global constraints* as an important tool to make the structure of constraint problems explicit and achieve strong propagation. Dedicated propagation algorithms for many different global constraints are available.

Propagation-based constraint solving. At the heart of a propagation-based constraint solver, *propagators* realize the constraints of a CSP by pruning the variable domains. A propagator removes values from variable domains that cannot be part of any solution of its constraint. Propagators for particular constraints are usually implemented as specialized algorithms. The constraint solver computes a fixed point of all propagators, maximizing the amount of inference they can contribute. It then splits the problem and solves the resulting smaller problems recursively.

Literature. A thorough discussion of the historical foundations and an overview of all aspects of constraint programming appears in the Handbook of Constraint Programming (Rossi et al., 2006). A more didactic approach to constraint programming is provided in the textbooks by Marriott and Stuckey (1998), Apt (2003), Dechter (2003), Frühwirth and Abdennadher (2003), as well as Apt and Wallace (2007).

1.2 The Thesis

The thesis of this dissertation is that principled models and a careful design enable the implementation of correct, well-understood, modular, comprehensive and efficient propagation-based constraint solvers. This section motivates the approach and summarizes the contributions of this dissertation.

Motivation and approach

Constraint propagation is the essential ingredient that makes constraint solvers feasible and efficient. Complex problems can only be modeled and solved with an *efficient infrastructure* for constraint propagation, as well as a *comprehensive* library of efficient propagator implementations.

To substantiate the thesis that efficient, comprehensive implementations follow from a principled design, this dissertation develops mathematical models, implementation architectures, and concrete algorithms and data structures for a propagation-based constraint solver. These different levels of abstraction are closely linked: The models are abstract enough for reasoning about important properties like correctness and strength of propagation. At the same time, they do not oversimplify, but capture the essence of the implementation.

A mathematical model of constraint propagation. This dissertation establishes a mathematical model of constraint propagation, based on a minimal definition of the central concept, the *propagator*. The model elegantly captures essential properties of propagators such as correctness, (non-)monotonicity, and strength of propagation. Furthermore, it explains techniques for efficiently computing the fixed points of a set of propagators.

A propagation kernel. Based on the mathematical model, this dissertation presents a principled design for a propagation-based constraint solver. The mathematical model explains the overall architecture, and carefully justified and evaluated design decisions lead to the concrete algorithms and data structures. From a software architecture viewpoint, a constraint solver should be *modular*. In a modular solver, a *propagation kernel* provides the *domain-independent* infrastructure for constraint propagation; on top of the kernel, *domain modules* realize the domain-specific parts, including data structures for variable domains and actual propagation algorithms. Both the model and the implementation architecture in this dissertation force a clean separation of domain-independent from domain-specific tasks.

Deriving propagators from existing propagators. Providing a comprehensive library of propagation algorithms is challenging, because designing and implementing them is a tedious, time-consuming, and error-prone task. Moreover, the task is often also *repetitive*, because many propagators follow a common algorithmic pattern but are subtly different (for example a simple sum and a weighted sum; Boolean conjunctions and disjunctions; set union and intersection). This dissertation therefore develops a technique for *reusing* existing propagators for variants of their original constraints, by systematically *deriving* propagators from existing propagators using *views*. Derived propagators inherit crucial properties like correctness and propagation strength from the original propagators, and can be implemented efficiently. The technique of deriving propagators using views is widely applicable.

Deriving propagators from constraint specifications. Certain classes of constraints expose a regular structure. For a class of constraints over set-valued variables, the underlying structure, Boolean algebra, is well understood. Taking advantage of this structure, this dissertation derives correct, strong propagators and propagation algorithms directly from formal specifications of these Boolean set constraints.

Gecode. All abstractions and techniques that this dissertation presents have been implemented in the Gecode (2009) constraint solver. Gecode is one of the fastest constraint solvers available and comes as a production-quality, widely deployed open-source C++ library. Gecode meets all of the claims: its high performance, its comprehensive library of propagation algorithms, and its modularity and clean architecture result directly from the principled models and powerful techniques presented here. Gecode is thus evidence of the viability and the success of the principled design approach.

Contributions

The central contributions of this dissertation can be summarized as follows.

1. This dissertation develops a solid mathematical foundation for a constraint solver. Particular contributions are the elegance and uniformity of the model, the thorough discussions of non-monotonicity and strength of propagation, and an implementation model of event-directed propagator scheduling.
2. Based on the mathematical model, this work presents a clean design for an efficient, modular propagation kernel that strictly separates the domain-specific from the domain-independent parts of the constraint solver. The design decisions for the central data structures are carefully justified and evaluated.
3. This dissertation introduces the novel technique of deriving propagators using views, discussed on both the mathematical and the implementation level. The technique is widely applicable and leads to more comprehensive libraries of efficient propagation algorithms.
4. For the first time, this dissertation presents a theory and implementation of propagators for Boolean set constraints. Efficient propagation algorithms are generated systematically from constraint specifications.
5. The models and techniques developed here are paramount for the efficiency, comprehensiveness, and modularity of the Gecode constraint solver.

Detailed summaries of the contributions of each of the two parts of this dissertation are given after the respective parts, on pages 101 and 167.

1.3 Overview

The next chapter recapitulates the fundamental ideas behind propagation-based constraint solving by means of examples. The technical content of this dissertation then follows in two parts. The first part presents the infrastructure for propagation, a propagation kernel. The second part develops techniques for deriving propagators from other propagators and from declarative constraint specifications.

The first part comprises Chapter 3 to Chapter 6.

A Model of Constraint Propagation. Chapter 3 lays the foundations for the remaining chapters with a mathematical model of constraint propagation. The model is the basis for a propagation kernel, and is later used to prove properties of views, derived propagators, and propagators for set constraints.

Propagation Strength. Chapter 4 characterizes propagators by their strength, that is, by how much a propagator prunes the search space. Propagation strength is defined with respect to domain approximations.

Efficient Propagator Scheduling. Chapter 5 extends the model from Chapter 3 towards a realistic propagation kernel, incorporating techniques for efficient scheduling of propagators.

Implementing a Propagation Kernel. Chapter 6 develops techniques for the implementation of an efficient, domain-independent, modular constraint propagation kernel, based on the models from the previous chapters. The presented techniques are implemented in the Gecode constraint solver, which is used to evaluate the approach empirically.

Chapter 7 to Chapter 11 make up the second part of the dissertation.

Views. Chapter 7 introduces views as powerful abstractions that are used to derive propagators from existing propagators. The chapter discusses properties of derived propagators like correctness and propagation strength. Chapter 8 presents generic techniques that can be used to derive propagators using views.

Implementing Views. Chapter 9 presents implementation strategies for views and derived propagators, and Chapter 10 introduces range iterators as an implementation strategy for set-valued variable and view operations. Both chapters evaluate different possible designs empirically, and present evidence for the wide applicability of views and derived propagators in practice.

Propagating Boolean Set Constraints. Chapter 11 derives propagators and the corresponding propagation algorithms for Boolean set constraints from declarative specifications of the constraints. The chapter presents implementation techniques for the propagation algorithms and evaluates the implementation empirically.

Chapter 12 concludes the dissertation with a discussion of the presented material and an outlook to future work. Figure 1.1 shows the dependencies among the main chapters. The material of the first part appears in the dashed box.

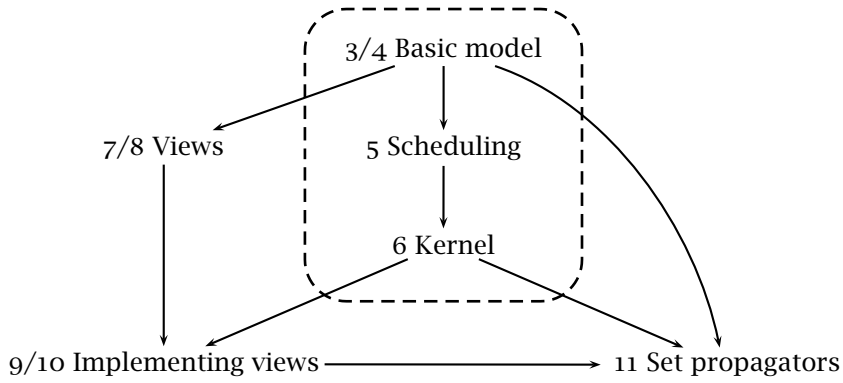


Figure 1.1: Organization of the dissertation

Source material

Parts of this dissertation are based on material that has already been published. These parts contain a unified and extended presentation of the following articles.

Views and iterators for generic constraint implementations.
In Mats Carlsson, François Fages, Brahim Hnich, and Francesca Rossi, editors, *Recent Advances in Constraints, 2005*, volume 3978 of LNCS, pages 118–132. Springer, 2006.
Joint work with Christian Schulte.

Generating propagators for finite set constraints.
In Frédéric Benhamou, editor, *Principles and Practice of Constraint Programming, 12th International Conference, CP 2006, Nantes, France, September 25-29, 2006, Proceedings*, volume 4204 of LNCS, pages 575–589. Springer, 2006.
Joint work with Christian Schulte and Gert Smolka.

Perfect derived propagators.
In Peter J. Stuckey, editor, *Principles and Practice of Constraint Programming, 14th International Conference, CP 2008, Sydney, Australia, September 14–18, 2008, Proceedings*, volume 5202 of LNCS, pages 571–575. Springer, 2008.
Joint work with Christian Schulte.

How to read this dissertation

The most important concepts in this dissertation are introduced in dedicated, numbered definitions. Other definitions appear in the main text, **highlighted like this**.

Throughout the dissertation, related work is usually discussed in separate sections. The interested reader may consult these sections for the technical and historical context, but they do not contain any material that is necessary for understanding the technical parts of this text.

2 Constraint Programming

This chapter contains a brief introduction to the field of constraint programming.

The term *constraint programming* should be understood in the tradition of *linear programming*, *integer programming*, or *dynamic programming*, in that it is not a general-purpose programming paradigm, but rather a technique for solving certain kinds of problems. Constraint programming refers to a methodology for solving combinatorial problems.

The success (and the beauty) of constraint programming is due to the clear separation between *model* and *solver:* The problem is stated declaratively, in terms of variables and constraints, modeling real-world objects and the relations the objects are engaged in. Then, this high-level model is passed to a constraint solver, which, given enough time, will return a solution to the problem.

This dissertation deals with constraint solvers that are based on constraint propagation and exhaustive search. In this chapter, we will explain these concepts by means of simple examples. Real-world constraint problems are solved using the same techniques.

Structure of the chapter. We start with modeling the well-known Sudoku puzzle as a constraint problem (2.1). Using the example of Sudoku, we then recapitulate the basic techniques of propagation-based constraint solving, propagation and search (2.2). Finally, we have a brief look at set constraints (2.3).

2.1 Modeling Constraint Problems: Sudoku

The first step in solving a combinatorial problem using constraint programming is to model it in terms of variables and constraints. In the following, we develop a model for solving Sudoku puzzles, which are probably the most well-known combinatorial problems today.

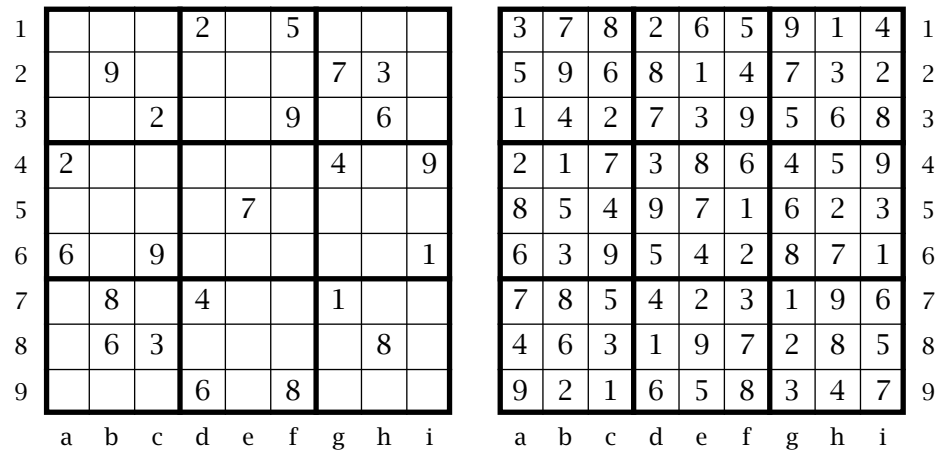


Figure 2.1: A Sudoku puzzle (left) and its solution (right)

Sudoku puzzles

A puzzle consists of a 9×9 matrix, which is to be filled with numbers from 1 to 9 in such a way that each row, each column, and each of the nine 3×3 blocks contains exactly the numbers from 1 to 9. The given matrix is partially filled so that there is a unique solution. Figure 2.1 shows a typical Sudoku puzzle on the left, and its solution on the right.

The history of the Sudoku puzzle cannot be traced completely. It is a refinement of the magic square problem (Euler, 1849), with the additional constraints on the nine 3×3 blocks. Sudokus first appeared in the May 1979 issue of *Dell Pencil Puzzles & Word Games* under the name "Number Place", probably designed by Howard Garns. The puzzle was made popular in 1984 by the Japanese company *Nikoli*, which also coined the name Sudoku, meaning "single number" in Japanese.

Modeling Sudoku

The straightforward model of Sudoku as a constraint problem identifies each of the 81 fields with a *variable*. Each variable can take a number from the set $\{1, \ldots, 9\}$ as its value. This initial set of values for each variable is called its *variable domain*. The model has 27 *constraints*, each stating that the variables that represent one particular row, column, or block, must take exactly the numbers from 1 to 9. Each constraint is an instance of the well-known *all-different* constraint, which states that a set of variables $x_1, \ldots, x_n$ must take pairwise different values.

2.2 Constraint Propagation and Search

Now that we have a model of Sudoku as a constraint satisfaction problem, we can pass it to a constraint solver to obtain a solution. This dissertation deals with a particular class of constraint solvers, based on *constraint propagation* and *exhaustive search*, which we will now introduce informally using the Sudoku example.

Constraint propagation

The unwritten law of Sudoku is that you play with a pen, not a pencil. This means that you are not supposed to guess and later backtrack, but to solve the puzzle by *inference* alone. Let us look at the top right block in Figure 2.1. The 1 is still missing from this block, but both column g and i already contain a 1, so it must be in column h. As there is only one empty field left, we can infer that the 1 must be at position h1. Now we can make further inferences from the fact that h1 is 1. Both the second and the third row already contain a 9, as well as column i. There is thus only one field left for the 9 in the top right block, which is field g1.

This process of inference is called *constraint propagation*. As the main inference method in constraint programming systems, constraint propagation infers that certain values cannot be part of certain variable domains any more because they violate some constraint. In the above example, we inferred that all values except the 1 cannot be part of the domain of the variable representing the field h1. The entities that perform constraint propagation are called *propagators*.

Two things are crucial for successfully solving a hard combinatorial problem with a propagation-based constraint solver: (1) a model that makes the structure of the problem explicit, stating it in terms of high-level constraints such as *all-different*; and (2) a solver that provides efficient implementations of a sufficient number of these high-level constraints as propagators. This dissertation develops an architecture for efficient constraint propagation, as well as techniques for implementing a comprehensive number of propagators.

Search

While Sudokus are usually designed such that they can be solved by propagation alone, this is not true for constraint problems in general. Constraint propagation alone is incomplete. A propagation-based constraint solver interleaves constraint propagation with search, thus providing a complete solution procedure. First, a mutual fixed point of all propagators is computed, making as many inferences as possible. Then, the problem is split into smaller problems, which are solved recursively. The recursion is implemented as a backtracking search procedure, exploring a tree of subproblems of the original problem.

	Group 1	*Group 2*	*Group 3*	*Group 4*	*Group 5*
Week 1	1, 2, 3	4, 5, 6	7, 8, 9	10, 11, 12	13, 14, 15
Week 2	1, 4, 7	2, 5, 10	3, 8, 13	6, 11, 14	9, 12, 15
Week 3	1, 5, 14	2, 4, 12	3, 7, 11	6, 9, 13	8, 10, 15
Week 4	1, 6, 15	2, 11, 13	3, 4, 9	5, 8, 12	7, 10, 14
Week 5	1, 8, 11	2, 9, 14	3, 5, 15	4, 10, 13	6, 7, 12
Week 6	1, 9, 10	2, 6, 8	3, 12, 14	4, 11, 15	5, 7, 13
Week 7	1, 12, 13	2, 7, 15	3, 6, 10	4, 8, 14	5, 9, 11

Figure 2.2: Scheduling 15 golfers in groups of three over seven weeks

2.3 Set Constraints

This section takes a brief look at constraints over set-valued variables. Many of the later chapters discuss propagation for this kind of constraints, so we give a short introduction here.

Modeling with sets

Many constraint problems can be formulated naturally in terms of *sets* of objects instead of just individual objects. A good example is assigning people to groups and then stating constraints on the groups, such as size limits or that people can only be in one group at a time. For this purpose, many constraint solvers provide variables that range over sets of objects, and corresponding set constraints.

Here is a classic problem that can be expressed elegantly with set constraints. We use this example to introduce the most important constraints over set variables.

Example 2.1 (The Social Golfer Problem) The task is to schedule $g \times s$ golfers in g groups per week, each group of size s, over a period of w weeks, such that no two golfers play against each other in a group more than once. *

A solution for the particular instance[1] $g = 5$, $s = 3$, $w = 7$ appears in Figure 2.2. Every golfer is represented by a number between 1 and $g \times s = 15$.

The social golfer problem has a natural model in terms of set variables and constraints. For each group i in each week j, there is a variable $x_{i,j}$. The initial domain of each such variable is the set of all s-element subsets of the golfers:

$$\forall 1 \leq i \leq g, 1 \leq j \leq w : \; x_{i,j} \subseteq \{1, \ldots, g \times s\} \wedge |x_{i,j}| = s$$

[1] This instance is known as *Kirkman's schoolgirl problem*, and according to Graham et al. (1995) was stated in 1850 by Thomas Kirkman in a magazine called *The Lady's and Gentleman's Diary*.

Let us now turn to the constraints linking the groups. Each week, the groups form a partition of the full set. This ensures that each golfer is assigned to exactly one group in each week:

$$\forall 1 \leq j \leq w : x_{1,j} \uplus \cdots \uplus x_{g,j} = \{1, \ldots, g \times s\}$$

The second requirement is that no two golfers play together more than once. Thus, for each group $x_{i,j}$, the intersection with any other group must not have more than one element. For this constraint, we introduce additional set variables for the intersections of each pair of groups, and constrain their cardinalities to be at most 1.

$$\forall i, i', j, j' : i' \neq i \vee j' \neq j \implies y_{i,j,i',j'} = x_{i,j} \cap x_{i',j'} \wedge |y_{i,j,i',j'}| \leq 1$$

Certain instances of the social golfer problem turn out to be surprisingly hard for constraint solvers. For example, we are not aware of any constraint solver that can produce a solution for the instance $g = 8$, $s = 4$, $w = 10$.

Set constraints

The social golfer problem exhibits the typical constraints that are used when modeling with set variables:

- The initial domains of the variables are often the subsets of a specific set of possible values (here: the set of all golfers).
- The cardinalities of the sets in the initial domains are often restricted (here: groups have size s).
- Constraints between set variables involve the usual set operations like intersection, union, partition, or complement, and typical relations such as equality, subset, or disjointness. We call these constraints *Boolean* set constraints, as the operations are taken from the Boolean algebra of sets.
- Constraints often involve set constants such as the empty set, or the set of all elements (here: the set of all golfers).

Additionally, solvers typically provide constraints that link integer and set variables, such as $x \in y$ or $x = |y|$ for an integer variable x and a set variable y.

Symmetries

Besides offering expressivity that leads to more natural models for some problems, set constraints have another advantage, as they avoid introducing *symmetry* into a model. For example, we could have modeled the social golfer problem using s integer variables per group instead of one set variable. This however would have

introduced symmetry, as any permutation of the integer variables within a group would still be a valid solution.

Symmetries can increase the size of the search space dramatically. The search does not only have to enumerate all the symmetric solutions, but also the *symmetric non-solutions* that are not pruned by propagation. An integer model would have to use special additional techniques for avoiding this symmetry in order to be solvable efficiently. Avoiding symmetry is an active area of research. For a detailed overview, see the chapter in the Handbook of Constraint Programming by Gent et al. (2006c).

Approximating set variable domains

Many constraint solvers do not represent the domain of a set variable completely, as a complete domain representation is in general exponential in size. The alternative is to *approximate* the variable domain as an interval described by a lower and an upper bound $[l, u]$. The lower bound contains all the elements that are *known* to be in the set, while the upper bound contains the elements that are *possibly* members of the set. Only this approximation technique, developed by Puget (1992) and Gervet (1994, 1995), has made set constraints successful in practice. We will cover this technique in detail in Section 4.5. Chapter 11 of this dissertation is concerned with propagation algorithms for Boolean set constraints based on the interval approximation.

In its simplest form, the set interval approximation cannot represent cardinality information: the set variable domain $\{\{1\}, \{2\}\}$ can only be approximated by the interval $[\emptyset, \{1, 2\}]$, losing the information that all possible sets are singletons. Cardinality must hence be realized by propagation. Assume that we want to propagate the cardinality constraint $m \leq |x| \leq n$. Then, if the lower bound of x already contains n elements, we can assign x to its lower bound. Dually, if only m elements are left in the upper bound of x, we can assign x to its upper bound.

More elaborate techniques for cardinality reasoning have been developed, and will be discussed in the section on related work at the end of Chapter 11.

Part I

A Propagation Kernel

3 A Model of Constraint Propagation

This chapter introduces a mathematical model of constraint propagation. The model serves as the basis for the theoretical results in this dissertation, and at the same time justifies the implementation architectures we develop.

We present a framework that captures a denotational model of constraint problems, expressing constraints in extension and defining constraint satisfaction problems (CSPs), the problems we aim to solve. Within the same framework, we develop an operational model, realizing constraints using propagators, and yielding the operational equivalent to CSPs, called propagation problems.

The framework we present is inspired by many sources, as discussed in several paragraphs on related work in this chapter. One of the contributions of this chapter is to present the state of the art in a concise and uniform way. The further contributions, such as the notion of induced constraints, the discussion of non-monotonic and idempotent propagators, and the modular treatment of different variable sorts, will be pointed out in the text.

Structure of the chapter. We start with the denotational model that expresses constraint problems in terms of assignments, constraints, and domains (3.1). The operational model defines propagators as refinements of constraints (3.2). Next, we show that constraint propagation can be expressed as a transition system (3.3). We discuss idempotency and monotonicity of propagators (3.4), and finally show how the framework can be extended to support different sorts of variables (3.5).

3.1 A Denotational Model of Constraint Problems

The aim of this section is to clearly define *what* we want to solve: Constraint satisfaction problems.

Constraint satisfaction problems are modeled with respect to a **finite set of variables** X and a **finite set of values** V. We typically write variables as $x, y, z \in X$, and refer to values as $v, w \in V$.

Assignments and constraints

A solution of a constraint satisfaction problem must assign a single value to each variable. A constraint restricts which assignments of values to variables are allowed. The following definition captures assignments and constraints.

Definition 3.1 An **assignment** a is a function mapping variables to values. The set of all assignments is $\mathsf{Asn} := X \to V$. A **constraint** c is a set of assignments, $c \in \mathsf{Con} := \mathscr{P}(\mathsf{Asn}) = \mathscr{P}(X \to V)$ (we write $\mathscr{P}(S)$ for the power set of S). It corresponds to a relation over the variables in X. Any assignment $a \in c$ is a **solution** of c. *

We base constraints on full assignments, defined for all variables in X. However, for typical constraints, only a subset $\mathrm{vars}(c)$ of the variables is *significant*; the constraint is the full relation for all $x \notin \mathrm{vars}(c)$. More formally, a constraint c is the **full relation** for a variable x if and only if $\forall v \in V\ \forall a \in c : a[v/x] \in c$, where $a[v/x]$ is the assignment a' where $a'(x) = v$ and $a'(y) = a(y)$ for all variables $y \neq x$. Consequently, the **significant variables** of c are defined as

$$\mathrm{vars}(c) := \{x \in X \mid \exists v \in V\ \exists a \in c : a[v/x] \notin c\}$$

Constraints are either written as sets of assignments, or just stated as mathematical expressions with the usual meaning. We use the notation $[\![\cdot]\!]$ when we want to stress that we mean the constraint; for example, we write $[\![x < y]\!]$ to denote the constraint $\{a \in \mathsf{Asn} \mid a(x) < a(y)\}$.

Example 3.2 (A sum constraint) Let $X = \{x, y, z\}$ be the set of variables, and let $V = \{1, 2, 3, 4\}$ be the set of values. Then the constraint $[\![x = y + z]\!]$ corresponds to the following set of assignments:

$$\begin{aligned}[\![x = y + z]\!] = \{&(x \mapsto 2, y \mapsto 1, z \mapsto 1),\\ &(x \mapsto 3, y \mapsto 1, z \mapsto 2),\\ &(x \mapsto 3, y \mapsto 2, z \mapsto 1),\\ &(x \mapsto 4, y \mapsto 2, z \mapsto 2)\}\end{aligned}$$ *

Domains and constraint satisfaction problems

Constraints constitute one of the two crucial ingredients of constraint satisfaction problems. The other part is the initial set of values that each variable can take. For example in a Sudoku (as introduced in Section 2.1), each variable must take a value from the set $\{1, \ldots, 9\}$. A mapping from variables to sets of possible values is a *domain*.

Definition 3.3 A **domain** d is a function mapping variables to sets of values, such that $d(x) \subseteq V$. The set of all domains is $\mathsf{Dom} := X \to \mathscr{P}(V)$. The set of values in d for a particular variable x, $d(x)$, is called the **variable domain** of x. A domain d represents a set of assignments, a constraint, defined as

$$\mathrm{con}(d) := \{a \in \mathsf{Asn} \mid \forall x \in X : a(x) \in d(x)\}$$

We say that an assignment $a \in \mathrm{con}(d)$ is **licensed** by d. *

Now we have all the definitions in place to introduce the denotational model of a constraint problem. It consists of a domain that restricts the initial values that the variables can take, and a set of constraints that express the relations over the variables.

Definition 3.4 A **constraint satisfaction problem** (CSP) is a pair $\langle d, C\rangle$ of a domain d and a set of constraints C. The constraints C are interpreted as a conjunction of all $c \in C$ and are thus equivalent to the constraint $\{a \in \mathsf{Asn} \mid \forall c \in C : a \in c\}$. The **solutions** of a CSP $\langle d, C\rangle$ are the assignments licensed by d that satisfy all constraints in C, defined as $\mathrm{sol}(\langle d, C\rangle) := \{a \in \mathrm{con}(d) \mid \forall c \in C : a \in c\}$. *

Example 3.5 (Sudoku as a CSP) The Sudoku model from Section 2.1 can be cast into a CSP as follows. We have 81 variables $\{x_1, \ldots, x_{81}\}$, and the initial domain maps each variable to either the set $\{1, \ldots, 9\}$, or the singleton set $\{i\}$ for fields which are pre-filled with the number i.

We have 27 constraints. Each constraint expresses the fact that a certain row, column, or block takes exactly the numbers from 1 to 9. For example, assume that the first row is represented by $x_1 \ldots x_9$, then the constraint for the first row is

$$c_{r1} = \left\{ a \in \mathsf{Asn} \;\middle|\; \bigcup_{i=1}^{9} a(x_i) = \{1, \ldots, 9\} \right\}$$

This constraint consists of $9! \times 9^{72}$ assignments (9! for the permutations of $1 \ldots 9$ in variables $x_1 \ldots x_9$, the remaining factor of 9^{72} for the unconstrained variables $x_{10} \ldots x_{81}$). In practice, it is clearly infeasible to represent a constraint in extension like this. Even if we restricted ourselves to the significant variables, storing (and computing with) 9! different assignments for each of the 27 constraints would not lead to an efficient solution procedure. *

More on domains

Domains are not only used to specify the initial sets of possible values in a CSP. They are also a vital part of the operational model introduced in the next section. We therefore now identify some of the properties of domains and establish useful notation.

A domain d represents the set of assignments $\mathrm{con}(d)$. However, domains cannot represent arbitrary sets of assignments. They are restricted to a *Cartesian* representation, expressing exactly conjunctions of unary constraints (those that have a single significant variable). The following inequation describes the relation between domains and constraints:

$$\{\mathrm{con}(d) \mid d \in \mathsf{Dom}\} \subset \mathsf{Con}$$

In this sense, any domain is also a constraint. We therefore take the liberty to omit writing the conversion and simply use domains as constraints, resulting in a more uniform presentation. In particular, we will write $a \in d$ (instead of $a \in \mathrm{con}(d)$) for an assignment a that is licensed by d, and we will write $c \cap d$ (instead of $c \cap \mathrm{con}(d)$) for the intersection of a constraint with a domain.

A domain d that maps some variable to the empty set of values is called **failed**, and we write $d = \emptyset$, as it represents no valid assignments ($\mathrm{con}(d) = \emptyset$). We carefully designed all definitions in this dissertation to not distinguish between different failed domains, so we will sometimes identify all failed domains and talk about *the* failed domain $\emptyset$. A domain d that represents a single assignment, $\mathrm{con}(d) = \{a\}$, is called **assigned**. We will sometimes write assigned domains as $\{a\}$.

We define a partial order on domains as the point-wise lifting of the subset order: $d \subseteq d' :\Leftrightarrow \forall x \in X : d(x) \subseteq d'(x)$. We say that d is **stronger** than d', and that d' is **weaker** than d. A domain d is **strictly stronger** than a domain d' (written $d \subset d'$) if and only if d is stronger than d', d' is not failed, and $d(x) \subset d'(x)$ for some variable x. We will see in the next section that the goal of constraint propagation is to prune values from variable domains, thus inferring stronger domains, without removing solutions of the constraints.

3.2 An Operational Model of Constraint Propagation

As we have just seen in Example 3.5, representing (let alone solving) constraint satisfaction problems directly on the extensional representation is infeasible. In practice, constraint solvers employ *propagators* to realize constraints. This section develops a mathematical model of constraint propagation.

Propagators

The basis of a propagation-based constraint solver is a *search procedure*, which systematically enumerates the assignments licensed by the domain d of a CSP $\langle d, C\rangle$. For each assignment, the solver uses a *decision procedure* for each constraint to

determine whether the assignment is a solution of the CSP. Enumerating *all* assignments would be infeasible in practice, so in addition to the decision procedure, the solver employs a *pruning procedure* for each constraint, which may rule out assignments that are not solutions of the constraint.

These two tasks, the decision and the pruning procedure for a constraint, are realized by *propagators*. Each propagator *induces* a particular constraint. A propagator decides for a given assignment whether it satisfies the induced constraint, and it may prune those assignments from a domain that do not satisfy the constraint. Interleaving propagation and search yields a sound and complete solution procedure for the CSP. It is complete, because only assignments that are not solutions are pruned by the propagators, and all remaining assignments are enumerated. It is sound, because for each of the enumerated assignments, the propagators decide whether it is a solution.

The formal definition of propagators we develop below captures the *minimal* properties that are required in order to get a sound and complete solver. In this way, our model differs from the definitions usually found in the literature, which will be discussed later. Furthermore, to our knowledge the characterization of propagators by unique induced constraints is novel.

We define propagators in terms of domains. A propagator is a function p that takes a domain as its argument and returns a stronger domain, it may only *prune* assignments. If the original domain was an assigned domain $\{a\}$, the propagator either accepts it ($p(\{a\}) = \{a\}$) or rejects it ($p(\{a\}) = \emptyset$), realizing the decision procedure for its constraint. In fact, each propagator induces a unique constraint, the set of assignments that it accepts. To make this setup work, we need one additional restriction. The decision procedure and the pruning procedure must be *consistent:* if the decision procedure accepts an assignment, the pruning procedure must never remove this assignment from any domain—this property is called *soundness*. We compress all these considerations into the following central definition:

Definition 3.6 A **propagator** is a function $p \in \mathsf{Dom} \to \mathsf{Dom}$ that is

- **contracting:** $p(d) \subseteq d$ for any domain d
- **sound:** for any domain $d \in \mathsf{Dom}$ and any assignment $a \in \mathsf{Asn}$, if $\{a\} \subseteq d$, then $p(\{a\}) \subseteq p(d)$

The set of all propagators is Prop. If a propagator p returns a *strictly* stronger domain ($p(d) \subset d$), we say that p **prunes the domain** d. The propagator p **induces** the constraint c_p defined by the set of assignments accepted by p:

$$c_p := \{a \in \mathsf{Asn} \mid p(\{a\}) = \{a\}\} \qquad *$$

Soundness expresses exactly that the decision and the pruning procedure realized by a propagator are consistent. A direct consequence is that a propagator never removes assignments that satisfy its induced constraint.

Proposition 3.7 Let p be a propagator and d a domain. Then $c_p \cap d = c_p \cap p(d)$. $*$

Proof. We prove the two subset relations.

$\subseteq$ Let a be an assignment. By soundness of p, $a \in c_p \cap d$ implies that $a \in p(d)$. So $c_p \cap d \subseteq c_p \cap p(d)$.

$\supseteq$ As p is contracting, $p(d) \subseteq d$, and thus $c_p \cap p(d) \subseteq c_p \cap d$. ∎

Soundness is a weak form of monotonicity. Traditionally, propagators are often defined to satisfy two more properties, idempotency and (strong) monotonicity. However, these stronger properties are not required for the constraint solver to be sound or complete. We will discuss idempotency and monotonicity in detail in Section 3.4.

Propagation problems

We have defined propagators as a refinement of constraints—each propagator induces one particular constraint, but in addition has an operational meaning, its pruning procedure. We can now define the operational equivalent of a CSP, a *propagation problem*. Propagation problems realize all constraints of a CSP using propagators. All techniques we develop in this dissertation aim at implementing and efficiently solving propagation problems.

Definition 3.8 A **propagation problem** (PP) is a pair $\langle d, P\rangle$ of a domain d and a set of propagators P. The **induced constraint satisfaction problem** of a propagation problem $\langle d, P\rangle$ is the CSP $\langle d, \{c_p \mid p \in P\}\rangle$. The **solutions** of a PP $\langle d, P\rangle$ are the solutions of the induced CSP, $\mathrm{sol}(\langle d, P\rangle) := \mathrm{sol}(\langle d, \{c_p \mid p \in P\}\rangle)$. $*$

The set of solutions of a PP $\langle d, P\rangle$ can be defined equivalently as $\mathrm{sol}(\langle d, P\rangle) := \{a \in \mathsf{Asn} \mid \forall p \in P : p(\{a\}) = \{a\}\}$, just applying the definitions of induced constraints and solutions of CSPs.

Existence of strongest and weakest propagators

Propagators combine a decision procedure with a pruning procedure. While the decision procedure determines the constraint a propagator induces, there is some liberty in the definition of the pruning, as long as it is sound. Thus, there are different propagators for the same constraint, and they can be arranged in a partial order according to their *strength:*

Definition 3.9 Let p_1 and p_2 be two propagators that induce the same constraint. Then p_1 is **stronger** than p_2 (written $p_1 \sqsubseteq p_2$) if and only if for all domains d, $p_1(d) \subseteq p_2(d)$. *

Propagation strength has important practical implications. A stronger propagator performs more pruning, and therefore leaves a smaller search space to explore. Solving hard problems typically requires strong propagation. On the other hand, stronger propagators are often algorithmically more complex than weaker versions for the same constraint. It is hence important to strike the right balance between pruning power and run-time complexity of a propagation algorithm.

The remainder of this section develops the basics of propagation strength, showing that for each constraint c, there are unique strongest and weakest propagators that induce c. The next chapter develops a detailed characterization of propagation strength, helping us to identify classes of propagators that lie in between the weakest and the strongest propagators for their induced constraint.

A first observation that leads to a more precise characterization of propagation strength is that the set of propagators is closed under functional composition, point-wise union, and point-wise intersection. We write $p_1 \cap p_2$ for the point-wise lifting of the intersection on domains, $\lambda d.p_1(d) \cap p_2(d)$, and similarly for union.

Proposition 3.10 Let p_1 and p_2 be propagators. Then $p_1 \circ p_2$ and $p_1 \cap p_2$ are also propagators, both inducing the constraint $c_{p_1} \cap c_{p_2}$. Furthermore, $p_1 \cup p_2$ is a propagator, inducing the constraint $c_{p_1} \cup c_{p_2}$. *

Proof. All operations clearly preserve contraction:

- $p_2(d) \subseteq d$, so $p_1(p_2(d)) \subseteq d$
- $p_1(d) \subseteq d$ and $p_2(d) \subseteq d$, so $p_1(d) \cap p_2(d) \subseteq d$ and $p_1(d) \cup p_2(d) \subseteq d$

Soundness is also preserved, as the combined propagators prune at most the assignments that the individual propagators prune.

The composition $p_1 \circ p_2$ and the intersection $p_1 \cap p_2$ induce the constraint $c_{p_1} \cap c_{p_2}$, because if either propagator rejects an assignment, the combined propagator rejects it, too. The union $p_1 \cup p_2$ induces $c_{p_1} \cup c_{p_2}$, because if at least one of the two propagators accepts an assignment, the combined propagator accepts it, too. ∎

As the partial order on propagators is just lifted point-wise from the partial order on domains, which is lifted point-wise from the partial order on the subset lattice of values, the set of propagators Prop forms a complete lattice. The lattice is finite, since the basic sets X and V are finite. If two propagators p_1 and p_2 induce the same constraint c, then clearly both $p_1 \cap p_2$ and $p_1 \cup p_2$ also induce c. But $p_1 \cap p_2$ is stronger than both p_1 and p_2, and $p_1 \cup p_2$ is weaker than both. It thus turns out that $p_1 \cap p_2$ is exactly the weakest propagator that is stronger than both p_1 and p_2,

and therefore the meet in the propagator lattice. Conversely, $p_1 \cup p_2$ is the join, the strongest propagator weaker than both p_1 and p_2.

Complete lattices have unique suprema and infima, so there is a unique weakest and a unique strongest propagator for each constraint c. We call the strongest propagator $p_c^{\max}$, and the weakest propagator $p_c^{\min}$, accordingly.

The strongest propagator $p_c^{\max}$ for a constraint c is an important concept, as it represents the maximal pruning that we can achieve for c. We call a propagator p that is the strongest propagator for its induced constraint ($p = p_{c_p}^{\max}$) **domain-complete**.

Domain completeness is not a compositional property. Let p_1 and p_2 be domain-complete propagators inducing different constraints. Recall that the induced constraint of $p_1 \circ p_2$ is $c_{p_1} \cap c_{p_2}$. However, the composition $p_1 \circ p_2$, even when iterated to idempotency, is in general not domain-complete for $c_{p_1} \cap c_{p_2}$.

Propagation algorithms

In order to implement a constraint solver, propagators have to be realized algorithmically. Just as there are several propagators that induce a single constraint, there can be several propagation algorithms for a single propagator, and, of course, different implementations of the same propagation algorithm. On these levels of abstraction, different properties become important. We can talk about soundness and completeness on all levels, for example stating that a propagation *algorithm* is domain-complete if the propagator it realizes is. On the algorithmic level, asymptotic run-time and memory complexity become important, which also carry over to the implementation. The implementation finally fixes the concrete data structures as well as the implementation language that is used, and thus determines the actual performance in practice.

Related work

► A whole line of research is concerned with solving a constraint satisfaction problem directly, where constraints are given in extension as tables of allowed (or sometimes forbidden) tuples of values. This *CSP approach* was pioneered by Montanari (1974), Mackworth (1977), Freuder (1982) and Dechter and Pearl (1987). While this approach provides insight into the fundamental structure of CSPs, it is no efficient solution procedure for hard, real-world combinatorial problems.

► The *propagation approach* we follow here, as opposed to the CSP approach to solving constraint problems, has evolved from constraint logic programming, pioneered by the CHIP system (Dincbas et al., 1988; Van Hentenryck, 1989).

► The term *propagator* was coined in the context of the Oz programming model (Smolka, 1995). Other terms for the same concept are filter functions, narrowing operators (Benhamou et al., 1994), or reactive constraints (Maher, 2002).

► The definitions of propagators by Saraswat et al. (1991) are close to ours, they require propagators to be contracting, monotonic, and idempotent functions. In their work, propagators describe the semantics of concurrent processes that implement constraints.

3.3 Propagation as a Transition System

The previous section introduced propagation problems, which are the operational equivalents of constraint satisfaction problems. A propagation problem is operational in the sense that its propagators induce constraints by operating on domains, successively strengthening the propagation problem.

A propagation-based solver interleaves constraint propagation and search, where constraint propagation means to prune the domain as much as possible using propagators, before search resorts to enumerating the assignments in the domain. Propagating as much as possible means, in the context of propagation problems, to compute a *mutual fixed point* of all propagators. This section presents transition systems whose terminal states are the desired fixed points.

Transitions

Let $\langle d, P\rangle$ be a propagation problem. If there is a propagator $p \in P$ that can prune the domain d, that is, if $p(d) \subset d$, then applying p yields a new, simpler propagation problem, $\langle p(d), P\rangle$. Soundness of p makes sure that the new problem has the same set of solutions as the original problem, $\mathrm{sol}(\langle d, P\rangle) = \mathrm{sol}(\langle p(d), P\rangle)$.

A propagation problem thus induces a transition system, where a transition is possible from a domain d to a domain $d' \subset d$ if there is a propagator $p \in P$ such that $p(d) = d'$. We write such a transition $d \vdash p\rightarrow d'$. Figure 3.1 shows how the transitions from a given initial domain d may look like.

Definition 3.11 Let d be a domain. A **transition** $d \vdash p\rightarrow d'$ with a propagator p to a domain d' is possible if and only if $d' = p(d)$ and $d' \subset d$. The **transition system** of a propagation problem $\langle d, P\rangle$ consists of all the transitions that are possible with propagators $p \in P$, starting from d. A terminal domain, that is, a domain d such that there is no transition $d \vdash p\rightarrow p(d)$ for any propagator $p \in P$, is called **stable**. We write $d \Rightarrow d'$ if there is a sequence of transitions that transforms d into a stable domain d'. This sequence is empty, $d \Rightarrow d$, if d is stable. *

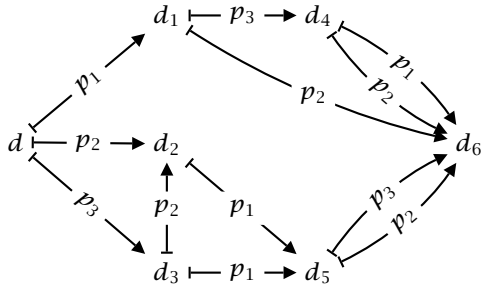


Figure 3.1: The transition system of a propagation problem

Note that a domain d' is defined to be strictly stronger than a domain d only if d is not failed. Thus, any failed domain $d = \emptyset$ is stable. We can lift the transitions to propagation problems, such that $\langle d, P\rangle \vdash p\rightarrow \langle d', P\rangle$ if and only if $d \vdash p\rightarrow d'$ for some $p \in P$. A propagation problem $\langle d, P\rangle$ is called stable if and only if its domain d is stable.

Example 3.12 (Transitions) Let d be a domain such that $d(x) = d(y) = d(z) = \{1, 2, 3, 4\}$, and assume three domain-complete propagators such that $c_{p_1} = [\![x < y]\!]$, $c_{p_2} = [\![x + y = z]\!]$, and $c_{p_3} = [\![y < z]\!]$. Then Figure 3.1 shows the transitions that are possible for the propagation problem $\langle d, \{p_1, p_2, p_3\}\rangle$. The transition system has a unique stable domain d_6. The values of the domains are

$d_1(x) = \{1, 2, 3\}$	$d_1(y) = \{2, 3, 4\}$	$d_1(z) = \{1, 2, 3, 4\}$
$d_2(x) = \{1, 2, 3\}$	$d_2(y) = \{1, 2, 3\}$	$d_2(z) = \{2, 3, 4\}$
$d_3(x) = \{1, 2, 3, 4\}$	$d_3(y) = \{1, 2, 3\}$	$d_3(z) = \{2, 3, 4\}$
$d_4(x) = \{1, 2, 3\}$	$d_4(y) = \{2, 3\}$	$d_4(z) = \{3, 4\}$
$d_5(x) = \{1, 2\}$	$d_5(y) = \{2, 3\}$	$d_5(z) = \{2, 3, 4\}$
$d_6(x) = \{1, 2\}$	$d_6(y) = \{2, 3\}$	$d_6(z) = \{3, 4\}$

Section 3.4 will show that not all propagation problems have transition systems with unique stable domains, and argue that monotonicity of propagators guarantees confluence of the transitions. Furthermore, we will prove in Section 4.1 that domain-complete propagators are monotonic. So the situation in Figure 3.1 is not coincidental, but due to the properties of p_1, p_2, and p_3. *

The transition system of a propagation problem is non-deterministic, as there are many possible chains of propagation that result in a stable domain. Chapter 5 explains how an implementation of a constraint propagation engine determines an efficient order of propagator invocation.

Fixed points

The important theorem that ensures that constraint propagation is useful in practice is that, given a propagation problem $\langle d, P\rangle$, its transition system is finite and terminating. No matter in what order the propagators are applied, we reach a stable propagation problem after a finite number of steps.

Theorem 3.13 Let $\langle d, P\rangle$ be a propagation problem. The transition system of $\langle d, P\rangle$ is finite and terminating. A terminating transition sequence $d \Rightarrow d'$ consists of at most $k = 1 + \sum_{x \in X}(|d(x)| - 1)$ steps. *

Proof. Termination is an immediate consequence of the fact that each transition yields a strictly stronger domain. As d is finite, the order on domains is well-founded—a domain can only be made strictly stronger a finite number of times. As the number of propagators is finite, there can only be finitely many transition sequences $d \Rightarrow d'$. Any step $d_i \vdash p \rightarrow d_j$ in such a sequence yields a domain d_j that contains at least one element less in some variable domain, $d_j(x) \subset d_i(x)$ for some variable x. There can hence be at most $k = 1 + \sum_{x \in X}(|d(x)| - 1)$ transitions before the empty domain is reached. ∎

Let $\langle d, P\rangle$ be a propagation problem, and let d' be a stable domain reachable from d. Then d' is a **mutual fixed point** of all propagators $p \in P$, as by definition of the transition relation, no propagator can prune d.

A simple propagation-based solver

The naive approach to solving a propagation problem $\langle d, P\rangle$ is to generate all assignments $a \in d$, and then use the propagators $p \in P$ to check whether a satisfies all constraints. This approach makes use of the fact that propagators realize decision procedures for their induced constraints, but does not use their pruning capabilities. A solver that proceeds naively in this fashion is said to follow the *generate-and-test* approach.

The generate-and-test solver is too inefficient to solve real-world problems, as it requires enumeration of all assignments. We now improve on this naive method by *pruning* the set of enumerated assignments using propagation.

A propagation-based solver interleaves propagation and enumeration (search). For a propagation problem $\langle d, P\rangle$, the solver determines a stable propagation problem $\langle d', P\rangle$ such that $d \Rightarrow d'$. Then the domain d' is split into two non-empty domains d_1 and d_2 such that $\text{sol}(\langle d_1, P\rangle) \cup \text{sol}(\langle d_2, P\rangle) = \text{sol}(\langle d, P\rangle)$, and the resulting, smaller propagation problems are solved recursively. Pseudo-code for this algorithm appears in Figure 3.2. The PROPAGATE method returns a fixed point of the propagators. We will come back to it in Chapter 5. The task of the BRANCH procedure is to split the domain into two stronger domains, which are solved recursively. By splitting

```
SOLVE(⟨d, P⟩)
1  d′ ← PROPAGATE(⟨d, P⟩)
2  if d′ = ∅ then return ∅                ▷ failed
3  if d′ = {a} then return {a}            ▷ solved
4  ⟨d1, d2⟩ ← BRANCH(d′)
5  return SOLVE(⟨d1, P⟩) ∪ SOLVE(⟨d2, P⟩)

PROPAGATE(⟨d, P⟩)
1  return d′ such that ⟨d, P⟩ ⇒ ⟨d′, P⟩
```

Figure 3.2: Propagation-based constraint solving

domains using the BRANCH procedure, the algorithm explores a tree of propagation problems, the **search tree**. More elaborate branching schemes are possible, for example adding propagators to the branches instead of splitting domains. As the topic of this dissertation is propagation, we content ourselves with this simple scheme and leave the implementation of BRANCH abstract, just assuming that it satisfies the condition that $\mathrm{sol}(\langle d_1, P\rangle) \cup \mathrm{sol}(\langle d_2, P\rangle) = \mathrm{sol}(\langle d, P\rangle)$. This guarantees that all assignments are generated eventually.

Propagation does not discard solutions, and for an assignment a, PROPAGATE decides whether a is a solution. The solver will thus only return solutions, and it will find all solutions—it is **sound** and **complete**.

The advantage over generate-and-test is that PROPAGATE is applied to domains, not only to assignments. In practice, propagation can prune big parts of the search space—only a fraction of the assignments have to be enumerated.

Let us briefly look again at soundness and completeness of the solver. A solver returns solutions, soundness therefore means that a solver returns *only* solutions, and completeness means that it returns *all* solutions of a given propagation problem. Interestingly, these notions are exactly dual for propagators: a propagator *prunes non-solutions*. Hence, a propagator is sound if it prunes *only* non-solutions, and, as we will see in Chapter 4, it is complete if it prunes *all* non-solutions. Propagation-based solvers are sound, because each propagator is complete when applied to an assignment (it realizes a decision procedure for its induced constraint on assignments), and they are complete, because all assignments are enumerated and because each propagator is sound (it does not remove solutions).

Related work

► The generate-and-test approach is sometimes referred to as the "British Museum method". An early reference to this folklore term is Prawitz (1960). He refers to

the fact that one could in theory produce all the books in the British Museum by enumerating all the finitely many strings of appropriate length. Prawitz also calls it the "Fifty Million Monkeys method".

► Solving constraint satisfaction problems by constraint propagation and backtracking search dates back to the 1960's, with the famous Davis-Putnam algorithm for solving propositional satisfiability problems (Davis and Putnam, 1960; Davis et al., 1962).

► The presented solver stands in the tradition of algorithms like CS2 (Gaschnig, 1974), MAC (maintaining arc consistency, Sabin and Freuder, 1994) and FC (forward checking, Golomb and Baumert, 1965; Haralick and Elliott, 1980), in the sense that it interleaves propagation and search. However, no particular propagation strength (such as arc consistency) is built in.

► Hard problems can often only be solved if branching is done according to a good *heuristic*, such as fail-first (Haralick and Elliott, 1980). This dissertation concentrates on propagation, so we just assume an exhaustive branching procedure.

3.4 Idempotency, Monotonicity, and Confluence

In addition to being contracting and sound, propagators are traditionally required to be idempotent and monotonic. This section explains why our definition of propagators is sufficient, and what the stronger properties imply.

Definition 3.14 A propagator p is **idempotent** if and only if for all domains d, $p(p(d)) = p(d)$. It is **monotonic** if and only if for any two domains d_1 and d_2, $d_1 \subseteq d_2$ implies $p(d_1) \subseteq p(d_2)$. *

Idempotency

The result of applying an idempotent propagator is a domain that is a fixed point for the propagator. However, for solving propagation problems, we are not interested in fixed points of individual propagators, but in *mutual* fixed points of *all* propagators. As we have seen, the transition systems produce mutual fixed points, independent of whether the individual propagators are idempotent. In this dissertation, we thus do not require propagators to be idempotent. However, *if* a propagator is idempotent, we can take advantage of this fact in practice to compute the fixed point more efficiently (see Section 5.4).

A more direct way to see that idempotency is not important for propagators is to show that it can be achieved for any propagator by iteration.

Proposition 3.15 For each propagator p, there is a number $n > 0$ such that p^n (which means p iterated n times, $p \circ p \circ \cdots \circ p$) is idempotent. *

Proof. Any domain d is finite, and each application of p either arrives at a fixed point or makes d smaller. As p is a function, $p(d) = d$ implies $p(p(d)) = d$. So the length of any chain $p(p(\ldots p(d)))$ where each application yields a strictly stronger domain is bounded by the size of the domain d. ∎

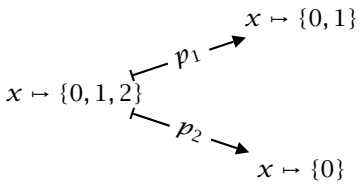
Given a propagator p, we define $p^* := p^n$ to be its **idempotent closure**, where n is the constant that ensures idempotency of p^n.

Monotonicity

The terminal domains of the transition systems are always mutual fixed points of all propagators. However, these fixed points are not necessarily unique—the transition systems are not *confluent*. Traditionally, confluence has been considered essential for constraint propagation, and the usual way to guarantee confluence is to restrict propagators to being monotonic, as we will see below.

Before discussing monotonicity and confluence in more detail, let us look at an example that exhibits non-monotonic, non-confluent behavior.

Example 3.16 (Propagation may not be confluent) Let $\langle d, \{p_1, p_2\}\rangle$ be a propagation problem with $d(x) = \{0, 1, 2\}$, $c_{p_1} = [\![x \in \{0, 1\}]\!]$, and $c_{p_2} = [\![x = 0]\!]$, without specifying the propagators any further for the moment. Assume that the propagation problem has the following transition system:



Clearly, $p_2(d)$ is a fixed point of p_1, too, so the domain $x \mapsto \{0\}$ is stable. But how can $x \mapsto \{0, 1\}$ be stable? We can define p_2 to only propagate if the domain of y is assigned or has more than two elements:

$$p_2(d)(x) = \begin{cases} x \mapsto d(x) \cap \{0\} & \text{if } |d(x) = 1| \text{ or } |d(x)| > 2 \\ d(x) & \text{otherwise} \end{cases}$$

The transition system of $\langle d, \{p_1, p_2\}\rangle$ is not confluent. *

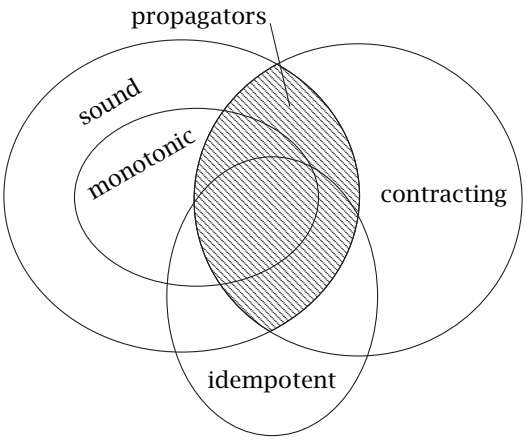


Figure 3.3: Functions in Dom → Dom

The source of the non-confluence in the above example is that the propagator p_2 is contracting and sound, but not monotonic. Clearly, $x \mapsto \{0,1\}$ is stronger than $x \mapsto \{0,1,2\}$, but in the example, $p_2(x \mapsto \{0,1\}) \supset p_2(x \mapsto \{0,1,2\})$, showing that p_2 is not monotonic. Monotonicity is in fact a sufficient condition for confluence, and in addition guarantees that the stable domain reached by transitions is the weakest mutual fixed point of the propagators.

Theorem 3.17 Let $\langle d, P\rangle$ be a propagation problem where all propagators $p \in P$ are monotonic. Then the transition system of $\langle d, P\rangle$ is confluent, and its unique stable domain is the weakest mutual fixed point of all $p \in P$ that is stronger than d. *

Proof. We will show that for any stable domain d' reachable by transitions from d, any mutual fixed point $d'' \subseteq d$ of the propagators in P is stronger than d'. Confluence then follows from the fact that all stable domains reachable by transitions from d are mutual fixed points of the propagators in P.

Let $d \vdash p_1 \rightarrow d_1 \vdash p_2 \rightarrow d_2 \ldots \vdash p_n \rightarrow d_n = d'$ be a sequence of transitions leading to the stable domain d', and let $d'' \subseteq d$ be an arbitrary mutual fixed point of all propagators in P. We will show by induction over the length of the transition sequence that $d'' \subseteq d'$. The base case is clear, as we assumed $d'' \subseteq d$. For the induction step, assume $d'' \subseteq d_i$. Then, by monotonicity, $p_{i+1}(d'') \subseteq p_{i+1}(d_i)$. As d'' is a fixed point of all $p \in P$, we get $d'' \subseteq p_{i+1}(d_i) = d_{i+1}$. ∎

Monotonicity subsumes soundness: if a function $f \in \mathsf{Dom} \rightarrow \mathsf{Dom}$ is monotonic, then $\{a\} \subseteq d$ implies $f(\{a\}) \subseteq f(d)$, which is the definition of soundness. Figure 3.3 shows a diagram of the different classes of functions in Dom → Dom. Propagators are at the intersection of sound and contracting functions, and can in addition be idempotent and/or monotonic.

Non-monotonicity in practice

The practical consequence of allowing non-monotonic propagators is that the size and shape of the search tree, as well as the order in which solutions are found, depends on the order of propagator application. The *set* of solutions that is found is however independent of the application order, as each propagator is still sound.

The main reason why confluent propagation is desirable is that it makes debugging of a constraint model easier. The fundamental rule in a confluent system is that adding propagators results in a smaller search space. With non-monotonic propagators, this may no longer be true: adding a propagator can *prevent* another propagator from pruning, leading to less pruning and a *bigger* search space. Depending on the concrete implementation, the order of propagation may even be non-deterministic. In this case, non-monotonic propagators lead to different search trees in different runs of the solver using the same propagators.

In practice, most propagators are monotonic. The only reason to resort to a non-monotonic propagation algorithm is usually its asymptotic run-time. For instance, Baptiste (1994) shows that edge finding with task intervals (a propagation algorithm used for scheduling problems and first described by Caseau and Laburthe, 1994) is non-monotonic. Instead of considering all (exponentially many) subsets of activities, the algorithm restricts itself to only quadratically many subsets. While this makes the algorithm tractable in terms of run-time, the choice of these subsets depends on the current domain, and hence propagation becomes non-monotonic.

In this dissertation, we do not require monotonicity of propagators. None of the properties or techniques we discuss in the following chapters rely on propagators being monotonic. However, as monotonicity plays such an important role in practice, we will show how the techniques we develop behave for monotonic propagators. For example, Chapter 5 discusses the relation of (non-)monotonicity and propagator completeness. In Chapter 7, we show that a propagator that is derived from a monotonic propagator using a view is again monotonic. Finally, Chapter 11 shows that the propagators we derive for set constraints are always monotonic.

Monotonic and idempotent propagators in the lattice

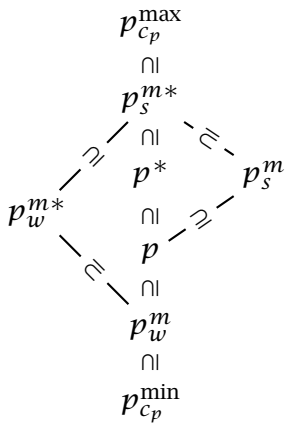
Given an arbitrary propagator p, its idempotent closure p^* is always stronger than p. We can define the weakest monotonic propagator stronger than p as p^m_s, and the strongest monotonic propagator weaker than p as p^m_w. These two exist because monotonic propagators are closed under intersection and union.

Proposition 3.18 Monotonic propagators are closed under intersection and union.$_*$

Proof. Let p_1 and p_2 be monotonic propagators, and let d and d' be domains such that $d' \subseteq d$. From monotonicity, $p_1(d') \subseteq p_1(d)$ and $p_2(d') \subseteq p_2(d)$, it follows that

$p_1(d') \cup p_2(d') \subseteq p_1(d) \cup p_2(d)$ and $p_1(d') \cap p_2(d') \subseteq p_1(d) \cap p_2(d)$, as both union and intersection are monotonic. ∎

These propagators, together with their idempotent closures, are arranged in the propagator lattice as follows:



If p is monotonic, all these propagators collapse into $p_{c_p}^{\min}$, p, p^*, and $p_{c_p}^{\max}$.

Related work

Propagators are traditionally required to be monotonic. To the best of our knowledge, this dissertation presents the first systematic discussion of non-monotonic propagators.

► A function that operates on a lattice (like Dom) and that is contracting, idempotent, and monotonic is called a *closure operator* (Ward, 1942) or *consequence operator* (Tarski, 1930, 1983). In the context of logical deduction, in which Tarski developed the consequence operator, idempotency characterizes the fact that all consequences of consequences of a set of sentences are themselves consequences. Monotonicity captures that from more facts, more consequences can be deduced. In constraint programming, however, this philosophical point of view is not important: When computing a fixed point of a set of propagators, the idempotent closure is computed automatically; and fixed points are only required to still contain all solutions of the original problem, so that soundness is sufficient.

► Saraswat et al. (1991) explicitly relate their definition of propagators to Tarski's consequence operator, and thus require both monotonicity and idempotency. The propagators in their system are models for *concurrent processes*. Idempotency thus means that a process only stops if it cannot contribute any more information. Monotonicity guarantees unique fixed points. Thus, Saraswat et al. can identify processes with their sets of fixed points.

► Benhamou et al. (1994) also essentially model propagators as closure operators. Benhamou (1996) drops idempotency, but still lists monotonicity as a requirement.

► In Constraint Handling Rules (see Frühwirth, 1998), propagators are implemented by sets of rules. Confluence plays a prominent role: any (terminating) set of rules can be made confluent using a method similar to Knuth-Bendix completion.

► Indexicals, as defined by Van Hentenryck et al. (1991) or Carlson (1995), are constructed to be monotonic and idempotent propagators. Van Hentenryck et al. (1991) explicitly link indexicals with closure operators.

► The Mozart (2009) system allows non-monotonic propagators, but fixes their order of application, so that fixed points remain unique (Müller, 2001). This is a vital prerequisite for the correctness of *recomputation* as described by Schulte (2002). Section 6.1 reconciles recomputation with non-monotonicity.

3.5 A Many-Sorted Model

In the model presented so far, there is a single set of values V common to all variables. Constraint solvers typically offer different *sorts* of variables, such as integer, Boolean, or set variables. This section presents the minimal extension of the mathematical model that is necessary to capture different sorts of variables.

The fundamental difference in a many-sorted system is that instead of a single set of values V, we have a *family* of sets V_x, one per variable $x \in X$. The set V_x is the *sort* of the variable x.

The basic entities in our model—assignments and domains—are mappings from variables to values or sets of values, respectively. Instead of functions $X \to V$, we now need a mapping $x \mapsto v$ where $v \in V_x$. This cannot be captured in a simple functional type. A **dependent type** $\prod x \in X.\ Y_x$ describes a mapping from variables to a *family* Y_x for each x. Using this notation, we can bootstrap the remaining definitions:

- Assignments $a \in \mathsf{Asn} := \prod x \in X.\ V_x$
- Domains $d \in \mathsf{Dom} := \prod x \in X.\ \mathscr{P}(V)$

All other definitions are not affected. For instance, constraints are still sets of assignments, propagators are still functions in $\mathsf{Dom} \to \mathsf{Dom}$, and the definition of the operator $\mathrm{con}(\cdot)$ does not change.

As none of the proofs in this dissertation takes advantage of the fact that two variables are mapped into the same set of values, all results immediately generalize to the many-sorted case by simple textual replacement of the relevant types.

4 Propagation Strength

This chapter presents a characterization of the strength of propagators. The characterization serves two purposes. It yields a more complete mathematical model of propagation, explaining propagation strength precisely and in an abstract way. More importantly for this dissertation, it is a tool that we will use later to show how techniques we develop preserve or influence propagation strength (Chapter 7), and to show that propagators we generate have a particular strength (Chapter 11).

We already saw in the previous chapter that propagators differ in strength, and that there is a unique weakest and a unique strongest propagator for each constraint. We now identify classes of propagators that lie in between the weakest and the strongest propagators for their respective induced constraints. As a measure for propagation strength, we weaken the notion of domain completeness and introduce *completeness with respect to domain approximations*, restricting the domains that a propagator may return to be at least of a certain strength.

Weaker propagators matter in practice because for many constraints, domain-complete propagation algorithms are computationally expensive or even intractable. Therefore, often propagators are used that for example only propagate the bounds of integer variable domains, or even only prune the domain if some variables are assigned. We define these classes of propagation strength, and relate our definitions to the more classical notions of domain or bounds consistency known from the literature.

This chapter builds on the notions of domain approximations and completeness with respect to approximations developed by Benhamou (1996). We use these notions slightly differently, though. While Benhamou *combines* heterogeneous constraint solvers, we *characterize* the propagation strength of different propagators within the same constraint solving framework.

Structure of the chapter. We start the discussion of propagation strength by again looking at the unique weakest and strongest propagators for a given constraint (4.1). Next, we discuss domain approximations (4.2), and how they serve as a measure for propagation strength (4.3). Finally, we define the integer interval approximation (4.4) and the set interval approximation (4.5), two domain approximations that are important in practice.

4.1 Weakest and Strongest Propagators

The previous chapter showed that for each constraint, there is a unique weakest propagator $p_c^{\min}$ and a unique strongest propagator $p_c^{\max}$. Before looking at propagators in between $p_c^{\min}$ and $p_c^{\max}$ in the following sections, this section develops more concrete definitions of $p_c^{\min}$ and $p_c^{\max}$. We will see later how the definition of $p_c^{\max}$ can be weakened in order to yield the desired classes of propagation strength.

The weakest propagator for a constraint c, $p_c^{\min}$, only realizes the decision procedure for c on assignments, checking for an assignment a whether it satisfies c or not. Otherwise, it does not prune the domain. We can define it as

$$p_c^{\min}(d) := \textbf{if } d = \{a\} \textbf{ and } a \notin c \textbf{ then } \emptyset \textbf{ else } d$$

Proposition 4.1 The function $p_c^{\min}$ as defined above is a propagator, it is idempotent and monotonic, it induces the constraint c, and it is the unique weakest propagator that induces c. *

Proof. Contraction, idempotency, and monotonicity are all obvious from the definition of $p_c^{\min}$, and the propagator clearly induces the constraint c. Let p be any other propagator with $c_p = c$. In order to prove that $p_c^{\min}$ is the weakest propagator for c, we have to show that $p(d) \cup p_c^{\min}(d) = p_c^{\min}(d)$ for any domain d. There are two cases.

- Case $p_c^{\min}(d) = \emptyset$: Then $d = \{a\}$ for some assignment a, and as $c_p = c$, we know $p(d) = \emptyset$. So $p(d) \cup p_c^{\min}(d) = \emptyset = p_c^{\min}(d)$.
- Case $p_c^{\min}(d) = d$: Then $p(d) \cup p_c^{\min}(d) = p(d) \cup d = d = p_c^{\min}(d)$. ∎

At the other end of the spectrum is the strongest propagator $p_c^{\max}$. A propagator returns domains, so propagation strength is limited by what a domain can represent. Given a domain d, $p_c^{\max}$ must yield the strongest domain that contains all assignments of c that are still licensed by d. This is captured in the following definition.

Definition 4.2 The **domain relaxation** of a constraint c is defined as

$$[\![c]\!] := \bigcap \{d \in \mathsf{Dom} \mid c \subseteq \mathrm{con}(d)\}$$ *

As domains are closed under intersection, $[\![c]\!]$ is the strongest domain that licenses all assignments of the constraint c. The propagator $p_c^{\max}(d)$ must therefore compute the domain relaxation of $c \cap d$:

$$p_c^{\max}(d) := [\![c \cap d]\!]$$

Proposition 4.3 The function $p_c^{\max}$ is a propagator, it is monotonic and idempotent, it induces the constraint c, and it is domain-complete. *

Proof. The full set of domains Dom is closed under intersection, so that $\llbracket c \cap d \rrbracket \in \mathsf{Dom}$. From $c \cap d \subseteq d$, it follows that $\llbracket c \cap d \rrbracket \subseteq d$. Set intersection is monotonic, so $\llbracket c \cap d \rrbracket$ is a contracting, monotonic function in d.

From the definition of $p_c^{\max}$, we know that $p_c^{\max}(p_c^{\max}(d)) = \llbracket c \cap p_c^{\max}(d) \rrbracket = \llbracket c \cap \llbracket c \cap d \rrbracket \rrbracket$. Furthermore, the definition of $\llbracket \cdot \rrbracket$ yields that $c \cap d$ is a subset of $\llbracket c \cap d \rrbracket$, and $\llbracket c \cap d \rrbracket$ is a subset of d. Together, this gives $c \cap d = c \cap \llbracket c \cap d \rrbracket$. So $p_c^{\max}$ is idempotent:

$$p_c^{\max}(p_c^{\max}(d)) = \llbracket c \cap \llbracket c \cap d \rrbracket \rrbracket = \llbracket c \cap d \rrbracket = p_c^{\max}(d)$$

All together, $p_c^{\max}$ is a monotonic, idempotent propagator. Given an assignment a, $p_c^{\max}(\{a\}) = \llbracket c \cap a \rrbracket$, and $\llbracket c \cap a \rrbracket = \{a\}$ if and only if $a \in c$. The induced constraint of $p_c^{\max}$ is therefore c. As Dom is closed under intersection, $\llbracket c \cap d \rrbracket$ is the unique strongest domain that contains all assignments of $c \cap d$. Hence, $p_c^{\max}$ is the strongest propagator inducing c, it is domain-complete. ∎

Example 4.4 (Propagating linear constraints) Let us look at the simple ternary linear constraint $c = [\![x = 3y + 5z]\!]$. The weakest propagator for c, $p_c^{\min}$, only decides whether an assigned domain satisfies c. For example, it accepts the domain d_1 where $d_1(x) = \{8\}$, $d_1(y) = \{1\}$, $d_1(z) = \{1\}$: $p_c^{\min}(d_1) = d_1$. And it rejects d_2 where $d_2(x) = \{6\}$, $d_2(y) = \{2\}$, $d_2(z) = \{1\}$: $p_c^{\min}(d_2) = \emptyset$. For a non-assigned domain d_3 where $d_3(x) = \{3, \ldots, 7\}$, $d_3(y) = \{0, 1, 2\}$, $d_3(z) = \{0, 1\}$, $p_c^{\min}$ does not perform any pruning: $p_c^{\min}(d_3) = d_3$.

The strongest propagator $p_c^{\max}$ removes all values from a variable domain which cannot be part of any assignment $a \in c$. For the above domain d_3, it yields $p_c^{\max}(d_3)(x) = \{3, 5, 6\}$, $p_c^{\max}(d_3)(y) = \{0, 1, 2\}$, $p_c^{\max}(d_3)(z) = \{0, 1\}$, removing the 4 and the 7 from the domain of x. *

The following section refines the concept of domain relaxation, introducing the weaker notion of relaxation with respect to a domain approximation. Similar to domain completeness, we can then define weaker notions of completeness.

4.2 Domain Approximations

Domain completeness is a very strong property. In fact, for many important constraints such as linear equations, deciding whether $\llbracket c \cap d \rrbracket \subset d$ (which a domain-complete propagator would have to do) is NP-complete (Choi et al., 2004). Therefore

weaker notions of completeness have been developed, which we define here with respect to *domain approximations*.

Domains approximate constraints. This is already witnessed in their type. We saw in Section 3.1 that, modulo the $\mathrm{con}(\cdot)$ operator, domains are a proper subset of constraints ($\{\mathrm{con}(d) \mid d \in \mathsf{Dom}\} \subset \mathsf{Con}$).

We can approximate even further. A domain approximation is defined by restricting the sets of values a variable is mapped to, for example to *intervals* according to a given order of the values. Two approximations that are used in almost all constraint solvers motivate that domain approximations are highly relevant in practice.

Interval reasoning for integer variables. In practice, many constraints over integer variables are realized as propagators using *interval reasoning*. The propagators regard a variable domain as an interval, ignoring any holes, and only prune the interval bounds. This often leads to a significant reduction of the asymptotic run-time complexity of propagation algorithms. For instance, a domain-complete propagation algorithm for the *all-different* constraint (Régin, 1994) has a run-time complexity of $O(n^{2.5})$, whereas the run-time of the weaker propagation algorithm that uses interval reasoning (Puget, 1998) is in $O(n \log n)$.

Interval reasoning for set variables. The value of a set variable x is a subset of a finite universe $\mathscr{U}$, so the set of values is $V = \mathscr{P}(\mathscr{U})$. This means that set variable domains are sets of sets, and their size is exponential in the size of the universe $\mathscr{U}$. Most constraint solvers that support set variables approximate their domains as intervals. The idea is similar to the interval approximation for integer variables: a set interval contains all the sets between a lower and an upper bound, only that the subset order on sets is used instead of the natural order on numbers. However, while most solvers implement the variable domains of integer variables completely and only provide certain propagators that use interval reasoning, set variable domains are typically *stored* as intervals. The set interval approximation was introduced by Puget (1992) and formalized by Gervet (1995, 1997).

Domain systems

We will now define approximations of domains formally, as sets of allowed, or representable, domains. Such a set is called a *domain system*, and will later be used to define the relaxation of a constraint with respect to an approximation, and finally weaker notions of propagator completeness.

The definition of domain systems we give below results from several requirements. A domain system must contain all assigned domains in order to represent solutions, as well as the empty domain for failure. Moreover, a domain system must be able to approximate *any* other domain, so it must at least include the full domain. Finally, in order to generalize the definition of domain relaxation of a constraint, we require

domain systems to be closed under intersection. The following definition reflects these requirements.

Definition 4.5 A **domain system** is a set of domains $\mathscr{D} \subseteq \mathsf{Dom}$ with the following properties:

- $\mathscr{D}$ is closed under intersection
- $\mathscr{D}$ contains the full domain $\lambda x.V$
- $\mathscr{D}$ contains the assigned domains $\{a\}$ for all assignments a

Any domain $d \in \mathscr{D}$ is called a $\mathscr{D}$**-domain**. *

The empty domain is an element of any domain system $\mathscr{D}$ because $\mathscr{D}$ is closed under intersection, and the intersection of two different assigned domains $\{a\}$ and $\{a'\}$ is empty.

The domain relaxation operator $[\![\cdot]\!]$ expresses the fact that domains approximate constraints, by computing the strongest domain that licenses all solutions of a constraint. We can generalize this notion to relaxation with respect to a domain approximation. Definition 4.5 requires that $\mathscr{D}$ be closed under intersection, so that the strongest $\mathscr{D}$-domain that licenses all solutions of a constraint is well-defined:

Definition 4.6 For a domain system $\mathscr{D}$, the $\mathscr{D}$**-relaxation** of a constraint c is defined as

$$[\![c]\!]_{\mathscr{D}} := \bigcap \{d \in \mathscr{D} \mid c \subseteq d\}$$ *

Example 4.7 (Interval approximation) The interval approximation for integer variables corresponds to the domain system

$$\mathscr{D}^{[\mathbb{Z}]} := X \to \mathscr{P}_{\text{int}}(V) = X \to \{[i,j] \mid i \in V, j \in V\}$$

If the set of values V is an interval, then $\mathscr{D}^{[\mathbb{Z}]}$ is a domain system according to the definition: all singleton intervals $[i,i]$, the full interval $[\min(V), \max(V)]$, as well as the empty interval (denoted by $[1,0]$) are present for all variables x, and the intersection of two intervals is again an interval (possibly empty).

For any constraint c, the interval relaxation $[\![c]\!]_{\mathscr{D}^{[\mathbb{Z}]}}$ computes the strongest $\mathscr{D}^{[\mathbb{Z}]}$-domain that captures c. It can be defined equivalently as

$$[\![c]\!]_{\mathscr{D}^{[\mathbb{Z}]}} := \lambda x.[\min([\![c]\!](x)), \max([\![c]\!](x))]$$ *

Domain systems are naturally ordered by inclusion.

Definition 4.8 A domain system $\mathscr{D}_1$ is **stronger** than a domain system $\mathscr{D}_2$ if and only if $[\![c]\!]_{\mathscr{D}_1} \subseteq [\![c]\!]_{\mathscr{D}_2}$ for all constraints c. *

The smaller a domain system is, the coarser the approximation it represents.

Proposition 4.9 A domain system $\mathscr{D}_1$ is stronger than a domain system $\mathscr{D}_2$ if and only if $\mathscr{D}_1 \supseteq \mathscr{D}_2$. *

Proof. We prove both directions of the equivalence:

- $\Leftarrow$ Given domain systems $\mathscr{D}_1 \supseteq \mathscr{D}_2$, we know that for all constraints c, $[\![c]\!]_{\mathscr{D}_2} \in \mathscr{D}_1$. Therefore, $[\![c]\!]_{\mathscr{D}_1}$ can only be the same or stronger.
- $\Rightarrow$ For the reverse direction, consider domain systems $\mathscr{D}_1$ and $\mathscr{D}_2$ such that $[\![c]\!]_{\mathscr{D}_1} \subseteq [\![c]\!]_{\mathscr{D}_2}$ for all constraints c. We know that for all domains $d' \in \mathscr{D}_2$, $[\![\mathrm{con}(d')]\!]_{\mathscr{D}_2} = d'$. On the other hand, we have $d' \subseteq [\![\mathrm{con}(d')]\!]_{\mathscr{D}_1}$. Together, we get $[\![\mathrm{con}(d')]\!]_{\mathscr{D}_1} = d'$, which means that $d' \in \mathscr{D}_1$ and consequently $\mathscr{D}_2 \subseteq \mathscr{D}_1$. ∎

According to the definition, the strongest domain system is $\mathscr{D}^{\max} := \mathsf{Dom}$, and the weakest one is $\mathscr{D}^{\min} := \{\emptyset, \lambda x.V\} \cup \{\{a\} \mid a \in \mathsf{Asn}\}$.

The interval domain system $\mathscr{D}^{[\mathbb{Z}]}$ is strictly stronger than $\mathscr{D}^{\min}$, as it does contain non-singleton domains other than $\lambda x.V$. At the same time, $\mathscr{D}^{[\mathbb{Z}]}$ is strictly weaker than $\mathscr{D}^{\max}$, because it does not contain non-interval domains.

4.3 Strength with Respect to a Domain System

Section 3.2 defined domain completeness using the domain relaxation operator $[\![\cdot]\!]$. This section generalizes the definition and defines completeness with respect to domain systems, which can characterize classes of weaker propagators.

Completeness with respect to approximations

The definition of completeness with respect to a domain system $\mathscr{D}$ is straightforward.

Definition 4.10 Let $\mathscr{D}$ be a domain system. A propagator p is **$\mathscr{D}$-complete**, if and only if for any domain d, $p(d) \subseteq [\![c_p \cap [\![d]\!]_{\mathscr{D}}]\!]_{\mathscr{D}}$. *

Compared to domain completeness, we accept solutions in $[\![d]\!]_{\mathscr{D}}$ instead of the stricter d, and we only require the result to be at least a $\mathscr{D}$-domain (or stronger). For the full domain system Dom, the definition coincides with domain completeness. However, $\mathscr{D}$ completeness is only defined as a lower bound on the pruning capability of a propagator (or, equivalently, an upper bound on the domains a propagator may return). As a consequence, a domain-complete propagator is always $\mathscr{D}$-complete for any domain system $\mathscr{D}$. For every constraint c, there is a unique weakest $\mathscr{D}$-complete propagator, which we call $\mathscr{D}$-canonical.

Definition 4.11 The $\mathscr{D}$-**canonical** propagator for a constraint c is defined as $p(d) := \lceil c \cap \lceil d \rceil_{\mathscr{D}} \rceil_{\mathscr{D}} \cap d$. *

It is easy to see that the $\mathscr{D}$-canonical propagator for a constraint c is the weakest $\mathscr{D}$-complete propagator for c. Any $\mathscr{D}$-complete propagator must satisfy two equations: $p(d) \supseteq \lceil c \cap \lceil d \rceil_{\mathscr{D}} \rceil_{\mathscr{D}}$, and $p(d) \subseteq d$ (so that it is contracting). The weakest propagator that satisfies both equations is therefore the $\mathscr{D}$-canonical propagator.

Example 4.12 ($\mathscr{D}^{[\mathbb{Z}]}$-complete all-different) Let p be a $\mathscr{D}^{[\mathbb{Z}]}$-complete propagator for the *all-different* constraint on three variables, $c_p = [\![x \neq y \wedge x \neq z \wedge y \neq z]\!]$, and let the domain d be given such that $d(x) = d(y) = d(z) = \{1, 3\}$. Then $\lceil d \rceil_{\mathscr{D}^{[\mathbb{Z}]}}(x) = \lceil d \rceil_{\mathscr{D}^{[\mathbb{Z}]}}(y) = \lceil d \rceil_{\mathscr{D}^{[\mathbb{Z}]}}(z) = \{1, 2, 3\}$, and $c_p \cap \lceil d \rceil_{\mathscr{D}^{[\mathbb{Z}]}} = \lceil d \rceil_{\mathscr{D}^{[\mathbb{Z}]}}$. The propagator p therefore does not have to prune the domain d. On a domain d' where $d'(x) = d'(y) = d'(z) = \{1, 2\}$, however, we get $\lceil d' \rceil_{\mathscr{D}^{[\mathbb{Z}]}} = d'$, and $c_p \cap \lceil d' \rceil_{\mathscr{D}^{[\mathbb{Z}]}} = \emptyset$. Thus the propagator must detect failure, $p(d') = \emptyset$. Puget (1998) develops a $\mathscr{D}^{[\mathbb{Z}]}$-complete propagation algorithm for *all-different*. *

Consistency

The definition of $\mathscr{D}$ completeness establishes a relationship between the propagator as a function and the constraint that it induces. For historic reasons, propagation strength is usually defined differently, as a *property of domains* called *consistency*. Here, we define different notions of consistency and show how they are related to completeness of propagators. For the historic context, see Section 4.6.

The strongest consistency property is called domain consistency.

Definition 4.13 A domain d is **domain-consistent** for a constraint c if and only if $d = \lceil c \cap d \rceil$. *

A propagator p **establishes domain consistency** if and only if its result is domain-consistent for its induced constraint: $p(d) = \lceil c_p \cap p(d) \rceil$ for any domain d. Note the difference to our definition of domain-complete propagators: the returned domain is required to be a fixed point. However, as domain-complete propagators are idempotent, they establish domain consistency:

$$p(d) = p(p(d)) = \lceil c_p \cap p(d) \rceil$$

Weaker consistency notions can be defined with respect to a domain system.

Definition 4.14 A domain d is $\mathscr{D}$-**consistent** for a constraint c and a domain system $\mathscr{D}$ if and only if $d \subseteq \lceil c \cap \lceil d \rceil_{\mathscr{D}} \rceil_{\mathscr{D}}$. *

For the full domain system Dom, this definition again coincides with domain consistency. However, for weaker domain systems $\mathscr{D} \subset \mathscr{D}^{\max}$, the result of a $\mathscr{D}$-complete propagator is not always $\mathscr{D}$-consistent for its induced constraint, as $\mathscr{D}$ completeness does not imply idempotency. A $\mathscr{D}$-complete propagator can be either too weak or too strong to be idempotent.

Example 4.15 (Too weak for idempotency) Consider a $\mathscr{D}^{[\mathbb{Z}]}$-complete propagator for the equation $x = y$, defined as

$$p(d) = (x \mapsto d(x) \cap \llbracket d \rrbracket_{\mathscr{D}^{[\mathbb{Z}]}}(y), \\ y \mapsto d(y) \cap \llbracket d \rrbracket_{\mathscr{D}^{[\mathbb{Z}]}}(x))$$

Given the domain $d = (x \mapsto \{0,2,3\}, y \mapsto \{1,2,3\})$, the propagator returns the domain $p(d) = (x \mapsto \{2,3\}, y \mapsto \{1,2,3\})$, which is clearly not a fixed point, and not $\mathscr{D}^{[\mathbb{Z}]}$-consistent. The propagator is *too weak* to achieve $\mathscr{D}^{[\mathbb{Z}]}$ consistency. ∗

Example 4.16 (Too strong for idempotency) As an example for a propagator that is too strong, consider the *all-different* constraint. Assume we have three variables and the domain $d(x_1) = d(x_2) = d(x_3) = \{1,3\}$. A domain-complete propagator for *all-different* will detect failure immediately, while a $\mathscr{D}^{[\mathbb{Z}]}$-complete propagator could return the domain unchanged (as seen in Example 4.12). A propagator that lies between domain and $\mathscr{D}^{[\mathbb{Z}]}$ completeness might remove the 1 from the domain of x_1, yielding a domain that is not a fixed point and not $\mathscr{D}^{[\mathbb{Z}]}$-consistent with *all-different*. This propagator is *too strong* for achieving $\mathscr{D}^{[\mathbb{Z}]}$ consistency. ∗

At its fixed points, however, any $\mathscr{D}$-complete propagator achieves $\mathscr{D}$ consistency.

Proposition 4.17 Let p be a $\mathscr{D}$-complete propagator. Each fixed point of p is $\mathscr{D}$-consistent for c_p. ∗

Proof. Consider a fixed point d of p. $\mathscr{D}$ completeness of p means $p(d) \subseteq \llbracket c \cap \llbracket d \rrbracket_{\mathscr{D}} \rrbracket_{\mathscr{D}}$. Substituting $p(d)$ for d, we get $p(p(d)) \subseteq \llbracket c \cap \llbracket p(d) \rrbracket_{\mathscr{D}} \rrbracket_{\mathscr{D}}$, and as d is a fixed point of p, it follows that $p(d) \subseteq \llbracket c \cap \llbracket p(d) \rrbracket_{\mathscr{D}} \rrbracket_{\mathscr{D}}$. ∎

When a $\mathscr{D}$-complete propagator p is used in a propagation-based solver, the fixed point that the solver computes is thus always $\mathscr{D}$-consistent for c_p.

In Section 4.1, we saw that complete propagators are monotonic. This result does not generalize to completeness with respect to a domain system $\mathscr{D}$. As for idempotency, this is due to $\mathscr{D}$ completeness being defined only as a lower bound. However, the $\mathscr{D}$-canonical propagator for a constraint c, defined as $p(d) := \llbracket c \cap \llbracket d \rrbracket_{\mathscr{D}} \rrbracket_{\mathscr{D}} \cap d$, is monotonic. The argument is the same as for domain completeness, both $\llbracket \cdot \rrbracket_{\mathscr{D}}$ and set intersection are monotonic operations. Another monotonic $\mathscr{D}$-complete propagator is p^*.

4.4 The Integer Interval Approximation

Let us come back to the interval approximation $\mathscr{D}^{[\mathbb{Z}]}$ for integer variables to get an intuition for consistency and completeness with respect to an approximation.

Traditionally, $\mathscr{D}^{[\mathbb{Z}]}$ consistency is called *bounds* consistency. To distinguish it from other notions of bounds consistency (for an overview, see Choi et al., 2006), it has also been given the more specific name bounds($\mathbb{Z}$) consistency. The usual definition reads as follows:

Definition 4.18 A domain d is **bounds($\mathbb{Z}$)-consistent** for a constraint c, if and only if for each variable x, there exist assignments $a, b \in c$ such that $a(x) = \min(d(x))$, $b(x) = \max(d(x))$, and $\min(d(y)) \leq a(y) \leq \max(d(y))$, $\min(d(y)) \leq b(y) \leq \max(d(y))$ for all other variables y. *

This definition is equivalent to our generic definition of $\mathscr{D}^{[\mathbb{Z}]}$ consistency. Recall that $\mathscr{D}^{[\mathbb{Z}]}$ consistency means that $d \subseteq \llbracket c \cap \llbracket d \rrbracket_{\mathscr{D}^{[\mathbb{Z}]}} \rrbracket_{\mathscr{D}^{[\mathbb{Z}]}}$. The fact that we only require solutions for each $\min(d(x))$ and $\max(d(x))$ is reflected in the outer $\llbracket \cdot \rrbracket_{\mathscr{D}^{[\mathbb{Z}]}}$. The inner $\llbracket \cdot \rrbracket_{\mathscr{D}^{[\mathbb{Z}]}}$ makes sure that solutions may take values from the intervals instead of the full domains, ignoring any holes.

The bounds(D) and range approximations

Two stronger notion of bounds consistency are known from the literature. They are called bounds(D) consistency (discussed for example by Choi et al., 2006) and range consistency (as presented for instance by Quimper, 2006), and strengthen bounds($\mathbb{Z}$) consistency in different ways as follows.

Definition 4.19 A domain d is **bounds(D)-consistent** for a constraint c, if and only if for all variables x, there exists an assignment $a \in c$ such that $a(x) = \min(d(x))$ and $a(y) \in d(y)$ for all other variables y, and analogously for $\max(d(x))$.

A domain d is **range-consistent** for a constraint c, if and only if for all variables x and all values $v \in d(x)$, there exists an assignment $a \in c$ such that $a(x) = v$ and $\min(d(y)) \leq a(y) \leq \max(d(y))$ for all other variables y. *

We can generalize the two notions to arbitrary domain systems $\mathscr{D}$. Let us repeat the definition of $\mathscr{D}$ consistency of a domain d for a constraint c here:

$$d \subseteq \llbracket c \cap \llbracket d \rrbracket_{\mathscr{D}} \rrbracket_{\mathscr{D}}$$

Replacing either of the two $\mathscr{D}$ relaxations by a domain relaxation yields the stronger notions of consistency and completeness we want to define.

Definition 4.20 A domain d is $\mathscr{D}$-**Dom-consistent** for a domain system $\mathscr{D}$ and a constraint c if and only if $d \subseteq \llbracket c \cap d \rrbracket_{\mathscr{D}}$. A propagator p is $\mathscr{D}$-**Dom-complete** for c if and only if $p(d) \subseteq \llbracket c \cap d \rrbracket_{\mathscr{D}}$ for all domains d. The $\mathscr{D}$-**Dom-canonical** propagator is the weakest $\mathscr{D}$-Dom-complete propagator, $p(d) := \llbracket c \cap d \rrbracket_{\mathscr{D}} \cap d$.

A domain d is **Dom-**$\mathscr{D}$**-consistent** for a domain system $\mathscr{D}$ and a constraint c if and only if $d \subseteq \llbracket c \cap \llbracket d \rrbracket_{\mathscr{D}} \rrbracket$. A propagator p is **Dom-**$\mathscr{D}$**-complete** for c if and only if $p(d) \subseteq \llbracket c \cap \llbracket d \rrbracket_{\mathscr{D}} \rrbracket$ for all domains d. The **Dom-**$\mathscr{D}$**-canonical** propagator is the weakest Dom-$\mathscr{D}$-complete propagator, $p(d) := \llbracket c \cap \llbracket d \rrbracket_{\mathscr{D}} \rrbracket \cap d$. ∗

With this definition, bounds(D) consistency and completeness are exactly $\mathscr{D}^{[\mathbb{Z}]}$-Dom consistency and completeness, and range consistency and completeness correspond to Dom-$\mathscr{D}^{[\mathbb{Z}]}$ consistency and completeness.

The Dom-$\mathscr{D}$-complete propagators behave like $\mathscr{D}$-complete propagators in that they can be both too weak and too strong to achieve Dom-$\mathscr{D}$ consistency. However, a $\mathscr{D}$-Dom-complete propagator can only be too strong to achieve $\mathscr{D}$-Dom consistency. This is a consequence of the fact that the $\mathscr{D}$-Dom-canonical propagator is idempotent and achieves $\mathscr{D}$-Dom consistency. It is also monotonic.

Proposition 4.21 The $\mathscr{D}$-Dom-canonical propagator $p(d) = \llbracket c \cap d \rrbracket_{\mathscr{D}} \cap d$ for a constraint c is monotonic and idempotent. ∗

Proof. Monotonicity follows with the same argument as for complete propagators. For idempotency, we first apply the definition $p(d) = \llbracket c \cap d \rrbracket_{\mathscr{D}} \cap d$. Just as for $\llbracket \cdot \rrbracket_{\mathsf{Dom}}$, it is easy to see that $c \cap d = c \cap \llbracket c \cap d \rrbracket_{\mathscr{D}}$, and consequently $\llbracket c \cap d \rrbracket_{\mathscr{D}} = \llbracket c \cap d \cap \llbracket c \cap d \rrbracket_{\mathscr{D}} \rrbracket_{\mathscr{D}}$. So we can rewrite the definition of p to $p(d) = \llbracket c \cap d \cap \llbracket c \cap d \rrbracket_{\mathscr{D}} \rrbracket_{\mathscr{D}} \cap d$. With the original definition, this yields $p(d) = \llbracket c \cap p(d) \rrbracket_{\mathscr{D}} \cap d$. As we know that $p(d) \subseteq d$, it follows that $p(d) = \llbracket c \cap p(d) \rrbracket_{\mathscr{D}} \cap p(d) = p(p(d))$. ∎

According to these definitions, given a domain system $\mathscr{D}$, any domain-complete propagator is $\mathscr{D}$-Dom-complete and Dom-$\mathscr{D}$-complete, and any $\mathscr{D}$-Dom-complete and any Dom-$\mathscr{D}$-complete propagator is $\mathscr{D}$-complete. Furthermore, $\mathscr{D}^{\max}$ completeness (domain completeness) is the same as $\mathscr{D}^{\max}$-Dom completeness and Dom-$\mathscr{D}^{\max}$ completeness. The complete picture of propagator strength appears in Figure 4.1.

$\mathscr{D}^{[\mathbb{Z}]}$-complete and Dom-$\mathscr{D}^{[\mathbb{Z}]}$-complete propagation algorithms are relatively common, examples are given by Leconte (1996), Quimper et al. (2004), Choi et al. (2006), and Quimper (2006). Algorithms for $\mathscr{D}^{[\mathbb{Z}]}$-Dom-complete propagators are rarely found in practice.

Bounds$(\mathbb{R})$ completeness

For linear equation constraints, already bounds$(\mathbb{Z})$ completeness is algorithmically intractable: a bounds$(\mathbb{Z})$-complete propagator decides whether $\llbracket c \cap \llbracket d \rrbracket_{\mathscr{D}^{[\mathbb{Z}]}} \rrbracket_{\mathscr{D}^{[\mathbb{Z}]}} \subset d$,

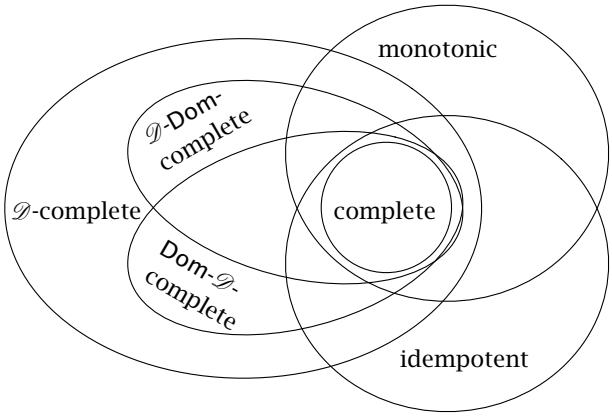


Figure 4.1: Classes of propagator strength

and this is NP-complete for a linear constraint $[\![\sum_{i=1}^{n} a_i x_i = k]\!]$ (Choi et al., 2004). Therefore, another important approximation has been developed that we have not mentioned yet: bounds($\mathbb{R}$).

The bounds($\mathbb{R}$) approximation is different from the approximations discussed so far, in that it uses a different set of values for the variables: instead of some subset of the integers, we use a subset of the real numbers. The domain system $\mathscr{D}^{[\mathbb{R}]}$ is defined as $\mathscr{D}^{[\mathbb{R}]} := X \to \{[i, j] \subseteq \mathbb{R} \mid i \in V, j \in V\}$. A domain is bounds($\mathbb{R}$)-consistent if and only if $d \subseteq [\![c_{\mathbb{R}} \cap [\![d]\!]_{\mathscr{D}^{[\mathbb{R}]}}]\!]_{\mathscr{D}^{[\mathbb{R}]}}$, where $c_{\mathbb{R}}$ is the constraint c, relaxed to the real numbers. Intuitively, the difference to $\mathscr{D}^{[\mathbb{Z}]}$ consistency is that it suffices to find real-valued instead of integer solutions. The approach is similar to integer linear programming, in that it solves a *linear relaxation* (see for example Hooker, 2007) of the original problem, and uses the result to infer new lower and upper (integer) bounds.

Of course, the notion of bounds($\mathbb{R}$) completeness only makes sense for certain constraints. For example, the *all-different* constraint relaxed to the real numbers is not very useful: we can always find a real-valued assignment between the integer bounds that satisfies the constraint as long as the variables are not assigned.

The following example shows what kind of inferences are required on the different levels of completeness for linear equation constraints.

Example 4.22 (Bounds completeness for linear constraints) Let us again consider the ternary linear constraint from Example 4.4, $c = [\![x = 3y + 5z]\!]$. We already saw that for the domain d where $d(x) = \{3, \ldots, 7\}$, $d(y) = \{0, 1, 2\}$, $d(z) = \{0, 1\}$, a domain-complete propagator for c would produce the domain $p(d)(x) = \{3, 5, 6\}$, $p(d)(y) = \{0, 1, 2\}$, $p(d)(z) = \{0, 1\}$.

A bounds($\mathbb{Z}$)-complete propagator does not have to remove the 4 from the domain

of x, but could return $p(d)(x) = \{3,4,5,6\}$, $p(d)(y) = \{0,1,2\}$, $p(d)(z) = \{0,1\}$. A bounds($\mathbb{R}$)-complete propagator does not have to contribute any propagation, as even for $x = 7$, we can find a real-valued solution $y = 2/3$, $z = 1$. *

4.5 The Interval Approximation for Set Variables

Section 4.2 sketched the set interval approximation as an example for a commonly used domain approximation. This section presents its mathematical definition. We will use the set interval approximation in later chapters, in particular in Chapter 10 and Chapter 11.

The set interval approximation regards a set variable domain as an interval of sets between a greatest lower and a least upper bound. The greatest lower bound contains those elements shared by all sets in the variable domain, the elements that must be part of any assignment. The least upper bound is the union of all sets in the variable domain, it holds all the elements that can possibly be part of an assignment.

The representation of the domain of a set variable using bounds can be exponentially smaller than the full domain representation. Consider the set of all subsets of $\{1,\ldots,n\}$. Representing this set by its lower bound $\emptyset$ and upper bound $\{1,\ldots,n\}$ takes $O(n)$ space. The full set of sets has size $O(2^n)$. For this reason, solvers not only propagate with respect to the set interval approximation, but actually *implement* set domains using bounds.

We now define the set interval domain system formally. Let the set of values be given as $V = \mathscr{P}(\mathscr{U})$, the subsets of a fixed, finite universe $\mathscr{U}$. Then we can define intervals of sets with respect to the subset order.

Definition 4.23 A **set interval** $[l,u]$ is the set of sets defined as

$$[l,u] := \{s \subseteq \mathscr{U} \mid l \subseteq s \wedge s \subseteq u\}$$

A **set interval domain** maps each variable to a set interval. We define the **domain system** $\mathscr{D}^{[\mathscr{P}(\mathscr{U})]}$ as the set of all set interval domains:

$$\mathscr{D}^{[\mathscr{P}(\mathscr{U})]} := \{d \in \mathsf{Dom} \mid \forall x \in X\ \exists l \subseteq \mathscr{U}, u \subseteq \mathscr{U} :\ d(x) = [l,u]\}$$

Given a set interval domain d, we access the two components of a variable domain $d(x) = [l,u]$ as $\mathrm{glb}(d(x)) = l$ and $\mathrm{lub}(d(x)) = u$. *

We can verify that $\mathscr{D}^{[\mathscr{P}(\mathscr{U})]}$ satisfies all properties of a domain system: it is closed under intersection, it represents the full domain $\lambda x.V$, and it contains all assignments. The $\mathscr{D}^{[\mathscr{P}(\mathscr{U})]}$ approximation is similar to the integer interval domain system $\mathscr{D}^{[\mathbb{Z}]}$, as it is also based on a lower and upper bound, but with respect to the inclusion order on sets instead of the natural order on numbers.

An equivalent definition of $\mathscr{D}^{[\mathscr{P}(\mathscr{U})]}$ states that the lower and upper bounds of each variable x in a $\mathscr{D}^{[\mathscr{P}(\mathscr{U})]}$-domain d are the intersection and union, respectively, of all assignments $a \in d$:

$$\mathscr{D}^{[\mathscr{P}(\mathscr{U})]} = \{d \in \mathsf{Dom} \mid \forall x \in X : \; d(x) = [\bigcap_{a \in d} a(x), \bigcup_{a \in d} a(x)]\}$$

This definition will be useful for reasoning about set interval completeness of propagators in Chapter 11.

4.6 Related Work

► The characterization of propagation strength goes back to the notion of arc consistency, introduced by Mackworth (1977). The term arc consistency alludes to the view of a CSP as a graph, where variables are nodes and binary constraints are edges (arcs). An arc of this so-called *constraint network* is consistent if all values of the adjacent variable nodes are *supported* by the constraint of the arc, which means that for any value of one variable, one finds a value of the other such that the constraint is satisfied. Hypergraphs generalize this framework to non-binary constraints. The corresponding notion of consistency is called hyperarc consistency, generalized arc consistency, or domain consistency, the term that we use. Apt (2003) provides a detailed overview of further consistency notions, such as path consistency, k-consistency, and directional consistency. These stronger notions of consistency are usually not realized in propagation-based constraint solvers.

► Benhamou (1996) describes a model for cooperating constraint solvers based on domain approximations. Our model builds on his idea of defining propagation strength with respect to a domain system (which he calls an *approximate domain*). Maher (2002) defines completeness with respect to a class of constraints, which is equivalent to our definition of completeness with respect to an approximation. The discussion of consistency versus completeness, and what role propagator idempotency plays, is a novel contribution of this dissertation.

► A straightforward extension of the set interval approximation is to add cardinality information. For instance, the set domain $\{\{1\},\{2\},\{3\},\{1,2\},\{1,3\},\{2,3\}\}$ cannot be represented exactly using only bounds. With additional cardinality information, it can be represented using an empty lower bound, the upper bound $\{1,2,3\}$, and

the cardinality $\{1, 2\}$. Cardinality reasoning was already described by Puget (1992) and later refined by Azevedo (2007).

► Recently, several alternative approximations for set domains have been proposed. Sadler and Gervet (2004) add two more bounds, based on a (total) lexicographic order instead of the partial subset order. This so-called *hybrid* representation can express some cardinality constraints more directly. A variation of this technique as proposed by Gervet and Van Hentenryck (2006) is to use *only* lexicographic bounds, further ordered by cardinality. This *length-lex ordering* is reported to be at least as strong as the hybrid approach, while propagation is at the same time computationally cheaper. Finally, solvers have emerged that represent the full set of sets for set variable domains. These solvers make use of *Reduced Ordered Binary Decision Diagrams* (ROBDDs) to get small representations in practice (Hawkins et al., 2005).

► Domain approximations as introduced in this dissertation always aim at weakening the full domain system Dom. Recently, there has also been research on making it stronger. While all current constraint solvers are based on a Cartesian domain representation, Andersen et al. (2007) develop non-Cartesian domains and corresponding propagators.

► We have argued that weaker notions of propagation strength are important in practice because domain-complete propagation is often algorithmically expensive or even intractable. Schulte and Stuckey (2005, 2008a) identify conditions under which the stronger propagation of domain-complete propagators even does not yield smaller search trees, and therefore the computationally cheaper, weaker propagators should be used.

5 Efficient Propagator Scheduling

Chapter 3 modeled constraint propagation as a transition system. The transitions transform a propagation problem into a stable propagation problem, where the propagators are at a mutual fixed point. While the transition systems are essentially non-deterministic, an implementation of a constraint solver must decide deterministically in which order to apply the propagators.

This chapter recapitulates traditional as well as more recently developed techniques for the efficient computation of propagator fixed points. The techniques are embedded into our mathematical model, yielding refined transition systems that are closer to an implementation. All techniques are based on two insights: (1) We maintain a set of propagators for which we know that they are at a fixed point, and only apply the remaining propagators. This is called *propagator-centered propagation*. (2) We use *events* to describe modifications of variable domains, and only consider propagators for propagation if an event has occurred that can cause them to not be at a fixed point any longer. This leads to *event-directed propagation*.

In addition to reviewing the existing techniques, this chapter introduces *propagation conditions* and *modification events*. These two concepts explain what kinds of events are relevant for event-directed propagation, and how to determine the set of events that describes the modifications between two variable domains. The model based on propagation conditions and modification events yields a straightforward and efficient implementation.

Structure of the chapter. We develop transition systems that are based on queues of propagators that are possibly not at a fixed point (5.1). Then we show how events can be used to efficiently determine the set of propagators that depend on certain domain modifications and thus have to be added to the queue (5.2). Next, we make the transition systems more dynamic, by allowing propagators to change their dependencies, and by rewriting propagators to other, equivalent propagators (5.3). We then discuss two more advanced scheduling techniques, fixed point reasoning and propagator staging (5.4). Finally, we introduce propagation conditions and modification events (5.5).

5.1 Propagator-Centered Propagation

This section presents transition systems that are based on maintaining an agenda of propagators that are possibly not at a fixed point.

An agenda of propagators

Given a propagation problem $\langle d, P\rangle$, its transition system non-deterministically selects a propagator $p \in P$ that is not at a fixed point ($p(d) \neq d$) and applies it. A perfect deterministic transition system would have to determine an order of propagation that corresponds to the most efficient sequence of propagation in a practical implementation.

Constraint solvers are not perfect in this sense. The only properties we know about propagators in general are that they are contracting and sound, and this information is not sufficient for determining a perfect propagation order. The naive approach would be to try all propagators $p \in P$ until finding one that can prune the domain.

Instead of trying all propagators in P, constraint solvers keep track of which propagators are *known* to be at a fixed point. The remaining propagators are kept in a data structure called the **agenda**. Processing propagators from the agenda one by one, the solver maintains the invariant that all propagators that are not on the agenda are at a fixed point. Propagation terminates when the agenda is empty, and hence a mutual fixed point has been computed. When a propagator is added to the agenda, we say that it is **scheduled** for propagation. The propagators on the agenda are called **active**, and the propagators not on the agenda are called **idle**.

Transitions between spaces

We define agenda-based propagation again as a transition system. Instead of transitions $d \vdash p\rightarrow d'$ between domains, we now consider transitions between pairs of a domain and an agenda, $\langle d, Q\rangle$, which we call **propagation spaces** or **spaces** for short. The agenda must contain all propagators that are not at a fixed point. As this property is vital for the further discussion, it deserves a dedicated definition.

Definition 5.1 A space $S = \langle d', Q\rangle$ is **admissible** for a propagation problem $\langle d, P\rangle$ if and only if $d' \subseteq d$, and all propagators that are not on the agenda are at a fixed point: $\forall p \in P \setminus Q : \; p(d') = d'$. *

We can now define **agenda-based transition systems** formally. A transition now does not necessarily prune the domain any more. Instead, each transition applies one propagator and updates the agenda, transforming an admissible space $S = \langle d, Q\rangle$ into a space $S' = \langle d', Q'\rangle$ (written $S \vdash p\rightarrow S'$) using a propagator $p \in Q$ if and only if

1. $d \neq \emptyset$,
2. $d' = p(d)$,
3. S' is admissible,
4. $Q' \supseteq Q \setminus \{p\}$, and
5. if $d' = d$, then $Q' = Q \setminus \{p\}$ (which is needed for termination as we will see shortly).

We call conditions 3–5 the **agenda invariant**. A space where $Q = \emptyset$ or $d = \emptyset$ is called **stable**, coinciding with stability of propagation problems: if $\langle d, \emptyset \rangle$ is an admissible, stable space for a propagation problem $\langle d, P \rangle$, then $\langle d, P \rangle$ is stable, as d is a fixed point for all propagators in P. A space $\langle \emptyset, Q \rangle$ is called **failed**, and this notion also coincides with failure for propagation problems.

Theorem 5.2 The agenda-based transition system of a propagation problem $\langle d, P \rangle$ is terminating. Each terminal space $\langle d', Q \rangle$ is stable. If $\langle d', Q \rangle$ is a terminal, stable space, then $d \Rightarrow d'$ in the original transition system. *

Proof. The lexicographic ordering of spaces according to the domain (with respect to strength) and the agenda (using the subset order) is well-founded. Each transition either prunes the domain, or decreases the agenda size due to condition 5 of the transition system. The transition system is hence terminating.

If no transition is possible, then either $Q = \emptyset$, or $d = \emptyset$, so the terminal spaces are stable. Consider the transitions $\langle d, Q \rangle \vdash p_1 \rightarrow \langle d_1, Q_1 \rangle \vdash p_2 \rightarrow \ldots \vdash p_n \rightarrow \langle d_n, Q_n \rangle$ where $\langle d_n, Q_n \rangle$ is stable. Let $q_1, \ldots, q_m$ be those propagators p_i that produce strictly stronger domains ($d_i \subset d_{i-1}$), in the same order as the p_i. Then $d \vdash q_1 \rightarrow \ldots \vdash q_m \rightarrow d_m$ is a valid chain of transitions in the original transition system, $d_m = d_n$ because the same propagators have been applied in the same order (except for those that do not modify the domain), and hence $\langle d_m, P \rangle$ is stable. ∎

In summary, agenda-based propagation for a propagation problem $\langle d, P \rangle$ yields a fixed point that can also be reached using the original transition system. With Theorem 3.17, it follows that if all propagators in P are monotonic, then agenda-based propagation results in the unique weakest mutual fixed point of all propagators in P that is stronger than d.

In the following sections, we present strategies for maintaining the agenda invariant. Before we discuss sophisticated strategies, let us first look at a naive, but correct, alternative. Assume $\langle d, Q \rangle$ is an admissible space with $d \neq \emptyset$, and there is a propagator $p \in Q$. Then we can make a transition to the space $\langle p(d), Q' \rangle$ where

$$Q' = \begin{cases} P & \text{if } p(d) \neq d, \\ Q \setminus \{p\} & \text{otherwise} \end{cases}$$

This strategy maintains the agenda invariant, as it simply schedules *all* propagators whenever the domain changes. The following sections successively improve on this scheme, presenting transition systems that avoid to schedule propagators that are known to be at a fixed point. The transition systems are defined as extensions of agenda-based propagation, and usually only the rules that differ from the rules presented here are given.

Queues and priorities

So far, we left the concrete data structure used for Q open. Of course, choosing Q as a symbol already suggests the typical data structure, a FIFO (First In First Out) queue. The FIFO strategy ensures *fair* scheduling of propagators, where no group of propagators can dominate propagation while another group is never scheduled and *starves*. Starvation is an important issue, because a starving propagator may be able to perform significant pruning or detect failure earlier than any propagator in the dominating group. Starvation is a characteristic of LIFO (Last In First Out) stacks.

The natural extension of a queue is a *priority queue*. Each propagator is assigned a priority (out of a fixed, small set of priority levels), and instead of choosing the oldest propagator, the oldest propagator of the highest priority is chosen. Priorities can be used for example to prefer computationally cheap over more expensive propagators, or to order propagators by their potential impact, that is, the amount of pruning they are expected to contribute. A prioritized system always computes a fixed point of all the high-priority propagators before applying a propagator of a lower priority.

From now on we assume that Q is a priority queue with a fixed set $\{1, \ldots, i_{\max}\}$ of priority levels. Three functions implement the queue operations.

- $\mathrm{head}(Q)$ returns the propagator p that is the oldest propagator at the highest priority level in Q.
- $\mathrm{deq}(Q, p)$ returns Q where p has been removed.
- $\mathrm{enq}(Q, p, i)$ adds the propagator p with priority $i \in \{1, \ldots, i_{\max}\}$. If p is already in the queue at a priority $j \geq i$, Q is returned unchanged. If p is already in the queue Q at a priority $j < i$, then Q is returned where p has been promoted to priority i. If p is not already in the queue, Q is returned where p has been added at priority level i.

For the next two sections, we will just treat Q as a set again to simplify presentation. A transition system based on a priority queue appears in Section 5.4, where priorities play a vital role in more advanced propagator scheduling techniques.

5.2 Event-Directed Scheduling

Most agenda-based constraint solvers determine which propagators are not at a fixed point any longer using *events*. An event describes how a certain variable has changed. Only propagators that have claimed interest in an event that has occurred since their last application are scheduled for propagation. All other propagators are known to be still at a fixed point. This section employs event-directed scheduling to maintain the agenda invariant in agenda-based transition systems.

Overview

In typical propagation problems, most propagators only deal with a subset of the variables. For example, a propagator for the constraint $[\![x < y]\!]$ will only ever prune the domains of x and y, and the result of the propagation does not depend on the domain of any other variable, either. Usually, propagators not only depend on the fact that a variable domain has changed, but that it has changed *in a certain way*.

Example 5.3 (Events for less-than) Consider the following propagator for the constraint $[\![x < y]\!]$:

$$p_{x<y}(d)(z) = \begin{cases} d(x) \cap \{-\infty, \ldots, \max(d(y)) - 1\} & \text{if } z = x \\ d(y) \cap \{\min(d(x)) + 1, \ldots, \infty\} & \text{if } z = y \\ d(z) & \text{otherwise} \end{cases}$$

The propagator can only prune the domain if $\min(d(x))$ or $\max(d(y))$ has changed since its previous invocation; it *depends* on changes to these *bounds*. If only values at the other bounds or inner values have been removed, $p_{x<y}$ cannot prune the domain. *

Example 5.4 (Events for all-different) The *all-different* constraint restricts a set of integer variables $\{x_1, \ldots, x_n\}$ to take pairwise different values. A simple propagator for this constraint removes the values of all assigned variables x_i where $i \neq j$ from the domain of x_j:

$$p_{\textit{all-different}}(d)(z) = \begin{cases} d(x_j) \setminus \{d(x_i) \mid i \neq j, |d(x_i)| = 1\} & \text{if } x_j = z \\ d(z) & \text{otherwise} \end{cases}$$

The propagator can only do any pruning if one of the x_i is assigned. More powerful propagators for *all-different* (for example the domain-complete propagator by Régin, 1994), on the other hand, can potentially prune the domain if an arbitrary value is removed from the domain of any x_i. *

The examples show that different kinds of domain changes can result in different propagators not being at a fixed point any more. *Bounds* changes, the fact that a

variable has become *assigned*, or an *arbitrary* domain change can be a necessary condition for propagation. The different kinds of domain changes are called *events*.

Events

An event describes the change between a variable domain $d(x)$ and a stronger variable domain $d'(x) \subseteq d(x)$. We call a set of events that describe all possible changes an **event system**. Before defining events formally, the following examples explore part of the design space for event systems for different types of variables.

Example 5.5 (A single asn event) The most basic event system consists of the single event asn, signaling a domain change that resulted in a variable becoming assigned. Although so simple, this event system is widely used in practice: Boolean variables (with 0/1 domains) require no more than a single event, as any modification means that the variable is assigned. For arbitrary variable types, the system implements forward checking (Golomb and Baumert, 1965; Haralick and Elliott, 1980). ∗

Example 5.6 (Variable-directed scheduling) If the variable domains may contain more than two elements, the simplest event system has only a dmc event, signaling an arbitrary domain change. In this system, a propagator is scheduled if any of the variables it depends on is modified—the system corresponds to the so-called *variable-directed scheduling* scheme without events. ∗

Example 5.7 (Events for integer variables) A typical set of events that describe the changes between integer variable domains $d(x)$ and $d'(x) \subseteq d(x)$ is the following:

- asn: the variable is assigned ($|d(x)| > 1$ and $|d'(x)| = 1$)
- lbc: the lower bound changes ($\min(d(x)) < \min(d'(x))$)
- ubc: the upper bound changes ($\max(d(x)) > \max(d'(x))$)
- dmc: the domain changes ($d(x) \supset d'(x)$)

This example shows that events typically overlap. Both lbc and ubc imply that also dmc happens, and asn implies dmc and at least one of lbc and ubc. We will use this event system for many examples in the rest of this dissertation. ∗

Example 5.8 (Simplified integer events) A simplified event system for integer variables is asn, bnd, dmc. Compared to the system from Example 5.7, the two events lbc and ubc are collapsed into one event, bnd, signaling an arbitrary bounds change. Schulte and Stuckey (2008c) show that this overall simpler system in fact leads to more efficient scheduling in practice. ∗

Example 5.9 (Events for set variables) The domain of a set variable, a set of sets of values, is exponential in size. Most systems therefore use the set interval approximation $\mathscr{D}^{[\mathscr{P}(\mathscr{U})]}$ (see Section 4.5), representing a set variable domain by a greatest

lower and a least upper bound. Some systems additionally represent the cardinality of the set, usually also as an interval. A suitable event system for this setup has events asn, lbc, ubc, and card. The events lbc and ubc correspond to modifications of the greatest lower and least upper bound, respectively. A card event signals a change in cardinality, not specifying more precisely how the cardinality changed. We do not need a dmc event, since the approximation only permits bounds and cardinality changes. *

Let us now define events formally.

Definition 5.10 An **event** e is characterized by a condition $e(d(x), d'(x))$ for two variable domains $d'(x) \subseteq d(x)$. For variable domains $d''(x) \subseteq d'(x) \subseteq d(x)$, it must satisfy $e(d(x), d''(x))$ if and only if $e(d(x), d'(x))$ or $e(d'(x), d''(x))$. An event always describes an actual domain modification: if $d(x) = d'(x)$, then $e(d(x), d'(x))$ must be false. *

We define the set $\text{events}(d(x), d'(x)) := \{e \mid e(d(x), d'(x))\}$ for two variable domains $d'(x) \subseteq d(x)$. This construction ensures that events are *monotonic*, they are never discarded by further changes to a variable domain. For three variable domains $d''(x) \subseteq d'(x) \subseteq d(x)$, monotonicity implies that $\text{events}(d(x), d''(x)) = \text{events}(d(x), d'(x)) \cup \text{events}(d'(x), d''(x))$. Non-monotonic events would lead to non-monotonic propagation, as the next example shows.

Example 5.11 (A non-monotonic event) Consider the property itv, which we define to be true whenever an integer variable domain $d(x)$ is an interval $[a, b]$. We cannot turn this property into an event: with variable domains $d(x) = \{1, 2, 3, 5\}$, $d'(x) = \{1, 2, 3\}$, $d''(x) = \{1, 3\}$, we would get $\text{itv} \in \text{events}(d(x), d'(x))$, but $\text{itv} \notin \text{events}(d(x), d''(x))$.

It is important that events are monotonic, as we want to base propagator scheduling on the presence of certain events. In the above case, a propagator that is only scheduled when the (hypothetical) itv event happens would have been scheduled, because $\text{itv} \in \text{events}(d(x), d'(x))$. However, if $d'(x)$ was $\{1, 3, 5\}$ instead (leading to the same $d''(x)$, but first pruning 2, then 5), the propagator would not have been scheduled. Because of the dependency invariant, the propagator would even be at a fixed point (or it would have to be subscribed in addition with a different event). The consequence is that propagation still produces fixed points, but it becomes non-monotonic. While this is not a problem per se, we do not want to introduce unnecessary non-monotonicity. *

Event-directed scheduling

We want propagator scheduling to depend on events. For every variable and every event, we therefore define the set of propagators that must be scheduled when the event happens.

For each variable x and event e, we collect a set of **dependent propagators** in a mapping $deps(x)(e) \subseteq P$. If a propagator p appears in the dependencies $deps(x)(e)$ for some variable x and event e, we say that p is **subscribed to x with e**. For a propagation problem $\langle d, P\rangle$, we require that the dependencies *deps* satisfy the following **dependency invariant** for all $p \in P$:

$$p(d) = d \wedge d' \subseteq d \wedge p(d') \neq d$$
$$\Longrightarrow$$
$$\exists x \in X\ \exists e \in \text{events}(d(x), d'(x)) :\ p \in deps(x)(e)$$

Intuitively, the dependency invariant states that whenever a propagator is not at a fixed point, then the propagator must be subscribed to one of the modified variables with one of the events that has happened. For now, the dependency mapping is static, it is not modified by transitions. Section 5.3 introduces transition systems with dynamic dependencies.

We update the agenda in an event-directed transition system as follows:

$$Q' = Q \setminus \{p\} \cup \{deps(x)(e) \mid e \in \text{events}(d(x), d'(x))\}$$

This scheduling scheme maintains the agenda invariant. If $p(d) = d$, no propagator is scheduled since $\text{events}(d(x), d'(x))$ is empty. Otherwise, the dependency invariant guarantees that all propagators that are not at a fixed point any longer are scheduled. In Section 5.5, we will show how to implement the agenda update in practice using propagation conditions and modification events.

The dependency invariant implies that a propagator must only use information of a variable domain that corresponds to the events it is subscribed to, as the following example demonstrates.

Example 5.12 (Propagators must be honest) Consider two propagators p, p' on integer variables x and y with $c_p = [\![y = 1 \leftrightarrow x \geq 3]\!]$, and $c_{p'} = [\![x = y]\!]$. Both subscribe with lbc and ubc to both variables. However, assume that p' is domain-complete, so it can remove inner values during propagation.

For both propagators, the domain d with $d(x) = \{0, 1, 2, 3\}$ and $d(y) = \{0, 1, 2, 3\}$ is a fixed point. Now removing 3 from $d(x)$ will schedule both propagators, but it is not defined in which order. There is one scenario that demonstrates the problem: Assume p' is applied first, pruning $d(y)$ to $\{0, 1, 2\}$. As p' modified the bound of one of its own variables, it is scheduled. Assume p' is applied again immediately, and thus does not prune the domain, as it is at a fixed point. Then p is applied and prunes $d(y)$ to $\{0, 2\}$. However, this means that neither lbc nor ubc occurs, and p' is not scheduled. As $d(x)$ is still $\{0, 1, 2\}$, we have not computed a fixed point.

Thus, propagators must be "honest" about their subscriptions. If they handle events that they are not subscribed to, the terminal states of the transition systems may not be fixed points. *

Most propagation-based constraint solvers use some form of event-directed scheduling; Section 5.6 gives a detailed overview. The following two sections further refine this technique, so that we can avoid to schedule even more propagators that are known to be at a fixed point.

5.3 Dynamic Dependencies and Propagator Sets

The transition systems as presented so far neither modify the set of propagators P nor the dependencies *deps*. Under certain conditions, however, the stronger domains obtained by propagation yield more accurate knowledge about which propagators are at a fixed point. This section shows how this knowledge can be expressed by modifying P and *deps* dynamically, leading to more efficient scheduling.

In the following, scheduling is based on knowledge about propagator fixed points that depends on the current domain as well as the concrete propagators. The implementation architecture we will see in the next chapter distinguishes between the *propagation kernel*, implementing the domain-independent services like scheduling and propagator invocation, and the *domain modules*, implementing the actual variable domains and propagation algorithms. The more accurate fixed point knowledge clearly has to be realized in a domain module. For our more abstract model here, we will introduce helper functions that encode this knowledge and represent the boundary between the kernel and a domain module.

Losing interest in variables

Depending on the domain d, certain variables may not be interesting for a propagator any more. For example, consider a propagator $p_{\max}$ for the ternary maximum constraint $[\![y = \max\{x_1, x_2, x_3\}]\!]$. Given the domain $d(x_1) = \{4, 5\}$, $d(x_2) = \{3, 5\}$, $d(x_3) = \{2, 3\}$, it is clear that no matter how the domain of x_3 changes, this will not change the maximum. So the subscription of $p_{\max}$ to x_3 could be removed. That way, future domain changes of x_3 will not cause the propagator to be scheduled and thus applied gratuitously.

Dependencies now become a part of spaces, so that transitions can update them. A space is now a 3-tuple $\langle d, Q, \mathit{deps}\rangle$, and admissibility for spaces includes the dependency invariant: a space $\langle d, Q, \mathit{deps}\rangle$ is admissible if and only if the propagators not on the agenda are at a fixed point ($\forall p \in P \setminus Q : \; p(d) = d$), and the dependency invariant holds for *deps*.

The fundamental dependency invariant is not changed. But a transition can now replace one mapping *deps* that satisfies the invariant with another mapping $deps'$ that also satisfies the invariant. The transition systems have to be changed only slightly to accommodate for dynamic dependencies: we now permit transitions of the form $\langle d, Q, deps\rangle \vdash p \rightarrow \langle p(d), Q', deps'\rangle$, where *deps* is modified to $deps'$. The transition system already requires the target space to be admissible, and admissibility includes that the dependency invariant holds.

In the above example, the dependency invariant still holds if $p_{\max}$ is not subscribed to x_3. Thus, all dependencies $(x_3, e) \mapsto p_{\max}$ can be removed from *deps*. We model this as a function cancel(p, d), which returns a set of dependencies $(x, e) \mapsto p$ that can be removed. As mentioned above, this models the boundary between domain module and kernel: the domain module determines which subscriptions to cancel through the cancel function, and the kernel modifies the dependencies accordingly.

Given an admissible space $\langle d, Q, deps\rangle$ where $d \neq \emptyset$ and a propagator $p \in Q$, a transition is possible to a space $\langle d', Q', deps'\rangle$ such that

$$\begin{aligned} deps' &= deps \setminus \text{cancel}(p, d) \\ Q' &= Q \setminus \{p\} \\ &\quad \cup \{deps'(x)(e) \mid e \in \text{events}(d(x), d'(x))\} \end{aligned}$$

The system maintains the dependency invariant if cancel only removes dependencies that are in fact no longer required. Note that the scheduling is performed with respect to the updated dependencies $deps'$.

Subsumption

If the dependency invariant allows to remove *all* subscriptions of a propagator p in a space $\langle d, Q, deps\rangle$, then it guarantees that for all stronger domains $d' \subseteq d$, $p(d') = d'$. We say that p is **subsumed** by the domain d. A subsumed propagator does not contribute to propagation any more, so we can cancel its dependencies and remove it from the set P.

To enable modifications of the set of propagators P, we include it in the space and arrive at a 4-tuple $\langle d, P, Q, deps\rangle$.

We introduce a function subsumed(p, d) that may return $\{p\}$ if p is subsumed by d, and must return $\emptyset$ otherwise. The function again represents the domain-dependent part of the reasoning. Then a transition from $\langle d, P, Q, deps\rangle$ to $\langle d', P', Q', deps'\rangle$ additionally computes P' as

$$P' = P \setminus \text{subsumed}(p, d)$$

Determining subsumption is coNP-complete in general. However, it can usually be decided easily for special cases. For instance, when all significant variables of a propagator's constraint are assigned, the propagator must be subsumed after propagation. A propagator for $[\![x < y]\!]$ can report subsumption if the upper bound of x is already smaller than the lower bound of y. A propagator for *all-different* is subsumed if no two variable domains overlap. Carlson et al. (1994a) develop a technique for detecting subsumption of integer constraints implemented by indexicals.

In the implementation as described in the next chapter, detecting subsumption is vital, because it saves both memory and run-time. In fact, we therefore require for any propagator p that subsumption is detected eventually. More precisely, if all significant variables of c_p are assigned in a domain d, then subsumed$(p, d) = \{p\}$.

Fully dynamic dependencies

Up to now, the dependencies *deps* were only modified monotonically, by removing subscriptions that were no longer needed.

Under certain conditions, it can make sense for propagators to also add new subscriptions. The dependency invariant only requires that a propagator has enough subscriptions so that it is scheduled when needed. It does not have to be subscribed to *all* variables that are significant for its induced constraint.

Example 5.13 (Boolean disjunction) Consider a propagator p for a Boolean disjunction $b_1 \vee b_2 \vee \cdots \vee b_n = 1$, where we assume that the b_i are 0/1 integer variables, 0 represents false and 1 stands for true. The propagator can only prune the domain when all but one variable are assigned 0: in that case, it can set the remaining variable to 1 in order to satisfy the constraint.

Without dynamic dependencies, the propagator would have to subscribe to all n variables with the event `asn`. With dynamic dependencies, p only has to subscribe to two variables b_i and b_j that are not yet assigned to 0: if any of the remaining variable domains is modified, there are still less than $n - 1$ variables that are assigned to 0.

Assume that the domain of b_i is modified (the reasoning is the same for b_j), then there are two situations: Either it is assigned to 1, in which case the propagator is subsumed. Or it is assigned to 0, in which case the propagator has to find a new variable b_k, different from b_i and b_j, which is not yet assigned to 0. The propagator cancels the subscription to b_i and adds a subscription to b_k. If no such b_k can be found, the propagator sets $b_j \neq 0$, possibly resulting in failure if b_j was already 0. This technique was developed in the context of SAT solvers, where the two dynamic subscriptions are called *watched literals* (Moskewicz et al., 2001). *

To support new subscriptions of propagators, the function subscribe is introduced: subscribe(p, d) returns the dependencies $(x, e) \mapsto p$ that are to be added to the

mapping *deps*. Only the computation of $deps'$ has to be adjusted in a transition from $\langle d, P, Q, deps\rangle$ to $\langle d', P', Q', deps'\rangle$:

$$deps' = (deps \setminus \text{cancel}(p, d)) \cup \text{subscribe}(p, d)$$

Propagator rewriting

As a last step, we also make the set of propagators P fully dynamic. Given a space $\langle d, P, Q, deps\rangle$, a propagator $p \in P$ can be replaced by a set of propagators R (and corresponding dependencies $deps'$ to fulfill the dependency invariant) if and only if $\text{sol}(\langle d, \{p\}\rangle) = \text{sol}(\langle d, R\rangle)$. The propagator p is *rewritten* to the set of propagators R.

Example 5.14 (Rewriting reified propagators) Propagator rewriting increases code reuse. It is particularly interesting for reified constraints of the form $c = [\![c' \leftrightarrow b]\!]$, expressing that constraint c' holds if and only if the Boolean variable b is true. Given a propagator $p_{c'}$ that induces c', and a propagator $p_{\neg c'}$ that induces $\neg c'$, a reified propagator p_c can use rewriting as follows:

- If $d(b) = \{1\}$, rewrite to $p_{c'}$.
- If $d(b) = \{0\}$, rewrite to $p_{\neg c'}$.
- If c' is entailed by the domain d, return $p_c(d)(b) = \{1\}$ and report subsumption.
- If $\neg c'$ is entailed by the domain d, return $p_c(d)(b) = \{0\}$ and report subsumption. *

Without rewriting, the propagation of c' and $\neg c'$ would have to be embedded into p_c, and the check whether b is assigned would have to be performed each time p_c is applied.

We model rewriting as a function $\text{rewrite}(p, d)$, which returns the set of propagators R that p is rewritten to, or $\emptyset$ if no rewriting is performed. Additionally, if $\text{rewrite}(p, d) \neq \emptyset$, then $\text{subsumed}(p, d) = \{p\}$, $\text{cancel}(p, d)$ cancels all subscriptions, and $\text{subscribe}(p, d)$ establishes the subscriptions for the newly added propagators. Only the computation of the new set of propagators P' has to be adjusted to

$$P' = (P \setminus \text{subsumed}(p, d)) \cup \text{rewrite}(p, d)$$

5.4 Self-Rescheduling Propagators

The refinements of the transition systems presented so far were fairly standard. This section adds more advanced scheduling strategies, developed only recently by Schulte and Stuckey (2004, 2008c).

In a prioritized setting, it can be beneficial for a propagator to only perform partial propagation, and then reschedule itself to perform the remaining propagation later. That way, propagators at higher priorities can kick in. When the lower-priority propagator is run again, the domain is already stronger and applying the propagator may be more profitable.

For the techniques we discuss in this section, we assume that priorities model the estimated *cost* of propagation. A straightforward way to estimate the cost is to classify the propagators according to their algorithmic complexity. We will use the following system of costs and priorities: `unary` = 7, `binary` = 6, `ternary` = 5, `linear` = 4, `quadratic` = 3, `cubic` = 2, `veryslow` = 1. The names suggest the arity of the corresponding propagator (for the highest three priorities), or the asymptotic run-time for n-ary propagation algorithms. The cost of propagation often changes dynamically. For instance, a typical algorithm for propagating linear equations has an asymptotic run-time linear in the number of *unassigned* variables. Accordingly, when all but three variables are assigned, the cost should be reported as `ternary` instead of `linear`. We define a function $cost(p, d')$ that returns the approximated cost of running the propagation algorithm for p given the current domain d'. As in the previous section, this function models the boundary between kernel and domain module.

To make notation simpler, we lift the enq and deq functions to sets of propagators and priorities.

$$\begin{gathered}\mathrm{enq}(Q, \{\langle p_1, i_1\rangle, \ldots, \langle p_n, i_n\rangle\}) = \\ \mathrm{enq}(\mathrm{enq}(\ldots \mathrm{enq}(Q, p_n, i_n), p_2, i_2), p_1, i_1)\end{gathered}$$

The deq lifting is analogous. The concrete order of the individual operations is not specified. In a typical solver, the order will depend on implementation details.

Fixed points

The simplest form of self-rescheduling has already been introduced: When a propagator p itself modifies one of the variables it is subscribed to, it is automatically rescheduled. If the variable modifications it performed caused the scheduling of any other propagators at higher priorities, these will be run first, before propagating p again.

So the transition systems presented so far always reschedule propagators that are not at a fixed point. But they also reschedule propagators for which it is easy to determine that they *are* at a fixed point, either because they are idempotent, or because it is obvious from the current domain. These propagators should be able to *prevent* the self-rescheduling.

We introduce a function $\mathrm{fix}(p, d')$ that may return $\{p\}$ if d' is a fixed point of the propagator p, and must return $\emptyset$ otherwise.

Given an admissible space $\langle d, P, Q, \mathit{deps}\rangle$ where $d \neq \emptyset$ and a propagator p with $p = \mathrm{head}(Q)$, a transition is possible to a space $\langle d', P', Q', \mathit{deps}'\rangle$ such that

$$\begin{aligned} Q' = \mathrm{deq}(\mathrm{enq}(\mathrm{deq}(Q, p),\\ \{\langle p', i\rangle \mid \exists x \in X\ \exists e \in \mathrm{events}(d(x), d'(x)) :\\ p' \in \mathit{deps}'(x)(e) \wedge i = \mathit{cost}(p', d')\}),\\ \mathrm{fix}(p, d')) \end{aligned}$$

The propagator p is first removed from the queue, then potentially re-added because it modified one of its own variables, and then removed again if it is known to be at a fixed point. Section 6.3 develops an efficient mechanism that ensures that a propagator is not removed, added, and removed again, but only the operation that is actually necessary is performed.

Obviously, in a real system, the fixed point status has to be determined without actually computing $p(d')$, as otherwise no efficiency is gained compared to just rescheduling p. For idempotent propagators, $\mathrm{fix}(p, d') = \{p\}$ independent of the domain d'. When a propagator is not known to be idempotent, a safe approximation is always $\mathrm{fix}(p, d') = \emptyset$.

Staged propagation

For a single constraint, there is often a choice of propagation algorithms, differing in propagation strength as well as asymptotic run-time (see Chapter 4). Typically, stronger propagators are also computationally more expensive: the domain-complete *all-different* propagation algorithm has an asymptotic run-time of $O(n^{2.5})$, while the bounds($\mathbb{Z}$)-complete version is in $O(n \log n)$; a bounds($\mathbb{R}$) propagation algorithm for linear equations has linear run-time, whereas both bounds(D)-complete and domain-complete propagation in this case is NP-hard.

Recall the reason for introducing priority-based scheduling in Section 5.1: it is beneficial to compute a fixed point of the cheaper propagators before executing a more expensive one. This also works with different propagators for the same constraint. Let two propagators p_1 and $p_2 \subseteq p_1$ be given that induce the same constraint. Furthermore, assume that the cost for running p_2 is higher than the cost for running

p_1. Then p_1 is assigned a higher priority, and will always be run before p_2. When p_2 is eventually run, it can take advantage of the propagation already done by p_1.

Staged propagators. The two main drawbacks of using multiple propagators per constraint are that (1) it wastes resources in a concrete implementation, as each propagator requires memory and run-time for scheduling; and that (2) if one propagator detects subsumption or a fixed point for all the propagators, the other propagators still need to be run. Schulte and Stuckey (2004, 2008c) introduce *staged propagators*, which address these shortcomings. A staged propagator combines several propagation algorithms in a single propagator. Depending on the events that caused its scheduling, the propagator is put in a certain *stage*, determining which of the available algorithms it is going to use next. When the cheaper, weaker propagation algorithm is run, the propagator performs only *partial propagation:* it may have to be rescheduled with a different stage, at which the more expensive but stronger propagation algorithm can still prune the domain.

Example 5.15 (A staged all-different) For the *all-different* constraint, consider the two propagators p_{dom}, performing domain-complete propagation, and p_{asn}, performing the cheap value propagation when variables are assigned (as in Example 5.4). We assume the event system from Example 5.7. A staged propagator combines the two propagation algorithms as follows:

- If the propagator is scheduled because of the event `asn`, it is put in stage A and gets priority `linear`.
- If the propagator is scheduled because of the event `dmc`, and it is not already in stage A, it is put in stage B and gets priority `quadratic`.
- When the propagator is executed in stage A, it uses p_{asn}. If it is not subsumed, it reschedules itself with priority `quadratic` and in stage B.
- When the propagator is executed in stage B, it uses p_{dom}, and is not rescheduled (because p_{dom} is idempotent). *

A propagator's stage can be identified with the priority level it is assigned. We extend our concept of propagators $p \in P$. Each $p \in P$ is now a function Stage $\to$ (Dom $\to$ Dom), which, given the current propagation stage (the propagator's priority), returns a propagator for that stage.

Several functions now have to be aware of the stage:

- head(Q) returns a pair $\langle p, i\rangle$ of a propagator and its priority.
- $cost(p, d', e)$ gets the event as an argument, so that staged propagators can determine the priority (which corresponds to the stage).
- nextStage(p, d', i) is a new function that returns either $\{\langle p, j\rangle\}$, scheduling p for another stage j, or the empty set, if no other stage needs to be run.

- $\mathrm{fix}(p(i), d')$ is given the actual propagator $p(i)$.

The result is called **prioritized, event-directed, dynamic, staged transition system** and looks as follows. Given an admissible space $\langle d, P, Q, deps \rangle$ with $d \neq \emptyset$ and $\langle p, i \rangle = \mathrm{head}(Q)$, there is a transition to the space $\langle d', P', Q', deps' \rangle$ such that

$$
\begin{aligned}
d' &= p(i)(d) \\
P' &= (P \setminus \mathrm{subsumed}(p, d)) \cup \mathrm{rewrite}(p, d) \\
deps' &= (deps \setminus \mathrm{cancel}(p, d)) \cup \mathrm{subscribe}(p, d) \\
Q' &= \mathrm{deq}(\mathrm{enq}(\mathrm{deq}(Q, p), \{\langle p', i' \rangle \mid \exists x \in X\ \exists e \in \mathrm{events}(d(x), d'(x)) : \\
&\qquad\qquad p' \in deps'(x)(e) \\
&\qquad\qquad \wedge\ i' = cost(p', d', e)\} \\
&\qquad\qquad \cup\ \mathrm{nextStage}(p, d', i)), \\
&\quad \mathrm{fix}(p(i), d'))
\end{aligned}
$$

5.5 Propagation Conditions and Modification Events

The preceding sections recapitulated traditional as well as more recent techniques for computing a mutual fixed point of a set of propagators. An important concept was the notion of *event-based scheduling*, where propagators are only scheduled if certain events occur.

In order to be useful in practice, event-based scheduling relies on two operations to be efficient: determining the set $\mathrm{events}(d(x), d'(x))$ when a variable domain has been modified from $d(x)$ to $d'(x)$, and determining the set of propagators that depend on these events.

In this section, we refine event-based scheduling so that these two operations can be implemented efficiently. As a first refinement, we introduce the notion of *propagation conditions*, which will be used to describe the events that propagators subscribe with. The second refinement deals with determining efficiently which events occurred. For this, we introduce *modification events*.

Propagation conditions

In the model as presented so far, propagators may be subscribed to a single variable with multiple events, as every propagator has to react to the `asn` event. This is because `asn` is the only event that is guaranteed to happen eventually, and every propagator has to be applied at least once to an assignment in order to impose its constraint. For example, it is not sufficient to only subscribe with an `lbc` event, signaling changes of the lower bound. If the variable is assigned by a change of the

upper bound, the propagator would not get scheduled. Another example is a propagator that can react to both lower and upper bound changes. It has to subscribe with both lbc and ubc.

In an implementation, it is beneficial to keep every propagator at most once in the dependency mapping for a variable, as this saves memory and a bookkeeping overhead. We therefore subscribe propagators with *sets of events* E, and establish a mapping $deps(x)(E) \subseteq P$. Some event sets are equivalent as it comes to propagator scheduling. In order to use event sets as an efficient index into the *deps* mapping, we consider them modulo this equivalence, and call the resulting equivalence classes *propagation conditions*.

For example, the event sets $\{\mathsf{lbc}, \mathsf{dmc}\}$ and $\{\mathsf{dmc}\}$ are equivalent, as a lbc event always implies that a dmc event has also happened. We say that an event e **implies** an event e' (written $e \to e'$) if and only if for all variable domains $d(x)$ and $d'(x)$, $e(d(x), d'(x))$ implies $e'(d(x), d'(x))$. Any event system always contains the asn event, or only a single dmc event (as in Example 5.6), in which case we can simply add asn. All propagators then depend on the asn event, as they must implement the decision procedure for their constraint on assignments. Together, this yields the following definition.

Definition 5.16 A **propagation condition** π is a set of events such that $\mathsf{asn} \in \pi$ and such that π is closed under the converse of implication: for any two events e and e', if $e \in \pi$ and $e' \to e$, then $e' \in \pi$. *

If a propagator subscribes to a variable, it can now do that using a *unique* propagation condition. The dependency mapping is defined in terms of propagation conditions instead of arbitrary event sets: $deps(x)(\pi) \subseteq P$. We have to adapt the dependency invariant accordingly:

$$p(d) = d \wedge d' \subseteq d \wedge p(d') \neq d$$
$$\Longrightarrow$$
$$\exists x \in X\ \exists \pi :\ \pi \cap \text{events}(d(x), d'(x)) \neq \emptyset \wedge p \in deps(x)(\pi)$$

The resulting transition systems remain the same. The number of distinct events is usually small, so the number of propagation conditions is small, too. Therefore, each propagation condition can be represented by a small integer number, which yields an efficient implementation of the mapping *deps*.

The implications between the events for integer variables from the introductory Example 5.7, together with the corresponding propagation conditions, appear in Figure 5.1.

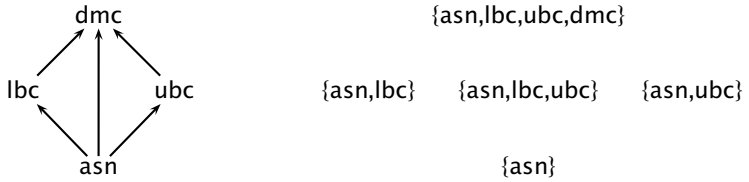


Figure 5.1: Implications between integer events, and the corresponding propagation conditions

Modification events

Event-directed scheduling requires determining the set events$(d(x), d'(x))$ when the domain changes from d to d'. Instead of computing this set from two given domains, an implementation will maintain a set of events incrementally during propagation. Such a set of events represents the modifications between two variable domains, and we call it a *modification event*.

Not all sets of events actually occur as modification events in practice. For instance, the set {lbc} cannot occur, as an lbc event always implies that a dmc event has happened, too. Furthermore, the asn event plays a special role, as it is contained in all propagation conditions. For scheduling, we therefore do not need both the sets {asn, lbc} and {asn, ubc}—just having {asn} is sufficient.

Definition 5.17 A **modification event** *me* is a set of events such that either $me = \{\mathsf{asn}\}$, or $\mathsf{asn} \notin me$ and *me* is closed under implication. *

The definition implies that modification events are closed under union. Again, as there are only few events in typical event systems, we can enumerate all modification events for a particular event system. An implementation can therefore represent modification events as small integers.

The last step towards event-based scheduling with propagation conditions and modification events is to introduce a function modifications(p, d), which returns a set of pairs $\langle x, me\rangle$ of the variables x modified when applying propagator p to domain d, and the corresponding modification events *me*. This function again lies on the boundary between kernel and domain module. The domain module determines which variables have been changed, and the kernel performs the required scheduling:

$$Q' = Q \setminus \{p\} \cup \{deps(x)(\pi) \mid \langle x, me\rangle \in \text{modifications}(p, d) \wedge \pi \cap me \neq \emptyset\}$$

We omit the straightforward generalization to prioritized or staged transition systems. As we will see in the next chapter, encoding both propagation conditions and modifications events as small integers yields efficient implementations of the mapping *deps* and the agenda update.

Our event system from Example 5.7 yields the following modification events:

$\mathsf{me}_{\mathrm{asn}}$	:= {asn}	(the variable is assigned)
$\mathsf{me}_{\mathrm{lbc}}$	:= {lbc, dmc}	(the lower bound changes, the variable is not assigned)
$\mathsf{me}_{\mathrm{ubc}}$	:= {ubc, dmc}	(the upper bound changes, the variable is not assigned)
$\mathsf{me}_{\mathrm{bbc}}$	:= {lbc, ubc, dmc}	(both bounds change, the variable is not assigned)
$\mathsf{me}_{\mathrm{inner}}$	:= {dmc}	(an inner value is removed, no bound changes)

5.6 Related Work

Most constraint solvers are based on propagator-centered, event-directed, prioritized propagation as presented in this chapter.

► SICStus Prolog (Carlsson et al., 1997) employs a priority queue of propagators, using two priority levels.

► Mozart (2009) maintains a two-level priority queue of propagators, and scheduling is based on events. Mozart offers additional priorities for non-monotonic propagators (as described by Müller, 2001): each non-monotonic propagator gets its own priority level, effectively fixing the order in which non-monotonic propagators are run and hence maintaining the guarantee to compute a unique fixed point.

► ECL^{i}PS^e (Wallace et al., 1997) has a feature called *suspension*, which attaches a Prolog goal to finite domain variables. When the variable domain changes, the goal, which may implement a propagator, is scheduled. The ECL^{i}PS^e system features twelve priority levels, but like SICStus and Mozart, its finite domain solver only makes use of two levels.

► B-Prolog (Zhou, 2006) queues *action rules*, which correspond to propagator invocations. A particularity of B-Prolog is that the same propagator can appear several times in the queue, once for each variable that triggered its scheduling.

► CHOCO (Laburthe, 2000) provides a sophisticated priority system with seven levels and both FIFO and LIFO scheduling, but is not propagator-centered, as explained below.

Digression: Variable-centered propagation

In our setup, the agenda holds the propagators that are not necessarily at a fixed point. Some solvers, notably ILOG Solver (2009), CHOCO (2009), and Minion (2009), use an alternative approach: an *agenda of modified variables* instead of an agenda of propagators. It is straightforward to build transition systems around this different kind of agenda, using an adjusted agenda invariant: For all variables $v \in X \setminus Q$, all

propagators subscribed to v are at a fixed point. A transition system that bases scheduling on an agenda of variables performs **variable-centered propagation**.

Both ILOG Solver and CHOCO actually implement a hybrid approach. When a modified variable is taken from the queue, its dependent propagators can either be run immediately, or put into a queue of propagators.

The advantage of variable-centered over propagator-centered propagation is that whenever a propagator is invoked, the information which variable exactly triggered the propagation is directly available. The propagator can take this information into account in order to compute the new domain *incrementally*, without recomputing from scratch. For example, a bounds($\mathbb{R}$)-complete propagator for the linear equation $y = \sum_{i=1}^{k} x_i$ typically computes the new lower bound of y as the sum of the lower bounds of the x_i. Incremental propagation in this case means that when some x_j is modified, the propagator can *adjust* the lower bound of y by the difference of the old and the new lower bound of x_j.

Lagerkvist and Schulte (2007) show how *advisors* can be used to implement incremental propagation in a propagator-centered system.

Other related work

► It is folklore knowledge that propagators should be scheduled in a FIFO fashion. Similarly, using events to prevent gratuitous scheduling of propagators has been used in constraint solvers for a long time—one can argue that it was already present in the early DPLL algorithm (Davis et al., 1962). Schulte and Stuckey (2008c) perform detailed experiments with different agenda strategies as well as priority queues, substantiating this folklore knowledge with empirical evidence. They also provide a comprehensive study of events, including a detailed experimental evaluation of different event schemes, fixed point reasoning, and staged propagation. Our formalization builds on their work. We embed their model into our setup of transition systems, and extend it with exact definitions of propagation conditions and modification events, to make it more directly suitable for implementation.

► The use of dynamic propagation conditions in Example 5.13 is closely related to the watched literals approach, which is used extensively in SAT solving to improve unit propagation (Moskewicz et al., 2001). The idea of watched literals has been transferred to constraint solving for the Minion system (Gent et al., 2006b). Dynamic dependencies as described here are not fully equivalent to watched literals, in that the latter are not reset upon backtracking, and, for an efficient implementation, require information about exactly which variable changed.

6 Implementing a Propagation Kernel

This chapter develops an implementation architecture for a propagation kernel based on the mathematical framework developed in the previous chapters.

A propagation kernel implements the infrastructure for constraint propagation. It controls the propagators and computes their fixed points, using the scheduling techniques developed in the previous chapter. Furthermore, it provides the facilities for backtracking search, as we will see in this chapter. These basic *services* are domain-independent. On top of the propagation kernel, domain modules implement the domain-specific parts of the constraint solver, such as variable domain data structures or propagation algorithms. The kernel provides the necessary *abstractions* that the domain-specific parts are built upon.

Constraint programming aims at solving real-world combinatorial problems. It is therefore essential to validate the presented models using an efficient implementation. The presented architecture has been implemented in the propagation kernel of the Gecode (2009) constraint solver, a modular, efficient open-source C++ library.

The contribution of this chapter is to not only present the design of a propagation kernel, but to make *justified design decisions*. We identify the performance-critical operations of the main data structures, the priority queue and the dependency mapping, and then choose the optimal implementation. An empirical evaluation validates the design decisions. A further contribution is to base the implementation architecture on strong invariants, established by *contracts* between propagators, variables, and the kernel. The invariants lead to a streamlined implementation and a clear separation of concerns between the kernel and the domain modules.

Structure of the chapter. A central decision in the design of a propagation kernel is whether to use trailing or copying and recomputation for backtracking during search. The chapter starts by reviewing the two strategies (6.1). Next, we introduce the basic object-oriented architecture (6.2). We then follow a top-down approach, first showing how domain modules are implemented on top of the kernel (6.3), and afterwards presenting the design of the kernel components realizing dependency management (6.4), the priority queue (6.5), and the overall control (6.6). The final component of the kernel deals with copying and memory management (6.7). Finally, we discuss how this kernel architecture is realized in the Gecode library (6.8), and present an empirical evaluation of some of our design decisions (6.9).

6.1 Copying Versus Trailing

When designing a propagation kernel, there is one decision that influences the entire architecture: how the state of the system is represented. The previous chapters captured the state of a transition system in a *space*.

For the mathematical model, it is convenient to regard spaces as pure, immutable, mathematical objects. Operations on spaces are functions, transitions transform one space into another space, and propagation takes a domain and yields a stronger domain. The result is a compact model that allows us to prove important properties, as we have seen.

In order to achieve high performance, the architecture we describe in this chapter implements all the components of a propagation-based solver as stateful objects. Spaces, variables, propagators, queues, and all other data structures that comprise the solver provide operations that destructively update the state. This behavior suits most parts of the constraint solver well: after applying a propagator, the previous domain has become dispensable, as the new domain still contains all solutions. When updating a queue, the old queue can be discarded. A subsumed propagator can be safely deleted, since it will never contribute to propagation again.

Search, however, is the exception. We have to be able to undo a choice (and all its consequences due to propagation) and try a different branch instead. Let us reconsider the main search algorithm from Section 3.3, formulated in terms of spaces instead of propagation problems:

```
SOLVE(⟨d, P, Q, deps⟩)
1  ⟨d′, Q′, P′, deps′⟩ ← PROPAGATE(⟨d, P, Q, deps⟩)
2  if d′ = ∅ then return ∅
3  if d′ = {a} then return {a}
4  ⟨s1, s2⟩ ← BRANCH(⟨d′, Q′, P′, deps′⟩)
5  return SOLVE(s1) ∪ SOLVE(s2)
```

Here, the functional formulation makes it easy to create two branches (line 4), which are then recursively solved. But if spaces are stateful data structures, and propagation and branching destructively update a space, we need a mechanism that *undoes* the destructive updates. This is called **backtracking**.

Backtracking

Two basic techniques for backtracking exist: trailing, and copying with recomputation. Trailing records all destructive updates on a stack, the *trail*. On backtracking,

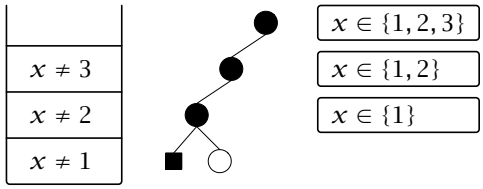


Figure 6.1: Trailing (left) versus copying (right)

the trail provides enough information to restore a previous state. A copying system, on the other hand, keeps full copies of the state from time to time, and redoes *(recomputes)* the remaining steps.

Figure 6.1 depicts the difference between backtracking using trailing (left) and copying (right). The picture shows a partial search tree, with the filled circles representing choice nodes, and the filled square representing a failure. The search has to backtrack to the last choice, and then take the other branch that leads to the white, unexplored node.

The trail, appearing on the left of the search tree, records changes made to the variable domains. Backtracking now undoes all the changes back to the last choice. A copying system stores complete copies of the space at each choice point, as shown to the right of the tree. Backtracking can simply discard the current, failed space, and restart from the copy at the previous choice. With recomputation, we do not place a copy at each choice node, but only at some choice nodes (always including the root node). Upon backtracking, the deepest copy is copied again, and the remaining steps, for which no copy is available, are simply recomputed.

There is in fact more to backtrack than just the variable domains. For example, spaces as defined in the previous chapter additionally contain the dependencies, the set of propagators, and the queues. Finally, some propagation algorithms such as for the *all-different* constraint (Régin, 1994) maintain complex internal data structures for incremental propagation.

Both copying with recomputation and trailing have advantages and disadvantages. For example, a system based on copying and recomputation can easily support arbitrary exploration strategies during search, like breadth-first search, as several nodes of the search tree can be kept "open" at once. Furthermore, copying and recomputation is naturally well-suited for concurrent (multi-threaded) search, and it supports fine tuning of required memory versus run-time. Trailing, on the other hand, is better suited for small variable domains that are not modified often (like Boolean variables), as in this case it requires very little memory.

As mentioned earlier, the choice how to represent the state of a propagation-based

system has a fundamental influence on the entire architecture. The implementation architecture described in this chapter, and consequently the architecture implemented in Gecode, is based on copying and recomputation. It is not a goal of this dissertation to discuss or evaluate this decision. Rather, we develop efficient data structures and algorithms for a propagation-based system *that is based on copying and recomputation*.

The following code implements a simple stateful search. Propagation now updates the space in place, and backtracking is based on copying. Instead of the BRANCH procedure, we use COMMIT to implement the branching. Committing to an alternative (here: 1 or 2) means modifying the domain and/or propagators in the space to represent that alternative.

```
SOLVE(⟨d, P, Q, deps⟩)
1  PROPAGATE(⟨d, P, Q, deps⟩)
2  if d = ∅ then return ∅
3  if d = {a} then return {a}
4  c ← COPY(⟨d, Q, P, deps⟩)
5  COMMIT(⟨d, P, Q, deps⟩ , 1)
6  COMMIT(c, 2)
7  return SOLVE(⟨d, P, Q, deps⟩) ∪ SOLVE(c)
```

Recomputation and non-monotonicity

Due to non-monotonicity, the result of recomputation may be a different fixed point than before. Assume that in Figure 6.1, only a copy of the root node is available when backtracking from the failed node. Recomputation proceeds by creating a copy of the root node copy, and redoing the remaining steps, committing twice to the first alternative.

If now the meaning of *first alternative* depends on the actual space (for example if a fail-first heuristic is used), non-monotonic propagators can can result in a completely different choice being made during recomputation than during the original exploration of that node. The search would possibly be incomplete.

Mozart (2009) solves this problem by fixing the order of application for non-monotonic propagators, so that fixed points remain unique (Müller, 2001). The alternative is to make the meaning of *n-th alternative* independent of the actual space. When a node is explored for the first time, a **branching description** is extracted from the space that describes which *constraints* have to be added to the space for the individual branches. This description is used for committing during recomputation.

Thus, during recomputation, the same set of constraints is imposed on a node as during its original exploration. Soundness of propagation then guarantees that the result of recomputation represents the same set of solutions, which is sufficient

for soundness and completeness of the search. As an additional advantage, only one fixed point needs to be computed during recomputation. Gecode implements this technique, which is known as **batch recomputation** and was first described by Choi et al. (2001).

Related work

► Trailing is a key technique in Warren's Abstract Machine (Warren, 1983; Ait-Kaci, 1991), the virtual machine model that most Prolog implementations are based on.

► A trailing system can deal with open nodes using recomputation, as described by Perron (1999), and that way implement more elaborate search strategies.

► The trail keeps a record of the order in which variables were modified, which can be useful for techniques like conflict analysis (Marques-Silva and Sakallah, 1996).

► As an interesting historical note, already the early DPLL algorithm (Davis et al., 1962) implemented backtracking by placing a copy of the state on magnetic tape. Depth first search was the result of organizing the records on tape "in the cafeterial stack-of-plates scheme: the last record written is the first to be read."

► The first constraint programming system that was based on copying was AKL (Janson and Haridi, 1991; Carlson et al., 1994b; Janson, 1994). The experiences from AKL were refined in the Mozart (2009) programming system, which introduced first-class computation spaces and was based on copying and recomputation, investigated in detail (including a comparison with trailing) by Schulte (1999, 2002).

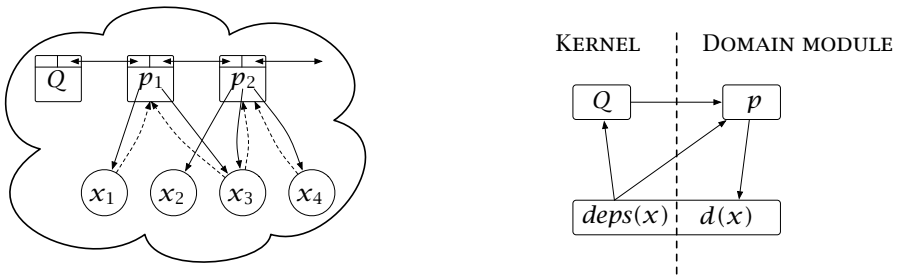
6.2 An Object-Oriented Design

This section presents the basic object model for a propagation-based constraint solver. The rest of this chapter develops the details of this model.

Propagators, variables, queues, dependencies

A propagation-based solver, as presented in the previous chapter, deals with the components of spaces: propagators, variables, priority queues, and dependencies. In an object-oriented architecture, each of these will be realized by or encapsulated in a stateful object.

Propagator objects encapsulate references to the variable objects they deal with, and possibly internal data structures required for the propagation algorithms. Variable objects encapsulate the variable domain data structure, and store the dependencies as references to propagators. Furthermore, propagator objects are kept in a data structure implementing the priority queue.



(a) A space

(b) The kernel/domain boundary

Figure 6.2: The basic object model

In addition to encapsulating the state, the objects provide *methods* that implement the required operations. Variables provide methods for accessing and updating the domain data structure, as well as for managing the dependencies. Propagators have a `propagate` method that uses the domain operations of the variable objects to implement the actual propagation algorithm. A control logic implements the main propagation loop, taking propagators from the queue, invoking them, and scheduling new propagators, until a fixed point has been reached.

This is the standard way of modeling propagators and variables as objects, and corresponds to the models of Puget and Leconte (1995) as well as Laburthe (2000).

As discussed in Section 6.1, our model will be based on copying. We therefore introduce another type of object, representing *spaces*, and encapsulating the complete state of the solver. Spaces thus contain variables, propagators, queues, and dependencies. Figure 6.2(a) shows a diagram of a space. Variables appear as circles, propagators as boxes, references from propagators to variables as arrows, and dependencies as dotted arrows. The box labeled Q, together with the arrows between the propagators, stands for the priority queue and will be explained in Section 6.5.

All the core functionality is provided as methods on spaces, the most important one being the method that applies the propagators until they are at a mutual fixed point and hence the space is stable. Furthermore, a space object provides a method that produces a copy of itself, so that search can be implemented as shown in the previous section.

Laburthe (2000) already identified the necessity to distinguish between the domain-independent parts of the solver, such as scheduling and dependency management, and the domain-specific parts, such as the actual domain representation and propagation algorithms. This distinction results in a modular system, to which new propagation algorithms or new types of variable domains can be added. As a side effect, it also forces us to design a clean architecture with well-defined interfaces.

We distinguish between the *propagation kernel*, implementing the domain-independent parts of the solver, and the *domain modules*, which provide the domain-specific parts. The priority queue, the dependencies, and the overall control that performs scheduling and propagator invocation are part of the propagation kernel. The actual propagation algorithms, the domain data structures, and the event systems are realized in domain modules. Figure 6.2(b) depicts this boundary. We will now see how this architecture maps to the implementation level.

Kernel and Domain Modules

The interface between kernel and domain modules is bi-directional. For instance, the kernel must be able to invoke the `propagate` methods, and on the other hand a variable modification in the domain module must trigger the insertion of propagators into the priority queue. To achieve this bi-directionality, the kernel defines *base classes* for variables and propagators. The domain modules then implement concrete variables and propagators on top of these abstractions. The propagator base class defines *virtual methods* that the kernel can call, and by dynamic binding, the concrete implementation in the domain module is used. At the same time, both base classes as well as the space objects define *services*, in the form of methods that the domain module can call to access kernel functionality.

- Variable base classes provide the data structure and methods for the dependencies of the particular variable. The domain module, implementing the event system, uses this functionality to schedule dependent propagators when the variable domain is modified.
- Propagator base classes define the virtual `propagate` method that the kernel uses to perform the actual propagation. Furthermore, they contain parts of the priority queue data structure.
- The space objects provide methods for rescheduling propagators. These are used by propagators for reporting fixed points or subsumption, and for staging.

For the rest of the chapter, we follow a top-down approach. The next section shows how domain modules interact with the kernel. Afterwards, we will design and discuss the kernel functionality in detail. Finally, all objects contain additional functionality for copying, which will be the topic of Section 6.7.

Related work

► Implementing a constraint solver in an object-oriented programming language was pioneered by Puget and Leconte (1995), who describe a C++ architecture for constraint solving. Their work is the basis of ILOG Solver (2009), which provides a feature-rich and efficient solver as a C++ library.

► Laburthe (2000) describes the design of the CHOCO constraint kernel, which introduced the notion of a generic, that is, domain-independent kernel, and an expressive event system. CHOCO (2009) was originally implemented in CLAIRE (Caseau and Laburthe, 1996) and later reimplemented as a Java library. Both ILOG Solver and CHOCO are based on trailing and implement variable-directed propagation.

► The Mozart (2009) constraint kernel (described by Müller, 2001) is built into the virtual machine of the Mozart programming system, and implemented in C++. The Oz programming language (which Mozart implements) is a concurrent language, which made the development of copying and recomputation for backtracking necessary (Schulte, 2002).

► The Minion (2009) constraint solver (Gent et al., 2006a) is implemented in C++ and based on copying for search. It uses a variable-centered scheduling scheme. Minion does not provide a domain-independent kernel, but it comes with different implementations for integer variables hard-wired into the system.

► While there are highly specialized solvers with a single sort of variables, such as MiniSat (2009) (described by Eén and Sörensson, 2004) for solving Boolean satisfiability problems, many solvers incorporate different variable domain types such as Boolean variables, finite domain integers, and sets. There is ongoing work on other domain types such as real intervals (for an overview see Benhamou and Granvilliers, 2006), multi-sets (Kiziltan and Walsh, 2002), or even graphs (Dooms et al., 2005; Dooms, 2006). A modular solver based on a domain-independent kernel provides an ideal test platform for novel types of variable domains.

6.3 Domain Modules

This section designs the interaction between the domain modules and the kernel.

For now, we regard the kernel as a black box, providing the scheduling and the dependency management. We will explain how propagators use variable domain operations to perform propagation, and how variable domain operations generate modification events and pass them to the kernel and back to the propagator. Based on this architecture, the later sections design the kernel components in detail.

The architecture is based on **contracts** between propagators, variables, and the kernel, which means that we require them to fulfill certain strong invariants. The invariants ensured by the contracts result in a streamlined and efficient implementation, and a clear separation of concerns. An example for such a contract is that propagators must return a status flag to the kernel, signaling whether failure has occurred.

The result is an efficient test for failure in the kernel. We will see more contracts throughout this section.

Domain operations

The variable objects provide methods for accessing and updating the variable domain. Propagators use these methods to implement the actual propagation algorithm.

We will use a simple implementation of integer variables as an example throughout this chapter. For accessing the domain of a variable x, the variable object provides methods x.`min`() and x.`max`(), returning the current minimum and maximum of the domain.

The domain operations for update are more complicated. For integer variables, let us assume that the two methods x.`adjmin`(i) and x.`adjmax`(i) update the minimum and maximum of the variable domain, respectively. Updating a variable domain has one of three effects:

1. The update results in no change at all. For the integer variable example, when the current minimum in the domain of x is 4, then x.`adjmin`(3) does not change the domain.
2. The update results in a failed domain.
3. The update prunes the domain.

We now define the contract between the domain update operations, the propagators, and the kernel. For each of the three cases, the method implementing the update operation must perform different tasks. In any case, it must inform the propagator about the new state of the domain. We will see shortly that this information is useful for determining the propagator's fixed point status.

1. If no change happens, the method returns the empty set to the propagator.
2. If the update results in failure, the method returns FAIL to the propagator.
3. If the update results in a domain change, the method determines the corresponding modification event *me*. It then calls a method `notify`(*me*), which takes care of scheduling the propagators depending on *me*. This method is also part of the domain module, and will be described in detail below. Finally, *me* is returned to the propagator.

Propagation status and fixed point reasoning

The next contract we have a closer look at controls the interaction between propagators and the kernel. The kernel invokes a propagator's `propagate` method. At the end of this method, the propagator must notify the kernel of its *status:* whether it has detected failure or not, and whether it has reached a fixed point or subsumption. The following example illustrates this contract.

Example 6.1 (A less-than propagator) The `propagate` method of a propagator for the constraint $[\![x < y]\!]$ can be implemented as follows.

```
p.propagate()
1  if x.adjmax(y.max() − 1) = FAIL then return FAIL
2  if y.adjmin(x.min() + 1) = FAIL then return FAIL
3  if x.max() < y.min() then subsumed() else fix()
4  return OK
```
*

The propagator must pass failure on to the kernel by returning FAIL (lines 1 and 2). Otherwise, the propagator must signal success to the kernel by returning OK (line 4). When the propagator detects failure, it can stop its execution immediately and return control to the kernel. Failure occurs frequently—assuming a binary search tree, usually more than half of the nodes are failed, so this optimization is important.

Furthermore, the propagator must notify the kernel of its fixed point status. The model from Section 5.4 uses a function fix to determine whether a propagator is at a fixed point. In the implementation model, the propagator itself is responsible for doing so, by calling one of three methods to notify the kernel:

1. If the propagator can determine that it has computed a fixed point, that is, if $\text{fix}(p, d') = \{p\}$ in the model from Section 5.4, it can be removed from the queue. In this case, the propagator calls the kernel method `fix`.
2. If the propagator detects subsumption, it calls the kernel method `subsumed`, which behaves like `fix`, but additionally removes the propagator, freeing its memory and canceling its subscriptions (as we will see in Section 6.4).
3. In all other cases ($\text{fix}(p, d') = \emptyset$ in the model), the propagator calls the kernel method `nofix`. This only means that the propagator is *possibly* not at a fixed point. When `nofix` is called, the kernel hence determines whether the propagator has modified its own variables, in which case it is left in the queue. Otherwise, it is at a fixed point and removed from the queue.

Note that a propagator stays in the queue until one of the kernel methods `subsumed`, `fix`, or `nofix` removes it. This implements the efficient scheduling mentioned in Section 5.4, avoiding to potentially remove, add, and again remove a propagator from the queue.

In order to determine its fixed point status, the propagator can use the status returned by the variable domain operations. The following example shows how this information can be put to use.

Example 6.2 (Iteration to fixed point) A bounds propagator for the equality constraint $[\![x = y]\!]$ can iterate until fixed point using the domain operation status.

```
p.propagate()
 1  repeat  me ← x.adjmin(y.min())          ▷ iterate until ...
 2          if me = FAIL then return FAIL
 3          me ← me ∪ y.adjmin(x.min())
 4          if me = FAIL then return FAIL
 5          me ← me ∪ x.adjmax(y.max())
 6          if me = FAIL then return FAIL
 7          me ← me ∪ y.adjmax(x.max())
 8          if me = FAIL then return FAIL
 9    until me = ∅                          ▷ ... no more changes, fixed point
10  fix()
11  return OK
```

Note that without the iteration, the propagator would not be idempotent. For instance, in a domain d with $d(x) = \{2,3\}$ and $d(y) = \{1,3\}$, one iteration of the **repeat** loop would result in a domain d' where $d'(x) = \{2,3\}$ and $d'(y) = \{3\}$, which is clearly not a fixed point. The propagator has "fallen into a domain hole". More complex propagators can use the concrete modification events to decide whether a fixed point has been reached, for instance, if no `asn` event has happened. *

The event system

We have just seen how propagation algorithms are implemented in terms of the domain access and update operations of variable objects. Updating a variable domain causes an *event*, and must consequently result in the scheduling of dependent propagators. The domain module must implement the event system and schedule propagators using methods provided by the kernel.

The event system is based on modification events, which are represented as small integers. Two operations are needed on modification events: computing the union of two modification events (for maintaining the modification event delta), and determining the propagation conditions that overlap with a certain modification event (for scheduling).

Section 5.5 already mentioned that modification events are closed under union, and that they are mapped to integers in the implementation. The first operation, computing $me_1 \cup me_2$, can thus be implemented by a simple table lookup. Let us look at the event system from Example 5.7 again. It has five modification events, $\mathsf{me}_{\mathsf{asn}}$,

me_{asn}	me_{asn}				
me_{lbc}	me_{asn}	me_{lbc}			
me_{ubc}	me_{asn}	me_{bbc}	me_{ubc}		
me_{bbc}	me_{asn}	me_{bbc}	me_{bbc}	me_{bbc}	
me_{inner}	me_{asn}	me_{lbc}	me_{ubc}	me_{bbc}	me_{inner}
	me_{asn}	me_{lbc}	me_{ubc}	me_{bbc}	me_{inner}

Table 6.1: Table for computing $me_1 \cup me_2$

me_{lbc}, me_{ubc}, me_{bbc}, and me_{inner}. Table 6.1 shows the lookup table for this event system.

The second operation, determining the propagation conditions that overlap with a given modification event, is embedded into the method `notify` we already mentioned above. When a domain update method actually modifies the domain, it determines the corresponding modification event *me* and calls `notify`(*me*).

The task of the `notify`(*me*) method is to schedule all propagators that are subscribed to the variable with a propagation condition π where $me \cap \pi \neq \emptyset$ (see Section 5.2). The set of propagation conditions $\{\pi_0, \ldots, \pi_{k-1}\}$ is, like modification events, fixed and small, so we identify each propagation condition π_i with the integer i. The `notify`(*me*) method then uses the kernel method `schedule`(π_i, π_j, me), which is provided by the variable base class. This method schedules all propagators that are subscribed to the variable with propagation conditions $\pi_i, \ldots, \pi_j$. The third argument, the modification event *me*, is needed for staging and will be discussed shortly.

We explain the `notify` method using the event system from Example 5.7 again. It has five propagation conditions:

$$\pi_1 = \{\mathsf{asn}\}$$
$$\pi_2 = \{\mathsf{asn}, \mathsf{lbc}\} \qquad \pi_3 = \{\mathsf{asn}, \mathsf{lbc}, \mathsf{ubc}\} \qquad \pi_4 = \{\mathsf{asn}, \mathsf{ubc}\}$$
$$\pi_5 = \{\mathsf{asn}, \mathsf{lbc}, \mathsf{ubc}, \mathsf{dmc}\}$$

Calling `notify`(*me*) schedules the propagators that are subscribed with any propagation condition π that overlaps with the modification event *me*, $me \cap \pi \neq \emptyset$. For the example event system, the following table shows which modification events overlap with which propagation conditions.

$$me_{asn} : \pi_1, \pi_2, \pi_3, \pi_4, \pi_5 \qquad me_{lbc} : \pi_2, \pi_3, \pi_5$$
$$me_{ubc} : \pi_3, \pi_4, \pi_5 \qquad me_{inner} : \pi_5$$

This mapping yields the following implementation of `notify`. Note that most sets of propagation conditions require just a single call to `schedule`. Only for me_{lbc} (line 3), two calls are needed, excluding the propagators subscribed with π_4.

```
notify(me)
1  case me of
2     me_asn   then schedule(π1, π5, me)
3     me_lbc   then schedule(π2, π3, me); schedule(π5, π5, me)
4     me_ubc   then schedule(π3, π5, me)
5     me_inner then schedule(π5, π5, me)
```

Delaying the scheduling. The `notify` method as presented above immediately schedules all dependent propagators when a variable domain is modified. This has the disadvantage that if a propagation algorithm modifies the same variable several times during a single run, the same propagators will be scheduled several times, too. In order to avoid this, one could instead make `notify` collect all modified variables in a list, and then let the kernel perform the scheduling after the propagator has finished. However, this requires additional infrastructure for collecting modified variables, and the potential performance gain in practice is low, as the experiments in Section 6.9 suggest.

Generating the event system. The event system part of the domain module only depends on the event system that is used. In order to simplify the implementation of a domain module, we *generate* the table for computing the union of two modification events, as well as the implementation of the `notify` method, from a simple specification of the event system.

Priorities and propagator staging

With the event system in place, a slight extension of the fixed point reasoning mechanism yields an implementation of propagator staging.

Staging, as introduced in Section 5.4, combines different propagation algorithms (stages) in a single propagator. Which stage is executed depends on the events that caused scheduling of the propagator.

The implementation handles staging slightly differently than the model presented earlier: instead of determining the stage based on the current priority, we use the set of events that has happened since the last invocation of the propagator. We call this set the **modification event delta**, and collect it for each propagator object p in a field $p.\Delta me$. We will see in Section 6.4 how the modification event delta is maintained by the kernel.

Before invoking the `propagate` method of a propagator p, the kernel empties the set $p.\Delta me$. The old set is passed to the `propagate` method as an argument and is used to determine the stage to execute. For example, a propagator with a bounds-complete and a domain-complete stage would execute the bounds-complete stage whenever there are bounds events in Δme, and the domain-complete stage otherwise.

This corresponds to the model we saw before, as a bounds event results in the propagator being scheduled at a different priority than a domain change event.

The domain module is also responsible for determining the priority of a propagator. For this, the propagator base class provides another virtual method, `cost`(Δme). A propagator must implement this method and compute the cost based on the modification event delta and the current variable domains.

After propagation, the `propagate` method may have to reschedule the propagator for the remaining stage. As with fixed point reasoning and subsumption, the kernel provides two methods that perform the scheduling:

- `nofixPartial`($\Delta me'$) sets $p.\Delta me \leftarrow p.\Delta me \cup \Delta me'$ and then reschedules the propagator.
- `fixPartial`($\Delta me'$) sets $p.\Delta me \leftarrow \Delta me'$ and then reschedules the propagator.

Both methods get the modification event that the propagator did not handle as their argument. The difference is that `nofixPartial` keeps the modification event delta set that was accumulated during propagation, while `fixPartial` only uses the remaining modification event $\Delta me'$. A propagator that invokes `fixPartial` thus has computed a *partial fixed point* for a subset of the events that caused propagation. If a propagator invokes `nofixPartial`, it is not sure to have computed such a partial fixed point.

Example 6.3 (Staged multiplication) Bounds-complete multiplication is algorithmically much cheaper than domain-complete propagation for the same constraint. It pays off to implement a domain-complete propagator using staging. Given two helper methods `propBnd` and `propDom` that perform the actual bounds and domain propagation, respectively, the staging logic is the following:

- If the propagator is scheduled because of the event `asn` or `bnd`, it is put in stage A and gets priority `ternary`.
- If the propagator is scheduled because of the event `dmc` (and not `asn` or `bnd`), and it is not already in stage A, it is put in stage B and gets priority `veryslow`.
- When the propagator is executed in stage A, it uses `propBnd`, which does not compute a fixed point of the bounds propagation. The propagator stages itself using `nofixPartial`. It has consumed all events except `dmc`.
- When the propagator is executed in stage B, it uses `propDom`, and signals a fixed point using `fix`.

This is the code for a staged multiplication propagator.

```
p.propagate(Δme)
1  if {bnd, asn} ∩ Δme ≠ ∅
2     then if propBnd() = FAIL then return FAIL;                    ▷ Stage A
3          nofixPartial(Δme \ {bnd})
4     else if propDom() = FAIL then return FAIL;                    ▷ Stage B
5          fix()
6  return OK
```

```
p.cost(Δme)
1  if {bnd, asn} ∩ Δme ≠ ∅ then return ternary else return veryslow   *
```

The modification event delta serves two more purposes apart from staging. First, the `nofix` method uses it to find out whether the propagator modified its own variables—if it did, the modification event delta is not empty, and the propagator must remain in the queue. The second purpose is that propagators can obtain partial information about what kind of domain change caused their scheduling. For example, if the modification event delta is $\mathsf{me}_{\mathrm{asn}}$, the propagator knows that at least one variable has been assigned, and can possibly perform particular propagation steps in that case. However, the information is not very accurate, as the events of *all* the variables are collected in a single set. More accurate information can be provided in a propagator-centered solver by adding a mechanism called *advisors*, as introduced by Lagerkvist and Schulte (2007).

6.4 Dependency Management

This section develops the first of the kernel data structures, the dependency mapping. In the mathematical model, we used a function $\mathit{deps}(x)(\pi)$, which yields the set of propagators that are subscribed to variable x with propagation condition π. In the implementation model, the dependencies are encapsulated in the variable objects, and implemented in the variable base class.

Design rationale

The data structure for the dependencies must provide three operations:

- `subscribe`(p, π) adds the propagator p to the dependencies at propagation condition π.
- `cancel`(p, π) removes the propagator p from the dependencies at propagation condition π.
- `schedule`$(\pi_i, \pi_j, \mathit{me})$ iterates over the propagators between propagation conditions π_i and π_j, schedules them, and adds *me* to each scheduled propagator's modification event delta.

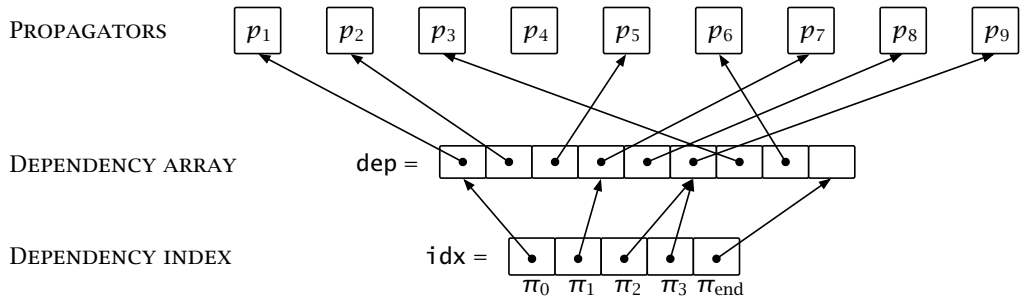


Figure 6.3: Dependency data structures

The most important operation is iteration. It is performed whenever an event happens, so we will design the data structure to be as efficient as possible in this case. For subscription and canceling, efficiency is not quite as important, as they happen less frequently.

We enforce a strong contract between propagators and the dependencies. A propagator must not cancel subscriptions that it has not established before, and it must cancel all its subscriptions when it reports subsumption. We will discuss below why the invariants enforced by this contract are important.

Indexed dependency arrays

The most efficient data structure for fast iteration is an array. So, in principle, we could have one array of propagators per propagation condition. However, in practice a single modification event often triggers several propagation conditions, as can be seen in the implementation of `notify` from the previous section.

The dependencies are therefore stored in a single *dependency array* `dep`, sorted by propagation condition. In addition, we maintain the *dependency index* `idx`, which partitions the dependency array by propagation condition. Figure 6.3 shows this architecture.

For each propagation condition π_i, the dependency index points to the first propagator in the dependency array that is subscribed with π_i. For example, the first propagator subscribed with π_1 in Figure 6.3 is $\texttt{dep}[\texttt{idx}[\pi_1]] = p_7$. To iterate over all propagators subscribed with a certain propagation condition π_i, we start at $\texttt{dep}[\texttt{idx}[\pi_i]]$ and finish at $\texttt{dep}[\texttt{idx}[\pi_{i+1}] - 1]$. There is one additional propagation condition, π_{end}, so that π_{i+1} and $\texttt{idx}[\pi_{i+1}]$ are defined for all propagation conditions π_i.

Again for the example in Figure 6.3, scheduling all propagators that are subscribed with propagation condition π_1 would amount to scheduling the propagators p_7 and p_8. No propagator is subscribed with π_2 (as $\texttt{idx}[\pi_2] = \texttt{idx}[\pi_3]$). Through this index

data structure, iterating over all propagators subscribed with a particular propagation condition is as efficient as possible, taking constant time per propagator, and with low constants in practice.

We can now define the method `schedule`(π_i, π_j, me), which schedules all propagators starting at propagation condition π_i and finishing at propagation condition π_j. In addition, it adds the modification event *me* to each propagator's modification event delta. For the above example, `schedule`(π_0, π_2, me) would thus schedule p_1, p_2, p_5, p_7, and p_8. The following code implements `schedule`, assuming that the kernel provides a method `enqueue` that puts a propagator into the right queue:

```
schedule(π_i, π_j, me)
1  for k ← idx[π_i] to idx[π_{j+1}] − 1
2      do if me ⊄ dep[k].Δme then
3          dep[k].Δme = dep[k].Δme ∪ me
4          enqueue(dep[k])
```

Line 2 makes sure that a propagator is only added to the queue if the new modification event *me* is not already contained in the propagator's modification event delta. This is correct because if $me \subseteq \texttt{dep}[k].\Delta me$, then the propagator has already been put into the queue before. We perform this optimization because `enqueue` must determine the propagator's priority by calling its `cost` method, which can be avoided with this simple test. However, the cost will thus only be re-evaluated when *new* events occur. From our experience, this does not represent a limitation in practice.

Subscribing and canceling

The remaining operations to be defined for the dependency data structure are subscribing and canceling. Subscribing a propagator p with propagation condition π_i means adding it at the appropriate position to the dependency array and modifying the index accordingly. Assuming that the dependency array is resized dynamically, subscription can be implemented to have amortized run-time $O(k - i)$ as shown in Figure 6.4(a). First, some space is cleared for the new subscription at `dep[idx[`π_i`]]` (lines 1–3). Then the new subscription is entered (line 4).

Canceling a subscription can be implemented to run in $O(\texttt{idx}[\pi_{i+1}] - \texttt{idx}[\pi_i] + i)$ (Figure 6.4(b)). The **while** loop in line 2 finds the index of p in the dependency array (note that the loop is only correct if the propagator is actually subscribed to the variable). After finding the index j_p, the position $\texttt{dep}[j_p]$ is reused (lines 3–7).

```
subscribe(p, π_i)
1  for j ← k downto i
2      do dep[idx[π_{j+1}]] ← dep[idx[π_j]]
3         idx[π_{j+1}] ← idx[π_{j+1}] + 1
4  dep[idx[π_i]] ← p
```

(a)

```
cancel(p, π_i)
1  j_p ← idx[π_i]
2  while dep[j_p] ≠ p do j_p ← j_p + 1
3  dep[j_p] ← dep[idx[π_{i+1}] − 1]
4  for j ← i + 1 to k
5      do dep[idx[j] − 1] ← dep[idx[π_{j+1}] − 1]
6         idx[π_j] ← idx[π_j] − 1
7  idx[π_end] ← idx[π_end] − 1
```

(b)

Figure 6.4: Subscribing (a) and canceling (b)

Propagator creation and destruction

Most subscriptions are created when the problem is set up. Each propagator has a *constructor* that initializes the internal data structures and subscribes to the variables. In addition, the propagator must be scheduled. This ensures that the dependency invariant (see Section 5.2) holds even before the first fixed point is computed. If the propagator subscribes with propagation condition {`asn`}, it only needs scheduling if the variable is already assigned.

During the lifetime of the propagator, the `propagate` method may create and cancel subscriptions, performing dynamic dependency updates as described in Section 5.3.

When a propagator reports subsumption using the kernel method `subsumed`, its lifetime ends. The propagator must implement another virtual method of the propagator base class, `dispose`, which is called by the kernel before it deallocates the propagator object. In this method, the propagator must cancel all its subscriptions. We will now see that this contract is crucial for making subsumption efficient.

Efficient subsumption

The whole point of detecting subsumption is to remove subsumed propagators from the space. Not only does this save memory in the current space, the propagators also do not have to be copied, saving run-time during copying and memory in the copies. Section 6.9 evaluates empirically how subsumption detection influences the performance. Another argument for removing subsumed propagators is that the graph formed by the dependencies between variables and propagators becomes a more accurate representation of the current state. This graph can be used for analyses like the ones presented by Schulte and Stuckey (2008a,b) or Mann et al. (2008).

A propagator can only be removed if there are no more references pointing at it, as otherwise these references would turn into *dangling pointers*. The scheme we

propose, which requires propagators to cancel their subscriptions, is a simple and efficient solution for this problem.

If the propagators were not responsible for canceling their subscriptions, we would need a much more complicated mechanism. For example, one could mark propagators as subsumed, and then remove subscriptions when a canceled propagator is found during scheduling. This would require some form of garbage collection in order to remove the propagator when the last subscription has been canceled.

We consider subsumption detection so important that we require all propagators to report subsumption at the latest when all their variables are assigned. Again, this is a contract between the propagator and the kernel. Alternatively, the kernel could check this property and remove propagators automatically, possibly making it slightly easier to implement a propagator. However, this would require domain knowledge in the kernel, and be less efficient for propagators that can detect subsumption earlier.

There are two situations in which it is not necessary to cancel subscriptions. Subscriptions to assigned variables do not have to be canceled, because on assigned variables, no further events will occur and thus no scheduling can happen. The `cancel` method therefore simply does nothing on assigned variables (and neither does `subscribe`). The second situation is when a space is failed. Failed spaces are simply discarded, including all propagators and their dependencies. This optimization will be evaluated in Section 6.9.

6.5 The Priority Queue

This section designs the second important kernel data structure, the priority queue. For our purposes, the priority queue must provide three operations:

- `enqueue`(p) adds propagator p, determining its priority using its `cost` method. If p is already in the queue, it is re-prioritized according to its current cost.
- `head`() returns the oldest propagator at the highest priority.
- `idle`(p) removes propagator p from the queue. This method is used internally by `fix` and similar methods.

All three operations are performed extremely often during the fixed point computation, and are hence crucial for the solver's performance.

Common algorithms for priority queues are based on variations of the *heap* data structure (see for example Mehlhorn and Näher, 1999; Cormen et al., 2001). Heaps support an arbitrary number of priority levels. For most types of heaps, the runtime complexity of the enqueue and dequeue operations depends on the number of

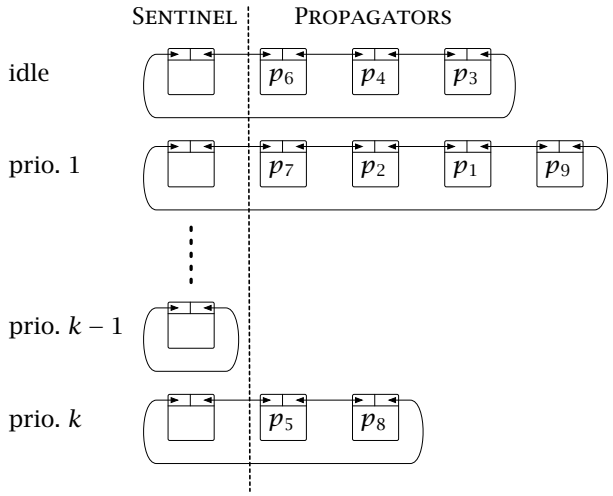


Figure 6.5: Propagators in prioritized queues

elements in the queue. For example, using binary heaps, both operations require time in $O(\log n)$ if n is the number of elements in the queue.

In order to make enqueue and dequeue as efficient as possible, we restrict the number of priority levels to a small, fixed set of integers. Then, a priority queue based on *buckets* can be used (see Mehlhorn and Näher, 1999), providing constant-time enqueue and dequeue operations. We will now see how a bucket-based priority queue of propagators can be implemented.

The bucket queue

A bucket-based priority queue consists of an array of doubly-linked lists of propagators. The list at array index i represents the queue of propagators at priority i. Furthermore, a propagator can only be in one queue at a time. We can hence embed the links for the doubly-linked lists into the propagator objects. In addition to the lists for each priority, the kernel maintains the list of idle propagators, the so-called *idle queue*. The invariant is then that a propagator is always in exactly one queue.

Each list of propagators is cyclic and terminated by a *sentinel element*. The sentinels are kept in an array that represents the priority queue and resides in the space. Figure 6.5 depicts an example of this architecture. An empty queue is depicted as a sentinel with a simple cycle (as at priority $k-1$).

This implementation of a bucket queue yields efficient access. Inserting and removing a propagator can be done in constant time—unlink it from its current queue, and link it at the position before the sentinel element of the target queue. Finding the next propagator to schedule costs at most k tests.

Queues are managed as follows:

- The kernel can access the queue with priority i as $Q[i]$.
- p.`next`() returns the propagator following p in the linked list.
- p.`unlink`() removes propagator p from its current queue.
- $Q[i]$.`tail`(p) adds p as the last propagator to the queue with priority i.

A propagator is added to the queue that corresponds to its cost, as described in Section 5.5. Each propagator reports its cost using the virtual method `cost`. The following code implements the `enqueue`, `head`, and `idle` methods, as well as a method `stable`() that reports whether all queues except the idle queue are empty, indicating that the space is stable.

```
enqueue(p)
1  p.unlink()                          ▷ remove p from current queue
2  Q[p.cost(p.Δme)].tail(p)            ▷ put p into new queue

head()
1  for i ← k downto 1
2       do if Q[i].next() ≠ Q[i] then return Q[i].next()

idle(p)
1  p.unlink()                          ▷ remove p from current queue
2  Q[0].tail(p)                        ▷ put p into idle queue

stable()
1  for i ← k downto 1
2       do if Q[i].next() ≠ Q[i] then return FALSE
3  return TRUE
```

Memory and run-time efficiency. The bucket queue is as efficient as possible, both in terms of memory requirements and run-time. The asymptotic run-time for all operations is a small constant if we restrict the priorities to a small, fixed set. Priority-based scheduling with a fixed number of priorities is the standard in all propagation-based solvers (see Section 5.6), and has proven effective in practice. In terms of memory, the kernel needs access to the set of all propagators anyway for copying, as we will see in Section 6.7. Embedding the double links for the queues in the propagator objects is therefore efficient, as no additional memory management is needed for the queues.

6.6 Control

With all the data structures in place, this section combines them in the main control loop, implementing agenda-based, event-directed propagation as a kernel service.

The main propagation loop is straightforward. We implement it as a method `status` on spaces, which applies all propagators until reaching a mutual fixed point, and then returns the overall status. Furthermore, the contract between the kernel and the propagators specifies that the `status` method clears the propagators' modification event deltas.

```
status()
1  while not stable()
2        do p ← head()
3           oldΔme ← p.Δme; p.Δme ← ∅
4           if p(oldΔme) = FAIL then return FAIL
5  return OK
```

The following invariants are maintained as a consequence of the contracts between propagators, variables, and the kernel:

- If the propagator modifies any variable domain, the dependent propagators are scheduled.
- For each propagator p that is scheduled, the events that caused scheduling are added to its modification event delta.
- If the propagator detects or causes failure (emptying a variable domain), it returns FAIL. Otherwise, it returns OK.
- According to the propagator's fixed point status, it will be in the correct queue after propagation, or removed completely in case of subsumption.

These invariants correspond to the agenda and dependency invariants developed in the previous chapter. They guarantee that invoking `status` on a space results in a stable space representing a mutual fixed point of all propagators. The space is failed if and only if `status` returns FAIL.

6.7 Copying and Memory Management

This section shows how the propagation kernel manages memory, using spaces as containers for propagators and variables. Furthermore, spaces provide the infrastructure for copying.

Allocation and deallocation

All propagators and variables reside inside a space. The space acts as a memory manager, providing allocation and deallocation for different kinds of memory:

Blocks can be allocated for data that usually lives as long as the space lives. Examples are the arrays that propagators use to hold pointers to their variables, but also the propagator and variable objects themselves.

Free-lists of small equally-sized blocks can be used to efficiently allocate data structures that change frequently. For instance, integer variable domains are implemented as lists of ranges, taken from the free-list pool of the space.

Scrap space can be used for temporary storage within a single method.

Memory allocated from spaces does not have to be freed explicitly, as it will be reclaimed when the space is destroyed. Spaces are destroyed when they are no longer needed by the search engine, for instance, when the space is failed and the search engine backtracks. If all propagators and variables allocate their entire memory from the space, destroying a space amounts to nothing more than reclaiming all its memory. Propagators and variables do not have to be recursively destroyed. This makes failure efficient, which is important as failure of course occurs extremely often in real-world searches.

For propagators and variables that do need additional resources, such as externally allocated memory, a space provides a mechanism for registering these propagators and variables. That way, only the objects that have external resources are destroyed recursively.

Copying

The memory management services that spaces provide are closely related to another important kernel service: copying. For backtracking search, the kernel must be able to produce a copy of a space that behaves just like the original. A space is a cyclic graph (recall Figure 6.2(a)). The central idea of copying a graph is to use **forwarding pointers**: after creating a copy x' of an object x, leave a pointer in x that points to its copy x'. Then copy the children of x recursively. When another object points to x again, the presence of the forwarding pointer to x' indicates that x has already been copied, and one can directly use x' instead of creating a new copy.

Copying spaces is slightly simpler due to their regular structure: the graph is bipartite, the only edges go from variables to propagators and from propagators to variables, but not between propagators or between variables. The copying functionality for both propagators and variables must be implemented partly in the domain module. In order to keep the overhead low, the space *delegates* copying of the variables to the propagators: a propagator knows the concrete types of its variables,

and already has virtual methods (`propagate`, `cost`, and `dispose`). We add another virtual method to all propagators, `copy`. The space copies the propagators, which in turn copy the variables.

Let us look at the individual steps for copying a space. We will assume that the space is stable, as in practice, search engines never need to copy unstable spaces. The letters in brackets denote who is responsible for the corresponding step, and therefore explain the boundary between the kernel and the domain module. We write **k** for the kernel, **p** for a propagator, and **v** for a variable.

1. **[k]** Create a new, empty space (the *target*).
2. **[k]** Iterate over the propagators in the idle queue. Call each propagator's `copy` method, supplying the target space as an argument.
3. **[p]** The propagator's `copy` method creates a new propagator in the target space and sets the forwarding pointer accordingly.
4. **[p]** The propagator copies its variables, calling each variable's `copy` method.
5. **[v]** The variable's `copy` method checks if the variable has a forwarding pointer, and if it has, returns it. Otherwise, it creates a copy, sets the forwarding pointer, and returns it. The dependencies in the target variable are left empty. The source variable is added to a list of copied variables.
6. **[p]** After copying a variable, the target propagator's link to that variable is set to point to the copy.
7. **[k]** After copying all the propagators, iterate over all the copied source variables. For each variable, find the copy using the forwarding pointer.
8. **[k]** For each copied variable, allocate a new dependency array and recreate the dependency index. Fill the dependency array by iterating over the source variable's dependencies, and for each propagator found there, enter its copy (accessed through the forwarding pointer) into the target dependency array.
9. **[k]** Finally, reset all forwarding pointers so that they can be reused when the space is copied again.

Copying a space is illustrated in Figure 6.6. The dashed arrows between the original space and the copy are the forwarding pointers. The figure also shows an important side-effect of copying spaces. The copy may be more compact than the original. In the concrete example, variable x_2 was not referenced by any propagator any more and hence does not have to be copied. In the same way, propagators may create more compact versions of internal data structures in the copy. The memory used by subsumed propagators may not have been reclaimed in the original space, but as subsumed propagators are not copied (they are not in the idle queue), they do not occupy memory in the copied space.

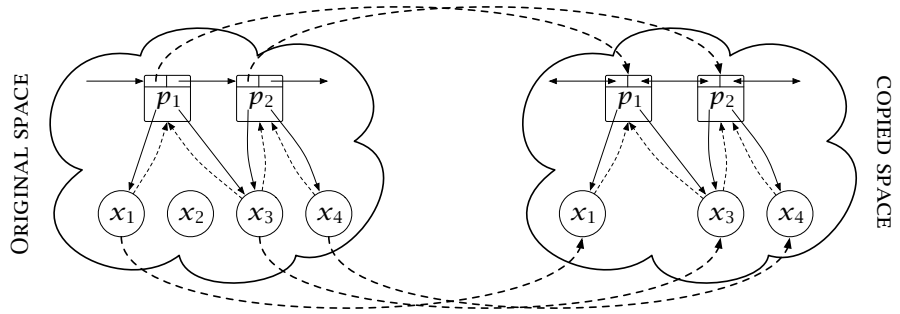


Figure 6.6: Spaces are copied using forwarding pointers

The dependencies can be treated specially during copying. The kernel keeps a record of the overall size of all dependency arrays, and then pre-allocates space for the dependencies in one block in the copy. This is more efficient than allocating new dependency arrays for each individual variable.

Reusing memory

As described above, copying requires some additional memory per variable and propagator: both have to accommodate for the forwarding pointers, and in addition, the variables have to be put into a list for later restoration of the dependency data structures. However, it turns out that we can *reuse* some of the data structures that are already present in variables and propagators.

The propagators store their forwarding pointer in the field that normally contains the pointer to the previous queue element (as shown in Figure 6.6). This pointer is not needed during copying (it suffices to iterate forwards through the idle queue), and can easily be restored afterwards. Additionally, the pointer is *tagged* so that it can be distinguished from the regular pointer to the previous queue element.

Variables reuse parts of the dependency index. When copying a variable, the contents of the original dependency index are saved in the copy. Then `idx`[0] is used for the (again tagged) forwarding pointer, and `idx`[1] points to the previously copied variable, establishing the list of copied variables. When updating the dependency data structures in step 8, the original dependency index is restored from the values saved in the copy. Note that independent of the concrete event system, the dependency index always has at least two slots, as there is at least one propagation condition plus π_{end}.

By superimposing the data structures needed for copying with those needed for propagation, we can implement copying without any memory overhead.

Related work

► Producing copies of graphs has received a lot of attention in the context of *copying garbage collection* (see Cheney, 1970 and Jones and Lins, 1996). The similarity to copying garbage collection goes even further. As we have seen, the copy of a space can be more compact, as variables and propagators that are no longer needed can be removed. Copying a space thus performs garbage collection.

► Search in the Mozart (2009) programming system is based on copying. According to Schulte (2002), the Mozart virtual machine uses the same copying routines for garbage collection and for spaces.

6.8 Gecode

The architecture presented in this chapter has been implemented in the Gecode constraint solver. This section gives an overview of Gecode.

The library

Gecode (2009) stands for *Generic Constraint Development Environment*, and is a state-of-the-art constraint solver, implemented as a C++ library. The core of the library is the propagation kernel, providing the basic services and abstractions we just saw. Gecode comes with several domain modules that are built on top of the kernel, implementing integer and Boolean variables, set variables using the set interval approximation (see Section 4.5), and set variables that represent the complete set domain using ROBDDs (see Section 4.6). The domain modules provide a comprehensive set of propagation algorithms, implementing constraints such as *all-different*, *regular*, the global cardinality constraint, linear equations and inequations, Boolean constraints, reified versions of many constraints, and many more.

Also built on top of the kernel, and orthogonal to the domain modules, are Gecode's search engines, providing standard search strategies such as depth first search or branch-and-bound search, as well as a graphical interactive search tool called *Gist*.

The library is available under the open-source MIT license, and written in standards-compliant C++, making it portable to most current platforms. The free availability of the library makes experiments conducted using Gecode reproducible, and hence makes Gecode suitable for research.

Implications of the architecture

Implementing the presented architecture in Gecode has the following practical implications.

1. The kernel is compact, the core functionality comprises less than 2 000 lines of C++ code. A small kernel is amenable to thorough code review as well as systematic testing. We are therefore confident that the code is correct.
2. The modularity and clean architecture enable experimentation, as most architectural changes require only local code modifications. Experimenting with different implementations for the main data structures enabled us to make informed design decisions.
3. The library approach enables integration. Interfaces that integrate Gecode into different programming systems, such as Java, Alice ML, Ruby, Common Lisp, and Mozart/Oz, as well as a parser for FlatZinc (Nethercote et al., 2007) are available.
4. The abstractions provided by the kernel make implementing a new domain module easy. The tedious bits involving the event system are generated from simple specifications. In addition to the modules provided by Gecode itself, we received external contributions, implementing graph variables (Dooms et al., 2005; Dooms, 2006) and variables over real-valued intervals.
5. The contracts that propagators have to comply with can be enforced by abstractions or tested systematically. Thus, implementing a propagator in Gecode is not more complicated than in systems that do not require strict contracts.

Performance

In terms of performance, Gecode is competitive with and in many cases exceeds the performance of commercial systems like ILOG Solver (2009) or SICStus Prolog (2009) (see Appendix A for a performance comparison with these systems). Competitive performance is a key requirement for the next section, which evaluates some of the design decisions presented in this chapter empirically, using Gecode. The results of such experiments are only practically relevant if the base system is efficient.

6.9 Performance Analysis

This section empirically evaluates some of the design decisions presented in this chapter.

Benchmark	*Without subsumption*		
	time %	*mem. %*	*prop. %*
BIBD	137.40	163.23	105.40
Alpha (smart)	115.96	100.00	100.00
Alpha (naive)	122.52	100.00	100.00
Knights (18)	183.11	194.18	117.99
Golomb R. (10)	105.38	100.00	113.25
Queens (N, 10)	158.68	100.00	204.37
Queens (S, 10)	128.16	100.00	101.29
Queens (N, 100)	225.03	209.39	356.89
Queens (S, 100)	165.25	100.00	100.00
Eq-20	107.45	100.00	100.00
Graph Coloring	116.38	100.00	129.94
M. Seq. (S, 500)	111.28	100.00	112.47
M. Seq. (N, 500)	1 340.81	200.95	100.00
M. Seq. (GCC, 500)	102.05	100.00	100.00
Photo Alignment	131.80	100.00	122.75
Partition (32)	133.97	100.00	110.61
Perfect Square	117.03	116.93	108.55

Benchmark	*Without subsumption*		
	time %	*mem. %*	*prop. %*
S. Golfers (8-4-9)	103.46	104.99	100.00
S. Golfers (5-3-7)	119.43	145.35	116.88
Crew Scheduling	115.59	100.00	111.01
Steiner T. (9)	157.90	100.00	109.53
Sudoku (Set, 1)	104.23	100.00	100.00
Sudoku (Set, 4)	99.83	100.00	100.00
Sudoku (Set, 5)	100.33	100.00	100.00
Queen Armies	134.98	100.00	163.88
Hamming	154.95	130.36	108.16

Table 6.2: Keeping subsumed propagators alive

Experimental platform

For the experiments, we used the upcoming version 3.0.0 of Gecode, and replaced or modified parts of its architecture to evaluate the impact of different design decisions. The concrete setup for the experiments, such as the used platform, operating system, and compiler, is explained in Appendix A. Most of the numbers we present here are relative numbers, given as percentages of the absolute results of the unmodified Gecode system reported in Appendix A. For example, a relative run-time of 200% means that the experiment needs twice the time of the unmodified Gecode, and a memory requirement of 50% means that it only needs half the memory. The experiments are partitioned into two groups. Models in the first group involve only integer and Boolean variables and propagators, and the second group deals mainly with set variables (and additional integer and Boolean variables).

Subsumption

The first experiment evaluates how important it is to detect subsumption and remove subsumed propagators. For the experiment, we simply changed the meaning of the `subsumed` method to be equal to `fix`. Thus, subsumed propagators are kept alive, and their subscriptions are kept active. Table 6.2 shows that the performance degrades for nearly all examples. This is due to two effects. First, the subsumed propagators will be scheduled and executed, although they cannot prune any more. This effect seems to be strong in examples where the number of propagation steps increases significantly, such as Graph Coloring, Photo Alignment, Naive Queens, or

Benchmark	*Sub.*	*A. sub.*	*Cancel*	*A. cancel*	*F. cancel*
BIBD	48 754	0	52 926	8 080	133 829
Alpha (smart)	119	0	0	0	2 672
Alpha (naive)	119	0	0	0	472 339
Knights (18)	30 593	185	14 929	13 723	66 877
Golomb Rulers (10)	171	0	48 906	18 636	461 808
Queens (Naive, 10)	270	0	213 987	315 667	38 933
Queens (Smart, 10)	30	0	1 562	0	9 361
Queens (Naive, 100)	29 700	0	16 495	17 103	607
Queens (Smart, 100)	300	0	0	0	135
Eq-20	140	0	0	0	3 440
Graph Coloring	21 529	135	29 806	5 242	417 008
Magic Sequence (Smart, 500)	282 628	0	313 692	94 652	4 396
Magic Sequence (Naive, 500)	751 499	0	288 913	332 619	59 791
Magic Sequence (GCC, 500)	1 999	0	0	997	1 508
Photo Alignment	8 592	7 854	52 905	349 095	496 682
Partition (32)	578	0	507 460	1 379 602	7 002 413
Perfect Square	25 457	162	44 339	61 957	1 968 875
Social Golfers (8-4-9)	6 984	9	0	7 125	200 151
Social Golfers (5-3-7)	13 087	27	37 021	136 503	1 952 445
Crew Scheduling	234	0	134	354	5 754
Steiner Triples (9)	2 178	0	24 532	89 626	642 435
Sudoku (Set, 1)	495	1	0	487	0
Sudoku (Set, 4)	495	1	0	649	209
Sudoku (Set, 5)	495	1	0	3 015	6 552
Queen Armies	25 381	1 070	78 867	169 277	350 544
Hamming Codes (20-3-32)	6 016	0	33 500	315 988	5 105 731

Table 6.3: Subscription statistics

Queen Armies. The second effect is that subsumed propagators have to be copied in the modified system, which costs run-time and increases memory usage. The biggest performance loss due to this effect can be observed for Knights, Naive Magic Sequence, and probably Hamming Codes.

Another experiment that is related to subsumption evaluates the decision to ignore `subscribe` and `cancel` operations on assigned variables (Section 6.4). Its results are reported in Table 6.3.

The number of `subscribe` operations that actually create a new subscription (column *Sub.*) is consistently far bigger than the number of `subscribe` operations on assigned variables (column *A. sub.*). Thus, optimizing subscription for assigned variables is not particularly important. The reason is that the majority of subscriptions happens when the problem is first set up and not many variables are assigned yet, and only few subscriptions are created later during search.

The numbers are very different for cancel operations. The number of `cancel` operations that actually remove a subscription (column *Cancel*) is often *lower* than

Benchmark	$O(n)$ *susp. lists*			$O(1)$ *susp. lists*		
	time %	*mem. %*	*prop. %*	*time %*	*mem. %*	*prop. %*
Alpha (smart)	110.61	185.71	91.52	114.48	185.71	84.68
Alpha (naive)	125.32	186.96	93.63	139.29	186.96	92.46
Queens (Naive, 10)	327.45	194.12	100.01	373.98	194.12	99.23
Queens (Smart, 10)	106.46	100.00	97.88	115.49	100.00	100.35
Queens (Naive, 100)	110.71	130.11	99.73	127.17	164.98	99.74
Queens (Smart, 100)	104.79	110.04	99.78	181.74	140.17	101.10
Eq-20	110.60	185.71	94.74	120.25	185.71	87.57
Graph Coloring	104.07	100.00	100.31	114.20	100.00	101.23
Magic Sequence (Smart, 500)	70.23	71.23	94.24	86.14	142.39	85.08
Magic Sequence (Naive, 500)	101.85	103.87	108.89	120.08	158.06	91.54
Magic Sequence (GCC, 500)	102.96	196.97	97.26	111.49	196.97	94.86
Partition (32)	106.24	105.78	98.97	118.46	123.10	97.44
Social Golfers (8-4-9)	109.58	133.70	100.00	124.28	161.79	99.96
Social Golfers (5-3-7)	107.25	130.23	99.89	116.76	163.49	99.51
Crew Scheduling	111.15	194.31	100.88	121.20	197.56	103.31
Steiner Triples (9)	118.74	199.45	108.65	141.90	199.45	111.99
Sudoku (Set, 1)	98.98	100.00	99.34	99.98	119.28	99.34
Sudoku (Set, 4)	101.77	100.00	102.53	102.21	100.00	100.27
Sudoku (Set, 5)	100.96	159.63	99.87	102.26	159.63	100.26
Queen Armies	106.90	100.00	100.46	117.10	196.39	101.49
Hamming Codes (20-3-32)	116.47	121.06	95.63	138.63	150.05	92.82

Table 6.4: Relative performance of suspension lists

the `cancel` operations on assigned variables (column *A. cancel*). The reason is that propagators cancel their subscriptions when they are subsumed, and they are often subsumed when some or all of their variables are assigned. The last column, *F. cancel*, reports the number of subscriptions still left when a space failed. These numbers are often orders of magnitude bigger than the number of cancel operations that are actually performed. The results make clear that performing cancel operations on assigned variables and in particular on failed spaces would incur a significant overhead.

Suspension lists versus dependency arrays

An alternative to the dependency and index arrays from Section 6.4 is a linked list of propagators for each propagation condition, a so-called *suspension list*. This list can be singly-linked, so that canceling a subscription takes $O(n)$ time if n is the number of entries in the list, or it can be doubly-linked, with a constant-time `cancel` operation.

The reason for choosing an array data structure over a list was that lists require more memory, and copying lists costs more run-time. On the other hand, lists enable constant time subscription and cancel operations. We thus experimented with both alternative designs in Gecode, and the results appear in Table 6.4. We could not run

Benchmark	$O(n)$	$O(1)$
	time %	
Alpha (smart)	120.86	135.18
Alpha (naive)	133.84	150.65
Queens (Naive, 10)	327.43	376.90
Queens (Smart, 10)	108.76	115.09
Queens (Naive, 100)	111.01	127.50
Queens (Smart, 100)	105.02	179.77
Eq-20	116.74	137.31
Graph Coloring	103.75	112.80
M. Seq. (Smart, 500)	74.53	101.24
M. Seq. (Naive, 500)	93.53	131.19
M. Seq. (GCC, 500)	105.86	117.53
Partition (32)	107.35	121.57

Benchmark	$O(n)$	$O(1)$
	time %	
Social Golfers (8-4-9)	109.58	124.33
Social Golfers (5-3-7)	107.37	117.33
Crew Scheduling	110.18	117.32
Steiner Triples (9)	109.29	126.71
Sudoku (Set, 1)	99.64	100.65
Sudoku (Set, 4)	99.25	101.94
Sudoku (Set, 5)	101.09	101.99
Queen Armies	106.41	115.38
Hamming Codes (20-3-32)	121.79	149.35

Table 6.5: Normalized run-time performance of suspension lists

all the benchmarks, as the changes to Gecode are rather involved and would require modifying some propagators substantially. The singly-linked suspension lists do not perform much worse for most examples (columns $O(n)$ *susp. lists*), and the increase in memory consumption is expected. Interestingly, the doubly-linked lists with constant-time `cancel` incur a significantly higher overhead. This is due to the additional back-link and the fact that each propagator needs to keep references of the subscriptions it made, resulting in even higher memory usage and increased cost during copying.

In many cases, fewer propagators are invoked than in the unmodified system (see columns *prop. %*). This is due to slightly different scheduling of the propagators, because `subscribe` and `cancel` on dependency arrays reorder the subscriptions (see Figure 6.4), whereas the order is preserved with suspension lists. The real overhead is therefore slightly higher. Table 6.5 presents the same run-time results, but normalized to the number of propagator invocations, which should give an upper bound of the actual overhead.

One should note that the run-time overhead of suspension lists is mostly due to the fact that the dependencies have to be copied. Suspension lists may hence be a viable alternative for trailing systems, where the overhead should be significantly lower.

Dependencies indexed by modification event

The dependency arrays are indexed by propagation condition. A single modification typically triggers more than one propagation condition, so scheduling may require iterating over different parts of the dependency array. For example, Section 6.3 defines `notify` for the modification event $\mathtt{me}_{\mathrm{lbc}}$ to schedule the propagators between propagation conditions π_2 and π_3, and in addition the propagators for π_5.

Benchmark	*time %*	*mem. %*
BIBD	107.91	105.75
Alpha (smart)	105.99	157.14
Alpha (naive)	104.06	169.57
Knights (18)	107.88	110.78
Golomb Rulers (10)	100.91	100.00
Queens (Naive, 10)	100.74	100.00
Queens (Smart, 10)	101.92	100.00
Queens (Naive, 100)	99.57	100.00
Queens (Smart, 100)	101.05	100.00
Eq-20	107.69	100.00
Graph Coloring	105.61	100.00
M. Seq. (Smart, 500)	244.25	292.29
M. Seq. (Naive, 500)	108.55	130.32
M. Seq. (GCC, 500)	99.33	100.00
Photo Alignment	103.99	106.35
Partition (32)	102.90	100.00

Benchmark	*time %*	*mem. %*
Social Golfers (8-4-9)	120.98	160.54
Social Golfers (5-3-7)	112.68	154.42
Crew Scheduling	116.76	197.56
Steiner Triples (9)	113.24	199.45
Sudoku (Set, 1)	114.67	157.83
Sudoku (Set, 4)	109.70	100.77
Sudoku (Set, 5)	104.60	159.63
Queen Armies	111.60	100.00
Hamming Codes (20-3-32)	119.80	133.36

Table 6.6: Relative performance of dependencies indexed by modification event

Alternatively, one could index the dependency array by modification event, and keep a single propagator multiple times in the array. Then `notify`($\mathsf{me}_{\mathsf{lbc}}$) would just have to iterate once over the propagators indexed by $\mathsf{me}_{\mathsf{lbc}}$. A propagator that subscribes for the propagation condition $\pi_3 = \{\mathsf{asn}, \mathsf{lbc}, \mathsf{ubc}\}$ would end up three times in the dependency array, once in the part indexed as $\mathsf{me}_{\mathsf{asn}}$, once in the part indexed as $\mathsf{me}_{\mathsf{lbc}}$, and once in the part indexed as $\mathsf{me}_{\mathsf{ubc}}$.

Table 6.6 lists the results of our experiments. Keeping propagators multiple times in the dependency arrays clearly increases the memory usage, but the performance also suffers because of increased copying costs. The overhead is highest for examples where many propagators subscribe with propagation conditions other than {`asn`}, such as the smart magic sequence, or the set constraint examples appearing in the right table.

Delaying the scheduling

As mentioned in Section 6.3, the `notify` method immediately schedules all dependent propagators when a variable domain is modified. Thus, propagators get scheduled several times if the same variable is modified several times during the same run of a `propagate` method. The experimental results shown in Table 6.7 suggest that this occurs only moderately often in practice. The column *Mod.* lists the number of variable modifications, and the column *Double mod. %* the relative number of these modifications that were double modifications during the same run of a `propagate` method.

Benchmark	*Mod.*	*Double mod. %*
BIBD	59 852	0.00
Alpha (smart)	2 853	17.88
Alpha (naive)	211 905	8.44
Knights (18)	54 336	28.85
Golomb Rulers (10)	1 335 772	16.10
Queens (Naive, 10)	117 163	1.13
Queens (Smart, 10)	119 055	15.52
Queens (Naive, 100)	7 331	0.10
Queens (Smart, 100)	7 332	2.22
Eq-20	1 133	8.56
Graph Coloring	83 970	8.89
M. Seq. (Smart, 500)	58 873	1.25
M. Seq. (Naive, 500)	723 190	0.07
M. Seq. (GCC, 500)	49 826	1.00
Photo Alignment	343 368	1.59
Partition (32)	10 300 108	5.10
Perfect Square	98 059	0.96

Benchmark	*Mod.*	*Double mod. %*
Social Golfers (8-4-9)	69 417	0.00
Social Golfers (5-3-7)	326 664	1.25
Crew Scheduling	1 792	15.46
Steiner Triples (9)	141 262	1.50
Sudoku (Set, 1)	890	4.38
Sudoku (Set, 4)	1 223	3.27
Sudoku (Set, 5)	5 795	2.09
Queen Armies	114 070	0.00
Hamming (20-3-32)	566 624	6.05

Table 6.7: Number of modifications and double modifications

The examples that have a high count of doubly modified variables use propagators that achieve idempotency by iteration, such as the propagator for linear equations (in Alpha and Eq-20) or the *all-different* propagators (in Knights, Smart Queens and Graph Scheduling). Note that the only overhead that arises from scheduling propagators twice in these cases is the check whether the new modification event is already contained in the propagator's modification event delta, as discussed in Section 6.3.

Copied versus shared propagators

One of our fundamental design decisions was to base our architecture on copying. More precisely, we decided to copy both variables and propagators. That way, we can backtrack the complete state during search, including the dependencies and the propagators' internal data structures. This enables techniques such as removing subsumed propagators or rewriting propagators (Section 5.3), and lets propagator implementations deal with complicated data structures easily (as long as they can be copied).

However, some propagators do not use these techniques and do not require any backtrackable state (for example, propagators for $x < y$ or $x = \max(y, z)$). Propagators for other constraints, such as Boolean clauses, have a *backtrack-safe* state: they adapt their dependencies dynamically (recall Example 5.13), but the state is still globally valid, at least in a DFS search.

To assess the overhead of copying for these propagators, we implemented a Boolean

Benchmark	*time %*	*mem. %*	*propagations %*
Dubois (20)	30.79	26.53	149.32
Towers of Hanoi (4)	70.37	86.69	147.52
Flat (200-1)	63.58	50.11	149.83
Pigeon Hole (7)	117.91	52.00	192.22
Pigeon Hole (8)	39.20	50.82	209.39
Ramsey (4-4-10)	36.49	25.70	94.60
Ramsey (4-4-13)	31.16	25.18	65.56

Table 6.8: Relative performance of non-copied propagators

clause propagator that is never copied. Every Boolean variable gets additional *global* dependencies, which point to the non-copied propagators and are shared between all copies of the variable. Table 6.8 reports the results for a number of SAT problems. Not copying the propagators clearly boosts performance, both in terms of run-time and memory consumption. Due to different scheduling, the number of propagation steps (column *prop.*) is even significantly higher, and still the solver is faster.

These experiments show the extreme case—a great number of essentially stateless propagators. For typical propagation problems, the benefits of copying outweigh the costs. One should also note that general-purpose constraint solvers do not handle SAT problems very well. Dedicated SAT solvers like MiniSat (Eén and Sörensson, 2004) solve any problem from Table 6.8 in a fraction of a second, due to elaborate techniques such as conflict clause learning and special branching heuristics.

Contributions of Part I

This first part of the dissertation established a mathematical model of constraint propagation, and developed an implementation architecture for a constraint solver based on the mathematical model. The main original contributions of these four chapters are as follows.

1. The mathematical model identifies contraction and soundness as the minimal properties that are required of propagators. Propagators are therefore more general than in previous models, which usually define them to be monotonic and often idempotent in addition. This work provides the first thorough discussion of idempotency and monotonicity of propagators. Non-monotonic propagators have no influence on soundness and completeness of a solver.
2. Previous models of constraint propagation typically defined conditions under which a propagator is correct for a particular constraint. Our definition of propagators is independent of constraints. Rather, it is a *consequence* of the definition that each propagator *induces a unique constraint.*
3. Based on existing models of domain approximations, this dissertation uses *domain systems* in a novel way as a tool for *characterizing propagation strength* generically. Propagation strength is defined as a property of propagators, in contrast to the traditional notions of *consistency*, which are defined as properties of domains. All classic notions such as bounds, range, or set interval consistency have corresponding notions in the general model.
4. While the scheduling techniques presented in Chapter 5 are no original contribution of this work, we consider the uniform presentation in our mathematical framework enlightening and valuable. We introduce the novel concepts of propagation conditions and modification events, which capture exactly the sets of events that are necessary for scheduling propagators and describing variable modifications, respectively. The model based on propagation conditions and modification events yields a straightforward implementation.
5. For the first time, this dissertation presents a complete implementation architecture for a constraint solver based on principled models. The architecture separates the domain-independent *kernel* of the solver from the *domain modules*, consisting of the domain-specific parts. *Contracts* between propagators,

variables, and the kernel dictate how these components interact, and establish strong invariants in the implementation architecture.

6. This dissertation contains the first systematic discussion and evaluation of the two central data structures in a propagation-based solver, the dependencies and the priority queue. We design dependency and index arrays for the dependencies, as well as a bucket queue with links embedded into the propagator objects for the priority queue.
7. An important and novel aspect is the detailed discussion of detecting propagator *subsumption*. In a copying solver, removing subsumed propagators saves not only memory, but also run-time during copying. Furthermore, subsumption is the basis for propagator *rewriting*, which simplifies the implementation of propagation algorithms.
8. The whole architecture presented here has been implemented in the Gecode constraint solver. Gecode provides a production-quality, efficient solver as a C++ library. The high performance as well as the modularity and clean architecture of Gecode result directly from the principled models and powerful techniques developed here.

Part II

Techniques for Deriving Propagators

7 Views

This chapter introduces *views*, which are used to *derive propagators* from existing propagators. A derived propagator induces a variant of the constraint that its original propagator induces. The goal is to reuse implementations of propagation algorithms, making it simpler to provide a comprehensive library of correct and efficient propagators.

Using the mathematical model of propagation from Chapter 3, we define views formally. We prove that derived propagators are *perfect*, in that they preserve all important properties like correctness, domain completeness, and completeness with respect to approximations.

Structure of the chapter. After an informal motivation of views and derived propagators (7.1), we present the basic mathematical model of views (7.2). Within this model, we can prove several results concerning correctness (7.3) and completeness (7.4) of derived propagators, and investigate properties such as compositionality, idempotency, subsumption, and events (7.5).

7.1 Motivation

When implementing a propagator for a constraint, one typically needs to decide whether to also implement some of its variants. Let us look at three examples of such constraint variants.

Example 7.1 (Minimum/maximum) When implementing a propagator for the constraint $[\![\max\{x_1,\ldots,x_n\} = y]\!]$, should one also implement $[\![\min\{x_1,\ldots,x_n\} = y]\!]$? These two constraints are related by the equivalence

$$[\![\min\{x_1,\ldots,x_n\} = y]\!] = [\![\max\{-x_1,\ldots,-x_n\} = -y]\!] \qquad *$$

Example 7.2 (Linear constraints) When implementing a propagator for the linear equation $[\![\sum_{i=1}^{n} a_i x_i = k]\!]$ for integer variables x_i and integers a_i and k, should one also implement the special case $[\![\sum_{i=1}^{n} x_i = k]\!]$ for better performance? *

Example 7.3 (Reified constraints) When implementing a propagator for the reified linear equation $[\![(\sum_{i=1}^{n} x_i = c) \leftrightarrow b]\!]$, should one also implement $[\![(\sum_{i=1}^{n} x_i \neq c) \leftrightarrow b]\!]$? These two constraints only differ by the sign of the variable b:

$$\left[\!\!\left[\left(\sum_{i=1}^{n} x_i \neq c \right) \leftrightarrow b \right]\!\!\right] = \left[\!\!\left[\left(\sum_{i=1}^{n} x_i = c \right) \leftrightarrow \neg b \right]\!\!\right]$$ *

The two straightforward approaches for implementing variants of constraints are to either implement dedicated propagators for the variants, or to decompose the constraints. In the last example, for instance, one could decompose the reified constraint using two propagators, one for $[\![(\sum_{i=1}^{n} x_i = c) \leftrightarrow b']\!]$, and one for $[\![b \leftrightarrow \neg b']\!]$, introducing an additional variable b'.

Implementing the variants inflates code and documentation. Given the potential code explosion, one may be able to only implement some variants (say, minimum and maximum). Other variants important for performance (say, minimum and maximum for two variables) may be infeasible due to excessive programming and maintenance effort. Decomposing, on the other hand, increases memory consumption and run-time, as we will see in the empirical evaluation in Chapter 9.

In this chapter, we follow a third approach: we *derive* propagators from already existing propagators using *views* on variables. This approach combines the efficiency of dedicated propagator implementations with the simplicity of decomposition.

Example 7.4 (Deriving a minimum propagator) Consider a propagator for the constraint $[\![\max(x, y) = z]\!]$. Given three more propagators for $[\![x' = -x]\!]$, $[\![y' = -y]\!]$, and $[\![z' = -z]\!]$, we could propagate the constraint $[\![\min(x', y') = z']\!]$ using the propagator for $[\![\max(x, y) = z]\!]$. In contrast to this decomposition, we propose to derive a propagator using views that perform simple transformations.

Views transform the input and the output of a propagator. For example, a minus view on a variable x transforms the variable domain of x by negating each element, then passes the transformed domain to the propagator, and performs the inverse transformation on the domain that the propagator returns. With views, the implementation of the maximum propagator can be reused: we derive a propagator for the minimum constraint from a propagator for the maximum constraint and three minus views. *

This chapter defines views formally, and reasons about the properties of the propagators we derive using views. The next chapter presents several generic techniques that one can use to derive propagators using views, and Chapter 9 and Chapter 10 present implementation approaches for views and derived propagators. We will see that views as introduced here have many applications, can be implemented easily, and result in a large number of useful and efficient derived propagators.

7.2 Views and Derived Propagators

After the informal introduction, let us now define views and derived propagators in the context of the mathematical model from Chapter 3. Given a propagator p, a view is represented by two functions, φ and φ^-, that can be composed with p such that $\varphi^- \circ p \circ \varphi$ is the desired derived propagator. The function φ transforms the input domain, and φ^- applies the inverse transformation to the propagator's output.

Definition 7.5 A **variable view** $\varphi_x \in V \to V'$ for a variable x is an injective function mapping values to values. The set V' may be different from V, and the corresponding sets of assignments, domains, constraints, and propagators are called Asn', Dom', Con', and Prop', respectively.

Given a family of variable views φ_x for all $x \in X$, we lift them point-wise to assignments: $\varphi_{\mathsf{Asn}}(a) := \lambda x.\varphi_x(a(x))$.

A **view** $\varphi \in \mathsf{Con} \to \mathsf{Con}'$ is a family of variable views, lifted to sets of assignments (constraints): $\varphi(c) := \{\varphi_{\mathsf{Asn}}(a) \mid a \in c\}$. The inverse of a view is $\varphi^-(c) := \{a \in \mathsf{Asn} \mid \varphi_{\mathsf{Asn}}(a) \in c\}$. *

Definition 7.6 Given a propagator $p \in \mathsf{Prop}'$ and a view φ, the **derived propagator** $\hat{\varphi}(p) \in \mathsf{Prop}$ is defined as $\hat{\varphi}(p) := \varphi^- \circ p \circ \varphi$. Similarly, we define a **derived constraint** to be $\varphi^-(c) \in \mathsf{Con}$ for a given constraint $c \in \mathsf{Con}'$. *

Example 7.7 Given a propagator p for the constraint $c = [\![x = y]\!]$, we want to derive a propagator for $c' = [\![x = 2y]\!]$ using a view φ such that $\varphi^-(c) = c'$.

Intuitively, the function φ leaves x as it is and scales y by 2, while φ^- does the inverse transformation. We thus define $\varphi_x(v) = v$ and $\varphi_y(v) = 2v$. Now it becomes clear why we need different sets V and V', as the latter must contain all elements of V multiplied by 2.

The derived propagator is $\hat{\varphi}(p) = \varphi^- \circ p \circ \varphi$. We say that $\hat{\varphi}(p)$ "uses a scale view on" y, meaning that φ_y is the function defined as $\varphi_y(v) = 2v$. Similarly, using an identity view on x amounts to φ_x being the identity function on V.

Given the assignment $a = (x \mapsto 2, y \mapsto 1)$, we first apply φ and get $\varphi(\{a\}) = \{(x \mapsto 2, y \mapsto 2)\}$. This is accepted by p and returned unchanged, so φ^- transforms it back to $\{a\}$. Another assignment, $a' = (x \mapsto 1, y \mapsto 2)$, is transformed to $\varphi(\{a'\}) = \{(x \mapsto 1, y \mapsto 4)\}$, rejected ($p(\varphi(\{a'\})) = \emptyset$), and the empty domain is mapped to the empty domain by φ^-. The propagator $\hat{\varphi}(p)$ induces $\varphi^-(c)$. *

Many-sorted views. Views directly generalize to the many-sorted model of constraint propagation introduced in Section 3.5. Only the definition of variable views changes to $\varphi_x \in V_x \to V'_x$, with a suitable family of sets V'_x, one for each variable

x. We will stick to the single-sorted model in the rest of the chapter, but implicitly use the many-sorted case in some examples (for instance when talking about propagators that involve both integer and Boolean variables).

7.3 Correctness of Derived Propagators

This section shows that derived propagators are well-defined and correct:

- A derived propagator $\hat{\varphi}(p)$ is in fact a propagator.
- The derived propagator induces the desired constraint: $c_{\hat{\varphi}(p)} = \varphi^-(c_p)$.
- A view φ preserves contraction of a propagator p: If $p(\varphi(d)) \subset \varphi(d)$, then $\hat{\varphi}(p)(d) \subset d$. This property makes sure that if p prunes the domain, then this pruning will not be lost after transforming the domain back using φ^-.

We will now prove these three statements. For the proofs, we employ the following properties of views, which are direct consequences of the definitions:

P1. φ and φ^- are monotonic by construction (as φ and φ^- are defined point-wise)

P2. $\varphi^- \circ \varphi = \mathrm{id}$ (the identity function)

P3. $|\varphi(\{a\})| = 1$, $\varphi(\emptyset) = \emptyset$

P4. For any view φ and domain d, we have $\varphi(d) \in \mathsf{Dom}$ and $\varphi^-(d) \in \mathsf{Dom}$ (as views are defined point-wise)

Proposition 7.8 For all propagators p and views φ, $\hat{\varphi}(p)$ is a propagator. If p is monotonic, then $\hat{\varphi}(p)$ is also monotonic. ∗

Proof. The derived propagator is well-defined because both $\varphi(d)$ and $\varphi^-(d)$ are domains (see P4 above). We have to show that $\varphi^- \circ p \circ \varphi$ is contracting and sound.

For contraction, we have $p(\varphi(d)) \subseteq \varphi(d)$ as p is contracting. From monotonicity of φ^- (with P1), it follows that $\varphi^-(p(\varphi(d))) \subseteq \varphi^-(\varphi(d))$. As $\varphi^- \circ \varphi = \mathrm{id}$ (with P2), we have $\varphi^-(p(\varphi(d))) \subseteq d$, which proves that $\hat{\varphi}(p)$ is contracting.

Soundness is shown as follows for all assignments a and domains d with $\{a\} \subseteq d$:

$$
\begin{array}{lcll}
 & \varphi(\{a\}) \subseteq \varphi(d) & & (\varphi \text{ monotonic, P1}) \\
\Longrightarrow & p(\varphi(\{a\})) \subseteq p(\varphi(d)) & & (|\varphi(\{a\})| = 1,\ p \text{ sound, P3}) \\
\Longrightarrow & \varphi^-(p(\varphi(\{a\}))) \subseteq \varphi^-(p(\varphi(d))) & & (\varphi^- \text{ monotonic, P1})
\end{array}
$$

In summary, for any propagator p, $\hat{\varphi}(p) = \varphi^- \circ p \circ \varphi$ is a propagator.

If p is monotonic, monotonicity of $\hat{\varphi}(p)$ can be shown just like soundness, replacing each $\{a\}$ with a domain d'. ∎

Proposition 7.9 Let p be a propagator, and let φ be a view. Then $\hat{\varphi}(p)$ induces the constraint $\varphi^-(c_p)$. *

Proof. As p induces c_p, we know $p(\{a\}) = c_p \cap \{a\}$ for all assignments a. With $|\varphi(\{a\})| = 1$ (P3), we have $p(\varphi(\{a\})) = c_p \cap \varphi(\{a\})$. Furthermore, we know that $c_p \cap \varphi(\{a\})$ is either $\emptyset$ or $\varphi(\{a\})$.

- *Case $\emptyset$:* We have $\varphi^-(p(\varphi(\{a\}))) = \emptyset = \left\{a' \in \mathsf{Asn} \;\middle|\; a = a' \wedge \varphi_{\mathsf{Asn}}(a) \in c_p\right\} = \varphi^-(c_p) \cap \{a\}$.
- *Case $\varphi(\{a\})$:* With P2, we have $\varphi^- \circ \varphi = \mathrm{id}$ and hence $\varphi^-(p(\varphi(\{a\}))) = \{a\}$. Furthermore, $\varphi^-(c_p) \cap \{a\} = \left\{a' \in \mathsf{Asn} \;\middle|\; a = a' \wedge \varphi_{\mathsf{Asn}}(a) \in c_p\right\} = \{a\}$.

Together, this shows that $\varphi^- \circ p \circ \varphi(\{a\}) = \{a\} \cap \varphi^-(c_p)$. ∎

Proposition 7.10 Views preserve contraction. Let p be a propagator, let φ be a view, and let d be a domain such that $p(\varphi(d)) \subset \varphi(d)$. Then $\hat{\varphi}(p)(d) \subset d$. *

Proof. The definition of $\varphi^-(c)$ is $\{a \in \mathsf{Asn} \mid \varphi_{\mathsf{Asn}}(a) \in c\}$. It clearly follows that $|\varphi^-(c)| \leq |c|$. Similarly, we know that $|\varphi(c)| = |c|$. From $p(\varphi(d)) \subset \varphi(d)$, it follows that $|p(\varphi(d))| < |\varphi(d)|$. Together, this yields $|\hat{\varphi}(p)(d)| < |\varphi(d)| = |d|$. We have seen in Proposition 7.8 that $\hat{\varphi}(p)(d) \subseteq d$, so we can conclude that $\hat{\varphi}(p)(d) \subset d$. ∎

7.4 Completeness of Derived Propagators

Given a domain system $\mathscr{D}$ and a $\mathscr{D}$-complete propagator p, ideally all propagators derived from p using a view φ would also be $\mathscr{D}$-complete. It turns out that this is not true in general, but depends on whether φ and φ^- commute with the $\mathscr{D}$-relaxation operator.

Definition 7.11 A constraint c is a **φ-constraint** for a view φ if and only if for all $a \in c$, there is a $b \in \mathsf{Asn}$ such that $a = \varphi_{\mathsf{Asn}}(b)$. A view φ is **$\mathscr{D}$-injective** if and only if $\varphi^-(\llbracket c \rrbracket_{\mathscr{D}}) = \llbracket \varphi^-(c) \rrbracket_{\mathscr{D}}$ for all φ-constraints c. It is **$\mathscr{D}$-surjective** if and only if $\varphi(\llbracket d \rrbracket_{\mathscr{D}}) = \llbracket \varphi(d) \rrbracket_{\mathscr{D}}$ for all domains d. It is **$\mathscr{D}$-bijective** if and only if it is $\mathscr{D}$-injective and $\mathscr{D}$-surjective. *

For the completeness proofs, we need the additional fact that views commute with set intersection.

Lemma 7.12 For any view φ, the equation $\varphi^-(c_1 \cap c_2) = \varphi^-(c_1) \cap \varphi^-(c_2)$ holds. *

Proof. By definition of φ^-, we have

$$\varphi^-(c_1 \cap c_2) = \{a \in \mathsf{Asn} \mid \varphi_{\mathsf{Asn}}(a) \in c_1 \wedge \varphi_{\mathsf{Asn}}(a) \in c_2\}$$

As φ_{Asn} is a function, this is equal to

$$\{a \in \mathsf{Asn} \mid \varphi_{\mathsf{Asn}}(a) \in c_1\} \cap \{a \in \mathsf{Asn} \mid \varphi_{\mathsf{Asn}}(a) \in c_2\} = \varphi^-(c_1) \cap \varphi^-(c_2) \qquad \blacksquare$$

Theorem 7.13 Let $\mathscr{D}$ be a domain system and let p be a $\mathscr{D}$-complete propagator. For any $\mathscr{D}$-bijective view φ, the propagator $\hat{\varphi}(p)$ is $\mathscr{D}$-complete. $*$

Proof. From Proposition 7.9, we know that $\hat{\varphi}(p)$ induces the constraint $\varphi^-(c_p)$. By monotonicity of φ^- (with P1) and $\mathscr{D}$ completeness of p, we know that

$$\varphi^- \circ p \circ \varphi(d) \subseteq \varphi^-(\llbracket c_p \cap \llbracket \varphi(d) \rrbracket_{\mathscr{D}} \rrbracket_{\mathscr{D}})$$

We now use the fact that both φ^- and φ commute with $\llbracket \cdot \rrbracket_{\mathscr{D}}$ and set intersection:

$$\begin{aligned}
\varphi^-(\llbracket c_p \cap \llbracket \varphi(d) \rrbracket_{\mathscr{D}} \rrbracket_{\mathscr{D}}) &= \varphi^-(\llbracket c_p \cap \varphi(\llbracket d \rrbracket_{\mathscr{D}}) \rrbracket_{\mathscr{D}}) && \mathscr{D}\text{-surjective} \\
&= \llbracket \varphi^-(c_p \cap \varphi(\llbracket d \rrbracket_{\mathscr{D}})) \rrbracket_{\mathscr{D}} && \mathscr{D}\text{-injective} \\
&= \llbracket \varphi^-(c_p) \cap \varphi^-(\varphi(\llbracket d \rrbracket_{\mathscr{D}})) \rrbracket_{\mathscr{D}} && \text{commute with } \cap \\
&= \llbracket \varphi^-(c_p) \cap \llbracket d \rrbracket_{\mathscr{D}} \rrbracket_{\mathscr{D}} && \text{definition of } \varphi
\end{aligned}$$

The second step uses $\mathscr{D}$ injectivity, so it requires $c_p \cap \varphi(\llbracket d \rrbracket_{\mathscr{D}})$ to be a φ-constraint. All assignments in a φ-constraint have to be the image of some assignment under φ_{Asn}. This is the case here, as the intersection with $\varphi(\llbracket d \rrbracket_{\mathscr{D}})$ can only contain such assignments. So in summary, we get

$$\varphi^- \circ p \circ \varphi(d) \subseteq \llbracket \varphi^-(c_p) \cap \llbracket d \rrbracket_{\mathscr{D}} \rrbracket_{\mathscr{D}}$$

which is the definition of $\hat{\varphi}(p)$ being $\mathscr{D}$-complete. $\blacksquare$

Stronger notions of completeness

We can formulate similar theorems for domain completeness, $\mathscr{D}$-Dom completeness, and Dom-$\mathscr{D}$ completeness. The theorems directly follow from the fact that any view φ is compatible with the domain relaxation, which means that φ is both Dom-injective and Dom-surjective. In order to prove this, we first need a lemma that lets us write the domain relaxation of a constraint in a slightly different way.

Lemma 7.14 Let d be the domain relaxation of a constraint c, $d = \llbracket c \rrbracket$. Then for all $x \in X$, we have $v \in d(x) \Leftrightarrow \exists a \in c : \ a(x) = v$. $*$

Proof. We prove both directions of the equivalence:

- $\Rightarrow$ There must be such an assignment a because otherwise we could construct a strictly stronger $d' \subset d$ with $v \notin d'(x)$ such that still $c \subseteq d'$.
- $\Leftarrow$ Each domain d' in the intersection $\bigcap \{d' \in \mathsf{Dom} \mid c \subseteq \mathrm{con}(d')\}$ must contain the value $v \in d'(x)$ as $c \subseteq d'$. So for the result of the intersection d, $v \in d(x)$. $\blacksquare$

Using this lemma, we can now prove that any view is compatible with the domain relaxation.

Lemma 7.15 Any view φ is Dom-injective and Dom-surjective. *

Proof. The Dom surjectivity follows directly from the definitions, as $[\![d]\!]_{\mathsf{Dom}} = d$ for any domain d (with P4). For Dom injectivity, we have to show that the equation $\varphi^-([\![c]\!]) = [\![\varphi^-(c)]\!]$ holds for any constraint c and any view φ. For clarity, we write the equation including the implicit $\mathrm{con}(\cdot)$ operations: $\varphi^-(\mathrm{con}([\![c]\!])) = \mathrm{con}([\![\varphi^-(c)]\!])$. By definition of φ^- and $\mathrm{con}(\cdot)$, we have

$$\varphi^-(\mathrm{con}([\![c]\!])) = \{a \in \mathsf{Asn} \mid \forall x \in X : \varphi_{\mathsf{Asn}}(a)(x) \in [\![c]\!](x)\}$$

Using Lemma 7.14 for $[\![\cdot]\!]$, this is equal to

$$\{a \in \mathsf{Asn} \mid \forall x \in X\ \exists b \in c : \varphi_{\mathsf{Asn}}(a)(x) = b(x)\}$$

As φ_{Asn} is an injective function, we can find such a b that is in the range of φ_{Asn}, if and only if there is also a $b' \in \varphi^-(c)$ such that $\varphi_{\mathsf{Asn}}(b') = b$. Therefore, we get

$$\begin{aligned}
&\{a \in \mathsf{Asn} \mid \forall x \in X\ \exists b' \in \varphi^-(c) : a(x) = b'(x)\} \\
&= \{a \in \mathsf{Asn} \mid \forall x \in X : a(x) \in [\![\varphi^-(c)]\!](x)\} \\
&= \mathrm{con}([\![\varphi^-(c)]\!]_{\mathsf{Dom}})
\end{aligned}$$ ∎

The following three theorems express under which conditions $\mathscr{D}$-Dom completeness, Dom-$\mathscr{D}$ completeness, and domain completeness are preserved when deriving propagators. We omit the proofs of these theorems, as they are analogous to the proof of Theorem 7.13, using Lemma 7.15.

Theorem 7.16 Let $\mathscr{D}$ be a domain system and let p be a $\mathscr{D}$-Dom-complete propagator. For any $\mathscr{D}$-injective view φ, the propagator $\hat{\varphi}(p)$ is $\mathscr{D}$-Dom-complete. *

Theorem 7.17 Let $\mathscr{D}$ be a domain system and let p be a Dom-$\mathscr{D}$-complete propagator. For any $\mathscr{D}$-surjective view φ, the propagator $\hat{\varphi}(p)$ is Dom-$\mathscr{D}$-complete. *

Theorem 7.18 Let p be a domain-complete propagator, and let φ be a view. Then $\hat{\varphi}(p)$ is domain-complete. *

7.5 More Properties of Derived Propagators

This section discusses how views can be composed, how derived propagators behave with respect to idempotency and subsumption, and how events can be used to schedule derived propagators.

Composing views

A derived propagator permits further derivation. Consider a propagator p and two views φ, φ'. Then $\widehat{\varphi'}(\hat{\varphi}(p))$ is a perfectly acceptable derived propagator, and properties like correctness and completeness carry over transitively. For instance, we can derive a propagator for $[\![x - y = c]\!]$ from a propagator for $[\![x + y = 0]\!]$, combining an *offset view* ($\varphi_y(v) = v + c$) and a *minus view* ($\varphi'_y(v) = -v$) on y. This yields a propagator for $[\![x + (-(y + c)) = 0]\!] = [\![x - y = c]\!]$.

Fixed points

Section 5.4 showed how to optimize the scheduling of propagators that are known to be at a fixed point. Views preserve fixed points of propagators, so the same optimizations apply to derived propagators.

Proposition 7.19 Let p be a propagator, let φ be a view, and let d be a domain. If $\varphi(d)$ is a fixed point of p, then d is a fixed point of $\hat{\varphi}(p)$. *

Proof. Assume $p(p(\varphi(d))) = p(\varphi(d))$. We have to show $\hat{\varphi}(p)(d) = \hat{\varphi}(p)(\hat{\varphi}(p)(d))$. With the assumption, we can write $\hat{\varphi}(p)(d) = (\varphi^- \circ p \circ p \circ \varphi)(d)$. We know that $\varphi \circ \varphi^-(c) = c$ if $|\varphi^-(c)| = |c|$. As we first apply φ, this is the case here, so we can add $\varphi \circ \varphi^-$ in the middle, yielding $(\varphi^- \circ p \circ (\varphi \circ \varphi^-) \circ p \circ \varphi)(d)$. With function composition being associative, this is equal to $\hat{\varphi}(p)(\hat{\varphi}(p)(d))$. ∎

Subsumption

A propagator is subsumed by a domain d if and only if for all stronger domains $d' \subseteq d$, $p(d') = d'$. Subsumed propagators cannot do any pruning in the remaining subtree of the search, and can therefore be removed (see Section 5.3). Deciding subsumption is coNP-complete in general, but for many practically relevant propagators an approximation can be decided easily. The following theorem states that the approximation is also valid for the derived propagator.

Proposition 7.20 Let p be a propagator and let φ be a view. The propagator $\hat{\varphi}(p)$ is subsumed by a domain d if and only if p is subsumed by $\varphi(d)$. *

Proof. The definition of φ implies that $\forall d' \subseteq d : \varphi^-(p(\varphi(d'))) = d'$ is equivalent to $\forall d' \subseteq d : \varphi^-(p(\varphi(d'))) = \varphi^-(\varphi(d'))$. As φ^- is a function, and because it preserves contraction (see Proposition 7.10), this is equivalent to $\forall d' \subseteq d : p(\varphi(d')) = \varphi(d')$. This can be rewritten to $\forall d'' \subseteq \varphi(d) : p(d'') = d''$ because all $\varphi(d')$ are subsets of $\varphi(d)$. ∎

Events and propagation conditions

Section 5.5 introduced propagation conditions for efficient scheduling of propagators. If a propagator p depends on a variable x with propagation condition π, what propagation condition does a propagator derived from p depend on?

Let us assume that we are dealing with an event system that provides at least the events asn (signaling assignment) and dmc (signaling arbitrary domain changes), and possibly others. Given a propagator p and a view φ, for each variable x that p is subscribed to with propagation condition π_x, we construct a set of events e_x as a safe approximation as follows. If $\mathsf{asn} \in \pi_x$, put $\mathsf{asn} \in e_x$. For any other event $e \in \pi_x$, put $\mathsf{dmc} \in e_x$. Finally, subscribe $\hat{\varphi}(p)$ to x with the smallest propagation condition π' such that for all $e \in e_x$, we have $e \in \pi'$.

We can use asn events because φ_x is injective, and injectivity implies that $|d(x)| = 1 \Leftrightarrow |\varphi_x(d(x))| = 1$. If and only if a variable is assigned, it also appears as assigned under the view.

Concerning events for integer variables, if φ_x is monotonic with respect to the order on V (meaning that $a < b$ implies $\varphi_x(a) < \varphi_x(b)$), we can also use bounds events. If φ_x is anti-monotonic with respect to that order, we have to switch lbc with ubc.

Example 7.21 (Deriving propagation conditions) Consider the following propagator for the constraint $[\![x \leq \max(y_1, y_2)]\!]$:

$$p(d)(z) = \begin{cases} d(x) \cap \{-\infty, \ldots, \max\{\max(d(y_1)), \max(d(y_2))\}\} & \text{if } z = x \\ d(z) & \text{otherwise} \end{cases}$$

Given the event system from Example 5.7 in Section 5.2, we subscribe the propagator to y_1 and y_2 with the propagation condition $\{\mathsf{ubc}\}$, as it can only contribute when an upper bound change happens.

Using minus views, we can derive a propagator $\hat{\varphi}(p)$ for $[\![x \geq \min(y_1, y_2)]\!]$ (which is equivalent to $[\![-x \leq \max(-y_1, -y_2)]\!]$, see Section 8.1). Minus views are anti-monotonic with respect to the natural order on integers: when the upper bound is decreased for a variable x, the lower bound of $-x$ increases. Hence, we have to subscribe $\hat{\varphi}(p)$ to y_1 and y_2 with the propagation condition $\{\mathsf{lbc}\}$. ∗

7.6 Related Work

While the idea to systematically derive propagators using views is novel, there are a few related approaches we can point out.

► Reusing functionality (like a propagator) by wrapping it in an adaptor (like a view) is of course a much more general technique—think of higher-order functions like fold or map in functional programming languages; or chaining command-line tools in Unix operating systems using pipes.

► Views that perform arithmetic transformations are related to the concept of indexicals (see Carlsson et al., 1997; Van Hentenryck et al., 1998). An indexical is a propagator that prunes a single variable and is defined in terms of range expressions. We will see a similar approach in the context of set constraints in Chapter 11. In contrast to views, range expressions can involve multiple variables, but on the other hand only operate in one direction. For instance, in an indexical for the constraint $[\![x = y + z]\!]$, the range expression $y + z$ would be used to prune the domain of x, but not for pruning the domains of y or z. Views must support reasoning in both directions, which is why they are limited in expressivity.

► Instead of regarding a view φ as *transforming* a constraint c, we can regard φ as *additional* constraints, implementing the decomposition. Assuming the significant variables of c are $x_1, \ldots, x_n$, we use additional variables $x'_1, \ldots, x'_n$. Instead of c, we use $c' = c[x_1/x'_1, \ldots, x_n/x'_n]$, which enforces the same relation as c, but on $x'_1, \ldots, x'_n$. Finally, we have n *view constraints* $c_{\varphi,i}$, each equivalent to the relation $x'_i = \varphi_i(x_i)$. The solutions of the decomposition model, restricted to the $x_1, \ldots, x_n$, are exactly the solutions of the original view-based model.

Every view constraint $c_{\varphi,i}$ shares exactly one variable with c and no variable with any other $c_{\varphi,i}$. Thus, the constraint graph is Berge-acyclic (Beeri et al., 1983), and we get a fixed point by first propagating all the $c_{\varphi,i}$, then propagating $c[x_1/x'_1, \ldots, x_n/x'n]$, and then again propagating the $c_{\varphi,i}$. This is exactly what $\varphi^- \circ p \circ \varphi$ does. Constraint solvers typically do not provide any means of specifying the propagator scheduling in such a fine-grained way. Thus, deriving propagators using views is also a technique for specifying perfect propagator scheduling.

On a more historical level, we can relate a derived propagator to the notion of *path consistency*. A domain is path-consistent for a set of constraints, if for any subset $\{x, y, z\}$ of its variables, $v_1 \in d(x)$ and $v_2 \in d(y)$ implies that there is a value $v_3 \in d(z)$ such that the pair (v_1, v_2) satisfies all the (binary) constraints between x and y, the pair (v_1, v_3) satisfies all the (binary) constraints between x and z, and the pair (v_3, v_2) satisfies all the (binary) constraints between z and y (Mackworth, 1977). If $\widehat{\varphi}(p)$ is domain-complete for $\varphi^-(c)$, then it achieves path consistency for the constraint $c[x_1/x'_1, \ldots, x_n/x'_n]$ and all the $c_{\varphi,i}$ in the decomposition model.

8 Deriving Propagators Using Views

This chapter explores a number of different techniques for deriving propagators using views, and thus reveals the broad scope of applications for views and derived propagators.

Structure of the chapter. We employ views for algebraic *transformations* such as Boolean negation or set complement (8.1), for *generalization*, a technique that derives more complex propagators from simpler propagators (8.2), for *specialization* of propagators using constant views (8.3), and for *converting* between different types of variable domains (8.4). We finally discuss limitations of views (8.5).

8.1 Transformation

This section discusses views and derived propagators for Boolean variables where $V = \{0, 1\}$. Not surprisingly, the only view apart from identity for Boolean variables captures negation. That is, using a *negation view* on x defines $\varphi_x(v) = 1 - v$ for $x \in X$ and $v \in V$.

Negation views are more widely applicable than one would initially believe. They demonstrate how views can be used systematically to obtain implementations of constraint variants by *transformation*.

Boolean connectives

The immediate application of negation views is to derive propagators for all Boolean connectives from just three propagators: A negation view for x in $x = y$ yields a propagator for $\neg x = y$. From disjunction $x \vee y = z$ one can derive conjunction $x \wedge y = z$ with negation views on x, y, z, and implication $x \rightarrow y = z$ with a negation view on x. From equivalence $x \leftrightarrow y = z$ one can derive exclusive or $x \oplus y = z$ with a negation view on z.

As Boolean constraints are widespread in models, it pays off to optimize frequently occurring cases. One important propagator is disjunction $\bigvee_{i=1}^{n} x_i = y$ for arbitrarily many variables; again conjunction can be derived with negation views on the x_i and on y. Another important propagator implements the constraint $\bigvee_{i=1}^{n} x_i = 1$, stating

that the disjunction must be true. A propagator for this constraint is essential as the constraint occurs frequently and as it can be implemented efficiently using watched literals, see for example Gent et al. (2006b). With views and derived propagators all implementation work is readily reused for conjunction. This shows a general advantage of views: effort put into optimizing a single propagator implementation directly pays off for all other propagators derived from it.

Boolean cardinality

Like the constraint $\bigvee_{i=1}^{n} x_i = 1$, the Boolean cardinality constraint $\sum_{i=1}^{n} x_i \geq c$ occurs frequently and can be implemented efficiently using watched literals (requiring $c+1$ watched literals, Boolean disjunction corresponds to the case where $c = 1$). But also a propagator for $\sum_{i=1}^{n} x_i \leq c$ can be derived using negation views with the following transformation:

$$\begin{array}{lllll} \sum_{i=1}^{n} x_i \leq c & \iff & -\sum_{i=1}^{n} x_i \geq -c & \iff & n - \sum_{i=1}^{n} x_i \geq n - c \\ & \iff & \sum_{i=1}^{n} 1 - x_i \geq n - c & \iff & \sum_{i=1}^{n} \neg x_i \geq n - c \end{array}$$

Reification

Many reified constraints (such as $(\sum_{x=1}^{n} x_i = c) \leftrightarrow b$) also exist in a negated version (such as $(\sum_{x=1}^{n} x_i \neq c) \leftrightarrow b$). Deriving the negated version is trivial by using a negation view on the Boolean control variable b. This contrasts nicely with the effort without views: either the entire code must be duplicated or the parts that perform checking whether the constraint or its negation is entailed must be factorized out and combined differently for the two variants.

Transformation using integer views

The integer equivalent to negation views for Boolean variables are *minus views:* a minus view on an integer variable x is defined as $\varphi_x(v) = -v$. Minus views help to derive propagators following simple transformations: for example, $\min(x, y) = z$ can be derived from $\max(x, y) = z$ by using minus views for x, y, and z.

Transformations through minus views can improve performance in subtle ways. Consider a bounds($\mathbb{Z}$)-complete propagation algorithm for multiplication $x \times y = z$. Propagation depends on whether zero is still included in the domains of x, y, or z. Testing for inclusion of zero each time the algorithm is executed is inefficient. Instead, one would like to rewrite the propagator to special variants where x, y, and z are either strictly positive or negative. These variants can propagate more efficiently, in particular because propagation can easily be made idempotent. Instead of implementing three different propagators (all variables strictly positive, x or y strictly positive, only z strictly positive), a single propagator assuming that all views are positive is sufficient. The other propagators can be derived using minus views.

Transformation using set views

Set constraints deal with variables whose domains are sets of finite sets. This power set lattice is a Boolean algebra, so typical constraints are constructed from the Boolean primitives disjunction (union), conjunction (intersection), and negation (complement), and the relations equality and implication (subset).

As for Boolean and integer variables, views on set variables enable transformation. Using *complement views* (analogous to Boolean negation) on x, y, z with a propagator for $x \cap y = z$ yields a propagator for $x \cup y = z$. A complement view on y gives us $x \setminus y = z$.

As Chapter 9 will show, views on integer variables can be implemented without any overhead compared to a specialized implementation. For set variables, this is often not the case, as the experiments we will see in Section 10.7 suggest. Chapter 11 will present an alternative technique for deriving propagators for constraints on set variables.

8.2 Generalization

Common views for integer variables capture linear transformations of the integer values: an *offset view* for $o \in \mathbb{Z}$ on x is defined as $\varphi_x(v) = v + o$, and a *scale view* for $a \in \mathbb{Z}$ on x is defined as $\varphi_x(v) = a \times v$.

Generalization using arithmetic views

Offset and scale views are useful for generalizing propagators. Generalization has two key advantages: simplicity and efficiency. A more specialized propagator is often simpler to implement than a generalized version. The possibility to use the specialized version when the full power of the general version is not required may save memory and run-time during execution.

We can devise an efficient propagation algorithm for a linear equality constraint $\sum_{i=1}^{n} x_i = c$ for the common case that the linear equation has only unit coefficients. The more general case $\sum_{i=1}^{n} a_i x_i = c$ can be derived by using scale views for a_i on x_i (the same technique of course applies to linear inequality and disequality rather than equality). Similarly, a propagator for *all-different*$(x_1, \ldots, x_n)$ can be generalized to *all-different*$(c_1 + x_1, \ldots, c_n + x_n)$ by using offset views for $c_i \in \mathbb{Z}$ on x_i. Likewise, from a propagator for the element constraint $a_x = y$ for integers $a_1, \ldots, a_n$ and integer variables x and y, we can derive the generalized version $a_{x+o} = y$ with an offset view, where $o \in \mathbb{Z}$ provides a useful offset for the index variable x.

The presented generalizations can be applied to domain- as well as bounds-complete propagators.

Derived bounds propagators

While most Boolean propagators are domain-complete, completeness with respect to approximations plays an important role for integer propagators. Section 4.2 introduced the interval approximation for integer variables, and the according notions of bounds($\mathbb{Z}$), bounds($\mathbb{R}$), bounds(D), and range completeness. Furthermore we have seen in Section 7.4 that, given appropriate $\mathscr{D}^{[\mathbb{Z}]}$-surjective and/or $\mathscr{D}^{[\mathbb{Z}]}$-injective views, the different notions of bounds consistency are preserved when deriving propagators.

The views for integer variables presented in this section have the following properties: minus and offset views are $\mathscr{D}^{[\mathbb{Z}]}$-bijective, whereas a scale view for $a \in \mathbb{Z}$ on x is always $\mathscr{D}^{[\mathbb{Z}]}$-injective and only $\mathscr{D}^{[\mathbb{Z}]}$-bijective if $a = 1$ or $a = -1$ (in which cases it coincides with the identity view or a minus view, respectively).

In Section 7.4, we have not looked at bounds($\mathbb{R}$)-complete propagators yet. It turns out that a propagator derived from a bounds($\mathbb{Z}$)-complete propagator and a $\mathscr{D}^{[\mathbb{Z}]}$-injective but not $\mathscr{D}^{[\mathbb{Z}]}$-surjective view is only bounds($\mathbb{R}$)-complete. This is exactly what we would expect from a propagator for linear equations, as the next example demonstrates.

Example 8.1 (Linear constraints) We have seen above how to derive a propagator for the constraint $\sum_{i=1}^{n} a_i x_i = c$ from a propagator for $\sum_{i=1}^{n} x_i = c$ using scale views. For the original constraint $\sum_{i=1}^{n} x_i = c$, we can implement an efficient bounds($\mathbb{Z}$)-complete propagator. Using scale views, we turn it into a bounds($\mathbb{R}$)-complete propagator for $\sum_{i=1}^{n} a_i x_i = c$. As already mentioned in Section 4.4, Choi et al. (2004) showed that bounds($\mathbb{Z}$)-complete propagation is NP-hard, so the derived propagator has exactly the same propagation strength as a propagator that one would implement by hand. *

8.3 Specialization

We employ *constant views* to specialize propagators. A constant view behaves like an assigned variable. In practice, specialization has two advantages: Fewer variables are needed, which means less memory consumption. And specialized propagators can be compiled to more efficient code, if constants are known at compile time.

Examples for specialization are

- a propagator for binary linear inequality $x + y \leq c$ derived from a propagator for $x + y + z \leq c$ by using a constant 0 for z;
- a Boolean propagator for $x \wedge y \leftrightarrow 1$ from $x \wedge y \leftrightarrow z$ and constant 1 for z;
- a propagator for the element constraint $a_x = y$ for a sequence of integer constants $a_1, \ldots, a_n$ and variables x and y derived from a propagator for $z_x = y$ for a sequence of integer variables $z_1, \ldots, z_n$;
- a reified propagator for $(x = c) \leftrightarrow b$ from $(x = y) \leftrightarrow b$ and a constant c for y;
- a propagator for the counting constraint $|\{i \mid x_i = y\}| = c$ from a propagator for $|\{i \mid x_i = y\}| = z$;
- a propagator for set disjointness from a propagator for $x \cap y = z$ and a constant empty set for z; and many more.

We have to extend our model to support constant views. Propagators may now be defined with respect to a superset of the variables, $X' \supseteq X$. A constant view for the value k on a variable $z \in X' \setminus X$ translates between the two sets of variables as follows:

$$\begin{aligned} \varphi(c) &= \{a[k/z] \mid a \in c\} \\ \varphi^-(c) &= \{a_{|X} \mid a \in c\} \end{aligned}$$

Here, $a[k/z]$ means augmenting the assignment a so that it maps z to k, and $a_{|X}$ is the functional restriction of a to the set X.

It is important to see that this definition preserves failure. If a propagator returns a failed domain d that maps z to the empty set, then $\varphi^-(d)$ is the empty set, too (recall that this is really $\varphi^-(\mathrm{con}(d))$, and $\mathrm{con}(d) = \emptyset$ if $d(z) = \emptyset$).

8.4 Type Conversion

A type conversion view lets propagators for one type of variable work with a different type, by translating the underlying representation. Our model already accommodates for this, as a view φ_x maps elements between different sets V and V'.

Converting between different variable types

Boolean variables are essentially integer variables with an implementation that is optimized for the special domain $\{0, 1\}$. It is thus straightforward to wrap a Boolean variable in an *integer view*. That way, all propagators for integer constraints can be directly reused with Boolean variables.

Another type conversion view is a *singleton view* on an integer variable x, defined as $\varphi_x(v) = \{v\}$. It presents an integer variable as a singleton set variable. Many useful constraints involve both integer and set variables, and some of them can be expressed with singleton views. The simplest constraint is $x \in y$, where x is an integer variable and y a set variable. Singleton views let us implement it as $\{x\} \subseteq y$, and just as easily give us the negated and reified variants. Obviously, this extends to $\{x\} \diamond y$ for all other set relations $\diamond$.

Singleton views can also be used to derive pure integer constraints from set propagators. For example, the constraint same($[x_1, \ldots, x_n], [y_1, \ldots, y_m]$) states that the two sequences of integer variables take the same values. With singleton views, $\bigcup_{i=1}^{n} \{x_i\} = \bigcup_{j=1}^{m} \{y_j\}$ implements this constraint.

Converting between different domain implementations

Most systems approximate set variable domains as set intervals defined by a lower and an upper bound (see Section 4.5). However, Hawkins et al. (2005) introduced a representation for the complete domains of set variables, using ROBDDs. Type conversion views can translate between set interval and ROBDD-based implementations. We can derive a propagator on ROBDD-based variables from a set interval propagator, and thus reuse set interval propagators for which no efficient ROBDD representation exists.

8.5 Limitations

Although views are widely applicable, they are no silver bullet. This section explores some limitations of the presented architecture.

Beyond injective views

Views as defined in the previous chapter are required to be injective. This excludes some interesting views, such as a view for the absolute value of a variable, or a view of a variable modulo some constant. None of the basic proofs makes use of injectivity, so non-injective views can be used to derive complete, correct propagators.

However, event handling changes when views are not injective:

- A domain change event dmc on a variable does not necessarily translate to a dmc event on the view. For instance, given a domain d with $d(x) = \{-1, 0, 1\}$, removing the value -1 from x produces a dmc event on x, but not on abs(x).
- A dmc event on a variable may result in a asn event on the view. For instance, removing 0 instead of -1 in the above example results in $d(x) = \{-1, 1\}$, but in abs(x) there is only a single value left.

These effects may lead to unnecessary propagator invocations, or even to incorrect behavior if a propagator relies on the accuracy of the reported event. We want events to be reliable in this sense, so we decided to not allow non-injective views.

Multi-variable views

Some multi-variable views that seem interesting for practical applications do not preserve contraction, for instance a view on the sum or product of two variables. The reason is that removing a value through the view would have to result in removing a *tuple* of values from the domain. As domains can only represent Cartesian products, this is not possible in general. For views that do not preserve contraction, Proposition 7.20 does not hold. That means that a propagator p cannot easily detect subsumption any longer, as it would have to detect it for $\widehat{\varphi}(p)$ instead of just for itself, p. We consider optimizing subsumption vital for performance, so we only allow contraction-preserving views.

For contraction-preserving views on multiple variables, all our theorems still hold. Some views we could identify are

- A set view of Boolean variables $[b_1, \ldots, b_n]$, behaving like $\{i \mid b_i = 1\}$.
- An integer view of Boolean variables $[b_1, \ldots, b_n]$, where b_i is 1 if and only if the integer has value i.
- The inverse views of the two views above.

These views are of limited use, and the decomposition approach will probably work just as well in these cases.

Propagator invariants

Propagators typically rely on certain invariants of a variable domain implementation. If idempotency or completeness of a propagator depend on these invariants, type conversion views lead to problems, as the actual variable implementation behind the view may not respect the same invariants.

For example, a propagator for set variables based on the set interval approximation can assume that adjusting the lower bound of a variable does not affect its upper bound. If this propagator is instantiated with a type conversion view for an ROBDD-based set variable, this invariant is violated: if, for instance, the current domain is $\{\{1,2\},\{3\}\}$, and 1 is added to the lower bound, then 3 is removed from the upper bound (in addition to 2 being added to the lower bound). If a propagator reports that it has computed a fixed point based on the assumption that the upper bound cannot have changed, it may actually not be at a fixed point. This potentially results in incorrect propagation, for instance if the propagator could detect failure if it were run again.

9 Implementing Views

This chapter presents an implementation architecture for views and derived propagators. The implementation is based on making propagators parametric, and is an orthogonal layer of abstraction on top of the actual solver implementation.

While Chapter 7 proved that derived propagators are perfect with respect to the mathematical model, this chapter shows that one can also obtain perfect implementations of derived propagators: in many cases, an implementation based on C++ templates incurs no performance penalties, as modern optimizing compilers perform aggressive inlining and constant folding. Furthermore, this chapter analyzes the massive impact views have on the amount of code that needs to be written: for Gecode, views save around 120 000 lines of code and documentation. Finally, we evaluate the performance of derived propagators empirically, using Gecode.

Structure of the chapter. This chapter first develops the implementation architecture for parametric propagators (9.1), and then presents implementation techniques for parametric and constant views (9.2). Next, we discuss how the implementation of views handles events (9.3). Finally, we empirically evaluate the applicability of the presented techniques, as well as the performance of derived propagators (9.4).

9.1 Parametric Propagators

Chapter 7 introduced a mathematical model for views. A view is a function that transforms the input and output of a propagator, which maps domains to domains. In the object-oriented implementation model from Section 6.3, a propagator is no longer a function, but an object with a `propagate` method that *modifies* the current domain, accessing and updating the domain through variable objects that provide methods for domain operations.

This section shows how to derive propagators in the implementation model. The main idea is to replace the propagators' references to variable objects by references to view objects. A view object has the same interface, the same domain operations, as a variable object, but performs the additional transformations like negation or scaling. The mechanism that we use for replacing a propagator's variables with different views is *parametricity*.

Parametricity

Abstraction through types (as defined by Reynolds, 1983) is one of the most powerful mechanisms of abstraction in typed programming languages. Essentially, the same code can be instantiated with different *parameters* of compatible types (in the case considered here, the same propagator will be instantiated with different views). Modern programming languages provide three forms of parametricity:

Functional parametricity means that in programming languages with higher-order functions such as Standard ML (Milner et al., 1997) or Haskell (Peyton Jones, 2003), a higher-order function is parametric over its arguments.

Dynamic binding is typically coupled with inheritance in object-oriented languages, and realized for instance as virtual function calls in C++ or method calls in Java.

Parametric polymorphism comes, for example, in the form of templates in C++, Java generics (Gosling et al., 2005), or Standard ML functors.

The following three examples show how to derive propagators using different kinds of parametricity. All three examples derive a propagator for $x = y + 2$ from a propagator for $x = y$ and an offset view. The code is simplified for presentation, it does not show a full implementation but conveys the general idea. In particular, we omit handling failure to keep the code more compact.

Example 9.1 (ML functions) Figure 9.1 sketches how integer variables, a propagator for equality, and offset views could be implemented in Standard ML. The propagator takes two variables as arguments, where the type of variables is only defined by the record of operations: `min`, `max`, `adjmin`, and `adjmax`. The function `mkOffsetView` turns a variable into an offset view. The original variable and the offset are bound in the scope of the view. In order to propagate $x = y + 2$, one can simply call the equality propagator with an offset view instead of a variable for `y`:

```
equal(x,mkOffsetView(y,2))
```
*

Example 9.2 (Java methods) When using dynamic binding in Java, an offset view class implements the same interface as integer variables, see Figure 9.2. The view uses a mechanism known as *delegation:* it encapsulates an integer variable and delegates the operations to the encapsulated variable, applying the transformations. The propagator calls the view operations via dynamic binding. Given integer variables `x` and `y`, a derived propagator for $x = y + 2$ is instantiated as follows:

```
new Eq(x, new OffsetView(y, 2));
```
*

Example 9.3 (C++ templates) Figure 9.3 shows an interface for integer variables, an implementation of offset views, and a parametric equality propagator in C++. The example still uses delegation as in Java. The difference is that the offset views do not inherit from the integer variables, they just have the same signature of operations. The propagator is *parametric* over the types of the two views it uses, and can be

```
type variable = { min : unit -> int, max : unit -> int,
                  adjmin : int -> unit, adjmax : int -> unit }

fun equal (x : variable, y : variable) =
  ( #adjmin x (#min y ());
    #adjmax x (#max y ());
    #adjmin y (#min x ());
    #adjmax y (#max x ()) )

fun mkOffsetView (x : variable, offset) =
  { min = fn () => offset + #min x (),
    max = fn () => offset + #max x (),
    adjmin = fn newMin => #adjmin x (newMin - offset),
    adjmax = fn newMax => #adjmax x (newMax - offset) }
```

Figure 9.1: Offset views and an equality propagator in ML

instantiated with any view that provides the necessary operations. This is how to instantiate the parametric equality propagator so that it implements $x = y + 2$, given two pointers to integer variables x and y:

```
new Eq<IntVar,OffsetView>(x,new OffsetView(y,2));
```

*

Views are orthogonal. The examples make clear that views are independent of the concrete solver implementation. They form an orthogonal layer of abstraction on top of any propagation-based constraint solver. As long as the implementation language provides some kind of parametricity, and variable domains are accessed through some form of variable objects, propagators can be derived using views.

Which kind of parametricity to choose

Most object oriented programming languages provide the choice between dynamic binding and parametric polymorphism. In particular in C++, the implementation language we chose for Gecode, both dynamic binding and parametric polymorphism have advantages and disadvantages as it comes to deriving propagators.

Parametric polymorphism is compiled by *monomorphization:* the code is replicated and specialized for each instance and then compiled individually. The compiler can generate optimized code for each instance, for example by inlining the transformations that a view implements. That way, a parametric propagator does not pay for additional function invocations when using views. All C++ compilers and some implementations of SML (such as MLTon, 2009) employ monomorphization.

Achieving high efficiency in C++ with templates sacrifices expressiveness. Instantiation can *only* happen at compile-time. Hence, either C++ must be used for modeling,

```
interface IntVar {
  public int min(void);     public int max(void);
  public void adjmin(int);  public void adjmax(int);
}

class OffsetView implements IntVar {
  protected IntVar v;  protected int offset;
  public OffsetView(IntVar v0, int o0) { v = v0; offset = o0; }
  public int min() { return v.min()+offset; }
  public int max() { return v.max()+offset; }
  public void adjmin(int newMin) { v.adjmin(newMin-offset); }
  public void adjmax(int newMax) { v.adjmax(newMax-offset); }
}

class Eq extends Propagator {
  protected IntVar x;  protected IntVar y;
  public Eq(IntVar x0, IntVar y0) { x = x0; y = y0; }
  public void propagate() {
    x.adjmin(y.min()); x.adjmax(y.max());
    y.adjmin(x.min()); y.adjmax(x.max());
  }
}
```

Figure 9.2: Offset views and an equality propagator in Java

or all potentially required propagator variants must be provided by explicit instantiation. The *choice* which propagator to use can however be made at run-time: for linear equations, for instance, if all coefficients are units, or all are positive, the respective optimized derived propagators can be posted.

For n-ary constraints, compile-time instantiation can be a real limitation, as all arrays must be monomorphic (only a single kind of view per array is allowed). For example, one cannot mix scale and minus views in linear constraints, or use constant views in n-ary set constraints. For some propagators, we can work around this restriction using more than a single array of views. For example, a propagator for a linear constraint can employ two arrays of different view types, one of which may then be instantiated with identity views and the other with minus views.

The advantage of dynamic binding is its greater flexibility, because instantiation happens at run-time. Thus arbitrary combinations of views can be used, and arrays can hold different types of views at the same time. This flexibility comes at the cost of reduced efficiency, as the transformations done by view operations typically cannot be inlined and optimized, but require additional virtual method calls.

In Java, parametric polymorphism is not implemented by monomorphization, but by type erasure and virtual method calls. The potential for optimization is thus not fully utilized. However, modern Java virtual machines employ just-in-time compila-

```
class IntVar {
private: int _min, _max;
public:  int min(void);     int max(void);
         void adjmin(int);  void adjmax(int);
};

class OffsetView {
protected:  IntVar* v;  int offset;
public:     OffsetView(IntVar* v0, int offset0) : v(v0), offset(offset0) {}
            int min(void) { return v->min()+offset; }
            int max(void) { return v->max()+offset; }
            void adjmin(int newMin) { v->adjmin(newMin-offset); }
            void adjmax(int newMax) { v->adjmax(newMax-offset); }
};

template <class View0, class View1>
class Eq : public Propagator {
protected: View0* x; View1* y;
public:    Eq(View0* x0, View1* y0) : x(x0), y(y0) {}
           virtual void propagate(void) {
             x->adjmin(y->min()); x->adjmax(y->max());
             y->adjmin(x->min()); y->adjmax(x->max());
           }
};
```

Figure 9.3: Offset views and a parametric equality propagator in C++

tion, optimizing the code at run-time, which may make up for missed optimization opportunities at (static) compile time.

In Gecode, polymorphic propagators are based on parametric polymorphism using C++ templates. Section 9.4 presents empirical evidence that this choice results in a highly efficient implementation. The examples in the rest of this chapter are presented as C++ code.

9.2 Parametric and Constant Views

Views do not have to be defined with respect to variables, but can themselves be parametric. This corresponds to the observation in Section 7.5 that views φ and φ' can be composed so that $\hat{\varphi}(\widehat{\varphi'}(p))$ is again a derived propagator.

The following code snippet implements a parametric minus view.

```
template <class View>
class MinusView {
protected:  View* v;
public:     MinusView(View* v0) : v(v0) {}
            int min(void) { return -v->max(); }
            int max(void) { return -v->min(); }
            void adjmin(int newMin) { v->adjmax(-newMin); }
            void adjmax(int newMax) { v->adjmin(-newMax); }
};
```

Section 8.3 introduced constant views for specializing propagators. A constant integer view looks as follows.

```
class ConstantIntView {
protected:  int k;
public:     ConstantIntView(int k0) : k(k0) {}
            int min(void) { return k; }
            int max(void) { return k; }
            void adjmin(int newMin) { if (newMin>k) fail(); }
            void adjmax(int newMax) { if (newMax<k) fail(); }
};
```

Recall that a constant view was defined as $\varphi^-(c) = \{a_{|X} \mid a \in c\}$, it reports failure when a propagator returns an empty domain (see Section 8.3). The methods `adjmin` and `adjmax` implement the test for emptiness and report failure accordingly (assuming a library function `fail`).

Compile-time versus run-time constants. Some of the views presented earlier involve a parameter, such as the coefficient of a scale view, the offset of an offset view, or the constant of a constant view. These parameters can again be instantiated at compile-time or at run-time. For instance, one can regard a minus view as a compile-time specialization of a scale view with coefficient -1. In the same way, a zero view or a one view may specialize a constant view. The advantage is again that the compiler can apply more aggressive optimizations, as the constants are now known at compile time. And again, the increased potential for optimization is paid for by a decrease in flexibility.

9.3 Event Handling

This section shows how views implement event handling. It combines the results from Section 7.5 with the implementation architecture from Section 6.3.

Subscribing and canceling

Events (and hence modification events) have to be adapted when working with views. For instance, Section 7.5 argued that for minus views on integer variables, the lower and upper bound events have to be swapped. Views therefore provide `subscribe` and `cancel` methods that perform the required transformations. Here is an example `subscribe` method for minus views (`cancel` works analogously).

```
void subscribe(Propagator& p, PropagationCondition pc) {
  switch (pc) { case {asn,lbc}: v->subscribe(p, {asn,ubc}); break;
                case {asn,ubc}: v->subscribe(p, {asn,lbc}); break;
                default: v->subscribe(p, pc); }
}
```

Domain operations and the modification event delta

As seen in Section 6.3, variable domain operations return status messages to the propagators. While the basic messages, $\emptyset$, FAIL, and even the modification event $\mathsf{me}_{\mathsf{asn}}$, are invariant under any view, the other modification events may require transformation. The domain operations on a minus view, for example, swap the $\mathsf{me}_{\mathsf{lbc}}$ and $\mathsf{me}_{\mathsf{ubc}}$ modification events before returning to the propagator.

In order to enable staging, Section 6.3 introduced the modification event delta. It represents the set of events that happened since the last invocation of the propagator. When using views, this set must be interpreted *under the particular view*. For example, the propagator must interpret an lbc event on a variable as an ubc event if it sees the variable under a minus view. Views therefore provide conversion methods that the propagators must use when dealing with the modification event delta.

9.4 Applicability and Performance Analysis

To conclude this chapter, we now argue that views have proven crucial for implementing Gecode, and that derived propagators are efficient in practice.

Applicability. Gecode makes heavy use of views. Table 9.1 shows the number of parametric propagators implemented in Gecode, and the number of derived instances. On average, every parametric propagator results in four instances. Propagators in Gecode account for more than 40 000 lines of code and documentation. As a rough estimate, deriving propagators using views thus saves around 120 000 lines of code and documentation to be written, tested, and maintained. On the other hand, the views are implemented in less than 8 000 lines of code, yielding a 1500% return on investment.

Variable type	*Parametric propagators*	*Derived propagators*	*Ratio*
Integer	78	304	3.90
Boolean	25	84	3.36
Set	24	126	5.25
Overall	127	514	4.05

Table 9.1: Applicability of views: number of parametric vs. derived propagators

Code inspection. A thorough inspection of the code generated by the GNU C++ compiler and the Microsoft Visual C++ compiler shows that they produce optimal code for derived propagators, actually performing the optimizations we consider essential. Operations on views are inlined entirely and thus implemented in the most efficient way. The abstractions do not impose a run-time penalty (compared to a system without views).

Example 9.4 Figure 9.4 shows a simplified implementation of an integer variable and a polymorphic propagator. When instantiated with an `IntVar` for `x` and an `OffsetView` for `y` (as defined previously in Figure 9.3), the GNU C++ compiler for Intel x86 translates the `propagate` function to the following assembly code:

```
P<IntVar,OffsetView>::propagate:
1       pushl   %ebp
2       movl    %esp,   %ebp
3       movl    8(%ebp),%eax
4       movl    8(%eax),%edx
5       movl    4(%eax),%eax
6       movl    (%eax), %eax
7       subl    4(%edx),%eax
8       movl    (%edx), %edx
9       cmpl    (%edx), %eax
10      jle     L4
11      movl    %eax,   (%edx)
    L4:
12      leave
13      ret
```

The pointer to the propagator object (the **`this`** pointer) is passed on the stack and loaded into register `eax` in line 3. Then the address of `y` is loaded into `edx` in line 4, and the address of `x` into `eax` in line 5. Line 6 loads `x->_min` into `eax`, and line 7 subtracts the offset (stored at the location pointed to by `edx+4`) from `x->_min`. Line 8 loads `y->_min`, which is compared to the result of the subtraction in line 9. In line 11, `y->_min` is set to the result of the subtraction if the latter is greater than the previous minimum. This example shows that neither the operations on the integer variable, nor the transformation of the offset view require any function call, but are compiled inline into the propagation function. *

```
class IntVar {
private:
  int _min, _max;
public:
  int min(void) { return _min; }
  int max(void) { return _max; }
  void adjmin(int m) {
    if (m>_min) _min = m;
  }
  void adjmax(int m) {
    if (m<_max) _max = m;
  }
};
```

```
template <class View0, class View1>
class P : public Propagator {
private:
  View0& x; View1& y;
public:
  P(View0& x0, View1& y0)
    : x(x0), y(y0) {}
  virtual void propagate(void) {
    y.adjmin(x.min());
  }
};
```

Figure 9.4: An integer variable and a simple propagator

Benchmarks

We now analyze the practical impact that views have on efficiency. Two aspects are evaluated: derived propagators versus decomposition, and compile-time polymorphism versus run-time polymorphism. The experiments use the same setup as described in Section 6.9 and Appendix A. All numbers are given as relative numbers compared to the standard Gecode system.

Views versus decomposition. Table 9.2 presents the overhead that results from replacing some view-based propagators by the respective decompositions. Both *Alpha* and *Eq-20* use mainly linear equations with coefficients, which we replaced by a decomposition. For *Queens 100*, we replaced the special *all-different*-with-offsets by its decomposition into an *all-different* propagator and binary equality-with-offset propagators. In *BIBD* and *Perfect Square*, we decomposed ternary Boolean propagators, implementing $x \wedge y \leftrightarrow z$ as $\neg x \vee \neg y \leftrightarrow \neg z$ in *BIBD*, and $x \vee y \leftrightarrow z$ as $\neg x \wedge \neg y \leftrightarrow \neg z$ in *Perfect Square*. The remaining seven examples involve set constraints, and we decomposed an intersection propagator into complement and union propagators.

Benchmark	*time %*	*mem. %*	*prop. %*
Alpha (smart)	457.24	357.14	478.91
Alpha (naive)	682.81	360.87	673.83
Eq-20	655.62	700.00	704.57
Queens (S, Dom, 10)	147.28	100.00	483.87
Queens (S, Dom, 100)	141.31	100.00	2 342.02
Partition (32)	105.66	100.00	117.48
BIBD	291.97	213.54	256.19
Perfect Square	110.49	108.47	104.42

Benchmark	*time %*	*mem. %*	*prop. %*
Steiner Triples (9)	118.45	100.00	85.41
Hamming (20-3-32)	113.37	104.92	100.10
Social G. (8-4-9)	319.58	234.82	160.83
Social G. (5-3-7)	212.65	178.60	149.56
Sudoku (Set, 1)	137.60	100.00	103.57
Sudoku (Set, 4)	132.97	100.00	103.70
Sudoku (Set, 5)	129.82	159.63	103.72

Table 9.2: Relative performance of decomposition, compared to views

Benchmark	*time %*
Alpha (smart)	178.36
Alpha (naive)	157.23
BIBD	143.55
Eq-20	222.92
Golomb Rulers (10)	135.53
Graph Coloring	103.36
Knights (18)	126.58
Magic Sequence (Smart, 500)	122.53
Magic Sequence (GCC, 500)	176.36
Partition (32)	137.74
Perfect Square	129.35
Photo Alignment	140.40
Queens (Smart, 10)	124.89
Queens (Smart, 100)	148.37

Benchmark	*time %*
Crew Scheduling	119.17
Hamming Codes (20-3-32)	120.16
Social Golfers (8-4-9)	130.97
Social Golfers (5-3-7)	120.90
Steiner Triples (9)	127.62
Sudoku (Set, 1)	113.44
Sudoku (Set, 4)	112.41
Sudoku (Set, 5)	111.58
Queen Armies	114.12

Table 9.3: Relative performance of virtual method calls

Some of the integer examples show a significant overhead when decomposed, requiring up to seven times the run-time and memory than using views. The overhead of most set examples as well as *Perfect Square* appears to be more moderate, which is partly due to the fact that no additional variable was introduced if the complement or negation of a variable was already present in the model. Interestingly, the number of propagation steps is in many cases significantly higher for the decomposition model, but as the additional steps are performed by cheap propagators (like $x = y + i$ or $x = \neg y$), the effect on the run-time is less drastic. The 100 Queens benchmark is an extreme example, where we see 23 times the propagation steps but only 41% more run-time. The benchmark *Partition 32* only contains a single linear equation with coefficients, all other constraints are simple linear equations and an *all-different*. Replacing the linear equation with coefficients by its decomposition has only little effect on the run-time (6% overhead).

Templates versus virtual methods. As suggested in Section 9.1, in C++, compile-time polymorphism using templates is far more efficient than virtual method calls. To evaluate this, we changed the basic operations of integer variables to be virtual methods, such that view operations need one virtual method call. This is a conservative approximation of the actual cost of fully virtual views. Set-valued operations (which will be discussed in the next chapter) cannot be made virtual, as they are based on templates. We therefore approximated the effect of virtual methods by preventing inlining of set-valued operations. An implementation based completely on virtual methods will typically exhibit an even higher overhead. The results of these experiments appear in Table 9.3. Virtual method calls cause a run-time overhead between 3% and 123% for the integer examples (left table), and 11% to 31% for the set examples (right table).

10 Range Iterators

This chapter proposes to implement set-valued domain operations using *range iterators*. The operations are shown to be simple, expressive, and efficient.

Domain operations in the preceding chapters only involved a single integer value, for instance adjusting the minimum or maximum of an integer variable. For propagators that perform domain reasoning on integer variables, and for propagators for set-valued variables, single-integer operations are not efficient. Instead, one would like to be able to update a whole integer variable domain to a new set, or exclude a set of elements from a set variable domain. When deriving propagators using views, these set-valued operations must be provided by the views, too, performing the necessary transformations on the given sets.

Range iterators are perfectly suited to act as the interface for set-valued domain operations. Instead of providing a set data structure for passing set-valued arguments around, we pass *operations* for sequentially accessing a set data structure instead, in the form of a range iterator. This additional abstraction yields compact and efficient implementations of views. As a further benefit, range iterators simplify the implementation of many propagation algorithms.

Iterators are a well-known implementation technique in object-oriented programming languages. However, we are not aware of any previous work that identifies iterators as a fundamental and powerful abstraction for implementing constraint solvers.

Structure of the chapter. After introducing range iterators (10.1), we use them to define set-valued operations on integer variables (10.2). Set operations such as union or intersection can be performed directly on range iterators (10.3), and yield set-valued operations on integer views (10.4) and set variables and views (10.5). Implementing propagation algorithms is simplified by using iterators as adaptors (10.6). At the end of the chapter, we analyze the performance of range iterators in practice (10.7).

10.1 Range Iterators

Object-oriented programming has introduced a useful abstraction for representing collections, a design pattern called *iterator*. An iterator provides access to the ele-

ments of a collection in sequential order, one at a time.

A set-valued domain operation supports simultaneous access or update of multiple values of a variable domain. For example, a propagation algorithm may compute with the whole domain of an integer variable, or the upper bound of a set variable. This section shows how range iterators can provide such operations efficiently.

Range sequences. A **range** $[m \,..\, n]$ denotes the set of integers $\{l \in \mathbb{Z} \mid m \leq l \leq n\}$. A **range sequence** ranges(S) for a finite set of integers $S \subseteq \mathbb{Z}$ is the shortest sequence $s = \langle [m_1 \,..\, n_1], \ldots, [m_k \,..\, n_k] \rangle$ such that $S = \bigcup_{i=1}^{k} [m_i \,..\, n_i]$ and the ranges are ordered by their smallest elements ($m_i \leq m_{i+1}$ for $1 \leq i < k$). We thus define the set covered by a range sequence as $\text{set}(s) = \bigcup_{i=1}^{k} [m_i \,..\, n_i]$. The above range sequence is also written as $\langle [m_i \,..\, n_i] \rangle_{i=1}^{k}$. Clearly, the range sequence of a set is unique, none of its ranges is empty, and $n_i + 1 < m_{i+1}$ for $1 \leq i < k$.

A **range iterator** for a range sequence $s = \langle [n_i \,..\, m_i] \rangle_{i=1}^{k}$ is an object that provides iteration over s: each of the $[m_i \,..\, n_i]$ can be obtained in sequential order but only one at a time. A range iterator r provides the following operations: r.`done`() tests whether all ranges have been iterated, r.`next`() moves to the next range, and r.`min`() and r.`max`() return the minimum and maximum value for the current range. By set(r) we refer to the set defined by an iterator r (which must coincide with set(s)).

A possible implementation of a range iterator for s maintains an index i which is initially $i = 1$, the operations can then be defined as:

```
class RangeIterator {
public:
  bool done(void) { return i > k; }
  int min(void) { return m_i; }
  int max(void) { return n_i; }
  void next(void) { i++; }
};
```

Iterators are consumed by iteration. Hence, if the same sequence needs to be iterated twice, a fresh iterator is needed. If iteration is cheap, a reset operation for an iterator can be provided so that multiple iterations are supported by the same iterator. A solution for more expensive iterators is discussed later.

A range iterator hides its implementation. It can iterate a sequence (for instance an array) directly by position as above, but it could just as well traverse a linked list or the leaves of a balanced tree. This abstractness of range iterators makes them perfectly suited as an interface for set-valued domain operations of variables and views.

Related work

► The standard template library (STL) of C++ uses iterators as a universal interface for collection types (Stroustrup, 1997). Here, an iterator does not by itself encode a sequence, but a pair of two iterators represents the sequence of elements *between them*. The Boost iterator library (Abrahams et al., 2009) extends and improves the STL iterators, and provides adaptors such as filters or reversion iterators.

► The Java collection libraries (Horstmann and Cornell, 2004) also use iterators as a fundamental abstraction. Java even provides special iteration syntax in the form of a **foreach** loop.

► Many constraint programming systems provide an abstract set-datatype for accessing and updating variable domains, as for example in CHOCO (2009), ECL^{i}PS^e (2009), SICStus Prolog (2009), and Mozart (2009). ILOG Solver (2009) only allows access by iterating over the values of a variable domain.

10.2 Set-Valued Operations for Integer Variables

The two basic set-valued operations on integer variables are domain access and domain update. For an integer variable x, the operation x.`getdom`() returns a range iterator for ranges$(d(x))$. For a range iterator r the operation x.`setdom`(r) updates the variable domain of x to set(r) provided that set$(r) \subseteq d(x)$. The responsibility for ensuring that set$(r) \subseteq d(x)$ is left to the programmer and hence requires careful consideration. The next section introduces richer (and safe) set-valued operations.

The operation x.`setdom`(r) is parametric with respect to r: any range iterator can be used. As for views, an implementor has to decide on the kind of parametricity to use. Gecode uses template-based parametric polymorphism, with the performance benefits due to monomorphization and consequent code optimization mentioned in Section 9.1.

Set-valued operations can offer a substantial improvement over single-value operations, if many values need to be removed from a variable domain simultaneously. Assume a typical implementation of a variable domain $d(x)$ which organizes ranges$(d(x)) = \langle [m_i \,..\, n_i] \rangle_{i=1}^{k}$ as a linked list. Removing a single element from $d(x)$ takes $O(k)$ time and might increase the length of the linked list by one (introducing an additional hole). Hence, in the worst case, removing l elements takes $O(l(k+l))$ time. With set-valued operations based on iterators, removal takes $O(k+l)$ time, as the update can be implemented as one linear pass over the linked list.

Range iterators serve as a simplistic abstract datatype to describe finite sets of integers. However, they provide some essential advantages over an explicit set representation. First, any range iterator regardless of its implementation can be used to

update the domain of a variable. This turns out to result in simple, efficient, and expressive updates of variable domains. Second, no costly memory management is required to maintain a range iterator as it provides access to only one range at a time. Third, iterators are abstract enough to be compatible with set-valued operations on views, as will be discussed in Section 10.4.

10.3 Computing with Iterators

This section shows that in addition to simple domain access and update, range iterators can also perform computations on sets directly. Iterator computations yield simple yet efficient propagation algorithms.

Propagation algorithms often have to operate with sets. For example, the following propagator induces an equality constraint on two integer variables x and y:

$$p(d)(z) = \begin{cases} d(x) \cap d(y) & \text{if } z = x \text{ or } z = y \\ d(z) & \text{otherwise} \end{cases}$$

Given that both domain access and domain update are provided as iterators, the propagation algorithm must compute the intersection of the sets represented by two iterators, and provide the result as an iterator. An important feature that makes iterators so useful is that many such operations can be implemented directly, without computing intermediate results.

Let us consider intersection as an example for computing with range iterators. Intersection is computed by an intersection iterator $r = \text{iinter}(a, b)$, taking two range iterators a and b as input where $\text{set}(r) = \text{set}(a) \cap \text{set}(b)$. The intersection iterator maintains integers m and n for storing the smallest and largest value of its current range. When initialized, the operation r.`next`() is executed once. The operations appear in Figure 10.1.

The **`do-while`**-loop iterates a and b until their ranges overlap. The tests whether a or b are done ensure that no operation is performed on an iterator that is already done. The remainder computes the resulting range and prepares for computing the next range.

The iterators a and b can be arbitrary iterators (again, the intersection iterator is parametric), so it is easy to obtain an iterator that computes the intersection of three iterators by using two intersection iterators. Intersection is but one example for a parametric iterator, other useful iterators are for instance: $\text{iunion}(a, b)$ for iterating the union of a and b, $\text{iminus}(a, b)$ for iterating the set difference of a and b, and $\text{icompl}(a)$ for iterating the complement of a with respect to some fixed universe.

```
template <class A, class B>
class IntersectionIterator {
private:
  int m, n; A a; B b;
public:
  IntersectionIterator(A a, B b) : a(a), b(b) { next(); }
  bool done(void) { return a.done() || b.done(); }
  int min(void) { return m; }
  int max(void) { return n;}
  void next(void) {
    if (a.done() || b.done()) return;
    do { while (!a.done() && (a.max() < b.min())) a.next();
         if (a.done()) return;
         while (!b.done() && (b.max() < a.min())) b.next();
         if (b.done()) return;
    } while (a.max() < b.min());
    m = max(a.min(), b.min()); n = min(a.max(), b.max());
    if (a.max() < b.max()) { a.next(); } else { b.next(); }
  }
};
```

Figure 10.1: An intersection iterator

Example 10.1 (Propagating equality) Consider a domain-complete propagator that induces equality $x = y$ (assuming that x and y are variables, views are discussed later). The propagator can be implemented as follows: get range iterators for x and y by $r_x = x.\texttt{getdom}()$ and $r_y = y.\texttt{getdom}()$, create an intersection iterator $r_i = \text{iinter}(r_x, r_y)$, update one of the variable domains by $x.\texttt{setdom}(r_i)$, and copy the domain from x to y by $y.\texttt{setdom}(x.\texttt{getdom}())$. *

Cache iterators. The above example suggests that for some propagators it is better to create an intermediate representation of the range sequence computed by an iterator. The intermediate representation can be reused as often as needed. This is achieved by a *cache iterator:* it takes an arbitrary range iterator as input, iterates it completely, and stores the obtained ranges in an array. Its operations then use the array. The cache iterator also implements a reset operation, so that the possibly costly input iterator is used only once, while the cache iterator can be used as often as needed.

Richer set-valued operations. With the help of iterator computations, richer set-valued operations are effortless. For an integer variable x and a range iterator r, the operation $x.\texttt{adjdom}(r)$ adjusts the domain $d(x)$ by $\text{set}(r)$, yielding $d(x) \cap \text{set}(r)$,

whereas $x.\texttt{excdom}(r)$ excludes $\text{set}(r)$ from $d(x)$, yielding $d(x) \setminus \text{set}(r)$:

$$x.\texttt{adjdom}(r) := x.\texttt{setdom}(\text{iinter}(x.\texttt{getdom}(), r))$$
$$x.\texttt{excdom}(r) := x.\texttt{setdom}(\text{iminus}(x.\texttt{getdom}(), r))$$

In contrast to the $x.\texttt{setdom}(\cdot)$ operation, the richer set-valued operations are inherently contracting, and thus safer to use when implementing a propagator.

Value versus range iterators. Set-valued operations could be based on *value iterators*, iterating individual values of a set, instead of range iterators. This is not efficient: a value sequence is considerably longer than a range sequence (in particular for the common case of a singleton range sequence). For implementing propagators, however, it can sometimes be simpler to iterate values. A value iterator v has the operations $v.\texttt{done}()$, $v.\texttt{next}()$, and $v.\texttt{val}()$ to access the current value. Two straightforward adaptors mediate between the value iterators that the propagator uses and the range iterators of the domain operations. A range-to-value iterator takes a range iterator as input and returns a value iterator iterating the values of the range sequence. The inverse is a value-to-range iterator: it takes as input a value iterator and returns the corresponding range iterator.

10.4 Integer Views with Set-Valued Operations

We have seen how range iterators can serve as an interface for set-valued variable operations. This section shows that the abstraction provided by range iterators is powerful enough to support set-valued operations for views.

Set-valued operations for constant integer views are straightforward. For a constant view v on constant k, the operation $v.\texttt{getdom}()$ returns an iterator for the singleton range sequence $\langle [k \mathinner{..} k] \rangle$. The operation $v.\texttt{setdom}(r)$ just checks whether the range sequence of r is empty (in order to detect failure).

Set-valued operations for an offset view are provided by an offset iterator. The operations of an offset iterator o for a range iterator r and an integer c (created by $\text{ioffset}(r, c)$) are as follows:

$$o.\texttt{min}() := r.\texttt{min}() + c \qquad o.\texttt{max}() := r.\texttt{max}() + c$$
$$o.\texttt{done}() := r.\texttt{done}() \qquad o.\texttt{next}() := r.\texttt{next}()$$

An offset view v on a variable x with offset c uses an offset iterator:

$$v.\texttt{getdom}() := \text{ioffset}(x.\texttt{getdom}(), c)$$
$$v.\texttt{setdom}(r) := x.\texttt{setdom}(\text{ioffset}(r, -c))$$

For minus views we just give the range sequence, iteration is obvious. For a given range sequence $\langle [m_i \mathinner{..} n_i] \rangle_{i=1}^{k}$, the negative sequence is obtained by reversal and sign change as $\langle [-n_{k-i+1} \mathinner{..} -m_{k-i+1}] \rangle_{i=1}^{k}$. The same iterator for this sequence can be used both for `setdom` and `getdom` operations. Note that implementing the iterator is quite complicated as it changes direction of the range sequence.

A scale iterator provides the necessary transformation for scale views. Assume a scale view on a variable x with a coefficient $a > 0$, and let $\langle [m_i \mathinner{..} n_i] \rangle_{i=1}^{k}$ be a range sequence for $d(x)$. If $a = 1$, the scale iterator does not change the range sequence. Otherwise, the corresponding scaled range sequence is $\langle \{a \times m_1\}, \{a \times (m_1 + 1)\}, \ldots, \{a \times n_1\}, \ldots, \{a \times m_k\}, \{a \times (m_k + 1)\}, \ldots, \{a \times n_k\} \rangle$. For the other direction, assume we want to update the domain using a set S through a scale view. Assume that $\langle [m_i \mathinner{..} n_i] \rangle_{i=1}^{k}$ is a range sequence for S. Then for $1 \leq i \leq k$ the ranges $[\lceil m_i/a \rceil \mathinner{..} \lfloor n_i/a \rfloor]$ correspond to the required variable domain for x, however they do not necessarily form a range sequence as the ranges might be empty, overlapping, or adjacent. Iterating the range sequence is however simple by skipping empty ranges and merging overlapping or adjacent ranges.

10.5 Set Variables and Views

In this section, set variables and views are equipped with domain operations based on range iterators.

Section 4.5 introduced a domain system for approximating set variable domains by a set interval defined by the greatest lower and least upper bound. The domain of a set variable x under this approximation is an interval $d(x) = [\mathrm{glb}(d(x)), \mathrm{lub}(d(x))]$.

The fundamental operations for variables and views based on this approximation are similar to set-valued operations on integer variables, and take iterators as arguments or return them. x.`glb`() returns a range iterator for ranges(glb($d(x)$)), x.`lub`() returns a range iterator for ranges(lub($d(x)$)), x.`adjglb`(r) updates the domain of x to $[\mathrm{glb}(d(x)) \cup \mathrm{set}(r), \mathrm{lub}(d(x))]$, and x.`adjlub`(r) updates the domain of x to $[\mathrm{glb}(d(x)), \mathrm{lub}(d(x)) \cap \mathrm{set}(r)]$.

Range iterators are not only perfectly suited for operations on set variables, but also provide exactly the operations that set propagators need: union, intersection, difference, and complement. Most propagators can thus be implemented directly using iterators and do not require any temporary data structures for storing set-valued intermediate results.

Constant set views. A set view v of a constant set S, used for specialization (see Section 8.3), can be implemented using a constant iterator r_S (with $\mathrm{set}(r_S) = S$):

$v.\texttt{glb}() := r_S$ $\quad$ $v.\texttt{adjglb}(r) :=$ **if** $\text{set}(r) \not\subseteq \text{set}(r_S)$ **then fail**()
$v.\texttt{lub}() := r_S$ $\quad$ $v.\texttt{adjlub}(r) :=$ **if** $\text{set}(r_S) \not\subseteq \text{set}(r)$ **then fail**()

Complement views. A complement view v of a set variable x uses a complement iterator:

$v.\texttt{glb}() := \text{icompl}(x.\texttt{lub}())$ $\quad$ $v.\texttt{adjglb}(r) := x.\texttt{adjlub}(\text{icompl}(r))$
$v.\texttt{lub}() := \text{icompl}(x.\texttt{glb}())$ $\quad$ $v.\texttt{adjlub}(r) := x.\texttt{adjglb}(\text{icompl}(r))$

Type conversion views. A singleton view v that provides a type conversion between an integer variable x and a set propagator (see Section 8.4) reuses the range iterators and set-valued operations of the integer variable, where $r_\emptyset$ stands for a range iterator r with $\text{set}(r) = \emptyset$:

$v.\texttt{glb}()$ $:=$ **if** $|x.\texttt{getdom}()| = 1$ **then** $x.\texttt{getdom}()$ **else** $r_\emptyset$
$v.\texttt{lub}()$ $:= x.\texttt{getdom}()$
$v.\texttt{adjglb}(r) :=$ **if** $|\text{set}(r)| = 1$ **then** $x.\texttt{adjdom}(r)$ **else if** $\text{set}(r) \neq \emptyset$ **then** fail()
$v.\texttt{adjlub}(r) := x.\texttt{adjdom}(r)$

10.6 Iterators as Adaptors

Section 10.3 already provided examples where iterators were constructed *on the fly*, for instance as the intersection of two other iterators. Here, we extend this idea and construct iterators over internal data structures of the propagation algorithms.

Propagation algorithms for global constraints typically use some advanced data structures, such as a variable-value graph for domain-complete *all-different* (Régin, 1994). After propagation, the new domains must be transferred from the data structure to the variables. Similar to iterators that compute with iterators, an iterator can now be used to compute with the propagator's data structure, and then be passed to the variable's set-valued domain update operation. The iterator serves as an *adaptor* between the data structure and the variable domain.

An adaptor for *all-different*

Régin's (1994) domain-complete propagator for the *all-different* constraint employs a *variable-value* graph, a bipartite graph with one set of nodes for the variables, and one set of nodes for the values in the variables' domains. The graph contains an edge from a variable node to a value node if the value is in the domain of the variable. Propagation removes edges that correspond to inconsistent variable-value pairs. The adaptor iterator for a variable x iterates the values whose nodes are adjacent to the variable node for x. A value-to-range iterator is used to update the variable domain of x.

An adaptor for the *regular* constraint

The *regular* constraint states that the sequence of values taken by a sequence of integer variables $x_0, \ldots, x_{k-1}$ is an element of a given regular language L. The propagator proposed by Pesant (2004) works on an unfolding of a finite automaton accepting L, called the *layered graph*. Each node in layer i in the graph represents a possible value for the variable x_i. If the node does not lie on a path from a node in layer 0 to a node in layer $k-1$, it can be pruned from the domain of x_i. The propagation algorithm records which nodes are still on such a path. The adaptor iterator then iterates the values for a certain layer i that are still on such paths, and transfers this information through a value-to-range iterator to the domain of x_i.

An adaptor for the *element* constraint

The element constraint states that $a_x = y$, given a sequence $a_0, \ldots, a_{k-1}$ of integers and two integer variables x and y. An efficient way of implementing a propagation algorithm for this constraint is to maintain the a_i in two doubly-linked lists: one of the lists is sorted by the index, the other is sorted by the a_i. During propagation, first all elements are removed whose indexes are no longer in the domain of x, then all elements are removed that are no longer in the domain of y. Finally, two adaptor iterators transfer the information in the data structure back to the variable domains: one iterator for the indexes, used to prune x, and one for the elements, used to prune y.

An adaptor for channeling

The constraint $\bigwedge_{i=1}^{n} (x = i) \leftrightarrow b_i$ for n Boolean variables b_i and an integer variable x is useful for channeling between an integer and a Boolean part of a model. Although not strictly an adaptor, the propagator for this constraint can use a value iterator that lists all the i such that b_i can still be 1. Through a value-to-range iterator, this iterator is then used to prune the domain of x.

10.7 Performance Analysis

In this section, we first show that using range iterators improves the efficiency of propagators, compared to the use of explicit set data structures for temporary results. We then explore the limitations of views based on iterators.

Benchmark	*time %*	*Benchmark*	*time %*
Alpha (smart)	99.78	Crew Scheduling	380.33
Alpha (naive)	99.14	Hamming Codes (20-3-32)	388.36
BIBD	98.39	Social Golfers (8-4-9)	766.41
Eq-20	102.45	Social Golfers (5-3-7)	552.78
Golomb Rulers (10)	98.82	Steiner Triples (9)	332.23
Graph Coloring	99.31	Sudoku (Set, 1)	538.44
Knights (18)	100.13	Sudoku (Set, 4)	494.67
Magic Sequence (Smart, 500)	99.60	Sudoku (Set, 5)	456.70
Magic Sequence (Naive, 500)	99.88	Queen Armies	293.46
Magic Sequence (GCC, 500)	99.26		
Partition (32)	98.40		
Perfect Square	99.94		
Photo Alignment	99.17		
Queens (Naive, 10)	99.38		
Queens (Smart, 10)	100.44		
Queens (Naive, 100)	99.21		
Queens (Smart, 100)	99.36		

Table 10.1: Relative performance of cached iterators

Explicit set data structures

One important claim is that iterators are advantageous because they avoid temporary data structures. Table 10.1 presents experiments where temporary data structures have been emulated by wrapping all iterators in a cache iterator as described in Section 10.3. The experiments again use the setup as described in Section 6.9 and Appendix A. The results show that computing with temporary data structures has hardly any impact on integer variables. For set constraints, however, the overhead is considerable (up to seven times the run-time). The obvious explanation is that propagators for set constraints heavily rely on set-valued operations, much more so than integer propagators. The memory consumption does not increase, because iterators are not stored, and only few iterators are active at a time.

Limits of compiler optimization

The efficiency of derived propagators relies on compiler optimizations like inlining and constant folding. For instance, a double-negation on integers or Booleans can be detected and removed easily.

For set-valued operations, the compiler will sometimes fail to produce optimal code. For example, a propagator for ternary intersection, $x = y \cap z$, will include an inference x.`adjglb`(y.`glb`() $\cap$ z.`glb`()). To derive a propagator for $x = y \cup z$, we instantiate the intersection propagator with complement views for x, y, and z, yielding

Benchmark	*time %*
Social Golfers (8-4-9)	146.80
Social Golfers (5-3-7)	126.90
Steiner Triples (9)	116.17
Hamming Codes (20-3-32)	121.22
Sudoku (Set, 1)	140.32
Sudoku (Set, 4)	135.01
Sudoku (Set, 5)	130.19

Table 10.2: Relative performance of views compared to dedicated set propagators

the following inference:

$$\overline{x}.\texttt{adjglb}(\text{glb}(\overline{y}) \cap \text{glb}(\overline{z}))$$

which amounts to computing

$$x.\texttt{adjlub}(\overline{\overline{\text{lub}(y)} \cap \overline{\text{lub}(z)}})$$

It would be more efficient to implement the equivalent $x.\texttt{adjlub}(\text{lub}(y) \cup \text{lub}(z))$ because this requires three set operations less. Unfortunately, no compiler will find this equivalence automatically, as it requires knowledge about the semantics of the set operations. Table 10.2 compares a dedicated propagator for the constraint $x \cap y = z$ with a version using complement views and a propagator for $x \cup y = z$. The overhead of 16% to 47% does not render views completely useless for set variables, but it is nevertheless significant.

The next chapter solves this problem by generating set propagators directly from specifications of the constraints.

11 Deriving Propagators for Boolean Set Constraints

The previous chapters were concerned with deriving propagators (and their implementations) from existing propagators using views. In this chapter, we shift the focus, as the goal is to derive propagators for set constraints and their implementations from *specifications of the constraints*.

Most of the literature on set constraints (discussed in detail in Section 11.6) presents propagation algorithms as *ad hoc* rules for a few basic constraints such as intersection $[\![x = y \cap z]\!]$ or union $[\![x = y \cup z]\!]$. We follow a *systematic approach* instead, *generating* propagator implementations from declarative constraint specifications.

We design a specification language for a particular class of set constraints called *Boolean* set constraints, and then translate a specification into a propagation algorithm. More specifically, a constraint specification is translated into a specification of a propagator that is set-interval-complete. The propagator specification directly yields an efficient algorithm. This chapter thus provides deeper insights into the structure of set constraints, and yields principled implementation strategies.

Structure of the chapter. We define a specification language for Boolean set constraints (11.1) and show how to derive a set-interval-complete propagation algorithm from a specification of a Boolean set constraint (11.2). We extend our specification language with negated Boolean set constraints, and again show how to propagate them (11.3). For certain n-ary set constraints, we can improve the run-time complexity by a factor of n using common subexpression elimination (11.4). Finally, we present and evaluate implementation techniques for Boolean set propagators (11.5).

11.1 Boolean Set Constraints

This section formalizes a specification language for the class of set constraints that the rest of this chapter is based on.

It is well-known that the algebra of sets with union, intersection, and complement operations is a Boolean algebra—union corresponds to disjunction, intersection to

conjunction, and complement to negation. The set constraints this chapter deals with are thus the **Boolean set constraints**, specified by the following grammar:

$C ::= S = S \mid S \subseteq S \mid C \wedge C$

$S ::= x \mid \emptyset \mid \mathscr{U} \mid S \cap S \mid S \cup S \mid \overline{S}$

An expression C corresponds to a constraint, the set of assignments that satisfy C, with the usual interpretation of the relation and function symbols. Let us define this formally. Set variables take their values from the set $V = \mathscr{P}(\mathscr{U})$ for a given finite **universe** $\mathscr{U}$. Assignments therefore map variables to subsets of $\mathscr{U}$. We can lift assignments to expressions S as follows.

Definition 11.1 The value of an expression S under an assignment a is defined by lifting assignments to expressions in the straightforward way:

$$\begin{aligned} a(S_1 \cap S_2) &:= a(S_1) \cap a(S_2) & \qquad a(\emptyset) &:= \emptyset \\ a(S_1 \cup S_2) &:= a(S_1) \cup a(S_2) & \qquad a(\mathscr{U}) &:= \mathscr{U} \\ a(\overline{S}) &:= \overline{a(S)} \end{aligned}$$

We define that an assignment a **satisfies an expression C**, written $a \vDash C$, as follows.

$$\begin{aligned} a \vDash C_1 \wedge C_2 &:\Leftrightarrow a \vDash C_1 \text{ and } a \vDash C_2 \\ a \vDash S_1 = S_2 &:\Leftrightarrow a(S_1) = a(S_2) \\ a \vDash S_1 \subseteq S_2 &:\Leftrightarrow a(S_1) \subseteq a(S_2) \end{aligned}$$

We use the notation $[\![C]\!]$ to denote the constraint $\{a \in \mathsf{Asn} \mid a \vDash C\}$. *

We say that two expressions C_1 and C_2 are **equivalent**, written $C_1 \equiv C_2$, if and only if they represent the same constraint, $[\![C_1]\!] = [\![C_2]\!]$.

The goal of this chapter is to generate $\mathscr{D}^{[\mathscr{P}(\mathscr{U})]}$-complete propagation algorithms that induce a constraint specified by an expression C. Recall (from Section 4.5) that the $\mathscr{D}^{[\mathscr{P}(\mathscr{U})]}$ approximation represents the domain of each set variable x as an interval $[l, u]$, where l and u are subsets of the universe $\mathscr{U}$. A $\mathscr{D}^{[\mathscr{P}(\mathscr{U})]}$-complete propagator has to enlarge the lower bound l and shrink the upper bound u for each of its variables as far as its constraint permits for a given domain.

Our approach to propagating a set constraint $[\![C]\!]$ is to transform C into an equivalent form $S_1 \subseteq x \wedge x \subseteq S_2$ for every variable x that appears in C, such that x does not appear in S_1 or S_2. The two set expressions S_1 and S_2 then describe a lower and an upper bound for the variable x. This section develops the theoretical background we need for the transformation. The next section shows how S_1 and S_2 can be used to specify a propagator, that the specified propagator is $\mathscr{D}^{[\mathscr{P}(\mathscr{U})]}$-complete for the constraint $[\![C]\!]$, and how it corresponds directly to a propagation algorithm that can be implemented.

While we can express many set constraints using conjunctions of equations and subset relations, some important constraints are missing, such as disequality of two sets $S_1 \neq S_2$. This is a fundamental restriction, in that for any non-trivial constraint $[\![C]\!]$, its complement is not expressible as $[\![C']\!]$ for any C'. We will prove this in Section 11.3, and show how these negated Boolean constraints can be propagated.

The Boolean algebra of sets

As we are dealing with a Boolean algebra, we can use all the equivalence transformations that are well-known from manipulating Boolean functions: if and only if two expressions S_1 and S_2 are equivalent in the two-valued Boolean algebra, they are equivalent in any Boolean algebra, and accordingly $S_1 = \mathscr{U} \equiv S_2 = \mathscr{U}$.

As a first step towards an expression $S_1 \subseteq x \wedge x \subseteq S_2$, we can transform any conjunction of set relations C into a single equation $S = \mathscr{U}$. This is a fundamental result from Boolean algebra (see for instance Whitesitt, 1995). We say that an expression $S = \mathscr{U}$ is in **equation normal form (ENF)**. The transformation to ENF uses the following identities:

$$\begin{aligned} S_1 = S_2 &\equiv S_1 \subseteq S_2 \wedge S_2 \subseteq S_1 \\ S_1 \subseteq S_2 &\equiv \overline{S_1} \cup S_2 = \mathscr{U} \\ S_1 = \mathscr{U} \wedge S_2 = \mathscr{U} &\equiv S_1 \cap S_2 = \mathscr{U} \end{aligned}$$

If $S = \mathscr{U}$ is the ENF of C, then $[\![S = \mathscr{U}]\!] = [\![C]\!]$. From now on, we will only consider constraints specified in ENF.

The following table lists some relations over set variables and the corresponding equation normal forms.

relation	*ENF*	
$x \subseteq y$	$\overline{x} \cup y = \mathscr{U}$	*subset*
$x = y$	$(\overline{x} \cup y) \cap (x \cup \overline{y}) = \mathscr{U}$	*equality*
$x = y \cup z$	$(\overline{x} \cup y \cup z) \cap (x \cup \overline{y \cup z}) = \mathscr{U}$	*union*
$x = y \cap z$	$(\overline{x} \cup y \cap z) \cap (x \cup \overline{y \cap z}) = \mathscr{U}$	*intersection*
$x \parallel y$	$\overline{x \cap y} = \mathscr{U}$	*disjointness*
$x = y \uplus z$	$(\overline{x} \cup y \cup z) \cap (x \cup \overline{y \cup z}) \cap \overline{y \cap z} = \mathscr{U}$	*partition*

Isolating variables

The machinery of Boolean algebras also yields a way to *isolate* any variable x in an equation normal form $S = \mathscr{U}$. Isolating x means to transform $S = \mathscr{U}$ into an equivalent expression $S_1 \subseteq x \wedge x \subseteq S_2$ such that x does not appear in S_1 or S_2.

The identity we make use of is an instance of *Shannon's expansion*. Let $S[a/x]$ denote the expression S where any occurrence of x has been substituted by a. For any expression S, the following holds:

$$S = (x \cup S[\emptyset/x]) \cap (\overline{x} \cup S[\mathscr{U}/x])$$

Consider an ENF $S = \mathscr{U}$. Then $S = \mathscr{U}$ is equivalent to $(S[\emptyset/x] \cup x) \cap (\overline{x} \cup S[\mathscr{U}/x]) = \mathscr{U}$, which, by translating from ENF backwards, is the same as $\overline{S[\emptyset/x]} \subseteq x \wedge x \subseteq S[\mathscr{U}/x]$.

As the two expressions $\overline{S[\emptyset/x]}$ and $S[\mathscr{U}/x]$ denote a set interval that encloses x, we call $\overline{S[\emptyset/x]} \subseteq x \wedge x \subseteq S[\mathscr{U}/x]$ an ***x*-interval normal form (INF$_x$)** of $S = \mathscr{U}$. For notational convenience, we will usually write it as

$$\overline{S[\emptyset/x]} \subseteq x \subseteq S[\mathscr{U}/x]$$

Example 11.2 (Intersection) Let us derive the interval normal forms for x, y, and z of the expression $x = y \cap z$. First, we transform the expression into equation normal form, and then isolate each of the three variables in turn:

$$\begin{array}{lllll}
x = y \cap z \equiv & (\overline{x} \cup (y \cap z)) \cap (x \cup \overline{y \cap z}) = \mathscr{U} & & & \text{ENF} \\
\\
\equiv & y \cap z & \subseteq x \subseteq & y \cap z & \text{INF}_x \\
\equiv & x & \subseteq y \subseteq & (\overline{x} \cup z) \cap (x \cup \overline{z}) & \text{INF}_y \\
\equiv & x & \subseteq z \subseteq & (\overline{x} \cup y) \cap (x \cup \overline{y}) & \text{INF}_z
\end{array}$$

*

Example 11.3 (Partition) The partition $x = y \uplus z$ can be represented as the two equations $x = y \cup z \wedge y \cap z = \emptyset$. The interval normal forms for partition are derived as follows:

$$\begin{array}{lllll}
x = y \cup z \wedge y \cap z = \emptyset \equiv & (\overline{x} \cup y \cup z) \cap (x \cup \overline{y \cup z}) \cap \overline{y \cap z} = \mathscr{U} & & & \text{ENF} \\
\\
\equiv & (y \cup z) \cup (y \cap z) & \subseteq x \subseteq & (y \cup z) \cap \overline{y \cap z} & \text{INF}_x \\
\equiv & (x \cap \overline{z}) \cup (\overline{x} \cap z) & \subseteq y \subseteq & x \cap \overline{z} & \text{INF}_y \\
\equiv & (x \cap \overline{y}) \cup (\overline{x} \cap y) & \subseteq z \subseteq & x \cap \overline{y} & \text{INF}_z
\end{array}$$

Note how every single interval normal form forces y and z to be disjoint. For instance, INF$_x$ requires by transitivity that $y \cup z$ (on the left of x) is a subset of $(y \cup z) \cap \overline{y \cap z}$.

*

11.2 Propagators for Boolean Set Constraints

This section shows how to derive a $\mathscr{D}^{[\mathscr{P}(\mathscr{U})]}$-complete propagator from a Boolean set constraint $[\![C]\!]$, and how the derived propagator yields an efficient propagation algorithm.

We have seen in the previous section how we can transform any expression C into an equivalent equation normal form $S = \mathscr{U}$, and from there into an equivalent x-interval normal form, for any variable x that appears in C. We will now use these normal forms to define a propagator for the constraint $[\![C]\!]$. In the rest of this section, we assume that $S = \mathscr{U}$ is the ENF of C.

To simplify presentation, we will from now on assume that any domain d is a $\mathscr{D}^{[\mathscr{P}(\mathscr{U})]}$-domain. This assumption is realistic for practical systems, which actually represent domains of set variables using lower and upper bounds. Section 4.3 defined the $\mathscr{D}^{[\mathscr{P}(\mathscr{U})]}$-canonical propagator for $[\![S = \mathscr{U}]\!]$ as $p_S^{\max}(d) = \llbracket [\![S = \mathscr{U}]\!] \cap \llbracket d \rrbracket_{\mathscr{D}^{[\mathscr{P}(\mathscr{U})]}} \rrbracket_{\mathscr{D}^{[\mathscr{P}(\mathscr{U})]}} \cap d$. With the assumption that all domains are $\mathscr{D}^{[\mathscr{P}(\mathscr{U})]}$-domains, this can be simplified to $p_S^{\max}(d) = \llbracket [\![S = \mathscr{U}]\!] \cap d \rrbracket_{\mathscr{D}^{[\mathscr{P}(\mathscr{U})]}}$. The definition of the $\mathscr{D}^{[\mathscr{P}(\mathscr{U})]}$-relaxation yields the equivalent

$$p_S^{\max}(d)(x) = \Big[\bigcap_{\substack{a \in d \\ a \vDash S = \mathscr{U}}} a(x), \bigcup_{\substack{a \in d \\ a \vDash S = \mathscr{U}}} a(x) \Big]$$

All results in this section generalize to the case of arbitrary domains by adding the inner $\mathscr{D}^{[\mathscr{P}(\mathscr{U})]}$-relaxation and the outer intersection with d.

The above definition determines the propagation strength we want to achieve, but it does not give us an efficient algorithm for $p_S^{\max}$, as computing the intersections and unions over all assignments is not tractable. We will now give an alternative, equivalent definition in terms of interval normal forms, and see that this definition leads to an efficient algorithm.

The propagator we develop prunes the lower and upper bound of each of its variables x using an x-interval normal form of S. More precisely, given a domain d, the propagator determines the greatest lower and least upper bound of x with respect to S that all the assignments $a \in d$ license, defined as follows.

$$\mathrm{glb}(S, x, d) := \bigcap_{a \in d} a(\overline{S[\emptyset/x]}) \qquad \mathrm{lub}(S, x, d) := \bigcup_{a \in d} a(S[\mathscr{U}/x])$$

We will now prove that the following defines a $\mathscr{D}^{[\mathscr{P}(\mathscr{U})]}$-complete propagator for $[\![S = \mathscr{U}]\!]$. For all variables x that appear in S and any domain d, we define p_S as follows:

$$p_S(d)(x) := [\mathrm{glb}(d(x)) \cup \mathrm{glb}(S, x, d), \mathrm{lub}(d(x)) \cap \mathrm{lub}(S, x, d)]$$

For the proof, we need several lemmas. Some of these lemmas exist in corresponding dual versions, which we omit to improve readability, but which will be used in the proof. The first lemma states that the evaluation of an expression S under an assignment a is *continuous*, in the sense that modifying a at any variable by a single value v changes the value of $a(S)$ at most by v.

Lemma 11.4 Let a be an assignment, $x \in X$ a variable, $v \in \mathscr{U}$ a value, and S an expression. Then changing a at variable x by adding or removing v can only add or remove v from $a(S)$:

$$\begin{aligned} & a(S) \setminus \{v\} \\ =& a[a(x) \setminus \{v\} \,/\, x](S) \setminus \{v\} \\ =& a[a(x) \cup \{v\} \,/\, x](S) \setminus \{v\} \end{aligned}$$ *

Proof. For any set W, adding (removing) an element to (from) W will only remove (add) that element from (to) $\overline{W}$. Similarly, for sets W and W', adding (removing) an element to (from) W and/or W' will only add (remove) that element to (from) $W \cup W'$ or $W \cap W'$. Thus, by induction over the structure of the expression S, the statement holds. ∎

The next lemma is a fact about set interval domains, and will help us to construct assignments from given assignments by *merging* them.

Lemma 11.5 Let S be an expression, let d be a domain, and let $a, a' \in d$ be assignments licensed by d. If there is a value $v \in a'(S) \setminus a(S)$, then there exists an assignment $a'' \in d$ such that $a''(S) = a(S) \cup \{v\}$. *

Proof. Without loss of generality, assume that S is in disjunctive normal form, $S = \bigcup_{i=1}^{n} \bigcap_{j=1}^{m_j} L_{i,j}$. Then v must have been contributed to $a'(S)$ by at least one disjunct $\bigcap_{j=1}^{m_j} L_{i,j}$, and hence by all the literals $L_{i,j}$ in that disjunct: $v \in a'(L_{i,j})$. We construct a'' such that

$$a''(x) = \begin{cases} a(x) \cup \{v\} & \text{if } x = L_{i,j} \text{ for some } j \\ a(x) \setminus \{v\} & \text{if } \overline{x} = L_{i,j} \text{ for some } j \\ a(x) & \text{otherwise} \end{cases}$$

Then for all j, $v \in a''(L_{i,j})$, and hence $v \in a''(S)$. But with Lemma 11.4, a'' behaves like a for any value except v, as we have only changed a with respect to v. Furthermore, $a'' \in d$, because we only added v to variables y where $v \in a'(y)$, and we only removed it from variables y where $v \notin a'(y)$. In summary, $a'' \in d$ and $a''(S) = a(S) \cup \{v\}$, concluding the proof. ∎

Using this lemma, we can show that for an expression S and a domain d, there is an assignment that maps x to its upper bound with respect to S.

Lemma 11.6 Let d be a domain, S an expression, and x a variable. If there is an assignment $a \in d$ such that $a \vDash x \subseteq S[\mathscr{U}/x]$, then there is an assignment $a' \in d$ such that $a' \vDash x \subseteq S[\mathscr{U}/x]$ and

$$a'(x) = \bigcup_{\substack{a \in d \\ a \vDash x \subseteq S[\mathscr{U}/x]}} a(x) \qquad *$$

Proof. Lemma 11.5 allows us to construct a' by merging all assignments a that satisfy $x \subseteq S[\mathscr{U}/x]$. This is possible because x does not occur in $S[\mathscr{U}/x]$. ∎

The next lemma takes a major step in the desired direction. It states that instead of the intersection and union over all assignments that satisfy $S = \mathscr{U}$, it suffices to take the intersection over the assignments that satisfy the lower bound $\overline{S[\emptyset/x]} \subseteq x$, and the union over the assignments that satisfy the upper bound $x \subseteq S[\mathscr{U}/x]$.

Lemma 11.7 Let $S = \mathscr{U}$ be given. Then the following equation holds:

$$\Big[\bigcap_{\substack{a \in d \\ a \vDash S = \mathscr{U}}} a(x), \bigcup_{\substack{a \in d \\ a \vDash S = \mathscr{U}}} a(x)\Big] = \Big[\bigcap_{\substack{a \in d \\ a \vDash \overline{S[\emptyset/x]} \subseteq x}} a(x), \quad \bigcup_{\substack{a \in d \\ a \vDash x \subseteq S[\mathscr{U}/x]}} a(x)\Big] \qquad *$$

Proof. We show the two subset relations.

- $\subseteq$ This direction is easy to show. The set of assignments that satisfy $S = \mathscr{U}$ is a subset of the set of assignments that satisfy either bound, $\overline{S[\emptyset/x]} \subseteq x$ or $x \subseteq S[\mathscr{U}/x]$. The left interval is hence smaller than the right interval, because its lower bound is an intersection over a subset of assignments and thus bigger, and its upper bound is a union over a subset of assignments and thus smaller.
- $\supseteq$ If the interval is empty, the relation holds. If there is no a that satisfies $x \subseteq S[\mathscr{U}/x]$, the union is empty. But then also the interval must be empty, as otherwise we could find an a that satisfies $\overline{S[\emptyset/x]} \subseteq x$ with $a(x) = \emptyset$, which would have to satisfy $x \subseteq S[\mathscr{U}/x]$, too. The dual reasoning holds for the lower bound. Hence assume the interval is not empty. Then, with Lemma 11.6 and its dual, we can find assignments $a_1 \in d$ and $a_2 \in d$ such that

  $$a_1 \vDash \overline{S[\emptyset/x]} \subseteq x \quad \text{and} \quad a_1(x) = \bigcap_{\substack{a \in d \\ a \vDash \overline{S[\emptyset/x]} \subseteq x}} a(x)$$

  $$a_2 \vDash x \subseteq S[\mathscr{U}/x] \quad \text{and} \quad a_2(x) = \bigcup_{\substack{a \in d \\ a \vDash x \subseteq S[\mathscr{U}/x]}} a(x)$$

  Then $a_1(x) \subseteq a_2(x) \subseteq a_2(S[\mathscr{U}/x])$. Using Lemma 11.5, we can construct an assignment a_3 by merging all values values $v \in a_2(x) \setminus a_1(S[\mathscr{U}/x])$ into the

assignment a_1, and letting $a_3(x) = a_2(x)$. That way, $a_3 \vDash x \subseteq S[\mathscr{U}/x]$. But at the same time, $a_3 \vDash \overline{S[\emptyset/x]} \subseteq x$, as we only added or removed values that are elements of $a_3(x)$. Together, $a_3 \vDash S = \mathscr{U}$, and thus

$$\bigcup_{\substack{a \in d \\ a \vDash x \subseteq S[\mathscr{U}/x]}} a(x) \subseteq \bigcup_{\substack{a \in d \\ a \vDash S = \mathscr{U}}} a(x)$$

A dual reasoning yields the lower bound. ∎

As a last step before proving our main theorem, we prove that we can further decompose the bounds. The upper bound of a variable x with respect to an expression S can be decomposed into the current upper bound of x and the bound of $S[\mathscr{U}/x]$.

Lemma 11.8 Let S be an expression, d a domain, and x a variable. If there exists no assignment $a \in d$ such that $a \vDash x \subseteq S[\mathscr{U}/x]$, then

$$\mathrm{glb}(d(x)) \supset \bigcup_{a \in d} a(x) \cap \bigcup_{a \in d} a(S[\mathscr{U}/x])$$

If otherwise there exists an $a \in d$ such that $a \vDash x \subseteq S[\mathscr{U}/x]$, then

$$\bigcup_{\substack{a \in d \\ a \vDash x \subseteq S[\mathscr{U}/x]}} a(x) = \bigcup_{a \in d} a(x) \cap \bigcup_{a \in d} a(S[\mathscr{U}/x]) \qquad *$$

Proof. We prove the first half by contradiction. Assume that there exists no assignment $a \in d$ that satisfies $x \subseteq S[\mathscr{U}/x]$, but

$$\mathrm{glb}(d(x)) \subseteq \bigcup_{a \in d} a(x) \cap \bigcup_{a \in d} a(S[\mathscr{U}/x])$$

Then, with Lemma 11.5, we start from an assignment that assigns x its greatest lower bound, and construct an assignment $a \in d$ such that $a(x) = \bigcup_{a \in d} a(x) \cap \bigcup_{a \in d} a(S[\mathscr{U}/x])$, and all other variables are chosen such that $a(x) \subseteq a(S[\mathscr{U}/x])$. Thus there is an assignment satisfying $x \subseteq S[\mathscr{U}/x]$, contradicting our assumption.

For the second half, let a' be an assignment that satisfies $x \subseteq S[\mathscr{U}/x]$. We prove the two subset relations.

- $\subseteq$ For any value $v \in a'(x)$, we know $v \in \bigcup_{a \in d} a(x)$ (as $a' \in d$), and at the same time $v \in \bigcup_{a \in d} a(S[\mathscr{U}/x])$ (as $a'(x) \subseteq a'(S[\mathscr{U}/x])$).
- $\supseteq$ Let $W = \bigcup_{a \in d} a(x) \cap \bigcup_{a \in d} a(S[\mathscr{U}/x])$. Then for each $v \in W$, we find an assignment $a \in d$ such that $v \in a(x)$, and at the same time $v \in \bigcup_{a \in d} a(S[\mathscr{U}/x])$. This is possible because x does not appear in $S[\mathscr{U}/x]$. Then we can construct an assignment a'' by merging all $v \in W$ into a' with Lemma 11.5. Hence

$$W \subseteq \bigcup_{\substack{a \in d \\ a \vDash x \subseteq S[\mathscr{U}/x]}} a(x)$$

∎

We can now attack the central theorem of this chapter.

Theorem 11.9 The function p_S is equal to the function $p_S^{\max}$, and hence defines a $\mathscr{D}^{[\mathscr{P}(\mathscr{U})]}$-complete propagator for the constraint $[\![S = \mathscr{U}]\!]$. *

Proof. Combining the above lemmas and their respective dual versions, we get

$$\begin{aligned}
p_S^{\max}(d)(x) &= \Big[\bigcap_{\substack{a\in d \\ a\vDash S=\mathscr{U}}} a(x), \bigcup_{\substack{a\in d \\ a\vDash S=\mathscr{U}}} a(x) \Big] && \text{(Def.)} \\
&= \Big[\bigcap_{\substack{a\in d \\ a\vDash \overline{S[\emptyset/x]}\subseteq x}} a(x), \bigcup_{\substack{a\in d \\ a\vDash x\subseteq S[\mathscr{U}/x]}} a(x) \Big] && \text{(L. 11.7)} \\
&= \Big[\bigcap_{a\in d} a(x) \cup \bigcap_{a\in d} a(\overline{S[\emptyset/x]}), \bigcup_{a\in d} a(x) \cap \bigcup_{a\in d} a(S[\mathscr{U}/x]) \Big] && \text{(L. 11.8)} \\
&= \big[\mathrm{glb}(d(x)) \cup \mathrm{glb}(S,x,d), \mathrm{lub}(d(x)) \cap \mathrm{lub}(S,x,d) \big] && \text{(Def.)} \\
&= p_S(d)(x) && \text{(Def.)}
\end{aligned}$$

Except for the third line, all steps should be clear. When applying Lemma 11.8, we have to distinguish two cases. If one of the bounds in the second line cannot be satisfied by any $a \in d$, then the third line will detect failure because of the first half of Lemma 11.8. Otherwise, the resulting interval is equal. ∎

Evaluating set expressions

The definition of p_S still does not yield an algorithm for computing $p_S(d)$ in a straightforward way. The problem is that the naive evaluation of $\mathrm{glb}(S,x,d)$ and $\mathrm{lub}(S,x,d)$ amounts to computing the union or intersection over all assignments $a \in d$. The number of assignments is not only exponential in the number of variables, but also in the size of their upper bounds, so this is clearly intractable in practice.

An efficient algorithm has to determine the values of $\mathrm{glb}(S,x,d)$ and $\mathrm{lub}(S,x,d)$ by only computing with the lower and upper bounds of *individual variables* in the domain d. We will now see how to achieve that for $\mathrm{lub}(S,x,d)$; the lower bound case is dual.

The upper bound, $\mathrm{lub}(S,x,d)$, is defined as $\bigcup_{a\in d} a(S[\mathscr{U}/x])$. Let us first look at a special case, when $S[\mathscr{U}/x]$ is just a single variable y or its complement $\overline{y}$. Then we can compute the upper bound efficiently from the domain d:

$$\bigcup_{a\in d} a(y) = \mathrm{lub}(d(y)) \qquad \text{and} \qquad \bigcup_{a\in d} a(\overline{y}) = \overline{\mathrm{glb}(d(y))}$$

We can reduce the general case to this special case by looking at a disjunctive normal form of $S[\mathscr{U}/x]$. Assume that S' is a disjunctive normal form that is equivalent to $S[\mathscr{U}/x]$, which means that $S' = \bigcup_{i=1}^{n} \bigcap_{j=1}^{m_i} L_{i,j}$. Now the following equations hold:

$$\begin{aligned}\mathrm{lub}(S,x,d) &= \bigcup_{a\in d} a(S') = \bigcup_{a\in d} a(\bigcup_{i=1}^{n}\bigcap_{j=1}^{m_i} L_{i,j}) \\ &= \bigcup_{a\in d}\bigcup_{i=1}^{n}\bigcap_{j=1}^{m_i} a(L_{i,j}) \\ &= \bigcup_{i=1}^{n}\bigcup_{a\in d}\bigcap_{j=1}^{m_i} a(L_{i,j})\end{aligned}$$

The problem has become significantly simpler: we only have to compute the union over all $a \in d$ of individual *clauses* of the form $\bigcap_{j=1}^{m_i} a(L_{i,j})$. There are two cases. In the first case, the clause contains two complementary literals, that is, x and $\overline{x}$. Then $\bigcap_{j=1}^{m_i} a(L_{i,j}) = \emptyset$. Otherwise, take the assignment $a_{\max} \in d$ that maximizes each literal. For a literal x, this means $a_{\max}(x) = \mathrm{lub}(d(x))$, and for a literal $\overline{x}$, it means $a_{\max}(x) = \mathrm{glb}(d(x))$. As the domain only represents set intervals, we know that for any other $a \in d$, $a(L) \subseteq a_{\max}(L)$ for any literal L in the clause. So the following equation holds:

$$\bigcup_{i=1}^{n}\bigcup_{a\in d}\bigcap_{j=1}^{m_i} a(L_{i,j}) = \bigcup_{i=1}^{n}\bigcap_{j=1}^{m_i}\bigcup_{a\in d} a(L_{i,j})$$

We now have reduced the problem to the initial special case, where the union over all assignments only considers literals $L_{i,j}$, which are variables or their negations.

In summary, given a disjunctive normal form of the upper bound $S[\mathscr{U}/x]$, we can compute $\mathrm{lub}(S,x,d)$ efficiently. And dually, a conjunctive normal form for $\overline{S[\emptyset/x]}$ yields an efficient algorithm for computing $\mathrm{glb}(S,x,d)$. The transformation to disjunctive and conjunctive normal forms can cause an exponential blow-up, but only in the number of variables, not in the sizes of the variables' upper bounds. The blow-up is expected, since a complete propagator for Boolean set constraints naturally decides the NP-complete problem of satisfiability of Boolean formulas.

Example 11.10 (Propagating partition) Let $S = \mathscr{U}$ be an equation normal form for the partition constraint $x = y \uplus z$ as seen in Example 11.3. We want to look at how the propagator p_S for this constraint prunes the lower bound of the variable y. The definition of p_S states that we have to compute the set $\bigcap_{a\in d} a(\overline{S[\emptyset/x]})$. The y-interval normal form for S yields the lower bound $\overline{S[\emptyset/y]} = (x \cap \overline{z}) \cup (\overline{x} \cap z)$. First of all, we have to transform this into conjunctive normal form, which yields $(x \cup z) \cap (\overline{x} \cup \overline{z})$. Now we have to compute $\bigcap_{a\in d} a((x \cup z) \cap (\overline{x} \cup \overline{z}))$. As set

intersection is commutative, this is equal to

$$\left(\bigcap_{a\in d} a(x \cup z)\right) \cap \left(\bigcap_{a\in d} a(\overline{x} \cup \overline{z})\right)$$

None of the conjuncts is trivial (containing a variable and its negation), so we can push the intersection further in and replace it with the lower and upper bound of the variable domains:

$$\left(\bigcap_{a\in d} a(x) \cup \bigcap_{a\in d} a(z)\right) \cap \left(\bigcap_{a\in d} a(\overline{x}) \cup \bigcap_{a\in d} a(\overline{z})\right)$$

$$= (\mathrm{glb}(d(x)) \cup \mathrm{glb}(d(z))) \cap (\overline{\mathrm{lub}(d(x))} \cup \overline{\mathrm{lub}(d(z))})$$

This last expression can be implemented efficiently, for example using range iterators as introduced in the previous chapter. Section 11.5 will discuss implementation issues in more detail. *

11.3 Negation of Boolean Set Constraints

The Boolean constraints we examined so far were all *positive*, expressed as conjunctions of equations and subset relations. In this section, we investigate their complements, the *negative* Boolean constraints.

We have seen that any constraint $[\![C]\!]$ can be represented in equation normal form $[\![S = \mathscr{U}]\!] = \{a \in \mathsf{Asn} \mid a \vDash S = \mathscr{U}\}$. Its negation is the complement of $[\![S = \mathscr{U}]\!]$ and can thus be written as

$$\left\{a \in \mathsf{Asn} \;\middle|\; a \vDash \overline{S} \neq \emptyset\right\} =: [\![\overline{S} \neq \emptyset]\!]$$

Additional expressivity

Adding negation increases the expressivity of our constraint language significantly. The following theorem states that given a universe with more than one element, a negative constraint cannot be expressed as a positive constraint unless the constraint is trivial.

Theorem 11.11 Let $|\mathscr{U}| > 1$. For any two expressions S and T, if $[\![S = \mathscr{U}]\!] = [\![T \neq \emptyset]\!]$, then $[\![S = \mathscr{U}]\!] = \emptyset$ or $[\![S = \mathscr{U}]\!] = \mathsf{Asn}$. *

Proof. We prove the contrapositive statement. Assume that $[\![S = \mathscr{U}]\!] \neq \emptyset$ and $[\![S = \mathscr{U}]\!] \neq \mathsf{Asn}$. Moreover, assume $[\![T \neq \emptyset]\!] \neq \emptyset$, as otherwise trivially $[\![S = \mathscr{U}]\!] \neq [\![T \neq \emptyset]\!]$.

Without loss of generality, we can assume that S is in conjunctive normal form $S = \bigcap_{i=1}^{n} \bigcup_{j=1}^{m_i} L_{i,j}$, and that there is a conjunct $\bigcup_{j=1}^{m_i} L_{i,j}$ that does not contain complementary literals, as otherwise the constraint would be trivial.

Consider an assignment $a \in [\![T \neq \emptyset]\!]$. We know $a \vDash T \neq \emptyset$, and there is a value $v \in a(T)$ that is a witness for that. Let $v' \in \mathscr{U} \setminus \{v\}$. Construct a new assignment a' as follows:

$$a'(x) = \begin{cases} \{v\} & \text{if } L_{i,j} = x \text{ for some } j, \text{ and } v \in a(x) \\ \emptyset & \text{if } L_{i,j} = x \text{ for some } j, \text{ and } v \notin a(x) \\ \{v, v'\} & \text{if } L_{i,j} = \overline{x} \text{ for some } j, \text{ and } v \in a(x) \\ \{v'\} & \text{if } L_{i,j} = \overline{x} \text{ for some } j, \text{ and } v \notin a(x) \\ a(x) & \text{otherwise} \end{cases}$$

Then $a' \in [\![T \neq \emptyset]\!]$, as for each $x \in X$, the new assignment a' behaves in exactly the same way as a with respect to the witness v. But $a \notin [\![S = \mathscr{U}]\!]$, because none of the disjuncts $L_{i,j}$ contributes v'. So $[\![S = \mathscr{U}]\!] \neq [\![T \neq \emptyset]\!]$. ∎

Propagating negative Boolean constraints

A negative Boolean set constraint $[\![S \neq \emptyset]\!]$ can prune the domain of a variable x under one single condition: if there is only one element v left that can serve as a witness of the set S not being empty, that is, if only including v in the lower bound of x or excluding it from the upper bound of x can satisfy the constraint.

Let us look at the constraint in more detail. We again use Shannon's expansion:

$$\begin{aligned} & [\![S \neq \emptyset]\!] \\ =& [\![(S[\emptyset/x] \cap \overline{x}) \cup (S[\mathscr{U}/x] \cap x) \neq \emptyset]\!] \end{aligned}$$

Thus propagation can only happen if one of the two disjuncts is already empty, and only a single v is left that can be used to make the other disjunct non-empty:

$$\bigcup_{a \in d} a(S[\emptyset/x]) \cap \overline{\mathrm{glb}(d(x))} = \emptyset \quad \text{and} \quad \mathrm{lub}(d(x)) \cap \bigcup_{a \in d} a(S[\mathscr{U}/x]) = \{v\}$$

in which case v must belong to the lower bound of x, or

$$\bigcup_{a \in d} a(S[\emptyset/x]) \cap \overline{\mathrm{glb}(d(x))} = \{v\} \quad \text{and} \quad \mathrm{lub}(d(x)) \cap \bigcup_{a \in d} a(S[\mathscr{U}/x]) = \emptyset$$

which means that v cannot belong to the upper bound of x. If both disjuncts are empty, that is, if

$$\bigcup_{a \in d} a(S[\emptyset/x]) \cap \overline{\mathrm{glb}(d(x))} = \emptyset \qquad \text{and} \qquad \mathrm{lub}(d(x)) \cap \bigcup_{a \in d} a(S[\mathscr{U}/x]) = \emptyset$$

then propagation has detected failure. The unions and intersections over all $a \in d$ can be computed algorithmically using the technique from the previous section. A propagation algorithm based on these rules for all variables x is complete for the constraint $[\![S \neq \emptyset]\!]$.

Subsumption and reification

Recall that a propagator p is subsumed by a domain d if and only if for all domains $d' \subseteq d$ we have $p(d') = d'$. Another interpretation is that a propagator can be regarded as subsumed as soon as the constraint it induces is entailed. A constraint c is entailed by a domain d if $c \cap d = d$, or equivalently, if $\overline{c} \cap d = \emptyset$.

For Boolean set constraints, this means that the constraint $[\![S = \mathscr{U}]\!]$ is entailed by a domain d if and only if $[\![\overline{S} \neq \emptyset]\!]$ is failed in d. We have just seen how to detect failure when propagating a negated Boolean set constraint, so we can detect entailment of $[\![S = \mathscr{U}]\!]$ as

$$\bigcup_{a \in d} a(\overline{S}[\emptyset/x]) \cap \overline{\mathrm{glb}(d(x))} = \emptyset \qquad \text{and} \qquad \mathrm{lub}(d(x)) \cap \bigcup_{a \in d} a(\overline{S}[\mathscr{U}/x]) = \emptyset$$

which is equivalent to

$$\bigcup_{a \in d} a(\overline{S[\emptyset/x]}) \subseteq \mathrm{glb}(d(x)) \qquad \text{and} \qquad \mathrm{lub}(d(x)) \subseteq \bigcap_{a \in d} a(S[\mathscr{U}/x])$$

Note the similarity to propagating $[\![S = \mathscr{U}]\!]$: if the biggest possibly interpretation of the lower bound expression is a subset of the lower bound of x, and the smallest possible interpretation of the upper bound expression is a superset of the upper bound of x, then the propagator is subsumed.

Checking subsumption is essential for reified constraints. A reified Boolean set constraint can be thought of as defined by an expression $C \leftrightarrow b$, for a Boolean 0/1 variable b. We can detect subsumption of and propagate $[\![C]\!]$ and its complement. Thus, we can derive a propagation algorithm for any reified Boolean set constraint.

As discussed in Section 5.3, a subsumption check can be used to improve propagator scheduling. However, the run-time costs of checking subsumption as presented here probably outweigh the gain from the improved scheduling.

Conjunctions of negative and positive constraints

Although negation adds significant expressivity, there are still many useful constraints that cannot be represented in this language. For example, the constraint $x \subset y$ would correspond to $x \subseteq y \wedge x \neq y$, a conjunction of a positive and a negative constraint. In our setup, we can only use two distinct propagators for this class of constraints, possibly giving up $\mathscr{D}^{[\mathscr{P}(\mathscr{U})]}$ completeness. It is a promising direction of future work to investigate which more general classes of set constraints can be translated to efficient and $\mathscr{D}^{[\mathscr{P}(\mathscr{U})]}$-complete propagators.

11.4 Techniques for n-ary Boolean Set Propagators

With the techniques introduced in this chapter, the asymptotic run-time of an algorithm for an n-ary Boolean set propagator is at least quadratic. In this section, we show how to get linear-time algorithms for a class of important n-ary constraints.

The run-time of an algorithm for a propagator p_S for a constraint $[\![S = \mathscr{U}]\!]$ depends on the size of the interval normal forms. We define the size of an expression S as the number of set operations (union, intersection, complement) S contains, and write it $|S|$. For each variable x that appears in S, let $S^l_x = \overline{S[\emptyset/x]}$ and $S^u_x = S[\mathscr{U}/x]$ such that S^l_x is in conjunctive normal form, and S^u_x is in disjunctive normal form. Then an algorithm for p_S has to perform $\sum_x |S^l_x| + |S^u_x|$ set operations. Abstracting from the cost of individual operations (as it depends on how sets are implemented), the run-time of an algorithm for p_S is therefore in $O(\sum_x |S^l_x| + |S^u_x|)$.

Example 11.12 (An n-ary propagator) The set constraint $[\![y = \bigcup_{1 \leq i \leq n} x_i]\!]$ yields $n + 1$ interval normal forms:

$$\bigcup_{i=1}^{n} x_i \subseteq y \subseteq \bigcup_{i=1}^{n} x_i \qquad \text{INF}_y$$

$$y \cap \bigcap_{j \neq i} \overline{x_j} \subseteq x_i \subseteq y \qquad \text{INF}_{x_i}$$

The size of the set expressions in each INF is in $O(n)$. The overall run-time of the propagation algorithm is therefore in $O(n^2)$. *

A generalized form of common subexpression elimination can be used in order to propagate in linear time. We now sketch this technique using the above example.

The lower bound expressions for the variables x_i are in conjunctive normal form. Thus, we know that $\text{glb}(S, x_i, d) = \text{glb}(d(y)) \cap \bigcap_{j \neq i} \overline{\text{lub}(x_j)}$. Assume we propagate in the order $x_1 \ldots x_n$. Then at step i, we know that for all $j > i$, we have not yet changed the domain of x_j. Thus, we can use a pre-computed table $\mathit{right}[i] =$

$\bigcap_{j>i} \overline{\mathrm{lub}(x_j)}$. The other half of the intersection, $left_i = \bigcap_{j<i} \overline{\mathrm{lub}(x_j)}$, can be maintained incrementally while moving from step $i-1$ to step i. This yields the following propagation algorithm (we only show the pruning of the x_i lower bounds):

```
PROPAGATE(d)
1  right[n] ← 𝒰
2  for i ← n − 1 downto 1
3       do right[i] ← right[i + 1] ∩ lub(d(x_{i+1}))‾
4  left ← 𝒰
5  for i ← 1 to n
6       do x_i.adjglb(glb(d(y)) ∩ right[i] ∩ left)
7            left ← left ∩ lub(d(x_i))‾
```

Computing the complete table $right[i]$ requires time $O(n)$. The set *left* is maintained incrementally in constant time in line 7. Thus, $\mathrm{glb}(S, x_i, d)$ is computed in line 6 in constant time. This yields a run-time of $O(n)$ for the whole propagator.

Generalizing this technique, for any constraint $S = \mathscr{U}$, we look for subexpressions of the form $\bigcap_{i\neq j} S'_i$ in the conjunctive normal forms of $\overline{S[\emptyset/x]}$, and $\bigcup_{i\neq j} S'_i$ in the disjunctive normal forms of $S[\mathscr{U}/x]$. These subexpressions can then be evaluated as presented above, improving the run-time by a factor of n.

11.5 Implementing Boolean Set Propagators

We have seen in this chapter how to derive propagators for Boolean set constraints from high-level specifications. This section develops and evaluates implementation strategies for both the translation from specification to propagator, as well as for the propagators themselves.

Translation using Binary Decision Diagrams

Generating a $\mathscr{D}^{[\mathscr{P}(\mathscr{U})]}$-complete propagator from a specification of a Boolean set constraint involves complex symbolic manipulations of set expressions. We have to transform the specification into conjunctive and disjunctive normal forms of the two parts of its interval normal forms for each variable. Fortunately, we do not have to implement these manipulations from scratch, as *binary decision diagrams* provide all the necessary operations.

A **Binary Decision Diagram (BDD)** represents a Boolean function $f \in \mathbb{B}^n \to \mathbb{B}$ as a rooted, directed, acyclic graph. Each node corresponds to a variable, or to one of the constants 0 and 1. Edges are labeled with either 0 or 1. Each path from the root node ends either in the 0 or the 1 node. A path from the root node to the 1

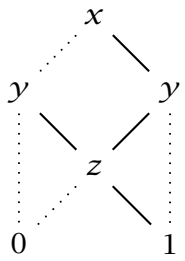


Figure 11.1: Binary decision diagram for the formula $(x \wedge (\neg y \vee z)) \vee (y \wedge z)$

node represents an assignment a such that $f(a) = 1$. Figure 11.1 shows a BDD for the Boolean function given by the formula $(x \wedge (\neg y \vee z)) \vee (y \wedge z)$. Dotted lines represent edges labeled with 0, solid lines correspond to edges with label 1.

BDDs are algorithmically interesting if they are *reduced* and *ordered*, meaning that no two distinct subgraphs are the same, and that variables on each path respect a global variable ordering. The BDD in Figure 11.1 is reduced and ordered. Reduced, ordered BDDs (ROBDDs) were introduced by Bryant (1986). An ROBDD is canonical: for every Boolean function f, and given a fixed variable ordering, there is exactly one ROBDD that encodes f. There are efficient algorithms that, given two ROBDDs, compute an ROBDD that represents the conjunction, disjunction, or implication of the two arguments. Another available operation, which will be especially useful for our purposes, is existential quantification.

Let $[\![S = \mathscr{U}]\!]$ be the Boolean constraint we want to propagate. In order to propagate $[\![S = \mathscr{U}]\!]$, we determine the x-interval normal form $\overline{S[\emptyset/x]} \subseteq x \subseteq S[\mathscr{U}/x]$ for each variable x that appears in S.

As the expression S is a Boolean formula, we can represent it as an ROBDD. We will continue to use set notation. The two components of the INF_x that we need are then easily acquired using existential quantification:

$$\overline{S[\emptyset/x]} = \overline{\exists x : \overline{x} \cap S}$$
$$S[\mathscr{U}/x] = \exists x : x \cap S$$

A suitable package for ROBDD computations can thus be used to implement variable isolation.

An implementation of a propagator p_S for a Boolean set constraint $[\![S = \mathscr{U}]\!]$ has to compute the values of the functions $\mathrm{glb}(S, x, d)$ and $\mathrm{lub}(S, x, d)$ for each variable x. In Section 11.2, we saw that $\mathrm{glb}(S, x, d)$ and $\mathrm{lub}(S, x, d)$ can be implemented efficiently on conjunctive normal forms of $\overline{S[\emptyset/x]}$ and disjunctive normal forms of $S[\mathscr{U}/x]$, respectively. Both normal forms can be obtained directly from the ROBDD. The disjunctive normal form is a disjunction of all paths that lead to the 1 node, where each path represents a conjunction of the literals on the path. For the example

in Figure 11.1, a disjunctive normal form would hence be $(\overline{x} \cap y \cap z) \cup (x \cap \overline{y}) \cup (x \cap y \cap z)$. A conjunctive normal form is obtained dually, by following the paths to the 0 node and negating the literals. Again for the above example, this yields $(x \cup y) \cap (x \cup \overline{y} \cup z) \cap (\overline{x} \cup \overline{y} \cup z)$.

Interpretation versus compilation

Given the disjunctive and conjunctive normal forms that allow us to compute the sets $\mathrm{glb}(S, x, d)$ and $\mathrm{lub}(S, x, d)$ for each variable x, there are two strategies for implementing the corresponding propagator. We can implement an *interpreter* for the set expressions of the normal forms, or we can *compile* the expressions directly to the implementation language of the constraint solver, in our case to C++ code. An interpreter has the advantage of being flexible: new propagators can be created at run-time, without recompiling the program. The compilation approach on the other hand is less flexible, but potentially yields better performance.

Both approaches have to compute the values of $\mathrm{glb}(S, x, d)$ and $\mathrm{lub}(S, x, d)$ for each variable x. Let us take the partition constraint $[\![x = y \uplus z]\!]$ from Example 11.10 again and discuss its implementation. In the example, we saw how to compute $\mathrm{glb}(S, y, d)$ from the lower and upper bounds of the current variable domains. The propagator has to determine the new lower bound of y as

$$\mathrm{glb}(d(y)) \cup \left((\mathrm{glb}(d(x)) \cup \mathrm{glb}(d(z))) \cap (\overline{\mathrm{lub}(d(x))} \cup \overline{\mathrm{lub}(d(z))})\right)$$

Section 10.5 developed set variable domain operations based on range iterators. The partition propagator constructs an iterator that represents the set $\mathrm{glb}(S, y, d)$, and then prunes the domain using the y.`adjglb`$(\cdot)$ domain operation.

The interpreter constructs the iterators at run-time, so it cannot use iterators based on templates in C++ (see Section 9.1). In Gecode, we therefore provide a second set of iterator implementations with dynamic binding using virtual methods.

The compilation approach simply emits C++ code that constructs the iterators directly and then prunes the variable bounds. The generated propagator implementation can therefore use parametric polymorphism through templates. The generated C++ code for the above example appears in Figure 11.2.

For the above example, binary union and intersection range iterators are sufficient. In general, the conjunctive or disjunctive normal forms of course require n-ary operations. Gecode provides n-ary union and intersection iterators, which are more efficient than a decomposition into binary iterators would be.

```
void propagate(void) {
  GlbRanges i1 = x.glb();
  GlbRanges i2 = z.glb();
  Union<GlbRanges,GlbRanges> i3(i1,i2);
  LubRanges i4 = x.lub();
  LubRanges i5 = z.lub();
  Complement<LubRanges> i6(i4);
  Complement<LubRanges> i7(i5);
  Union<Complement<LubRanges>,Complement<LubRanges> > i8(i6,i7);
  Intersection<Union<GlbRanges,GlbRanges>,
               Union<Complement<LubRanges>,Complement<LubRanges> > > i9(i3,i8);

  y.adjglb(i9);
}
```

Figure 11.2: Generated propagator implementation for the partition constraint

Performance analysis

We have implemented both a compiler and an interpreter for Boolean set propagators in Gecode. Table 11.1 shows how the two approaches compare as far as run-time and memory consumption are concerned. The setup for the experiments is the same as in Section 6.9 and Appendix A, but we replaced the hand-implemented ternary intersection propagator that comes with Gecode with a compiled and an interpreted version that were both generated from the simple specification $x = y \cap z$. Both types of propagators perform exactly the same pruning, so the number of propagator invocations is the same (and does not appear in the table). Unsurprisingly, the compiled propagators are much faster.

The set propagator implementations that come with Gecode perform additional cardinality reasoning, so their run-times are not directly comparable to the times reported in Table 11.1. Any implementation of $\mathscr{D}^{[\mathscr{P}(\mathscr{U})]}$-complete Boolean set propagators however has to perform the same inferences as the generated propagators. Hand-optimized implementations therefore only have a limited potential for outperforming generated propagators. For the ternary intersection propagator in the benchmark examples, the generated code is exactly what would have been written by hand. We believe that using standard compiler techniques such as common subexpression elimination, as well as minimization of the generated disjunctive and conjunctive normal forms, we can always generate propagators that are equivalent to hand-implemented versions in terms of performance.

Benchmark	*Compiled*		*Interpreted*	
	time (ms)	*mem. (KByte)*	*time %*	*mem. %*
Social Golfers (8-4-9)	182.90	10 254	1 134.62	120.60
Steiner Triples (9)	142.22	901	343.98	100.00
Hamming Codes (20-3-32)	1 195.78	23 402	486.40	104.92
Sudoku (Set, 1)	2.61	83	803.95	100.00
Sudoku (Set, 4)	6.10	130	675.88	100.00
Sudoku (Set, 5)	46.38	322	597.87	100.00

Table 11.1: Relative performance of interpreted set intersection propagators, compared to compiled versions

11.6 Related Work

► Set constraints have been investigated for a long time. While the term *set constraint* was coined as late as 1990 (Heintze and Jaffar, 1990), already the early ALICE constraint system (Laurière, 1978) provided set constraints as one of its fundamental constructs. In propagation-based solvers, set constraints only appeared after the set interval approximation (see Section 4.2) had been developed by Puget (1992) and later Gervet (1994, 1995, 1997).

► Concerning variable isolation for set constraints, Gervet (2006) states that "Since there is no inverse operation for $\cup,\cap,\setminus$ there is no way to move all the operation symbols on one side of the constraint relation." While this is certainly true in general, our results show that one *can* isolate variables if the set interval approximation $\mathscr{D}^{[\mathscr{P}(\mathscr{U})]}$ is used. This is in fact what makes the set interval approximation so useful: its structure coincides with how variables can be isolated in Boolean set constraints.

► The approach that is closest to ours was developed by Hawkins et al. (2005). They propose to represent set variable domains and propagators as ROBDDs. Both the variable domains and the constraints can be encoded as Boolean functions and thus represented using ROBDDs. In this encoding, not only the set interval approximation can be realized, but even complete domains become tractable. Although this representation may still be exponential in size, it works well for many practical examples. Similar to our approach, propagation using ROBDDs isolates variables using existential quantification. However, each set variable x is represented as its characteristic function using 0/1 variables x_v for each $v \in \mathscr{U}$. That way, the propagators do not compute new lower and upper bounds, but determine the value of each of the x_v directly. It is straightforward to derive ROBDD-based propagators from Boolean set constraint specifications. Consider an equation normal form $S = \mathscr{U}$. It can be turned into the 0/1 Boolean function $\bigvee_{v \in \mathscr{U}} S[x_{1,v}/x_1, \ldots, x_{n,v}/x_n]$, which is then represented as an ROBDD. Even though ROBDDs can be used to implement the set interval approximation, our approach still has significant advantages: (1) It can be used for existing systems that do not have a BDD-based set constraint solver. (2) A

direct implementation of the set interval approximation is more memory efficient. (3) Our propagators can be compiled statically, and independent of the size of $\mathscr{U}$. Still, they offer the same compositionality as ROBDD-based propagators.

► Müller (2001) develops set propagators in the context of the Mozart system. He defines *projectors*, propagators that only prune the domain of a single variable. Projectors are defined in terms of set expressions, and are therefore closely related to the Boolean set propagators we derive: the propagator p_S can be regarded as a collection of projectors, one for each variable x. Projectors as defined by Müller transfer the idea of indexicals (which is discussed below) from integer to set constraints. While Müller uses an implementation language based on set expressions, he does not present a general method for deriving projectors from constraint specifications, but just gives some examples for common constraints. As Mozart requires propagators to be idempotent, Müller describes methods for determining an efficient propagation order for projectors. Several projectors are then grouped into a single, idempotent propagator. The same techniques can be applied to the Boolean set propagators we derive, determining in which order a concrete implementation should prune the individual variable domains.

► Defining the pruning of a single variable domain using expressions that involve the remaining variables is exactly what *indexicals* do. The main idea of indexicals goes back to cc(FD) (Van Hentenryck et al., 1991, 1998) and was later elaborated in the context of clp(FD), AKL, and SICStus Prolog (see Codognet and Diaz, 1996; Carlson, 1995; Carlsson et al., 1997). Indexicals build on *range expressions* as a language for defining the projection of a constraint over integer variables. Range expressions are a more complicated language than our set expressions, as they provide arithmetic operations and can be evaluated both on integer intervals and on full integer domains. In contrast to the propagators we derive, indexicals are not contracting and monotonic by construction. Instead, the range expressions have to be designed carefully to yield a correct propagator. Carlson (1995) develops a compilation approach that generates indexicals from specifications of arithmetic constraints. He isolates every variable of the arithmetic expression and generates an indexical for it. However, variable isolation for arithmetic expressions is easier than for set constraints, as every arithmetic operation he considers has an inverse.

► We check entailment of a Boolean set constraint by computing the largest possible sets for the lower bound set expressions, and the smallest possible sets for the upper bound set expressions. In the indexical scheme, this technique corresponds to checking entailment using *anti-monotonic* indexicals (Carlson et al., 1994a).

► We saw in Section 2.2 that models with set constraints often involve the cardinalities of the sets. Azevedo (2007) defines ad-hoc propagation rules that perform cardinality reasoning for a number of Boolean set constraints. Bessière et al. (2004) present propagation algorithms that reason about cardinality for several Boolean set constraints, and show that for other Boolean set constraints, cardinality reasoning

is NP-hard. It is not clear how our systematic approach to propagation of Boolean set constraints can be extended with cardinality reasoning.

► Ågren et al. (2007) specify set constraints in the context of constraint-based local search using monadic second-order logic (MSO). The purely Boolean expressions we use can be regarded as the fragment of MSO that has only a single first-order universal or existential quantifier. Ågren et al. thus have a richer language, it supports existential second-order quantification as well as arbitrarily nested first-order quantifiers. They can deal with the additional expressivity because instead of performing constraint propagation for the specified constraints on full domains, they compute a *penalty* that describes how much a single given assignment violates a constraint.

Contributions of Part II

This second part of the dissertation developed novel techniques for deriving propagators and their implementations. We derived propagators from existing propagators, as well as from specifications of Boolean set constraints. The contributions of this part can be summarized as follows.

1. Deriving propagators using *views* is a novel technique, introduced, discussed thoroughly, and evaluated empirically in this dissertation. We provide formal definitions of views and derived propagators, show that derived propagators induce the desired constraints, and identify the conditions under which they preserve completeness of the original propagators. We develop an efficient implementation architecture based on *parametric* propagators. The experiments suggest that an implementation using parametric polymorphism in the form of C++ templates incurs no run-time or memory overhead. Views are widely applicable, as demonstrated by the four general *techniques* we present for deriving propagators: transformation, generalization, specialization, and type conversion. Using these techniques, views save the tremendous amount of 120 000 lines of code in the implementation of Gecode, which underlines their relevance in practice.
2. For an efficient implementation of propagation algorithms, this dissertation develops the notion of *range iterators*, which provide set-valued operations on variable domains. Using range iterators, set-valued operations on views can be implemented in a completely modular way, and without requiring additional memory for temporary set data structures. Furthermore, range iterators simplify propagator implementation, they serve as *adaptors* between internal propagator data structures and the variable domain operations during propagation.
3. This work is the first to present a systematic technique for propagating *Boolean set constraints*. While previous work on set constraints was confined to stating propagation algorithms as ad-hoc propagation rules, we derive set-interval-complete propagators and their implementations from specifications of the constraints. The key insights are that Boolean set constraints permit a limited form of variable isolation, yielding *interval normal forms* for each variable; and that in the set-interval approximation, one can derive efficient algorithms for computing the bounds resulting from the interval normal forms.

12 Conclusions

This dissertation aimed at developing a well-understood, modular, correct, comprehensive and efficient propagation-based constraint solver, based on a solid mathematical model of constraint propagation as well as a carefully designed implementation architecture. This chapter presents a summary of the contributions towards this goal, and an outlook on some further research questions that we did not answer in this dissertation.

12.1 Summary and Main Contributions

The first part of this dissertation developed a *propagation kernel*, the part of a constraint solver that provides the infrastructure for propagation.

As a solid foundation, we defined a mathematical framework that describes concisely how constraint propagation works. We defined *Constraint Satisfaction Problems* as a denotational model that captures *what* problems we want to solve, and *Propagation Problems* as a more operational model that explains *how* to solve these problems, on an abstract level. A propagation problem realizes the constraints of a CSP using *propagators*, which we defined as contracting and sound functions that transform a domain into a stronger domain. Each propagator realizes a decision procedure for one particular constraint, and additionally it can prune non-solutions from the domain, reducing the search space.

In order to solve real-world propagation problems, the propagators typically have to be *strong*, that is, they have to be able to prune as much of the non-solutions of their constraints as possible. We showed that there is a unique strongest and a unique weakest propagator for each constraint. We then characterized propagation strength using *domain approximations*, identifying propagators whose strength lies between the strongest and the weakest propagators. Completeness with respect to domain approximations generalizes the well-known notions of bounds and domain as well as set-interval consistency.

The mathematical model we presented is inspired by many sources. It combines the ideas from these sources in a compact model that completely describes a constraint solver. We went beyond previous models, making the induced constraint of

a propagator explicit, discussing monotonicity and idempotency in detail, and establishing the connection between consistency notions and propagator completeness with respect to approximations.

After laying the foundation with the mathematical framework, we turned to the architecture and implementation of a propagation kernel. The mathematical model describes the process of constraint propagation as transition system that chooses nondeterministically which propagator to apply next. A constraint solver must make this choice deterministically, so we applied several well-known techniques such as *agenda-based propagation* using a priority queue of propagators, and *event-directed propagation*, which only schedules propagators if the variables they are subscribed to change in certain ways. We also discussed how more advanced techniques such as dynamic fixed point reasoning and propagator staging fit into this framework. Finally, we refined the model with the novel concepts of propagation conditions and modification events, which are the basis for an efficient implementation of event-directed propagation.

Based on the mathematical model, we developed an object-oriented implementation architecture for a constraint solver. An important point is the clear distinction between the *domain-independent* propagation kernel, which provides services like propagator scheduling and copying, and the *domain module*, which contributes the domain-specific parts like the actual variable domains or propagation algorithms. *Contracts* between propagators, variables, and the kernel establish strong invariants that lead to a streamlined, efficient implementation. We carefully designed and evaluated the main data structures of the kernel, implementing the dependency mapping and the priority queue, and gave a detailed description of the copying mechanism that is used for backtracking during search. The implementation architecture is the basis of the Gecode C++ constraint solving library. Benchmarks showed that Gecode outperforms even state-of-the-art commercial constraint solvers. Based on Gecode, we empirically evaluated different design decisions for the kernel data structures and algorithms. The implementation thus validates the viability of the models and the success of the approach.

In the second part of this dissertation, we developed two new techniques for *deriving propagators*. Both techniques address the problem of implementing a comprehensive number of propagators.

The first technique *reuses* propagators by combining them with *views*, thereby realizing slight variations of constraints. We modeled views as functions that can be composed with a propagator, transforming its input and output domain. This composition is again a propagator, the *derived propagator*. By modeling views in our mathematical framework, we were able to prove several important properties. First, the derived propagator induces the intended constraint. Second, if the original propagator is domain-complete, then the derived propagator is also domain-complete.

Third, if the original propagator is complete with respect to a domain approximation, and the views are compatible with that approximation, then the derived propagator is also complete with respect to the approximation. We can therefore say that derived propagators are *perfect*, in that they inherit all the essential properties.

We presented four general techniques for deriving propagators using views. Algebraic *transformations* derive propagators using Boolean negation, integer negation, or set complement views. The *generalization* technique uses views such as scale or offset views to derive more complex propagators from simpler ones. Constant views allow us to *specialize* propagators, and *type conversion* views mediate between different types of variables.

The implementation architecture of views again followed the mathematical model. In the implementation, propagators are derived by *instantiating parametric propagators*. Parametric polymorphism in the form of C++ templates leads to a *perfect implementation* of derived propagators, as monomorphization and subsequent compiler optimizations make sure that deriving a propagator incurs no overhead compared to implementing a dedicated propagator by hand. *Range iterators* complemented the architecture by providing efficient set-valued domain operations for both variables and views, and by simplifying the implementation of propagation algorithms. Experiments showed that views provide superior performance compared to simple decompositions into smaller constraints, and that the compiler optimizations enabled by parametric polymorphism in C++ yield a significantly improved performance compared to an implementation based on dynamic binding. Furthermore, we saw that deriving propagators is an indispensable technique for the implementation of Gecode, as it saves more than 120 000 lines of code and documentation.

As a second technique for deriving propagators, we defined a specification language for *Boolean set constraints*, and, for the first time, showed how constraints specified this way can be automatically translated to propagation algorithms. Boolean set constraints elegantly describe a large class of useful constraints over set variables. At the same time, we can use all the equivalence transformations known from Boolean algebra. Boolean set constraints are transformed into x-interval normal forms for each of their variables x, and these normal forms provide pruning of the lower and upper bound of x. The main result is that one can automatically generate set-interval-complete propagation algorithms from Boolean constraint specifications. We extended the approach to support negated and reified constraints. For the implementation of Boolean set propagators, we employed ROBDDs, which provide an efficient implementation of the symbolic manipulations required to acquire the interval normal forms. The resulting set expressions are then evaluated using range iterators, which can be either constructed dynamically at run-time, or compiled to C++ code. Both approaches are available in Gecode.

12.2 Future Research

The work presented in this dissertation can be extended in a number of interesting directions.

Concurrent propagation. The architecture of personal computers has been changing for a number of years now. Instead of increased clock frequencies, modern computers have an increased number of processors (be it CPUs or GPUs). The challenge for all software development in the coming years is therefore parallelization. Constraint solvers, especially those based on copying, are in a good position, as search can be made parallel relatively easily. An equally interesting, yet largely unexplored approach is to parallelize constraint propagation. We should take advantage of the strong properties of propagators compared to arbitrary computations, like contraction and possibly monotonicity, and research implementation strategies that yield low synchronization overhead between several concurrently running propagators.

Monotonicity. We have seen that propagators do not have to be monotonic for the constraint solver to be correct. This opens up the development of approximative, randomized, or heuristic propagation algorithms. These would yield stronger pruning for constraints for which a complete propagator is computationally too expensive.

Copying versus trailing. With Gecode, we have evidence that a constraint solver based on copying and recomputation provides competitive performance compared to all trailing solvers on the market. However, for special problem classes such as SAT, trailing has its advantages. It should be enlightening to investigate how a hybrid approach can bridge the gap between copying and trailing solvers, and between general constraint solvers and dedicated SAT solvers. A first step in this direction has been taken by Reischuk (2008).

Set propagators. The approach presented in this dissertation yields set interval complete propagators only for positive and negative Boolean set constraints. It will be interesting to investigate which larger classes of Boolean set constraints still permit the generation of efficient and complete propagation algorithms.

Cardinality reasoning. Many problem formulations that use set variables constrain the cardinality of the sets. However, reasoning about the cardinality during propagation is extremely difficult. Already for simple constraints such as $\forall 1 \leq i < j \leq n, |x_i \cap x_j| \leq k$ for set variables $x_1, \ldots, x_n$ and an integer k, complete propagation taking into account the cardinality of the sets x_i is NP-hard (Bessière et al., 2004). It is a challenging task to develop techniques for deriving propagators from Boolean set constraint specifications that perform a limited, but effective form of cardinality reasoning.

A Benchmarks

This appendix presents a list of all the models that were used as benchmarks in this dissertation. Implementations for all problems are available as part of Gecode. Pointers to the corresponding entries in the CSPLib (Gent and Walsh, 1999) are provided where appropriate. Section A.5 gives an overview of the performance, search and propagation characteristics of the examples' Gecode implementations.

A.1 Models with Integer and Boolean Variables

Alpha

A well-known cryptoarithmetic puzzle of unknown origin. It consists of 20 linear equations over 26 variables (the letters of the alphabet). The *smart* version uses a first-fail branching heuristic, whereas the *naive* version enumerates the variables in the given order.

BIBD(v,k,λ)

Balanced incomplete block design, problem 28 in the CSPLib. We use one fixed instance with $v = 7$, $k = 3$, $\lambda = 60$.

Eq-20

A standard benchmark for solving a system of 20 linear equations.

Golomb Rulers(n)

Find a ruler of minimal size that has n markers, such that the distances between any two markers is different from the distance between any two other markers (problem 6 in the CSPLib).

Graph Coloring

Given an undirected graph, find the minimum number of colors such that you can assign each node of the graph a color and no two adjacent nodes have the same color. The description of the graph lists cliques of nodes explicitly, so that the *all-different* constraint can be used.

Knights(n)
Fill an $n \times n$ chessboard with knights such that the knights do a full tour by knights move (last knight reaches first knight again). The formulation is due to Gert Smolka.

Magic Sequence(n)
Find a sequence $[x_0, \ldots, x_{n-1}]$ of integers such that for each $i \in \{0, \ldots, n-1\}$, the number i occurs exactly x_i times in the sequence. Three different models are used in the benchmarks: using *reified constraints* (called *naive* in the tables), using *counting constraints* (called *smart*), and using a *global cardinality constraint* (called *GCC*).

Partition(n)
Partition n numbers into two groups, so that the sum of the first group equals the sum of the second, and the sum of the squares of the first group equals the sum of the squares of the second.

Perfect square packing
Packing squares into a rectangle without overlap. Problem 9 in the CSPLib.

Photo Alignment
A group of people wants to take a group photo. Each person can give preferences next to whom he or she wants to be placed on the photo. The problem to be solved is to find a placement that satisfies as many preferences as possible.

Queens(n)
Place n queens on an $n \times n$ chessboard so that no two queens attack each other. The *naive* version uses binary disequality propagators, the *smart* version uses special *all-different* propagators with offsets. The default propagation strength of the *all-different* is the simple value propagation; if domain propagation is used, this is noted in the example as Dom.

A.2 Models with Set Variables

Crew Allocation
Assign 20 flight attendants to 10 flights. Each flight needs a certain number of cabin crew, and they have to speak certain languages. Every cabin crew member has two flights off after an attended flight.

Hamming Codes(b,d,n)
Generate a Hamming code that fits in b-bit words to code n symbols where the Hamming distance between every two symbol codes is at least d. The Hamming distance between to words is the number of bit positions where they differ. This instance fixes b to 20, d to 3, and n to 32.

Social Golfers(g,s,w)
In a tournament of w weeks, schedule $g \times s$ golfers in g groups per week, each of size s, such that no two golfers play against each other in a group more than once. This is the problem from Example 2.1, and problem 10 in the CSPLib.

Steiner Triples(n)
Find a set of $n \times (n-1)/6$ triples of distinct integers in $\{1, \ldots, n\}$ such that no two triples share more than one element. This is problem 44 in the CSPLib.

Sudoku
The logic puzzle explained in Section 2.1, but using a model based on set variables. All instances are "classical" 9×9 Sudokus with a unique solution that cannot be solved by propagation alone. In order to make the problem slightly harder, an all-solution search (proving uniqueness of the solution) is performed.

Queen Armies
The goal of this problem is to place as many white and black queens on a chess-board without any two queens of different color attacking each other. The number of black queens should be greater than or equal to the number of white queens. The model is based on the one presented by Smith et al. (2004).

A.3 SAT Problems

The SAT (Boolean satisfiability) problems are all given in DIMACS clause format as used by SATLIB (Hoos and Stützle, 2000), and run through the DIMACS parser that comes with Gecode. All problems except for the Ramsey problem are from SATLIB.

Dubois (20)
An unsatisfiable, generated instance, DIMACS generated by Olivier Dubois.

Towers of Hanoi (4)
A towers of Hanoi problem with four blocks, DIMACS generated by Bart Selman.

Ramsey (n)
This problem determines whether n is a lower or upper bound for the Ramsey number $R(4,4)$, that is, whether there is a $K(4,4)$ subgraph in graphs with n vertices. DIMACS generated by Raphael Reischuk.

Pigeon Hole (n)
Place $n+1$ pigeons in n holes without placing two pigeons in the same hole. Classic unsatisfiable problem, DIMACS generated by John Hooker.

Flat ($200-1$)
A generated, quasi-random graph coloring, DIMACS generated by Joseph Culberson.

A.4 Stress Tests

Domain Stress
Cut holes into variable domains and then contract the domains again, until one domain is empty. This test measures the performance of domain operations on integer variables.

Propagation Stress
Prove unsatisfiability of the problem $x < y \wedge y < x$, where the domain size of both x and y is 1 000 000. This test measures mainly how fast the scheduling and execution of propagators is in a system.

Search Stress
Explore the complete search space of a problem with n variables of domain size n each and no constraints.

A.5 Gecode Performance

Table A.1 lists the run-time, memory requirements, number of failures in the search tree, and number of propagation steps for the individual benchmark examples. The experiments were run using the upcoming Gecode 3.0.0 on an Intel Core 2 Duo processor at 1.83 GHz, equipped with 2GB of RAM, running Mac OS 10.5.5. The programs were compiled using the GNU C++ compiler, version 4.2.1. All run-time

results are wall time, taken as the arithmetic mean of 20 runs, with a coefficient of deviation below 2% for all benchmarks, except for the SAT examples, where the run-time results are the arithmetic mean of five runs.

Figure A.1 and Table A.2 compare Gecode's run-time performance with ILOG Solver 6.5 and SICStus Prolog 4.0.2. Due to licensing issues, the experiments were not run on the same machine as all other experiments in this dissertation, but on an Intel Pentium 4 processor at 2.8 GHz, 1GB of RAM, running Linux. Both the Gecode and the ILOG examples were compiled using the GNU C++ compiler, version 4.3.2. All run-time results are the average of 20 runs, with a coefficient of deviation below 2% for all benchmarks. We only ran a subset of the benchmarks presented in Table A.1. For the other examples, we could not find models that lead to the same search space with all three solvers. This turned out to be particularly difficult for problems involving set constraints, because the propagation strength of set propagators that perform cardinality reasoning is not specified for ILOG Solver. SICStus Prolog does not provide set constraints.

Benchmark	*time (ms)*	*mem. (KByte)*	*failures*	*propagations*
Alpha (smart)	0.92	14	33	2 063
Alpha (naive)	81.57	23	7 435	136 179
BIBD	1 814.52	4 452	1 306	921 554
Eq-20	1.31	14	54	3 460
Golomb Rulers (10)	806.17	419	8 890	1 181 770
Graph Coloring	352.99	3 140	1 078	128 120
Knights (18)	65.81	11 281	15	102 259
Magic Sequence (Smart, 500)	172.50	4 227	251	84 302
Magic Sequence (Naive, 500)	681.87	19 843	251	1 254 911
Magic Sequence (GCC, 500)	327.43	330	251	3 908
Partition (32)	9 173.41	277	160 258	12 107 504
Perfect Square	179.01	3 780	150	305 391
Photo Alignment	130.94	63	10 345	468 610
Queens (Naive, 10)	55.26	51	4 992	269 819
Queens (Smart, 10)	33.27	27	4 992	43 448
Queens (Naive, 100)	34.56	4 038	22	16 821
Queens (Smart, 100)	1.17	239	22	455
Crew Scheduling	3.94	246	34	2 389
Hamming Codes (20-3-32)	1 377.77	23 402	2 296	903 217
Social Golfers (8-4-9)	175.50	10 254	32	181 343
Social Golfers (5-3-7)	1 323.41	2 117	1 174	891 423
Steiner Triples (9)	143.38	901	1 067	241 788
Sudoku (Set, 1)	3.46	83	0	1 821
Sudoku (Set, 4)	7.56	130	1	3 752
Sudoku (Set, 5)	57.93	322	25	28 038
Queen Armies	318.40	83	5 602	513 615
Dubois (20)	100 882.27	196	3 145 728	127 222 546
Towers of Hanoi (4)	156 613.64	1 924	888 424	289 149 334
Ramsey (4-4-13)	2 099 643.62	2 053	14 546 227	2 566 529 591
Ramsey (4-4-10)	1 140.24	323	19 575	2 389 124
Pigeon Hole (7)	258.21	100	32 781	1 099 849
Pigeon Hole (8)	10 394.45	244	378 344	13 894 407
Flat (200-1)	35 083.11	2 181	167 618	74 244 397
Domain Stress	16.90	131	1	0
Failure Stress	43.14	5	1	499 999
Search Stress	151.37	27	0	0

Table A.1: Benchmarks for the standard Gecode distribution

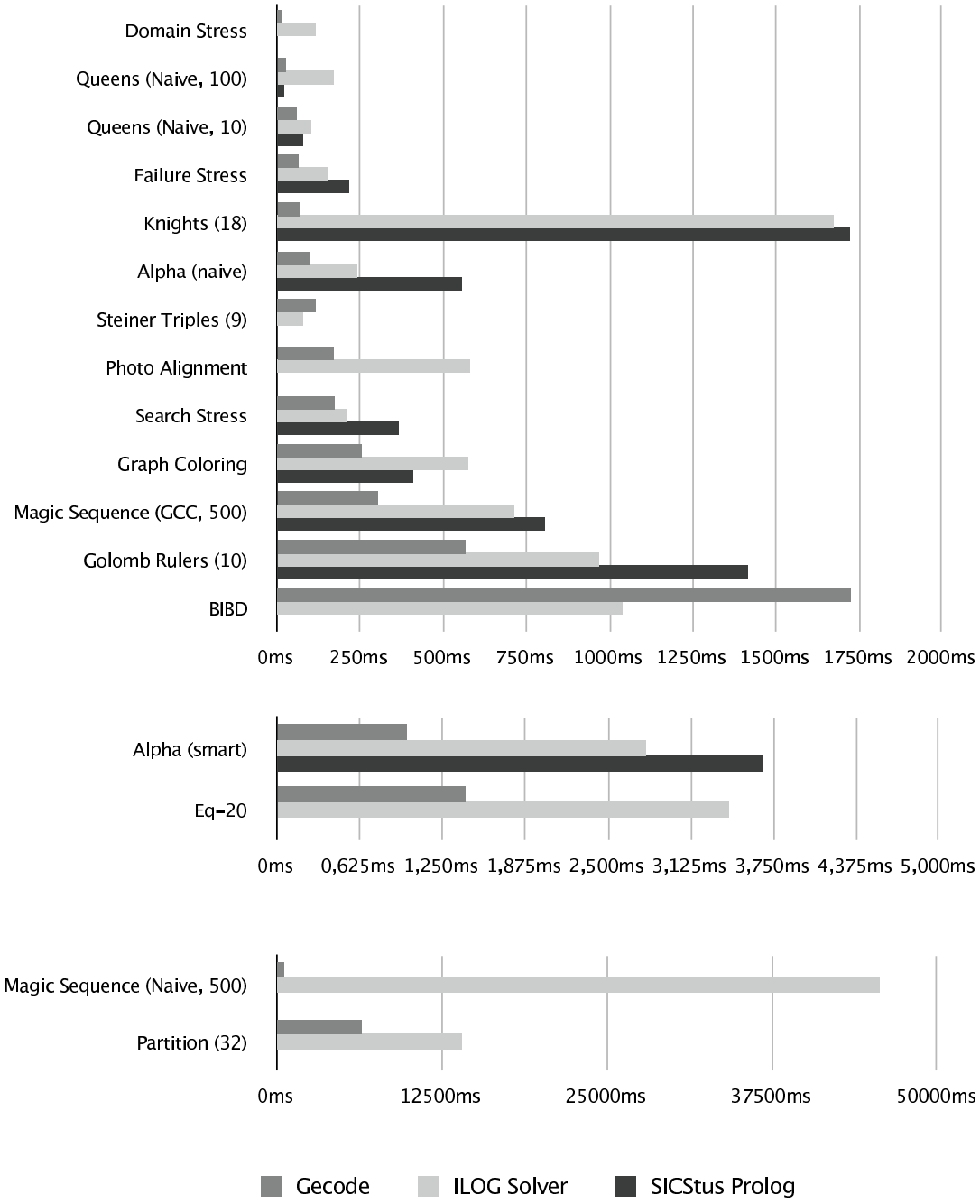


Figure A.1: Gecode versus ILOG Solver versus SICStus Prolog

Benchmark	*Gecode*	*ILOG Solver 6.5*	*SICStus Prolog 4.0.2*
	time (ms)	*time %*	*time %*
BIBD	1 726.00	60.34	—
Alpha (smart)	0.98	285.51	373.80
Alpha (naive)	99.00	247.78	562.20
Knights (18)	69.72	2 402.51	2 474.44
Golomb Rulers (10)	570.00	169.65	248.30
Queens (Naive, 10)	63.45	168.12	124.55
Queens (Naive, 100)	27.77	616.92	92.40
Eq-20	1.43	238.56	—
Graph Coloring	256.00	225.45	160.98
Magic Sequence (Naive, 500)	599.50	7 622.69	—
Magic Sequence (GCC, 500)	305.38	234.75	263.70
Photo Alignment	171.60	340.12	—
Partition (32)	6 487.50	216.99	—
Steiner Triples (9)	120.78	67.81	—
Search Stress	178.85	119.37	205.50
Failure Stress	67.05	227.03	327.32
Domain Stress	17.39	694.19	—

Table A.2: Gecode versus ILOG Solver versus SICStus Prolog

Bibliography

Abrahams, David, Jeremy Siek, and Thomas Witt. 2009.
The Boost.Iterator library.
URL: http://www.boost.org/doc/libs/1_37_0/libs/iterator/doc/.
Cited on page 135.

Ågren, Magnus, Pierre Flener, and Justin Pearson. 2007.
Generic incremental algorithms for local search. *Constraints*, 12(3):293–324.
Cited on page 165.

Ait-Kaci, Hassan. 1991.
Warren's Abstract Machine: A Tutorial Reconstruction. MIT Press, Cambridge, MA, USA.
Cited on page 71.

Andersen, Henrik Reif, Tarik Hadzic, John N. Hooker, and Peter Tiedemann. 2007.
A constraint store based on multivalued decision diagrams. In Bessière (2007), pages 118–132.
Cited on page 46.

Apt, Krzysztof R. 2003.
Principles of Constraint Programming. Cambridge University Press.
Cited on pages 2 and 45.

Apt, Krzysztof R., and Mark Wallace. 2007.
Constraint Logic Programming using Eclipse. Cambridge University Press.
Cited on page 2.

Azevedo, Francisco. 2007.
Cardinal: A finite sets constraint solver. *Constraints*, 12(1):93–129.
Cited on pages 46 and 164.

B-Prolog. 2009.
URL: http://www.probp.com/.
Cited on page 65.

Baptiste, Philippe. 1994.
Constraint-based scheduling: Two extensions. Master's thesis, University of Strathclyde, Glasgow, UK.
Cited on page 30.

Beeri, Catriel, Ronald Fagin, David Maier, and Mihalis Yannakakis. 1983.
On the desirability of acyclic database schemes. *Journal of the ACM*, 30(3):479–513.
Cited on page 114.

Benhamou, Frédéric. 1996.
Heterogeneous Constraint Solving. In Michael Hanus and Mario Rodríguez-Artalejo, *ALP*, volume 1139 of *LNCS*, pages 62–76. Springer.
Cited on pages 32, 33, and 45.

Benhamou, Frédéric, and Laurent Granvilliers. 2006.
Continuous and interval constraints. In Rossi et al. (2006), chapter 16, pages 571–604.
Cited on page 74.

Benhamou, Frédéric, David A. McAllester, and Pascal Van Hentenryck. 1994.
CLP(intervals) revisited. In *ILPS '94: Proceedings of the 1994 International Symposium on Logic programming*, pages 124–138, Cambridge, MA, USA. MIT Press.
Cited on pages 23 and 32.

Bessière, Christian, editor. 2007.
Principles and Practice of Constraint Programming - 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings, volume 4741 of *LNCS*. Springer.
Cited on pages 181, 188, and 190.

Bessière, Christian, Emmanuel Hebrard, Brahim Hnich, and Toby Walsh. 2004.
Disjoint, partition and intersection constraints for set and multiset variables. In Wallace (2004), pages 138–152.
Cited on pages 164 and 172.

Bryant, Randal E. 1986.
Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691.
Cited on page 160.

Carlson, Björn. 1995.
Compiling and Executing Finite Domain Constraints. PhD thesis, Uppsala University, Uppsala, Sweden.
Cited on pages 32 and 164.

Carlson, Björn, Mats Carlsson, and Daniel Diaz. 1994a.
Entailment of finite domain constraints. In Van Hentenryck (1994), pages 339–353.
Cited on pages 57 and 164.

Carlson, Björn, Seif Haridi, and Sverker Janson. 1994b.
AKL(FD) - a concurrent language for FD programming. In Maurice Bruynooghe, *Logic Programming, Proceedings of the 1994 International Symposium, November 13-17, 1994, Ithaca, NY, USA*, pages 521–535. MIT Press.
Cited on page 71.

Carlsson, Mats, Greger Ottosson, and Björn Carlson. 1997.
An open-ended finite domain constraint solver. In Hugh Glaser, Pieter H. Hartel, and Herbert Kuchen, *Programming Languages: Implementations, Logics, and Programs, 9th International Symposium, PLILP'97, Including a Special Track on Declarative Programming Languages in Education, Southampton, UK, September 3-5, 1997, Proceedings*, volume 1292 of *LNCS*, pages 191–206. Springer.
Cited on pages 65, 114, and 164.

Caseau, Yves, and François Laburthe. 1994.
Improved CLP scheduling with task intervals. In Van Hentenryck (1994), pages 369–383.
Cited on page 30.

Caseau, Yves, and François Laburthe. 1996.
Introduction to the CLAIRE programming language. Technical Report LIENS 96–9, École Normale Supérieure.
Cited on page 74.

Cheney, C. J. 1970.
A nonrecursive list compacting algorithm. *Communications of the ACM*, 13(11): 677–678.
Cited on page 92.

CHOCO. 2009.
URL: http://choco-solver.net.
Cited on pages 65, 74, and 135.

Choi, Chiu Wo, Martin Henz, and Ka Boon Ng. 2001.
Components for state restoration in tree search. In Walsh (2001).
Cited on page 71.

Choi, Chiu Wo, Warwick Harvey, Jimmy Ho-Man Lee, and Peter J. Stuckey. 2004.
Finite domain bounds consistency revisited.
URL: http://arxiv.org/abs/cs/0412021.
Cited on pages 35, 43, and 118.

Choi, Chiu Wo, Warwick Harvey, Jimmy Ho-Man Lee, and Peter J. Stuckey. 2006.
Finite domain bounds consistency revisited. In Abdul Sattar and Byeong-Ho Kang, *AI 2006: Advances in Artificial Intelligence*, volume 4304 of *LNCS*, pages 49–58. Springer.
Cited on pages 41 and 42.

Codognet, Philippe, and Daniel Diaz. 1996.
Compiling constraints in clp(FD). *Journal of Logic Programming*, 27(3):185–226.
Cited on page 164.

Cormen, Thomas M., Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2001.
Introduction to Algorithms. MIT Press, 2nd ed. edition.
Cited on page 85.

Davis, Martin, and Hilary Putnam. 1960.
A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215.
Cited on page 27.

Davis, Martin, George Logemann, and Donald Loveland. 1962.
A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397.
Cited on pages 27, 66, and 71.

Dechter, Rina. 2003.
Constraint Processing. Morgan Kaufmann.
Cited on page 2.

Dechter, Rina, and Judea Pearl. 1987.
Network-based heuristics for constraint-satisfaction problems. *Artificial Intelligence*, 34(1):1–38.
Cited on page 22.

Dincbas, Mehmet, Pascal Van Hentenryck, Helmut Simonis, Abderrahmane Aggoun, Thomas Graf, and Françoise Berthier. 1988.
The constraint logic programming language CHIP. In Institute for New Generation Computer Technology (ICOT), *Fifth Generation Computer Systems, Proceedings of the International Conference on Fifth Generation Computer Systems, Tokyo, Japan, November 28-December 2, 1988*, pages 693–702. OHMSHA Ltd. Tokyo and Springer.
Cited on page 22.

Dooms, Grégoire. 2006.
The CP(Graph) computation domain for constraint programming. PhD thesis, Université catholique de Louvain, Belgium.
Cited on pages 74 and 93.

Dooms, Grégoire, Yves Deville, and Pierre Dupont. 2005.
CP(Graph): Introducing a graph computation domain in constraint programming. In Peter van Beek, *Principles and Practice of Constraint Programming - 11th International Conference, CP 2005, Sitges, Spain, October 1-5, 2005, Proceedings*, volume 3709 of *LNCS*, pages 211–225. Springer.
Cited on pages 74 and 93.

Duchier, Denys. 1999.
Set constraints in computational linguistics – solving tree descriptions. In *Workshop on Declarative Programming with Sets (DPS'99)*, pages 91–98.
Cited on page 1.

ECL^{i}PS^e. 2009.
URL: http://www.eclipse-clp.org/.
Cited on pages 65 and 135.

Eén, Niklas, and Niklas Sörensson. 2004.
An extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, volume 2919 of *LNCS*, pages 502–518. Springer.
Cited on pages 74 and 100.

Euler, Leonhard. 1849.
De quadratis magicis. *Commentationes arithmeticae*, 2:593–602.
Cited on page 8.

Freuder, Eugene C. 1982.
A sufficient condition for backtrack-free search. *Journal of the ACM*, 29(1):24–32.
Cited on page 22.

Frühwirth, Thom W. 1998.
Theory and practice of constraint handling rules. *Journal of Logic Programming*, 37(1-3):95–138.
Cited on page 32.

Frühwirth, Thom W., and Slim Abdennadher. 2003.
Essentials of Constraint Programming. Springer.
Cited on page 2.

Gaschnig, John. 1974.
A constraint satisfaction method for inference learning. In *Proceedings of the Twelfth Annual Allerton Conference on Circuit and System Theory*, pages 866–874. University of Illinois.
Cited on page 27.

Gecode. 2009.
Generic constraint development environment.
URL: http://www.gecode.org.
Cited on pages 4, 67, and 92.

Gent, Ian P., and Toby Walsh. 1999.
CSPLib: a benchmark library for constraints. Technical report, APES-09-1999.
URL: http://www.dcs.st-and.ac.uk/~apes/apesreports.html.
Cited on page 173.

Gent, Ian P., Christopher Jefferson, and Ian Miguel. 2006a.
Minion: A fast scalable constraint solver. In Gerhard Brewka, Silvia Coradeschi, Anna Perini, and Paolo Traverso, *ECAI 2006, 17th European Conference on Artificial Intelligence, August 29 - September 1, 2006, Riva del Garda, Italy, Including Prestigious Applications of Intelligent Systems (PAIS 2006), Proceedings*, pages 98–102. IOS Press.
Cited on page 74.

Gent, Ian P., Christopher Jefferson, and Ian Miguel. 2006b.
Watched literals for constraint propagation in Minion. In Frédéric Benhamou, *Principles and Practice of Constraint Programming - CP 2006, 12th International Conference, CP 2006, Nantes, France, September 25-29, 2006, Proceedings*, volume 4204 of *LNCS*, pages 182–197. Springer.
Cited on pages 66 and 116.

Gent, Ian P., Karen E. Petrie, and Jean-François Puget. 2006c.
Symmetry in constraint programming. In Rossi et al. (2006), chapter 10.
Cited on page 12.

Gervet, Carmen. 1994.
Conjunto: Constraint logic programming with finite set domains. In *ILPS '94: Proceedings of the 1994 International Symposium on Logic programming*, pages 339–358. MIT Press.
Cited on pages 12 and 163.

Gervet, Carmen. 1995.
Finite Set Constraints. PhD thesis, L'Université de Franche-Comté, Besançon, France.
Cited on pages 12, 36, and 163.

Gervet, Carmen. 1997.
Interval propagation to reason about sets: Definition and implementation of a practical language. *Constraints*, 1(3):191–244.
Cited on pages 36 and 163.

Gervet, Carmen. 2006.
Constraints over structured domains. In Rossi et al. (2006), chapter 17, pages 605–638.
Cited on page 163.

Gervet, Carmen, and Pascal Van Hentenryck. 2006.
Length-lex ordering for set CSPs. In *Proceedings, The Twenty-First National Conference on Artificial Intelligence and the Eighteenth Innovative Applications of Artificial Intelligence Conference, July 16-20, 2006, Boston, Massachusetts, USA*. AAAI Press.
Cited on page 46.

Golomb, Solomon W., and Leonard D. Baumert. 1965.
Backtrack programming. *Journal of the ACM*, 12(4):516–524.
Cited on pages 27 and 52.

Gosling, James, Bill Joy, Guy Steele, and Gilad Bracha. 2005.
The Java Language Specification. Addison-Wesley Professional, 3rd edition.
Cited on page 124.

Graham, Ronald L., Martin Grötschel, and László Lovász, editors. 1995.
Handbook of combinatorics, volume 2. MIT Press, Cambridge, MA, USA.
Cited on page 10.

Haralick, Robert M., and Gordon L. Elliott. 1980.
Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14(3):263–313.
Cited on pages 27 and 52.

Hawkins, Peter, Vitaly Lagoon, and Peter J. Stuckey. 2005.
Solving set constraint satisfaction problems using ROBDDs. *Journal of Artificial Intelligence Research*, 24:109–156.
Cited on pages 46, 120, and 163.

Heintze, Nevin, and Joxan Jaffar. 1990.
A decision procedure for a class of set constraints (extended abstract). In *Proceedings, Fifth Annual IEEE Symposium on Logic in Computer Science, 4-7 June 1990, Philadelphia, Pennsylvania, USA*, pages 42–51.
Cited on page 163.

Hooker, John N. 2007.
Integrated Methods for Optimization. Springer.
Cited on page 43.

Hoos, Holger H., and Thomas Stützle. 2000.
SATLIB: An online resource for research on SAT. In Ian P. Gent, Hans van Maaren, and Toby Walsh, *SAT 2000, Highlights of Satisfiability Research in the Year 2000*, pages 283–292. IOS Press.
URL: http://www.satlib.org/.
Cited on page 175.

Horstmann, Cay S., and Gary Cornell. 2004.
Core Java 2, volume 2 – Advanced Features. Prentice Hall, 7 edition.
Cited on page 135.

ILOG Solver. 2009.
ILOG Solver, part of ILOG CP.
URL: http://www.ilog.com/products/cp.
Cited on pages 65, 73, 93, and 135.

Janson, Sverker. 1994.
AKL—A Multiparadigm Programming Language. PhD thesis, Uppsala Theses in Computing Science 19, Uppsala, Sweden.
Cited on page 71.

Janson, Sverker, and Seif Haridi. 1991.
Programming paradigms of the Andorra Kernel Language. In Vijay A. Saraswat and Kazunori Ueda, *Logic Programming: Proceedings of the 1991 International Symposium, San Diego, CA, USA, October 1991*, pages 167–186. MIT Press.
Cited on page 71.

Jones, Richard, and Rafael D. Lins. 1996.
Garbage Collection : Algorithms for Automatic Dynamic Memory Management. John Wiley & Sons.
Cited on page 92.

Kiziltan, Zeynep, and Toby Walsh. 2002.
Constraint programming with multisets. In *Proceedings of the 2nd International Workshop on Symmetry in Constraint Satisfaction Problems (SymCon-02)*.
Cited on page 74.

Laburthe, François. 2000.
Choco: Implementing a CP kernel. In Nicolas Beldiceanu, Warwick Harvey, Martin Henz, François Laburthe, Eric Monfroy, Tobias Müller, Laurent Perron, and Christian Schulte, *Proceedings of TRICS: Techniques foR Implementing Constraint programming Systems, a post-conference workshop of CP 2000*, pages 71–85.
Cited on pages 65, 72, and 74.

Lagerkvist, Mikael Z., and Christian Schulte. 2007.
Advisors for incremental propagation. In Bessière (2007), pages 409–422.
Cited on pages 66 and 81.

Laurière, Jean-Louis. 1978.
A language and a program for stating and solving combinatorial problems. *Artificial Intelligence*, 10(1):29–127.
Cited on page 163.

Leconte, Michel. 1996.
A bounds-based reduction scheme for constraints of difference. In *Constraint-96, Second International Workshop on Constraint-Based Reasoning*, pages 19–28.
Cited on page 42.

Mackworth, Alan. 1977.
Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118.
Cited on pages 22, 45, and 114.

Maher, Michael J. 2002.
Propagation completeness of reactive constraints. In Peter J. Stuckey, *Logic Programming, 18th International Conference, ICLP 2002, Copenhagen, Denmark, July 29 - August 1, 2002, Proceedings*, volume 2401 of *LNCS*, pages 148–162. Springer.
Cited on pages 23 and 45.

Mann, Martin, Guido Tack, and Sebastian Will. 2008.
Decomposition during search for propagation-based constraint solvers. Technical report.
URL: http://arxiv.org/abs/0712.2389.
Cited on page 84.

Marques-Silva, João P., and Karem A. Sakallah. 1996.
Grasp – a new search algorithm for satisfiability. In *ICCAD '96: Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*, pages 220–227, Washington, DC, USA. IEEE Computer Society.
Cited on page 71.

Marriott, Kim, and Peter J. Stuckey. 1998.
Programming with Constraints, An Introduction. MIT Press.
Cited on page 2.

Mehlhorn, Kurt, and Stefan Näher. 1999.
LEDA - A platform for combinatorial and geometric computing. Cambridge University Press.
Cited on pages 85 and 86.

Milner, Robin, Mads Tofte, and David MacQueen. 1997.
The Definition of Standard ML. MIT Press, Cambridge, MA, USA.
Cited on page 124.

Minion. 2009.
URL: http://minion.sourceforge.net/.
Cited on pages 65 and 74.

MiniSat. 2009.
URL: http://minisat.se/.
Cited on pages 74 and 100.

MLTon. 2009.
MLton, a whole program optimizing compiler for Standard ML.
URL: http://mlton.org/.
Cited on page 125.

Montanari, Ugo. 1974.
Networks of constraints: Fundamental properties and applications to picture processing. *Information Sciences*, 7:95 – 132. ISSN 0020-0255.
Cited on page 22.

Moskewicz, Matthew W., Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. 2001.
Chaff: engineering an efficient SAT solver. In *DAC '01: Proceedings of the 38th conference on Design automation*, pages 530–535, New York, NY, USA. ACM Press.
Cited on pages 57 and 66.

Mozart. 2009.
The Mozart programming system.
URL: http://www.mozart-oz.org.
Cited on pages 32, 65, 70, 71, 74, 92, and 135.

Müller, Tobias. 2001.
Constraint Propagation in Mozart. Doctoral dissertation, Universität des Saarlandes, Naturwissenschaftlich-Technische Fakultät I, Fachrichtung Informatik, Saarbrücken, Germany.
Cited on pages 32, 65, 70, 74, and 164.

Nethercote, Nicholas, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. 2007.
Minizinc: Towards a standard CP modelling language. In Bessière (2007), pages 529–543.
Cited on page 93.

Perron, Laurent. 1999.
Search procedures and parallelism in constraint programming. In Joxan Jaffar, *Principles and Practice of Constraint Programming - CP'99, 5th International Conference, Alexandria, Virginia, USA, October 11-14, 1999, Proceedings*, volume 1713 of *LNCS*, pages 346–360. Springer.
Cited on page 71.

Pesant, Gilles. 2004.
A regular language membership constraint for finite sequences of variables. In Wallace (2004), pages 482–495.
Cited on page 141.

Peyton Jones, Simon L. 2003.
Haskell 98. *Journal of Functional Programming*, 13(1).
Cited on page 124.

Prawitz, Dag. 1960.
An improved proof procedure. *Theoria*, 26:102–139.
Cited on page 26.

Puget, Jean-François. 1992.
PECOS: A high level constraint programming language. In *Proceedings of the first Singapore international conference on Intelligent Systems (SPICIS)*, pages 137–142.
Cited on pages 12, 36, 46, and 163.

Puget, Jean-Francois. 1998.
A fast algorithm for the bound consistency of alldiff constraints. In *AAAI '98/IAAI '98: Proceedings of the fifteenth national/tenth conference on Artificial intelligence/Innovative applications of artificial intelligence*, pages 359–366, Menlo Park, CA, USA. American Association for Artificial Intelligence.
Cited on pages 36 and 39.

Puget, Jean-François, and Michel Leconte. 1995.
Beyond the glass box: Constraints as objects. In John Lloyd, *Proceedings of the International Symposium on Logic Programming*, pages 513–527. MIT Press.
Cited on pages 72 and 73.

Quimper, Claude-Guy. 2006.
Efficient Propagators for Global Constraints. PhD thesis, University of Waterloo, Canada.
Cited on pages 41 and 42.

Quimper, Claude-Guy, Alejandro López-Ortiz, Peter van Beek, and Alexander Golynski. 2004.
Improved algorithms for the global cardinality constraint. In Wallace (2004), pages 542–556.
Cited on page 42.

Régin, Jean-Charles. 1994.
A filtering algorithm for constraints of difference in CSPs. In *AAAI '94: Proceedings of the twelfth national conference on Artificial intelligence (vol. 1)*, pages 362–367, Menlo Park, CA, USA. American Association for Artificial Intelligence.
Cited on pages 36, 51, 69, and 140.

Reischuk, Raphael. 2008.
Reconciling copying and trailing for constraint solvers. Bachelor's thesis, Saarland University, Saarbrücken.
Cited on page 172.

Reynolds, John C. 1983.
Types, abstraction and parametric polymorphism. In R. E. A. Mason, *Information Processing 83, Proceedings of the IFIP 9th World Computer Congress, Paris, France, September 19-23, 1983*, pages 513–523. North-Holland/IFIP.
Cited on page 124.

Rossi, Francesca, Peter van Beek, and Toby Walsh, editors. 2006.
Handbook of Constraint Programming. Foundations of Artificial Intelligence. Elsevier.
Cited on pages 1, 2, 182, and 186.

Sabin, Daniel, and Eugene C. Freuder. 1994.
Contradicting conventional wisdom in constraint satisfaction. In Alan Borning, *Principles and Practice of Constraint Programming, Second International Workshop, PPCP'94, Rosario, Orcas Island, Washington, USA, May 2-4, 1994, Proceedings*, volume 874 of *LNCS*, pages 10–20. Springer.
Cited on page 27.

Sadler, Andrew, and Carmen Gervet. 2004.
Hybrid set domains to strengthen constraint propagation and reduce symmetries. In Wallace (2004), pages 604–618.
Cited on page 46.

Saraswat, Vijay A., Martin C. Rinard, and Prakash Panangaden. 1991.
Semantic foundations of concurrent constraint programming. In *POPL, Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages, Orlando, Florida, January 1991*, pages 333–352. ACM Press.
Cited on pages 23 and 31.

Schulte, Christian. 1999.
Comparing trailing and copying for constraint programming. In Danny De Schreye, *Proceedings of the Sixteenth International Conference on Logic Programming*, pages 275–289, Las Cruces, NM, USA. MIT Press.
Cited on page 71.

Schulte, Christian. 2002.
Programming Constraint Services, volume 2302 of *LNCS (LNAI)*. Springer.
Cited on pages 32, 71, 74, and 92.

Schulte, Christian, and Peter J. Stuckey. 2004.
Speeding up constraint propagation. In Wallace (2004), pages 619–633.
Cited on pages 59 and 61.

Schulte, Christian, and Peter J. Stuckey. 2005.
When do bounds and domain propagation lead to the same search space? *Transactions on Programming Languages and Systems*, 27(3):388–425.
Cited on page 46.

Schulte, Christian, and Peter J. Stuckey. 2008a.
Dynamic analysis of bounds versus domain propagation. In Maria Garcia de la Banda and Enrico Pontelli, *Logic Programming, Proceedings of the Twenty Fourth International Conference on Logic Programming, December 9-13, 2008, Udine, Italy*, volume 5366 of *LNCS*, pages 332–346. Springer.
Cited on pages 46 and 84.

Schulte, Christian, and Peter J. Stuckey. 2008b.
Dynamic variable elimination during propagation solving. In Sergio Antoy and Elvira Alberts, *Proceedings of the 10th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 15-17, 2008, Valencia, Spain*, pages 247–257. ACM.
Cited on page 84.

Schulte, Christian, and Peter J. Stuckey. 2008c.
Efficient constraint propagation engines. *Transactions on Programming Languages and Systems*, 31(1):2:1–2:43.
Cited on pages 52, 59, 61, and 66.

SICStus Prolog. 2009.
URL: http://www.sics.se/sicstus/.
Cited on pages 65, 93, and 135.

Smith, Barbara M., Karen E. Petrie, and Ian P. Gent. 2004.
Models and symmetry breaking for 'peaceable armies of queens'. In Jean-Charles Régin and Michel Rueher, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, First International Conference, CPAIOR 2004, Nice, France, April 20-22, 2004, Proceedings*, volume 3011 of *LNCS*, pages 271–286. Springer.
Cited on page 175.

Smolka, Gert. 1995.
The Oz programming model. In Jan van Leeuwen, *Computer Science Today*, volume 1000 of *LNCS*, pages 324–343. Springer.
Cited on page 23.

Stroustrup, Bjarne. 1997.
The C++ Programming Language. Addison-Wesley, 3rd edition.
Cited on page 135.

Tarski, Alfred. 1930.
Fundamentale Begriffe der Methodologie der deduktiven Wissenschaften. I. *Monatshefte für Mathematik*, 37(1):361–404.
Cited on page 31.

Tarski, Alfred. 1983.
Logic, semantics, metamathematics, chapter V, pages 60–109. Hackett Publishing Company, 2nd ed. edition.
Cited on page 31.

van Beek, Peter, and Kent D. Wilken. 2001.
Fast optimal instruction scheduling for single-issue processors with arbitrary latencies. In Walsh (2001), pages 625–639.
Cited on page 1.

Van Hentenryck, Pascal. 1989.
Constraint satisfaction in logic programming. MIT Press, Cambridge, MA, USA.
Cited on page 22.

Van Hentenryck, Pascal, editor. 1994.
Logic Programming, Proceedings of the Eleventh International Conference on Logic Programming, June 13-18, 1994, Santa Margherita Ligure, Italy, Cambridge, MA, USA. MIT Press.
Cited on pages 182 and 183.

Van Hentenryck, Pascal, Vijay A. Saraswat, and Yves Deville. 1991.
Constraint processing in cc(FD). Technical report, Brown University.
Cited on pages 32 and 164.

Van Hentenryck, Pascal, Vijay A. Saraswat, and Yves Deville. 1998.
Design, implementation, and evaluation of the constraint language cc(FD). *Journal of Logic Programming*, 37(1-3):293–316.
Cited on pages 114 and 164.

Wallace, Mark, editor. 2004.
Principles and Practice of Constraint Programming - 10th International Conference, CP 2004, Toronto, Canada, September 27 - October 1, 2004, Proceedings, volume 3258 of *LNCS*. Springer.
Cited on pages 182, 190, 191, and 192.

Wallace, Mark, Stefano Novello, and Joachim Schimpf. 1997.
Eclipse: A platform for constraint logic programming. Technical report, IC Parc, Imperial College, London.
Cited on page 65.

Walsh, Toby, editor. 2001.
Principles and Practice of Constraint Programming - 7th International Conference, CP 2001, Paphos, Cyprus, November 26 - December 1, 2001, Proceedings, volume 2239 of *LNCS*. Springer.
Cited on pages 183 and 193.

Ward, Morgan. 1942.
The closure operators of a lattice. *Annals of Mathematics*, 43(2):191–196.
Cited on page 31.

Warren, David H. D. 1983.
An abstract Prolog instruction set. Technical Report 309, SRI International, Menlo Park, CA, USA.
Cited on page 71.

Whitesitt, J. Eldon. 1995.
Boolean Algebra and its Applications. Dover Publications.
Cited on page 147.

Yuan, Jun, Carl Pixley, and Adnan Aziz. 2006.
Constraint-Based Verification. Springer.
Cited on page 1.

Zhou, Neng-Fa. 2006.
Programming finite-domain constraint propagators in action rules. *Theory and Practice of Logic Programming*, 6:483–507.
Cited on page 65.