Composing Constraint Solvers

IPA Dissertation Series 2005-18

The work reported in this thesis has been carried out at the Center for Mathematics and Computer Science (CWI) in Amsterdam under the auspices of the research school IPA (Institute for Programming research and Algorithmics), and was supported by NWO, the Netherlands Organization for Scientific Research under project number 612.069.003.

Composing Constraint Solvers

Academisch Proefschrift

ter verkrijging van de graad van doctor aan de
Universiteit van Amsterdam
op gezag van de Rector Magnificus
prof. mr. P.F. van der Heijden
ten overstaan van een door het college voor
promoties ingestelde commissie, in het openbaar
te verdedigen in de Aula der Universiteit
op dinsdag 29 november 2005, te 14.00 uur

door

Peter Zoeteweij

geboren te Rotterdam.

Cover design by Robert Zoeteweij.
Printed and bound by PrintPartners Ipskamp, Enschede.

ISBN: 90–6196–533–0

Contents

Acknowledgments

Staat de wereld dan stil? Broeien de draken dan geen draakjes meer uit in de foreesten van het Land van Logres? Berijden er dan geen belaagde jonkvrouwen meer witte palafroeten door diezelfde foreesten? Moet keytievige ondeugd dan niet worden gefnuikt en zijn er geen interessante queste's meer te volbrengen?

Louis Couperus, in *Het Zwevende Schaakbord.*

This thesis describes my research in the four years that I had the privilege to be working at CWI, *het Centrum voor Wiskunde en Informatica* in Amsterdam. Soon after I started, I was told that doing research in computer science involves building your own castle. Since these words came from Kees Everaars, who is among the wisest of my CWI colleagues, I considered them with care, and concluded that it meant that first of all, some software had to be written.

The castle on the cover is drawn after Figure 3.1 on page 36, and depicts the software that I wrote to perform my experiments: my castle, OpenSolver. The creature sitting on top of Castle OpenSolver is a dragon. Instead of its more widespread use as a metaphor for the complexity of compiler design, the dragon illustrates that I have always thought of my PhD work as a personal quest: it is well known that in addition to castles, quests typically involve dangerous activities, such as conquering dragons. I like to think that I would have preferred to go on a real quest, build a real castle, and conquer a real dragon, but as is evident from the above quotation, dragons have become scarce long ago, so pursuing a PhD was the next best thing for me to do.

As things go on quests, I received help and friendship from many people on my way. At CWI, my colleagues from the group of Jan Rutten provided much of the social context of the work reported here. I was very lucky to have their everyday company, and I am particularly grateful to Kees Blom, Freek Burger, and Kees Everaars for help with technical issues. Dave Clarke played an important part (guitar, to be precise) in dragging me through the last months of writing up, and it was a great pleasure to share the office with Juan Guillen Scholten, who

often helped me to see matters of computer science, and life in general, in proper perspective.

During my time at CWI, three other PhD students were finishing a thesis in the area of constraints. They are Rosella Gennari, Sebastian Brand, and Willem Jan van Hoeve, and it was great to have them around for discussions on the subject, and to be able to follow their lead in going through the process of producing a PhD thesis. However, for some reason it seems to me that we spent more time together traveling to the CP conferences and CSCLP workshops than we did at CWI. No quest should be without faithful travel companions, and I want to thank them especially for their pleasant company on these occasions.

I am also very grateful to Eric Monfroy, Frédéric Goualard, and Laurent Granvilliers for discussions, advise, and software, and for facilitating a visit to Nantes in February 2004. While we have not met often, I have also greatly benefited from the few encouraging discussions I had with Maarten van Emden, and from his comments on an earlier version of Chapter 5.

Furthermore, I would like to thank everyone who has read parts, or even all of the material assembled in this thesis. Their comments have been of considerable influence on the quality of the final version.

Of course, none of this would have been possible without the support of my advisors, Krzysztof Apt and Farhad Arbab. I joined CWI only after a 4 year detour from the academic world, and I am most grateful for their confidence, encouragement, and advise. In particular, I want to thank Krzysztof for always looking after his student, even though most of the time we were separated by six or seven time zones, and Farhad for always being there to counter my skepticism with vision and optimism. And I thank both for letting me use the material of our joint papers.

Finally, before my story of knights, queens, and castles begins, I want to thank Rob for his work on the cover, and Marjan for her support, and for her patience with me, especially during the last few months of my quest, which has now finally come to an end.

Rotterdam, October 2005 Peter Zoeteweij

Chapter 1

Introduction

The subject of this thesis can be categorized as the implementation of constraint solvers. In this chapter we put constraint solving, and the programming style that it enables into context. We motivate our work, give an outline of the thesis, and summarize the scientific contributions.

1.1 Context

Constraint programming falls in the category of ***declarative*** programming styles, where essentially a program describes ***what*** must be computed, as opposed to ***imperative*** programming, where a program describes ***how*** the output must be computed. A constraint "program" is called a ***constraint satisfaction problem*** (CSP), and consists of the following elements.

- a set of variables, each with a set of allowed values, and
- a set of constraints, where every constraint applies to a subset of the variables, and restricts somehow the values that these variables may assume.

Writing a CSP to represent a combinatorial problem that occurs in practice is called ***modeling***. The purpose of ***constraint solving*** is to generate ***solutions*** to CSPs. These are assignments of values to variables that satisfy all constraints. We will also consider a modification of constraint satisfaction problems where the goal is to generate an optimal solution, according to some objective function. The combination of a CSP and an objective function is called a ***constrained optimization problem*** (COP). Programs, procedures, and algorithms for constraint solving are called ***constraint solvers***. For a tutorial, and textbooks on constraint programming, the reader is referred to [Smi95, Apt03, Dec03]. The subject of this thesis is the implementation of constraint solvers.

Because of the very general nature of a CSP, constraint programming subsumes several other forms of modeling, such as linear programming. For some specific forms of CSPs and COPs there exist very efficient solving methods, such as the simplex algorithm for linear programs. In this thesis we deal with general methods for constraint solving that are applied only when no efficient, problem-specific methods are available.

In particular, we deal with methods that are based on an exhaustive exploration of all possible assignments of values to variables. This excludes so-called ***local search*** methods, which start from an initial assignment, and try to improve it according to some notion of quality by iteratively making small (local) modifications. Local search methods naturally apply to COPs, but they can also be applied to CSPs by minimizing the number of violated constraints. Local search methods typically find good solutions quickly, but are not guaranteed to find the optimum, and therefore they cannot determine whether a solution to a CSP exists or not.

In contrast, the exhaustive methods that we consider in this thesis are guaranteed to find a solution if it exists, but in general they have a time complexity that is exponential in the size of the problem. Although computations with an exponential time complexity are considered to be intractable, this characterizes only the ***worst-case*** behavior, and despite their potentially intractable nature, such methods are successfully applied in practice. Examples of such applications are the generation of test patterns for digital circuits (see, e.g., [VHSD92]), the analysis of nonlinear functions [HMD97], and scheduling problems [BLPN01], such as sports tournament scheduling [Hen01].

1.2 Motivation

Constraint solving is based on a collection of largely independent techniques, that fall into two categories: search methods, and techniques for reducing the search space. In the context of constraint solving, techniques in the latter category are usually called ***constraint propagation*** techniques, and the approach to constraint solving that is considered in this thesis is known as ***branch-and-propagate search***. To a large extent, constraint programming consists of determining the combination of techniques that make a given CSP solvable. Many of these techniques have successfully been used in specific domains to build practical constraint programming tools, but in general it is not possible to combine them without reprogramming or re-engineering the tools, and a major challenge in this field is how to achieve a combination of various existing methods and techniques within a single framework. We will refer to realizing such combinations as ***solver composition***.

Recently, it was demonstrated that many constraint propagation algorithms proposed in the literature are actually instances of a generic iteration algorithm [Apt99, Gen02]. In this scheme, a constraint solver consists of a scheduler combined with the functions that are to be scheduled. If certain conditions are met, different schedulers will lead to the same result, and the schedulers can be configured to exploit several properties of the functions. Moreover, distributed versions of the generic iteration algorithm exist [MR99, Mon00a], and distributed constraint propagation can be realized simply by substituting a sequential scheduler by a distributed scheduler.

Our goal is to exploit the conceptual simplicity of this scheme, and to complement it with facilities for search to form an integrated framework for constraint solving. The central theme of this thesis is an exploration of the possibilities and limitations of such a framework. We have taken a practical approach, which has led to the development of OpenSolver, a highly configurable constraint solving engine that supports a wide range of relevant solver configurations. A major design goal was that OpenSolver can be used as a software component, to form the core of a solver, and to participate in several solver cooperation schemes. As a result, composing constraint solvers around OpenSolver involves various methods of software composition. We give an account of the design and implementation of OpenSolver, and of the experiments that were performed to verify the efficiency of the resulting constraint solvers.

1.3 Outline of the Thesis

The thesis can be read as a description of the OpenSolver software, plus a series of case studies that demonstrate how it can be configured for various application domains and methods of constraint solving. In addition to exploring the possibilities and limitations of our approach, some of these case studies address more specific research questions. The following chapters fall naturally in three parts. The first part contains the preliminary material:

Chapter 2. Here we introduce constraint solving, and branch-and-propagate search. To motivate our work further, we review several forms of solver composition that are found in the literature.

Chapter 3. A description of the OpenSolver software, which is used in the subsequent chapters as a platform for experimenting with solver composition. Part of the material presented here was submitted to the CP 2003 doctoral program, and an abstract appeared in the conference proceedings [Zoe03c].

The second part consists of three chapters that describe applications of a single OpenSolver instance:

Chapter 4. Here we demonstrate how OpenSolver can be configured as a solver for constraints on finite domains, real, and Boolean variables. We also describe a number of general-purpose facilities that are independent of the application domain, and we investigate how a number of existing techniques that are normally hard-wired in solvers can be realized through composition.

Chapter 5. In this chapter we demonstrate how OpenSolver can be configured for solving arithmetic constraints on integer variables. Two specific questions are addressed in this chapter: first, there are a number of natural approaches to implement these constraints, and we investigate which one of them can be expected to give the best performance. Second, we try to characterize the effect of constraint propagation for these constraints. This chapter is based on joint work with Krzysztof Apt, the preliminary results of which appear in [AZ04].

Chapter 6. This chapter is a case study describing a solver for the job-shop scheduling problem, based on OpenSolver. Like Chapter 4, this involves existing solving techniques, but this material serves two other purposes. First, job-shop scheduling is considered to be a non-trivial problem whose complexity is representative of many scheduling problems that occur in practice. As such, it shows that our approach leads to realistic solver implementations. Second, it demonstrates a technique that we refer to as constraining special-purpose data types. In an open-ended solver, this technique can be applied when the existing facilities do not support an efficient implementation of a solver for a given problem. Part of the material in this chapter was submitted to the CP 2004 doctoral program, and an abstract appeared in the conference proceedings [Zoe04b].

In computer science, ***coordination*** refers to the orchestration of the interaction among the various components of a software system [Arb98]. While all meaningful computation involves coordination, it is a particularly relevant aspect of ***concurrent*** systems, where several computations overlap in time. The design of OpenSolver allows that in addition to its use as a stand-alone constraint solver, it can be coordinated from the outside in many different ways, to support solver composition at a higher level. In the last part of the thesis we look at several ways in which OpenSolver instances can cooperate to solve a single constraint satisfaction problem. The techniques described here are orthogonal to those of the three preceding chapters, and can be used in combination with them. Two of these techniques involve concurrency, and the emphasis of their presentation is on the coordination aspects.

Chapter 7. There exist constraint propagation techniques that internally involve search. This internal search process occurs in the context of another, encompassing branch-and-propagate search, and is therefore called nested search. In this chapter we propose a generic reduction operator for nested search, and investigate the extent to which three existing techniques from different application domains can be expressed as applications of this generic operator. We also describe an implementation of the operator based on an almost autonomous OpenSolver instance, and we evaluate its performance on some benchmark problems. This chapter is based on a paper [Zoe04a] that was presented at the 2004 ERCIM/CologNet workshop on constraint solving and constraint logic programming.

Chapter 8. Parallel processing is used to reduce the turn-around time by distributing the workload among a number of threads or processes running on different processors or computers. In this chapter we describe a parallel constraint solver that uses a time-out mechanism for load balancing. To our knowledge, this is a novel approach to parallel search that supports the composition of a parallel solver from autonomous component solvers. The research question addressed in this chapter is whether this approach leads to efficient and scalable parallel solvers. This chapter is based on a paper with Farhad Arbab [ZA04].

Chapter 9. Here we discuss the use of OpenSolver as a software component for implementing a solver based on distributed constraint propagation. Several researchers have recognized the need for such a solver. A possible motivation is that in some cases, the CSP that we are trying to solve is distributed, while it is impossible, or undesirable to gather all constraints in a single solver. The chapter is based on two publications [Zoe03b, Zoe03a], that continue the research of Eric Monfroy and Farhad Arbab in the area of coordination-based constraint solving [Mon00a, AM00].

In Chapter 10 we review the material in the preceding chapters, and suggest directions for future work.

1.4 Contributions

This thesis demonstrates that solver composition can lead to efficient branch-and-propagate constraint solvers. Specific contributions are the following.

- An account of the design and implementation of a general purpose constraint solving engine, with a flexible architecture that supports a wide variety of relevant solver configurations. In particular
  - It is configurable with respect to low-level aspects such as the scheduling of the functions that implement constraint propagation. This allows the ***composition*** of techniques that are normally hard-wired in constraint programming tools. Solver composition in turn leads to reuse of code, and allows that solving techniques carry over to other data types and application domains.
  - It is designed as an autonomous application that communicates with its environment through a programmable interface and a solver configuration language. This design facilitates the component-based construction of constraint solvers around the solving engine, independent

of a particular computing environment or programming language. The configuration language gives unique possibilities for external manipulation of CSPs and solver configurations, which allow that special-purpose functionality can be implemented outside the solving engine.

- A demonstration and discussion of the technique of constraining special purpose data structures as a tool to implement solvers for problems that do not have a straightforward CSP formulation.

- A systematic study of several approaches to implementing arithmetic constraints on integers, using an interval representation for the variable domains, and integer interval arithmetic to describe and implement constraint propagation. For the most promising approach, we provide results that characterize the effect of constraint propagation.

- A demonstration that several operators for enforcing so-called stronger forms of consistency, which improve the efficiency of constraint solving in specific application domains, are actually instances of the same technique: nested search. We demonstrate that using a generic reduction operator for nested search, these operators can be ***composed*** from the operators that enforce weaker forms of consistency. Experiments show that this compositional approach leads to a viable implementation of the techniques that we are interested in.

- A study of a time-out mechanism for implementing parallel search. We demonstrate that by equipping constraint solvers with a time-out mechanism, these solvers can then be used as software components for building a parallel constraint solver, resulting in a very simple implementation that performs well on shared memory and distributed memory architectures, and gives a good load-balance in practice.

In addition, we believe that the OpenSolver software itself is potentially of interest to a wider audience. However, it was developed primarily for the experiments reported in this thesis, and has not been used for other purposes. Consequently, ease of use has not been a priority, and no user's manual or programmer's manual exists. The latter would be essential for exploiting the open-ended nature of the system. If time permits, we hope to be able to continue the development of OpenSolver, and to make it available as open source software.

Chapter 2

Constraint Solving

This chapter introduces the subject of constraint solving. The results in this thesis apply to one particular approach to constraint solving, namely ***branch-and-propagate*** search. We give a precise definition of this approach, and argue that it is desirable to be able to compose constraint solvers from components for different techniques, heuristics and other aspects of constraint solving. With this, we provide the main justification for the work reported in this thesis.

2.1 Introduction

Constraint solving deals with finding solutions to ***constraint satisfaction problems*** (CSPs). It refers to the techniques that enable ***constraint programming***, a branch of declarative programming where instead of implementing an algorithm, the programmer models the problem as a CSP, and uses a ***constraint solver*** to construct a solution. Constraint solving applies to combinatorial (optimization) problems, and many examples of successful applications exist, including scheduling [BLPN01, Hen01], analysis of nonlinear functions [HMD97], and testing of digital circuits [VHSD92].

Informally, a CSP consists of a set of ***variables***, each with an associated ***domain***, plus a set of ***constraints***. The domains are sets of possible values for the variables. Each constraint is defined on a subset of the variables, and restricts somehow the combinations of values that can be assigned to these variables. Constraint solving comes down to finding an assignment of values to variables that violates none of the restrictions imposed by the constraints. Constraints appear in various forms. They can be defined by explicitly enumerating allowed or disallowed combinations of values for the variables, but in most cases, the domains have some structure, and a more compact notation, such as a mathematical equation, can be used.

This chapter is organized as follows. Section 2.2 contains the definitions related to constraint solving that we will use throughout the thesis. Section 2.3

introduces branch-and-propagate search. In Section 2.4 we illustrate the need for a configurable constraint solver.

2.2 Definitions

In this section we define what we mean by constraint solving. This involves a definition of constraint satisfaction problems that conforms largely to the standard definitions as they are used, for example, in [Apt03]. Also we will introduce two notions of ***local consistency*** of CSPs that are widely used in the literature: arc consistency and hull consistency. For modeling the solving process we introduce the notion of a ***domain type***, and we will define a number of standard domain types that will be used throughout this thesis. Domain types provide a context for the solving process. We will call the combination of a CSP and a solving context an ***extended CSP***. Such combinations are essential to the model of constraint solving that this thesis is based on.

2.2.1 Sequences and Schemes

Several definitions used throughout this thesis rely on the notion of a ***sequence***, which is an ordered multiset. We consider only finite sequences, and for a sequence of length n we use the notation $\langle e_1, \ldots, e_n \rangle$. Tuples are finite sequences that are an element of a specific Cartesian product of sets. To simplify the notation, when it does not lead to confusion we will omit the angular brackets.

An ***n-scheme*** is a subsequence of $1, 2, \ldots, n$. Given a sequence $t := e_1, \ldots, e_n$ and an n-scheme $s := i_1, \ldots, i_l$, let $t[s]$ denote the sequence $e_{i_1}, \ldots, e_{i_l}$, which is called the subsequence of t with scheme s. Sequences of length one are identified with the element that they contain, so for $1 \leq i \leq n$ we have $t[i] = \langle e_i \rangle = e_i$.

Some notation: for a sequence A of length n, a sequence B of length 1, and a binary relation symbol r, we use ArB as shorthand for $A[1]rB[1], \ldots, A[n]rB[1]$.

2.2.2 Constraint Satisfaction Problem

Consider a sequence of variables $X := x_1, \ldots, x_n$ that have respective domains $D_1, \ldots, D_n$ associated with them. By a ***constraint*** C on X we mean a subset of $D_1 \times \ldots \times D_n$. The number n is the ***arity*** of the constraint.

A ***constraint satisfaction problem*** consists of a finite sequence of variables $X := x_1, \ldots, x_n$ with respective domains $\mathcal{D} := D_1, \ldots, D_n$, together with a finite set $\mathcal{C}$ of constraints, each on a subsequence of X. The scheme of this subsequence is the scheme of the constraint. We use the following notation for CSPs.

$$\langle \mathcal{C} \ ; \ x_1 \in D_1, \ldots, x_n \in D_n \rangle. \tag{2.1}$$

Instead of explicitly specifying the set of allowable tuples, we will often use an implicit specification of constraints, such as a mathematical equation.

2.2.1. Example. Consider the following CSP.

$$\langle x < y,\ y \neq z\ ;\ x, y, z \in \{1, 2, 3\}\rangle$$

The constraint $x < y$ denotes the subset $\{\langle 1, 2\rangle, \langle 1, 3\rangle, \langle 2, 3\rangle\}$ of $D_x \times D_y$. Its scheme is the sequence $1, 2$, identifying the first two elements of $X := x, y, z$. According to the notation introduced above, $x, y, z \in \{1, 2, 3\}$ is shorthand for $x \in \{1, 2, 3\}$, $y \in \{1, 2, 3\}$, $z \in \{1, 2, 3\}$. □

By a ***solution*** to a CSP of the form (2.1) we mean an element d of $D_1 \times \ldots \times D_n$ such that for each constraint $C \in \mathcal{C}$ with scheme s we have $d[s] \in C$. We call a CSP ***consistent*** if it has a solution, and ***inconsistent*** otherwise. Two CSPs with the same sequence of variables are called ***equivalent*** if they have the same set of solutions.

Further, given a CSP of the form (2.1), a sequence $\mathcal{D}' := D'_1, \ldots, D'_n$ having $D'_i \subseteq D_i$, for $1 \leq i \leq n$, and a constraint $C \in \mathcal{C}$ with scheme $s := i_1, \ldots, i_l$, let $C[\mathcal{D}'[s]]$ denote $C \cap D'_{i_1} \times \ldots \times D'_{i_l}$, the ***projection*** of $\mathcal{D}'[s]$ on C. $\mathcal{C}[\mathcal{D}]$ denotes the set of constraints obtained by replacing every constraint C in $\mathcal{C}$ with the projection $C[\mathcal{D}'[s]]$, where s is the scheme of C. Projections of domains on constraints are needed to maintain the property that constraints are subsets of Cartesian products of domains, when transforming CSPs by modifying their domains. They are seldom needed, because we mostly use implicit constraints as in Example 2.2.1.

2.2.3 Local Consistency

In addition to the distinction between consistent and inconsistent CSPs, several other notions of consistency of CSPs are commonly used. They are called ***local consistency*** notions, and in a CSP that complies to a local consistency notion, some values that do not contribute to any solution have been removed from the domains of variables. The various local consistency notions differ in the extent to which such values are absent. It is convenient to introduce a local consistency notion at this stage.

Consider a CSP P of the form (2.1), and a binary constraint $C \in \mathcal{C}$ on variables x and y. The constraint C is called ***arc consistent*** if

- for every $a \in D_x$ there is a value $b \in D_y$ such that $\langle a, b\rangle \in C$, and
- for every $b \in D_y$ there is a value $a \in D_x$ such that $\langle a, b\rangle \in C$.

P is called arc consistent if every binary constraint in $\mathcal{C}$ is arc consistent.

The following examples demonstrate that arc consistency does not imply consistency, and vice versa. This is true for local consistency in general.

2.2.2. Example. The inconsistent CSP

$$\langle x \neq y, y \neq z \ ; \ x, y, z \in \{0, 1\}\rangle$$

is arc consistent: for both disequality constraints, all values in the domains of the variables that it applies to occur in a tuple allowed by that constraint. Conversely, The consistent CSP of Example 2.2.1 is not arc consistent: the value 3 in the domain of x does not occur in any of the tuples $\langle x, y\rangle \in \{\langle 1, 2\rangle, \langle 1, 3\rangle, \langle 2, 3\rangle\}$ allowed by the constraint $x < y$, nor does the value 1 in the domain of y. However, this CSP can be transformed into an arc consistent CSP by removing these values from their respective domains. This yields the CSP

$$\langle x < y, \ y \neq z \ ; \ x \in \{1, 2\}, y \in \{2, 3\}, z \in \{1, 2, 3\}\rangle$$

which is equivalent to the original CSP. □

Arc consistency, which was introduced by Mackworth [Mac77], applies to binary constraints only. Its generalization to arbitrary constraints is called ***hyper-arc consistency***.

2.2.4 Domain Type

We will regard constraint solving as a process that performs a series of transformations on CSPs. Example 2.2.2 already demonstrated one such transformation. In principle, these transformations may affect the set of constraints, but in most cases that we consider, the transformations change only the domains of the variables. To model this process, we introduce the notion of a ***domain type***. A domain type is the set of all domains that can possibly be associated with a particular variable during the solving process.

Ideally, for a CSP of the form (2.1) and $1 \leq i \leq n$, we would like to be able to use $\mathcal{P}(D_i)$, the set of all subsets of D_i as a domain type. For finite D_i this is possible, but the cost of maintaining the data structures to represent such domains in a constraint solver can be high, and using $\mathcal{P}(D_i)$ may not be the most efficient choice.

Moreover, if D_i is a set of real numbers, using $\mathcal{P}(D_i)$ is generally not possible because most real numbers cannot be represented inside a computer. Instead we have at our disposal a set of binary ***floating-point*** numbers. This set of floating-point numbers is a finite subset of $\mathbb{R}$, and in general the only feasible representation of a real number that is not a floating-point number is an ***interval*** that contains this number, and whose bounds are consecutive floating-point numbers of a certain precision. By ***consecutive floating-point numbers*** we mean two floating-point numbers $a < b$, such that we do not have at our disposal a floating-point number c for which $a < c < b$. Finite $D_i \subset \mathbb{R}$ can be represented as finite sets of such intervals, but in practice, the smallest interval that contains D_i is used.

Domain types provide a level of abstraction that allows us to model constraint solving on finite domains, interval domains, and other domains in a uniform way. Examples of similar notions that are used in the literature are the ***approximate domains*** of Benhamou [Ben96], the ***subdefinite extensions*** of Telerman and Ushakov [TU96], and the collections of subsets based on a domain of Monfroy [Mon00a]. Borrowing from these, we will use the following definition.

2.2.3. Definition. A ***domain type*** $\mathcal{T}$ is a set of sets that is partially ordered with respect to set inclusion, and has the following properties:

- there is a largest element, denoted by $\mathcal{T}^\top$ that is a superset of all elements,
- it contains the empty set $\emptyset$,
- it is closed under intersection, and
- set inclusion is a well-founded relation over $\mathcal{T}$. □

The last property of this definition is due to [Ben96]. As a result of it, our domain types are specific forms of ***acceptable approximate domains***, introduced in that reference. They are specific in the sense that they contain the empty set. Recall that a well-founded relation over a set $\mathcal{T}$ is a partial order relation R such that every non-empty subset of $\mathcal{T}$ has an R-minimal element. This ensures that domain types do not contain any infinite decreasing sequences of elements. Because computer memory is finite, implementations of domain types will be implementations of ***finite*** domain types, and these correspond to subdefinite extensions of their largest element [TU96].

2.2.4. Example. The set consisting of $\mathbb{Z}$, and all finite subsets of $\mathbb{Z}$ is a domain type. The set of all sets of integers is not a domain type: it is a superset of the set $\{\{x \in \mathbb{Z} \mid x \geq l\} \mid l \in \mathbb{Z}\}$, which does not have a least element with respect to set inclusion. □

During the solving process, we may be able to associate a new set of allowable values with a variable. Instead of this set, we will use its representation in a particular domain type, which is defined as follows.

2.2.5. Definition. Given a domain type $\mathcal{T}$ and a set $D \subseteq \mathcal{T}^\top$, let $\mathcal{T}(D)$ denote the smallest element of $\mathcal{T}$ that is a superset of D. $\mathcal{T}(D)$ is called the ***representation*** of D in $\mathcal{T}$. □

2.2.6. Example. Let $\mathcal{T}$ denote the domain type containing the following 11 domains.

$$\begin{array}{c}
\{1,2,3,4,5,6,7\} \\
\{1,2,3,4,5\} \qquad\qquad \{3,4,5,6,7\} \\
\{1\} \quad \{1,2,3\} \quad \{3\} \quad \{3,4,5\} \quad \{5\} \quad \{5,6,7\} \quad \{7\} \\
\emptyset
\end{array}$$

The set $\{2,4\}$ is not in $\mathcal{T}$, so its representation $\mathcal{T}(\{2,4\}) = \{1,2,3,4,5\}$ is a proper superset of it. Only subsets of $\mathcal{T}^\top = \{1,2,3,4,5,6,7\}$ have a representation in $\mathcal{T}$, so $\mathcal{T}(\{7,8\})$ does not exist. □

The elements of a domain type that are representations of singleton sets play a special role in the solving process. These are called canonical domains, and are defined as follows.

2.2.7. Definition. For a domain type $\mathcal{T}$ we define

$$\lfloor \mathcal{T} \rfloor := \{\mathcal{T}(\{x\}) \mid x \in \mathcal{T}^\top\}.$$

The elements of $\lfloor \mathcal{T} \rfloor$ are called the ***canonical domains*** of $\mathcal{T}$. □

2.2.8. Example. The canonical domains of the domain type $\mathcal{T}$ of Example 2.2.6 are $\{1\}$, $\{1,2,3\}$, $\{3\}$, $\{3,4,5\}$, $\{5\}$, $\{5,6,7\}$, and $\{7\}$. □

In addition to some special purpose domain types, we will mainly be concerned with the four standard types defined below.

2.2.9. Definition. Let $\mathrm{I\!R}^\infty$ denote $\mathrm{I\!R} \cup \{-\infty, \infty\}$, the set of reals augmented with the symbols for plus and minus infinity. We define $-\infty < \infty$, and $x < \infty$ and $x > -\infty$ for all $x \in \mathrm{I\!R}$. $\mathrm{I\!F}$ denotes a finite subset of $\mathrm{I\!R}^\infty$ that contains $-\infty$ and ∞, and is used to model a set of floating-point numbers of unspecified, but fixed precision. For $a, b \in \mathrm{I\!R}^\infty$, let $[a,b]$ denote the set $\{x \in \mathrm{I\!R} \mid a \le x \le b\}$, and for two integers a and b, let $[a..b]$ denote the set $\{i \in \mathbb{Z} \mid a \le i \le b\}$.

- $\mathcal{B}$ denotes the domain type $\{\{\mathsf{true}, \mathsf{false}\}, \{\mathsf{true}\}, \{\mathsf{false}\}, \emptyset\}$, containing the domains for Boolean variables.
- $\mathcal{Z}$ denotes the domain type containing $\mathbb{Z}$ and all finite sets of integers, including $\emptyset$.
- $\mathcal{I}$ denotes the domain type containing $\mathbb{Z}$, $\emptyset$, and all intervals $[a..b]$ with $a, b \in \mathbb{Z}$ and $a \le b$. Elements of $\mathcal{I}$ are called ***integer intervals***.
- $\mathcal{F}$ denotes the domain type that consists of $\emptyset$ and all intervals $[a,b]$, where $a, b \in \mathrm{I\!F}$ and $a \le b$. Elements of $\mathcal{F}$ are called ***floating-point intervals***.

□

Variables with domain type $\mathcal{Z}$ are usually called ***finite domains*** variables. Note that $\mathrm{I\!R} \in \mathcal{F}$, and that $\mathcal{I} \subset \mathcal{Z}$, and $\lfloor \mathcal{I} \rfloor = \lfloor \mathcal{Z} \rfloor = \{\{x\} \mid x \in \mathbb{Z}\}$.

Domain type $\mathcal{F}$ is used for solving constraints on the reals. It is similar to the domain type of Example 2.2.6, in the sense that some of the domains overlap just on their bounds. The set of ***canonical intervals*** $\lfloor \mathcal{F} \rfloor$ contains a singleton set for each of the elements of $\mathrm{I\!F} - \{-\infty, \infty\}$, and an interval representation for all other real numbers. The representation of a set of real numbers is usually called the ***hull*** of this set, and to conform our notation, for $D \subseteq \mathbb{R}$ we define

$$\mathsf{hull}(D) := \mathcal{F}(D).$$

As an example, $-\frac{1}{10}, \frac{1}{10} \notin \mathbb{F}$ (no binary floating-point representation exists for these values), so $\mathsf{hull}([-\frac{1}{10}, \frac{1}{10}])$ is the interval $[a, b]$ with $a = \mathsf{max}(\{x \in \mathbb{F} \mid x < -\frac{1}{10}\})$ and $b = \mathsf{min}(\{x \in \mathbb{F} \mid x > \frac{1}{10}\})$.

Constraints on the reals are the topic of Section 4.5, but it is convenient at this point to introduce hull consistency, a local consistency notion that is specific to these constraints. This definition can be found in many publications concerning constraints on the reals, see for example [CDR99, BGGP99, BMVH94].

A constraint $C \in \mathbb{R}^n$ is ***hull consistent*** if for all $1 \leq i \leq n$,

$$D_i = \mathsf{hull}(\{x_i \in \mathbb{R} \mid \exists x_1 \in D_1, \ldots, x_{i-1} \in D_{i-1},\ x_{i+1} \in D_{i+1}, \ldots, x_n \in D_n \text{ for which } \langle x_1, \ldots, x_n \rangle \in C\}).$$

A CSP of the form (2.1) is hull consistent if all constraints $C \in \mathcal{C}$ are hull consistent.

As the following example demonstrates, hull consistency is an approximation of hyper-arc consistency that deals both with the fact that we represent domains by intervals, and with the imprecision inherent to computing with floating-point numbers.

2.2.10. Example. Let $a = \mathsf{max}(\{x \in \mathbb{F} \mid x < -\frac{1}{10}\})$ and $b = \mathsf{min}(\{x \in \mathbb{F} \mid x > \frac{1}{10}\})$. The CSP

$$\langle 100x^2 = 1 \ ; \ x \in \mathsf{hull}(\{-\tfrac{1}{10}, \tfrac{1}{10}\}) \rangle$$

is hull consistent. $x = -\frac{1}{10}$ and $x = \frac{1}{10}$ are the only solutions, but domain type $\mathcal{F}$ does not provide the means to represent the information that 0, or any of the other values in $[a, -\frac{1}{10}) \cup (-\frac{1}{10}, \frac{1}{10}) \cup (\frac{1}{10}, b]$ does not contribute to a solution. □

2.2.5 Extended CSP, Solved Form

Domain types specify what domains can be associated with the variables of a CSP during the solving process. The canonical domains and the empty set play special roles: if an equivalence preserving transformation changes the domain of a variable into the empty set, the CSP that we are trying to solve is inconsistent. Conversely, if all domains are singleton sets, while the CSP conforms to a notion of consistency that ensures that no constraint is violated, the values in these singleton sets constitute a solution to the CSP. For singleton domains, all practicable notions of local consistency have this property, but for constraints on the reals, the domain type may not support a precise representation of the solution. In this case, the best we can get is a sequence of canonical intervals for which the CSP conforms to some notion of local consistency, such as hull consistency, which generally does not imply consistency. In either case, when we reach canonical domains and local consistency, constraint solving is finished in the sense that the maximum precision allowed by the domain type has been reached. If we are not sure about consistency, other methods must be applied.

So constraint solving based on local consistency enforcing ends at canonical domains, but in some cases we are not interested in canonical domains for each of the variables. For example, a variable may have been introduced just to represent an intermediary result of a calculation, or the precision of the canonical intervals may be higher than what is needed, and CPU time can be saved by accepting domains of a lower precision. To specify such requirements, in addition to a domain type, we associate with every variable a set of final, or acceptable domains.

2.2.11. Definition. A set of domains $\mathcal{A} \subset \mathcal{T}$ qualifies as a set of ***final domains*** of domain type $\mathcal{T}$ if it has the following properties:

- The empty set is not a final domain, i.e., $\emptyset \notin \mathcal{A}$.
- All canonical domains are final domains, i.e., $\lfloor \mathcal{T} \rfloor \subseteq \mathcal{A}$.
- All non-empty subsets of final domains are final domains, insofar as they are elements of the corresponding domain type, i.e.,

$$\text{for all } D \in \mathcal{A},\ (\mathcal{P}(D) \cap \mathcal{T}) - \{\emptyset\} \subseteq \mathcal{A}. \qquad \square$$

We will be using three specific sets of final domains:

- In general we will use $\mathcal{A} = \lfloor \mathcal{T} \rfloor$ for all variables. For integers and Booleans, this entails that constraint solving yields solutions to CSPs. For constraint solving on the reals its yields a sequence of canonical domains for which the CSP complies to some notion of local consistency, as explained above.
- Alternatively, for constraints on the reals we may be interested in a limited precision ϵ only. In this case we use the set

$$\mathcal{A} = \lfloor \mathcal{F} \rfloor \cup \{[a, b] \in \mathcal{F} \mid 0 < b - a \leq \epsilon\}.$$

- We will also be looking at situations where we are interested in finding an assignment, or an interval of adequate precision for only some of the variables. The other variables are called ***auxiliary variables***, and for these we use $\mathcal{A} = \mathcal{T} - \{\emptyset\}$ to indicate that we will accept all non-empty domains of the domain type. The variables for which $\mathcal{A} \subset \mathcal{T} - \{\emptyset\}$ are called ***decision variables***.

We call the combination of a CSP, and a domain type and set of final domains for each of the variables an extended constraint satisfaction problem.

2.2.12. Definition. By an ***extended constraint satisfaction problem***, or ***ECSP*** we mean a structure of the form

$$\langle \mathcal{C} \ ; \ x_1 \in D_1, \ldots, x_n \in D_n \ ; \ \mathcal{T}_1, \ldots, \mathcal{T}_n \ ; \ \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle, \tag{2.2}$$

where $\langle \mathcal{C} \ ; \ x_1 \in D_1, \ldots, x_n \in D_n \rangle$ is a CSP, and for $1 \leq i \leq n$,

- $\mathcal{T}_i$ is a domain type,
- $D_i \in \mathcal{T}_i$, and
- $\mathcal{A}_i \subset \mathcal{T}_i$ is a set of final domains of $\mathcal{T}_i$.

An ECSP is called consistent if the corresponding CSP is consistent, and inconsistent otherwise. □

Instead of enumerating the full sequences of domain types and sets of final domains, we may use a more compact notation such as $D_x, D_y, D_z \in \mathcal{Z}$ and $\mathcal{A}_x, \mathcal{A}_y, \mathcal{A}_z = \lfloor \mathcal{Z} \rfloor$.

2.2.13. Example. In the ECSP

$$\langle x < y, \ y \neq z \ ; \ x, y, z \in \{0, 1, 2\} \ ; \ D_x, D_y, D_z \in \mathcal{Z} \ ; \ \mathcal{A}_x, \mathcal{A}_y, \mathcal{A}_z \rangle$$

having $\mathcal{A}_x = \lfloor \mathcal{Z} \rfloor$, $\mathcal{A}_y = \mathcal{Z} - \{\emptyset\}$, and $\mathcal{A}_z = \lfloor \mathcal{Z} \rfloor$, x and z are decision variables, and y is an auxiliary variable. Their domains are represented by elements of the domain type $\mathcal{Z}$. □

During the solving process, the domains of the variables are drawn from their respective domain types. Instead of an ECSP that constitutes a solution, in our model of constraint solving we will be concerned with creating a solved form.

2.2.14. Definition. Let γ refer to a local consistency notion, e.g., $\gamma =$ arc for arc consistency. An ECSP of the form (2.2) is said to be γ consistent iff the corresponding CSP $\langle \mathcal{C} \ ; \ x_1 \in D_1, \ldots, x_n \in D_n \rangle$ is γ consistent. The ECSP is said to be in ***γ solved form*** iff

- it is γ consistent, and
- $D_i \in \mathcal{A}_i$, for all $1 \leq i \leq n$.

Further, for two ECSPs

$$P := \langle \mathcal{C} \ ; \ x_1 \in D_1, \ldots, x_n \in D_n \ ; \ \mathcal{T}_1, \ldots, \mathcal{T}_n \ ; \ \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle$$

and

$$P' := \langle \mathcal{C}' \ ; \ x_1 \in D'_1, \ldots, x_n \in D'_n \ ; \ \mathcal{T}_1, \ldots, \mathcal{T}_n \ ; \ \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle$$

having $D'_1 \subseteq D_1$,. . . ,$D'_n \subseteq D_n$, we say that

- P' is a $\boldsymbol{\gamma}$ ***solved form of*** P if
  - P' is in γ solved form, and
  - P, with $D_1, \ldots, D_n$ substituted by $D'_1, \ldots, D'_n$ and $\mathcal{C}$ substituted by the projection $\mathcal{C}[D'_1, \ldots, D'_n]$ is in γ solved form.
- P' is a ***subproblem*** of P if every γ solved form of P' is also a γ solved form of P.
- P' is a ***proper subproblem*** of P if
  - P' is a subproblem of P, and
  - $D'_i \subset D_i$, for some $1 \leq i \leq n$. □

The definition of a subproblem can be extended to allow that subproblems have more variables than the ECSP that they are subproblems of. Because this leads to infinite sets of subproblems, and we do not need this facility to model our solving process, we have deliberately restricted the subproblem relation to ECSPs with equal numbers of variables. The modified constraints $\mathcal{C}'$ are needed because constraints are defined as subsets of Cartesian products of domains.

2.2.15. Example. The ECSP

$$\langle 100x^2 = 1 \ ; \ x \in \mathsf{hull}(\{-\tfrac{1}{10}, \tfrac{1}{10}\}) \ ; \ D_x \in \mathcal{F} \ ; \ \mathcal{A}_x = \lfloor \mathcal{F} \rfloor \rangle$$

is hull consistent, but is not in hull solved form because $D_x \notin \mathcal{A}_x = \lfloor \mathcal{F} \rfloor$.

The ECSP

$$\langle x < y, \ y \neq z \ ; \ x = 0, \ y \in \{1, 2\}, \ z = 0 \ ; \ D_x, D_y, D_z \in \mathcal{Z} \ ; \ \mathcal{A}_x, \mathcal{A}_y, \mathcal{A}_z \rangle$$

with $\mathcal{A}_x, \mathcal{A}_z = \lfloor \mathcal{Z} \rfloor$ and $\mathcal{A}_y = \mathcal{Z} - \{\emptyset\}$ is in arc solved form, because it is arc consistent, while the domains of all decision variables are elements of the corresponding sets of final domains. It is also a proper subproblem and arc solved form of the ECSP of Example 2.2.13. Note that we use $x = c$ as shorthand for $x \in \{c\}$. □

As we mentioned before, for integer and Boolean variables, all practicable consistency notions have the property that an inconsistent assignment of values to variables leads to a failed ECSP. A solved form corresponds to a solution of the original CSP, and the solving process is ***sound***.

For constraints on the reals, using $\mathcal{F}$ as a domain type, this is not the case. In general, proving the presence or absence of a solution in a solved form is difficult. The constraint solving process is ***complete***, though, and the best that can be expected is a set of solved forms whose domains are guaranteed to contain all solutions to the original, real valued problem.

With integer or Boolean variables and in the absence of auxiliary variables, the notions of a solution and a solved form coincide. If the distinction is not important, we will sometimes use the term "solution" also for solved forms.

2.2.6 Constraint Solvers

We can now give a precise definition of constraint solving, as we will use it in this thesis. Given

- a CSP $P = \langle \mathcal{C} \ ; \ x_1 \in D_1, \ldots, x_n \in D_n \rangle$,
- a sequence of domain types $\mathcal{T}_1, \ldots, \mathcal{T}_n$ such that for all $1 \leq i \leq n$, $\mathcal{T}_i(D_i)$ exists, and
- a sequence of sets of respective final domains $\mathcal{A}_1, \ldots, \mathcal{A}_n$,

let P_E denote the ECSP

$$\langle \mathcal{C} \ ; \ x_1 \in \mathcal{T}_1(D_1), \ldots, x_n \in \mathcal{T}_n(D_n) \ ; \ \mathcal{T}_1, \ldots, \mathcal{T}_n \ ; \ \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle.$$

Now by solving P we mean constructing an ECSP that is a γ solved form of P_E, for some notion of local consistency γ. A constraint solver is any algorithm, procedure, or application that works towards this goal. In particular

A complete constraint solver is guaranteed to deliver any number of solved forms that we are interested in (notably one, or all solved forms), or all solved forms, if the number of existing solved forms is less than the number that we are interested in.

An incomplete constraint solver transforms an ECSP into a set of ECSPs that are proper subproblems of the original ECSP. The sets of solved forms of these subproblems cover the set of solved forms of the original ECSP. Incomplete constraint solvers for which the set of subproblems always is of size one will play an important role in our model of constraint solving.

A distinct branch of constraint solving deals with solving ***constrained optimization problems*** (COPs). Here the goal is to find an assignment of values to variables that satisfies all constraints and in addition yields an optimal value for some ***objective function***. We will consider optimization as constraint solving, where every next solved form is constrained to be an improvement of the solved forms that have already been found (see Section 5.9.2). In this sense, by means of an all-solution search a complete constraint solver is guaranteed to find the solved form for which the objective function yields the optimum. We will be looking at optimization only in the context of integer domain types. The standard reference for optimization in presence of constraints on the reals is Numerica [HMD97].

2.3 Branch-and-Propagate Search

Generally, constraint solving comes down to a systematic exploration of all possible assignments of values to variables by means of a tree search algorithm. At

$$\begin{tikzcd}
 & x,y,z \in \{0,1,2\} \arrow[d, "\underline{x<y},\ y \neq z"] & & \\
 & x \in \{0,1\},\ y \in \{1,2\},\ z \in \{0,1,2\} \arrow[dl] \arrow[dr] & & \\
\boldsymbol{x=0},\ y \in \{1,2\},\ z \in \{0,1,2\} \arrow[d] & & \boldsymbol{x=1},\ y \in \{1,2\},\ z \in \{0,1,2\} \arrow[d, "\underline{x<y},\ \underline{y \neq z}"] & \\
{} & & x=1,\ y=2,\ z \in \{0,1\} \arrow[dl] \arrow[dr] & \\
 & {} & & {}
\end{tikzcd}$$

Figure 2.1: An illustration of branch-and-propagate constraint solving

every node of the search tree we try to reduce the remaining search space by removing values from the variable domains that will not contribute to any solution. This is called ***pruning*** the search tree, and the pruning techniques that are applied in constraint solving are referred to as ***constraint propagation***. These techniques enforce some form of local consistency on the subproblems represented by the nodes of the search tree. In many cases, the time saved by the reduced search space significantly outweighs the time spent on constraint propagation.

To illustrate constraint solving by branch-and-propagate search, consider the CSP

$$\langle x < y,\ y \neq z\ ;\ x, y, z \in \{0, 1, 2\}\rangle$$

We deal with integer domains, and all variables are decision variables, so no explicit reference to the ECSP is necessary. Before we do any search, we can already use the constraint $x < y$ and remove the value 2 from the domain of x: there is no value in the domain of the other variable involved in the constraint, y, that would make this constraint true for $x = 2$. Similarly, we can remove the value 0 from the domain of y. At that point the CSP is arc consistent, and we cannot reduce the problem any further by using the individual constraints, so we proceed by branching (see Figure 2.1). In the left branch we assume $x = 0$, and in the right branch we assume $x = 1$. Suppose now that search continues along the right branch. Here we can propagate the constraint $x < y$ again, and remove the value 1 from the domain of y. This effectively fixes the value of y, and now we can also use the constraint $y \neq z$ to remove the value 2 from the domain of z, because for $z = 2$ we would no longer be able to satisfy the constraint $y \neq z$. After this we reach arc consistency again, and we proceed by branching. Depending on whether we are interested in one solution or in all solutions, eventually we would also have to explore the branch $x = 0$.

2.3.1 Constraint Propagation

Domain Reduction Functions

The constraint propagation phase is usually implemented by repeated application of a number of reduction operators. In principle these operators can modify the set of constraints as well, and they could even be defined to add or remove variables, but we will mostly be concerned with reduction operators that modify the domains of variables. Such operators can be represented as functions on domain types.

2.3.1. Definition. For an ECSP of the form (2.2) a ***domain reduction function*** (DRF) with input scheme $s := i_1, \ldots, i_l$ and output scheme $t := j_1, \ldots, j_m$ is a function

$$f : \mathcal{T}_{i_1} \times \ldots \times \mathcal{T}_{i_l} \to \mathcal{T}_{j_1} \times \ldots \times \mathcal{T}_{j_m}$$

where s and t are both n-schemes.

Application of f transforms the sequence of domains $D_1, \ldots, D_n$ into the sequence $D'_1 \ldots, D'_n$ such that

$$\langle D'_{j_1}, \ldots, D'_{j_m} \rangle = f(D_{i_1}, \ldots, D_{i_l})$$

and $D'_i = D_i$ if i does not occur in scheme t.

We denote this transformation by

$$\langle D'_1, \ldots, D'_n \rangle = f^+(D_1, \ldots, D_n)$$

and

$$f^+ : \mathcal{D}_1 \times \ldots \times \mathcal{D}_n \to \mathcal{D}_1 \times \ldots \times \mathcal{D}_n$$

is called the ***domains extension*** of f. Applying it transforms an ECSP

$$P := \langle \mathcal{C} \ ; \ x_1 \in D_1, \ldots, x_n \in D_n \ ; \ \mathcal{D}_1, \ldots, \mathcal{D}_n \ ; \ \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle,$$

into

$$P' := \langle \mathcal{C}' \ ; \ x_1 \in D'_1, \ldots, x_n \in D'_n \ ; \ \mathcal{D}_1, \ldots, \mathcal{D}_n \ ; \ \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle,$$

where $\mathcal{C}'$ is the projection $\mathcal{C}[D'_1, \ldots, D'_n]$. □

The notion of a domains extension unifies the domains and codomains of all DRFs on a given ECSP. This allows us to treat the DRFs as univariate functions on ECSPs. Local consistency enforcing can now be described as the computation of a common fixed point of the domains extensions of DRFs.

2.3.2. Example. Consider an ECSP with variables x, y, z and domains $D_x, D_y, D_z \in \mathcal{Z}$. The DRF $f_{NE} : \mathcal{T}_y \times \mathcal{T}_z \to \mathcal{T}_y \times \mathcal{T}_z$, having $f_{NE}(D_y, D_z) = \langle D'_y, D'_z \rangle$, with

$$D'_y = \begin{cases} D_y - D_z & \text{if } D_z = \{z\} \\ D_y & \text{otherwise} \end{cases} \qquad D'_z = \begin{cases} D_z - D_y & \text{if } D_y = \{y\} \\ D_z & \text{otherwise} \end{cases}$$

enforces the constraint $y \neq z$. Its domains extension is $f^{+}_{NE} : \mathcal{T}_x \times \mathcal{T}_y \times \mathcal{T}_z \rightarrow \mathcal{T}_x \times \mathcal{T}_y \times \mathcal{T}_z$, having $f^{+}_{NE}(D_x, D_y, D_z) = \langle D'_x, D'_y, D'_z \rangle$ with $D'_x = D_x$, and D'_y and D'_z as for f_{NE}.

Alternatively, we could have used two DRFs, each updating one of the domains involved in the constraint. We demonstrate this for $x < y$, which is enforced by $f_{LT1} : \mathcal{T}_x \times \mathcal{T}_y \rightarrow \mathcal{T}_x$, and $f_{LT2} : \mathcal{T}_x \times \mathcal{T}_y \rightarrow \mathcal{T}_y$, having

$$f_{LT1}(D_x, D_y) = \{x \in D_x \mid \exists y \in D_y \; x < y\}$$

and

$$f_{LT2}(D_x, D_y) = \{y \in D_y \mid \exists x \in D_x \; x < y\}$$

The domains extensions of these two functions have the same signature as f^{+}_{NE}. We have $f^{+}_{LT1}(D_x, D_y, D_z) = \langle D'_x, D'_y, D'_z \rangle$, with $D'_x = f_{LT1}(D_x, D_y)$, and $D'_y = D_y$ and $D'_z = D_z$, and similarly for f^{+}_{LT2}.

Now the ECSP

$$\langle x \leq y,\ y \neq z \ ;\ x \in D_x, y \in D_y, z \in D_z \ ;\ D_x, D_y, D_z \in \mathcal{Z} \ ;\ \mathcal{A}_x, \mathcal{A}_y, \mathcal{A}_z \rangle$$

is arc consistent if

- none of the domains D_x, D_y, D_z is empty, and
- $\langle D_x, D_y, D_z \rangle$ is a common fixed point of f^{+}_{NE}, f^{+}_{LE1}, and f^{+}_{LE2}. □

Domain reduction functions, and the reduction operators that they represent, can be seen as an incomplete constraint solvers, as introduced in Section 2.2.6, for which the resulting set of subproblems is of size one. In general there are many options for implementing a constraint with DRFs. For more complex constraints, these will typically have a different trade-off between computation time and the amount of pruning, and hence the level of consistency that is achieved.

Also the domain type is of great influence on the level of consistency. For example if we use domain type $\mathcal{I}$, values can only be removed if they happen to be equal to the bounds of the domain, and in general we cannot enforce arc consistency for $x \neq y$.

Iteration Algorithm

Computing the common fixed point of (the domains extensions of) a set of domain reduction functions can be realized by repeated application of these functions, until none of them is able to reduce the domains any further. AC-1, the basic algorithm for computing arc consistency does just that: it keeps applying all domain reduction functions in sequence until a full sequence passes in which no domains are updated.

In order to reduce the number of DRFs that are applied, more advanced algorithms use the schemes of the DRFs, and information on updated variable domains to maintain a bookkeeping of functions that still need to be applied before we can be sure to have computed a common fixed point. Many such algorithms can be described as instantiations of generic iteration algorithms [Apt99, Gen02]. Different instantiations exploit various properties of domain reduction functions.

For expressing the properties that are of interest to us, consider the partially ordered set $(\mathcal{D}, \sqsubseteq)$, where $\mathcal{D}$ is a set of ECSPs on the same variables, and $P \sqsubseteq P'$ denotes that P' is a subproblem of P. We now have the following properties.

- If all DRFs correspond to inflationary functions on $(\mathcal{D}, \sqsubseteq)$, and $\mathcal{D}$ does not contain infinite increasing sequences, then the generic iteration algorithm is guaranteed to terminate [Gen02].
- If all DRFs correspond to monotonic functions on $(\mathcal{D}, \sqsubseteq)$, then any terminating execution of the generic iteration algorithm computes the same fixed point of these functions: the least common fixed point [Apt99].

Recall that a function f on $(\mathcal{D}, \sqsubseteq)$ is called ***inflationary*** if $P \sqsubseteq f(P)$ for all $P \in \mathcal{D}$, and that f is called ***monotonic*** if $P \sqsubseteq Q$ implies $f(P) \sqsubseteq f(Q)$ for all $P, Q \in \mathcal{D}$. An infinite increasing sequence $P_1, P_2, \ldots$ of elements of $\mathcal{D}$ has the property that for all $i \geq 1$, $P_i \sqsubseteq P_{i+1}$ and $P_i \neq P_{i+1}$. In our case, absense of such sequences (the ***ascending chain condition*** of [Gen02]) follows from the property that set inclusion is a well-founded relation over any domain type.

In this thesis we will mostly be working with Algorithm 2.1, having

$$\mathit{update}(F, D, D') := \{\ g \in F \mid \text{there exists an element } i \text{ in the input scheme of } g \text{ for which } D[i] \neq D'[i]\ \}$$

The resulting algorithm is equal to the CDA algorithm [Mon00a], except for the use of the *failed* flag (a common extension, see e.g., [AB03]). It is also a restriction of the generic iteration algorithm for compound domains (CD, [Apt99]). The restriction is the use of the intersection for updating the domains: $D'[t] := D[t] \cap f(D[s])$. Here we use $D[t] \cap f(D[s])$ as a shorthand for the sequence

$$D[t_1] \cap f(D[s])[1], \ldots, D[t_n] \cap f(D[s])[n],$$

where n is the length of the output scheme t. This restriction ensures that application of the DRF is inflationary, and that the algorithm terminates. If the DRFs are monotonic, which is often the case, the order of their application has no influence on the computed result, and we have complete freedom to implement their scheduling by means of an appropriate *select* function.

The set G of Algorithm 2.1 contains those DRFs for which we cannot yet be sure to have computed a fixed point. In principle, every function needs to be applied at least once, so initially, G equals the set of all DRFs F. However, if we start from a CSP that is already a common fixed point of the functions in F, except for some minor changes due to branching, efficiency of the propagation phase can be improved by initializing G with only those functions that are affected by the branching. This is exploited in our implementation, but here we define only the basic solving algorithms.

```
parameters: function $choose$,
            function $update$.

input:  domains $D_1, \ldots, D_n$,
        a set $F$ of DRFs,

output: domains $D_1, \ldots, D_n$.

$D := D_1, \ldots, D_n$
$D' := D$
$G := F$
$failed :=$ false
while $G \neq \emptyset$ and $\neg failed$
    $choose$ $f \in G$. Let $s$ and $t$ be the input scheme resp. output scheme of $f$
    $G := G - \{f\}$
    $D'[t] := D[t] \cap f(D[s])$
    $G := G \cup update(F, D, D')$
    $D[t] := D'[t]$
    if there exists an element $i$ in $t$ for which $D[i] = \emptyset$
    then
        $failed :=$ true
    end
end
```

Algorithm 2.1: Constraint propagation

2.3.2 Search

We use an additional operator to model the branching step of the branch-and-propagate search. Such an operator can be seen as incomplete constraint solver for which the set of resulting subproblems always contains at least two elements. As we pointed out before, we restrict ourselves to branching on the domains. In that case, the branching operator can be expressed as a function on domain types as well.

2.3.3. Definition. For an ECSP of the form (2.2) a ***domain branching function*** is a partial function

$$f : \mathcal{T}_1 \times \ldots \times \mathcal{T}_n \to \mathcal{P}(\mathcal{T}_1 \times \ldots \times \mathcal{T}_n)$$

such that if $\langle D_1, \ldots, D_n \rangle \in \mathcal{T}_1 \times \ldots \times \mathcal{T}_n$ has the property that

- none of the domains D_i is empty, and
- at least one of the domains is not a final domain,

then

$$\{D'_1 \times \ldots \times D'_n \mid \langle D'_1, \ldots, D'_n \rangle \in f(D_1, \ldots, D_n)\}$$

is both a proper covering and a minimal covering of $D_1 \times \ldots \times D_n$. □

Recall that a ***covering*** of a set X is a set of subsets of X, whose union equals X. A ***proper*** covering of X does not contain X, and a covering is a ***minimal*** covering if the omission of any element would destroy the covering property.

2.3.4. Example. The function $f : \mathcal{Z}^n \to \mathcal{P}(\mathcal{Z}^n)$ having

$$f(D_1, \ldots, D_n) = \{\langle D_1, \ldots, D_{j-1}, \{x\}, D_{j+1}, \ldots, D_n \rangle \mid x \in D_j\}$$

with $j = \mathsf{min}(\{i \mid 1 \leq i \leq n, |D_i| > 1\})$ is a domain branching function for an ECSP of the form (2.2) with $\mathcal{T}_i = \mathcal{Z}$ and $\mathcal{A}_i = \lfloor \mathcal{Z} \rfloor$, for $1 \leq i \leq n$. □

A straightforward branching strategy, which is also used in the previous example, is to split the domain of a single variable into a number of subdomains, and to keep the other domains unchanged. In this case, the two primary aspects of a domain branching function are:

- which variable to select, and
- how to construct the subdomains for that variable.

chronological	the variables are in some explicit order, and the first variable x_i in this order whose domain D_i is not yet in $\mathcal{A}_i$ is selected.
fail-first	selects a variable with the largest domain size. Used primarily with domain type $\mathcal{Z}$.
fail-last	the opposite of the previous strategy.
round robin	selects the variable x_i whose domain D_i is not in $\mathcal{A}_i$ that has least recently been selected

Table 2.1: Variable selection strategies

enumeration	used primarily with domain types $\mathcal{Z}$ and $\mathcal{B}$: a subdomain of size 1 is created for each of the values in the original domain.
L/R-enumeration	used with domain types $\mathcal{Z}$ and $\mathcal{I}$: the domain is split into two subdomains. One is a singleton set containing a specific value, and the other is the original domain minus the selected element. Obvious candidates for the selection are the leftmost and rightmost elements.
bisection	used primarily with domain types $\mathcal{I}$ and $\mathcal{F}$. The interval domain is split in two intervals of equal width.

Table 2.2: Value selection strategies; see also Figure 4.2 on page 70

We will call these aspects the ***variable selection strategy***, and the ***value selection strategy***, respectively. Tables 2.1 and 2.2 list the general-purpose variable and value selection strategies that are used in this thesis. The domain branching function of Example 2.3.4 uses a chronological variable selection strategy, and enumeration as a value selection strategy. In addition to these general-purpose strategies, specialized variable selection strategies are used in Section 4.4 and Chapter 6, but we implemented these strategies by manipulating domain sizes, and use one of the standard strategies for the actual selection.

The branch-and-propagate search process can now be specified as in Algorithm 2.2. The set F of this algorithm is called the ***search frontier*** [Per99]. It contains the sequences of domains for all subproblems that still need to be explored. These subproblems are nodes of the search tree. Initially the search frontier contains just the original problem.

propagate(D_w, R) applies the domain reduction functions in R to the domains in D_w, the node of the search tree (world) that was selected for further exploration. For *propagate* we can use an instance of Algorithm 2.1.

failed and *final* are predicates on sequences of domains:

$$\textit{failed}(\langle D_1, \ldots, D_n\rangle) \text{ is false iff } D_1 \neq \emptyset, \ldots, D_n \neq \emptyset,$$

$$\textit{final}(\langle D_1, \ldots, D_n\rangle) \text{ is true iff } D_1 \in \mathcal{A}_1, \ldots, D_n \in \mathcal{A}_n.$$

parameters: function *select*,
function *propagate*.

input: an ECSP $P := \langle \mathcal{C} \ ; \ x_1 \in D_1, \ldots, x_n \in D_n \ ; \ \mathcal{T}_1, \ldots, \mathcal{T}_n \ ; \ \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle$,
a domain branching function f,
a set R of domain reduction functions,

output: a set S of sequences of domains such that for all $\langle D'_1, \ldots, D'_n \rangle \in S$,

$$\langle \mathcal{C}[D'_1, \ldots, D'_n] \ ; \ x_1 \in D'_1, \ldots, x_n \in D'_n \ ; \ \mathcal{T}_1, \ldots, \mathcal{T}_n \ ; \ \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle$$

is a γ solved form of P, where γ is the notion of consistency enforced for the constraints in $\mathcal{C}$ by *propagate* and R.

```
$F := \{\langle D_1, \ldots, D_n \rangle\}$
$S := \emptyset$
while $F \neq \emptyset$ do
    $select\ D_w \in F$
    $F := F - \{D_w\}$
    $D'_w := propagate(D_w, R)$
    if $\neg failed(D'_w)$
    then
        if $final(D'_w)$
        then
            $S := S \cup \{D'_w\}$
        else
            $F := F \cup f(D'_w)$
        end
    end
end
```

Algorithm 2.2: Branch-and-propagate search

These predicates characterize a node either as a ***solution***, ***failure***, or ***internal*** node of the search tree. If after constraint propagation all domains are in their respective set of final domains, the *final* predicate holds, and the node is considered to be a solution node. Solution nodes constitute solved forms. If constraint propagation voids the domain of one or more variables, then the *failed* predicate holds, which characterizes the node as a failure. Note that *failed* and *final* are mutually exclusive. Nodes that are neither failures nor final (solutions) are called internal nodes.

Internal nodes are expanded by applying the branching function f. All nodes that are thus generated are added to the search frontier F, and the algorithm terminates when F is empty, so it performs an all-solution search. The algorithm can easily be modified for a first-solution search.

A very important aspect is still left unspecified. Selecting D_w from F determines in which subproblem of the search frontier the search algorithm will continue the exploration. This is called the ***traversal strategy***. In case of an all-solution search this may not seem important, because every node needs to be visited eventually, but even then it is of great influence on the size of the set F, and hence on the space complexity of the search algorithm. Two obvious alternatives are the following.

- If the set F is managed as a ***stack***, which implies that we always select one of the most recent additions, the algorithm essentially performs a ***depth-first*** search. In this case, there is a linear relation between the number of variables and the maximum number of nodes in the search frontier. If at every node we store the complete domains of all variables, then the space complexity of depth-first search is $O(n^2d)$, where n is the number of variables, and d is a bound on the size of the domains.

- Managing the set F as a ***queue***, selecting always one of the oldest additions, results in a ***breadth-first*** search. In this case, the maximum size of the search frontier is exponential in the number of variables, leading to a space complexity of $O((nd)^n)$.

Apart from the size of the search frontier, the traversal strategy is important if we are interested in only a limited number of solutions. In that case, using a good heuristic may bring the algorithm to these solutions faster, or at least improve the probability that this will happen. The same applies to optimization, where we do need to explore the full search space, but where some heuristics may discover good suboptimal solutions earlier than others, which will lead to a stronger pruning of the search tree.

In this thesis, depth-first search is the default traversal strategy, but alternatives are discussed in Sections 4.1.2 and 4.3.

2.4 Composing Constraint Solvers

In the introduction we indicated that there are various approaches to constraint solving. For some classes of CSPs there exist efficient methods that exploit properties of these problems, such as all constraints being linear, and if completeness is not important, local search may give a reasonably good solution quickly. In this thesis we deal with a specific approach to constraint solving, namely branch-and-propagate search, but even if we limit ourselves to this particular approach we have many options for various aspects such as how to build the search tree, how to explore it, what level of consistency to enforce, etc.

To a large extent, the deployment of constraint solving consist of determining the right combination of approaches, algorithms, and heuristics. For this reason, constraint systems must allow us to explore alternative combinations of solving techniques. Without assuming any level of granularity, we will refer to realizing such combinations as ***solver composition***. Many techniques have successfully been used in specific domains to build practical constraint programming tools, but in most cases the facilities for solver composition are limited to a small set of built-in alternatives, for only some aspects of constraint solving, and a major challenge in this field is how to achieve a combination of various existing methods and techniques in a single framework. In this section we will look at composite solvers that are described in the literature.

2.4.1 Combining Propagation Operators

In many cases, the best known approach to solving a certain class of CSPs involves a combination of several constraint propagation techniques that each have a set of reduction operators. In this section we will look at a number of examples of such combinations.

Subsuming Forms of Consistency

Reduction operators for stronger forms of local consistency are usually more computation-intensive than reduction operators for weaker forms of consistency. For certain strong forms of consistency it is more efficient first to compute a weaker form of consistency. The stronger form subsumes the weaker form, but the values that are removed as a part of computing the weaker form of consistency need not be considered, at a much higher cost, by the reduction operators for the stronger form. Such schemes can be explained as a combination of reduction operators for both forms of consistency.

An example is ***singleton arc consistency*** (SAC, see also Section 7.6), introduced by Debruyne and Bessière. This form of consistency entails that for every value in the domain of every variable, the CSP can still be made arc consistent if that value is assigned to the variable. The algorithm for enforcing SAC presented in [DB97] effectively tries all variable - value pairs, and enforces arc consistency for each such assignment. The values for which a failure is deduced are removed from the domain of the corresponding variables. Obviously, SAC subsumes arc consistency, but before entering the loop that tries all possible assignments, the SAC enforcing algorithm first enforces arc consistency on its argument CSP to reduce the number of arc consistency computations inside the loop.

Another example is the BC4 algorithm for enforcing ***box consistency***. For constraints on the reals, using $\mathcal{F}$ as a domain type, the obvious approximation of arc consistency is the notion of hull consistency, defined in Section 2.2.4. However, as we shall see in Section 4.5, for arbitrary constraints it is difficult to compute hull consistency. The usual way around this is to decompose the user constraints into atomic constraints, for which enforcing hull consistency is easy. The disadvantage of this approach is that hull consistency for the decomposed system is a weaker notion of consistency than hull consistency for the original system [CDR99]. The problem lies in the imprecision that is caused by evaluation of expressions with multiple occurrences of variables, using interval arithmetic and the natural interval extension. Therefore many solvers use an intermediate form of consistency, called box consistency [BMVH94] (see also Section 7.3.2). It can deal with multiple occurrences of a single variable. The procedure for enforcing box consistency effectively searches in the domain of this variable for bounds that do not fail the constraint, if all other domains are kept constant. Because of this search, the procedure is potentially costly, and the BC4 algorithm [BGGP99] applies the box consistency procedure only for projections of constraints with multiple occurrences of variables, and after hull consistency is computed for the decomposed constraints. The BC4 algorithm can be explained as a combination of the reduction operators for hull consistency and box consistency.

Such schemes can be implemented by computing a common fixed point of DRFs for both forms of consistency. In Algorithm 2.1, the selection of DRFs should then exhaustively apply the DRFs for the weaker form, before applying any DRF for the stronger form. For this purpose, the CHOCO system [Lab00] uses a layered propagation architecture, where operators are divided into eight layers of increasing computational cost. In [SS04] a dynamic scheme is described to recognize that different domain modifications may entail different computational costs for the same reduction operator.

Hybrid Forms of Consistency

For single occurrences of variables, hull and box consistency are the same, and the BC4 algorithm simply applies the most efficient set of reduction operators. In other cases, it may make sense to combine sets of reduction operators that enforce different forms of consistency to achieve some hybrid form of consistency.

Good examples of such combinations are found in constraint-based approaches to solving scheduling problems. Baptiste, Le Pape and Nuijten provide an overview of this area [BLPN01]. A basic solver for the job-shop scheduling problem combines the forms of consistency achieved by enforcing the disjunctive constraint, and by the ***edge finding*** procedure (see also Chapter 6). Job-shop scheduling allocates activities to machines, and the disjunctive constraint states that if two activities require the same resource, they cannot overlap in time. Edge finding aims at identifying activities that must execute first, or last, in a given set of activities. Enforcing the disjunctive constraint and applying edge finding make different, and complementary deductions. The basic solver can be described and implemented as a combination of the reduction operators for the disjunctive constraint and the edge finding procedure.

2.4.2 Hybrid Solvers

For many problems, a natural CSP formulation involves variables of different types. Mixed integer / real problems, combinations of Boolean and numerical variables, and combinations that involve more complex types like sets and multisets have been reported in literature. Solvers that support multiple domain types are sometimes called ***hybrid*** solvers, emphasizing that for different domain types different, specialized constraint propagation methods are used.

For example, RealPaver [Gra04b] supports both reals and integers, but it is essentially a solver on the reals. Integers are implemented as real variables that are constrained to have integer values. Therefore we do not consider RealPaver to be a hybrid solver. On the other hand, ILOG solver [Ilo01] will be considered to be a hybrid solver, because it supports both reals and integers, and uses a different representation for both of them.

The model of constraint solving introduced in this chapter is well suited for describing hybrid solvers on the level of constraint propagation: every variable has its own domain type, and they can be linked through DRFs of a mixed signature. For example, the DRF $f : \mathcal{F} \times \mathcal{I} \to \mathcal{F} \times \mathcal{I}$, having $f(\langle D_r, D_i \rangle) = \langle D'_r, D'_i \rangle$, where $D'_r := \mathsf{hull}(D_r \cap D_i)$ and $D'_i := [\lceil \mathsf{min}(D_i \cap D_r) \rceil \,..\, \lfloor \mathsf{max}(D_i \cap D_r) \rfloor]$, enforces the equality constraint on two variables of domain types $\mathcal{F}$ and $\mathcal{I}$.

Audemard et al. [ABC$^+$02] describe a solver for propositional formulas where in addition to propositional variables and their negations, literals can be (linear) mathematical constraints. This solver can be characterized as a hybrid solver, which combines Boolean and numerical domain types. As such, it fits our model of constraint solving: in principle, their solver uses the DPLL algorithm [DLL62] for generating assignments of truth values to the Boolean variables, for which the formula is satisfiable from a propositional point of view. Such assignments have an induced numerical problem, and a solution consists of an assignment of truth values for which the induced numerical problem is solvable. As we shall see in Section 4.4, the DPLL algorithm can largely be expressed in the framework of Section 2.3.

State-of-the-art solvers for checking satisfiability of propositional formulas (called SAT solvers) depend on non-chronological backtracking search, in particular on backjumping and learning (no-good recording) [LMS03]. In contrast, our framework relies heavily on constraint propagation. These are opposite approaches: constraint propagation aims at removing values in an attempt to avoid failures, while non-chronological backtracking happily works towards a failure, and then deduces the reason for the failure. This information is used to prevent the same failure from happening again. In Section 4.4 we will discuss the possibilities to incorporate backjumping and no-good recording into our framework.

2.4.3 Search

The SALSA language [LC02] supports composition of strategies for tree search, local search, and hybrid forms of search that combine tree search and local search. For tree search, SALSA supports the composition of branching strategies. For finite domains, for example, we may want apply a bisection branching until all domains are of a certain size, and proceed by an enumeration branching from that point on. We will see an example of the composition of branching strategies in SALSA in Section 6.4.

Possibilities for composition of traversal strategies also exist. We already mentioned that depth-first search has linear space complexity, while that of breadth-first search is exponential in the number of variables. Yet it may sometimes be beneficial to perform a limited amount of breadth-first search. A possible strategy would be to search breadth-first until a certain threshold amount of memory has been used. Then we switch back to depth-first search to clean up the search frontier, and only when enough memory becomes available again we return to breath-first search. This can be seen as a composite traversal strategy, built from two basic strategies.

Another possibility could be to have multiple instances of the same search strategy running inside a single solver. This would simulate a parallel search, and can be beneficial because of the ***speedup anomaly***, discussed in Section 8.5. However, the same effect could be achieved by actually running a parallel solver, and rely on the operating system for time sharing between the parallel processes or threads. Moreover, the interleaved depth-first (IDS) strategy [Mes97] was designed exactly to achieve this effect. But given that IDS is a useful thing to have, it is desirable to be able to ***compose*** an IDS-like strategy, instead of having to re-program a solver for it.

2.4.4 Solver Cooperation

Solver cooperation aims at combining individual solvers, in order to solve problems more efficiently, or to be able to solve problems that none of the combined solvers could handle on its own. While a solver that combines a number of specific algorithms could be classified as solver cooperation, and even individual reduction operators can be seen as atomic constraint solvers, a commonly used justification for research in the area of solver cooperation is that the development of new constraint solvers is a time-consuming and error-prone process, and that composing cooperations from pre-existing solvers will reduce the development costs of new constraint solvers. For this reason we would like to reserve the term for solvers that are composed of autonomous component solvers.

Let us look again at the solver of Audemard et al. as an example. Intuitively, we would like to think of this solver as a cooperation between a SAT solver and a numerical solver. Arguably, this would be a plausible explanation if indeed the mathematical solver is invoked only after a full model for the propositional part of the formula has been constructed. However, in [ABC$^+$02] it is demonstrated that the performance improves significantly if satisfiability of the induced numerical problem is verified each time the truth value of a mathematical constraint is fixed, as a part of the SAT solving. For this reason, it is better to say that the numerical solver is ***embedded*** in the SAT solver, comparable to the way that constraint propagation is embedded in the search procedure of Algorithm 2.2.

The embedded solver is used largely as a black box, but this is not the case for the solver in which it is embedded. In the ideal case, we would like to be able to compose a solver like that of Audemard et al. from software components, but in general, existing SAT solvers will not have facilities for performing checks like verifying the satisfiability of the growing induced numerical problem.

This illustrates a persistent problem with solver cooperation. Autonomous solvers are closed applications that run to completion, and there are few, if any, facilities for exchanging information with the environment, or for controlling the solving process once it has started. This justifies research towards a uniform interface for constraint solvers, as reported for example in [HSG01] and [AM98]. A further problem with cooperation of arbitrary solvers through a unified interface is the handling of disjunctions. These have to be handled on the level of the framework in which the solvers cooperate, which then has to implement a search algorithm. If more than one of the cooperating solvers is allowed to generate disjunctions simultaneously, the search space grows faster than with a centralized branching scheme. This becomes even more problematic if subsets of the constraints are sent to different solvers. Because these subsets likely form underconstrained problems, the resulting disjunctions will be large. We expect that this limits the use of such general frameworks to a small number of very specific cases.

Another, rather straightforward mode of solver cooperation would be to combine a local search solver and a complete solver for solving optimization problems. The local search solver will not be able to prove optimality, but each time it finds a better solution, the complete solver can take the updated bound into account to prune parts of the search space that will not improve on this bound. This simple branch-and-bound scheme (see Section 5.9.2) already requires that the complete solver can regularly check on new bounds. A black-box solver may not be able to do so, and the best option then is to restart it, with the additional constraint that it should improve the best solution found by local search.

Having said this, examples of successful cooperations of largely autonomous solvers do exist. In the area of numeric problems, a survey of cooperations of symbolic solvers and interval solvers is presented in [GMB01]. Examples of such cooperation schemes include the following.

- Symbolic solvers may be able to derive redundant constraints for a given problem, that strengthen the domain reduction when they are combined with the original constraints. The symbolic solver is applied as a preprocessing step to the branch-and-propagate solving.

- A dedicated solver for linear constraints checks consistency of the linear part of the problem each time a variable domain is reduced.

Currently there is also much interest in the combination of constraint solving and operations research methods. Because OR methods can deal only with specific classes of CSPs, such as linear programs, the models that can be solved by these methods are typically approximations, or ***relaxations*** of combinatorial problems. These relaxations can be solved very efficiently, and the results can be used to improve the efficiency of branch-and-propagate solving. For example, solving a linear relaxation of a problem first may give a good bound for the outcome of an objective function quickly. As another example, in [MvH02b], a linear relaxation of a combinatorial optimization problem is used to partition the domains of variables into two sets, of promising (good) and less promising (bad) values. With n variables, this partitioning gives rise to 2^n subproblems, that are solved in sequence, starting with the subproblem that is composed of only good subdomains, and gradually increasing the number of bad domains in a limited discrepancy search fashion (see Section 4.1.2).

2.4.5 Distributed Constraint Solving

Instead of composing a constraint solver from its constituent parts, it is sometimes necessary or desirable to distribute the solving process itself. In this case, the solver can be seen as to be composed from a number of cooperating processes. Reasons for doing so may be that the CSP that we want to solve is itself distributed, while it is impossible or undesirable to gather all constraints, and apply regular, centralized solving methods. Such problems are commonly referred to as ***distributed constraint satisfaction problems*** (DisCSPs). Another way in which distributed solving can be beneficial is by exploiting parallelism.

Distributed Constraint Propagation

Distributed versions of Algorithm 2.1 exists [MR99, Mon00a], which can be used to parallelize, or otherwise distribute constraint propagation. Because of the fine-grained communication involved, we expect that in general it will be difficult to obtain parallel speedup through distributed constraint propagation, but the approach can be justified in the DisCSP case. We return to the subject of distributed constraint propagation in Chapter 9.

Distributed Search

Yokoo [Yok01] has defined several algorithms for distributed search, specifically for the DisCSP case. The underlying assumption is that the DisCSP variables are distributed among a set of ***agents***, who propose values for their variables to each other. It is highly desirable that such algorithms are ***asynchronous***, i.e., they rely as little on synchronization and external coordination as possible, in order that the agents remain autonomous in the execution of the search algorithms.

Parallel Search

Parallelism in constraint solving is best exploited by parallel search, i.e., different solvers, running on different processors explore different parts of the search tree in order to reduce the turn-around time. A common issue with parallel processing is to achieve a good load balance, i.e., preventing that some processors become idle, while others do all the work. Because CSP search trees can be irregular and unbalanced, a dynamic load balancing scheme is required. A special issue is how to implement parallel optimization, where new bounds for the objective function have to be communicated between the cooperating solvers. Parallel constraint solving is the subject of Chapter 8.

2.5 Summary

In this chapter we introduced the subject of constraint solving. In addition to the regular notion of a constraint satisfaction problem, we defined domain types and extended constraint satisfaction problems. Domain types provide a uniform model for solving constraints on integer, real, and Boolean variables, and allow us to express several properties of the implementation of the domains of such variables. ECSPs augment a CSP with domain type information. They also provide the means to distinguish decision variables from auxiliary variables, and they specify the required precision for solving constraints on the reals.

Returning to the central theme of this thesis, as described in Section 1.2, we have now created a framework where a branch-and-propagate constraint solver is composed of the following elements.

- Domain types from which the domains of logical variables are drawn.
- Domain reduction functions that enforce constraints.

- Functions *choose* and *update* that instantiate a generic iteration algorithm to specify a scheduler for the domain reduction functions.
- A domain branching function that specifies how to construct a search tree. This typically involves a variable selection strategy and a value selection strategy.
- A selection function that specifies a traversal strategy. Together, these three strategies form a search strategy.

In addition, to give an idea of what we want to achieve, we gave an informal description in the context of this framework of several composite solvers that are used in practice.

In the next chapter we will describe our implementation of the formal framework defined here. In the chapters thereafter, in addition to addressing some more specific research questions, we will evaluate the framework and its implementation by composing constraint solvers for various problems. In Chapter 10 we will return to our description of existing composite solvers, and discuss what has been achieved, and what questions have been left unanswered.

Chapter 3

OpenSolver: a Software Component

This chapter describes the OpenSolver software, which is used in the remainder of the thesis as an experimental platform for composing constraint solvers. OpenSolver can best be described as a coordination-enabled abstract branch-and-propagate tree search engine. It is based on the solving algorithms of Section 2.3: branching, and pruning the search tree are implemented as the application of reduction operators that modify the domains of variables. It is abstract in the sense that its functionality is determined by software ***plug-ins*** that configure the basic solving algorithms.

These plug-ins come in a number of categories, corresponding to various aspects of branch-and-propagate tree search. A separate category of plug-ins covers the ***coordination layer*** of the algorithm (Figure 3.1). This category is special in the sense that it does not correspond to one specific aspect of Algorithm 2.1 or 2.2. Instead, plug-ins in this category control the execution of the solving algorithms, and facilitate the exchange of data between a solver and its environment.

No component technology is used to implement the branch-and-propagate constraint solving, but a major design goal was that OpenSolver itself can be used as a software component in several solver cooperation schemes. This is realized through the coordination layer mechanism. In Chapters 7, 8, and 9 we will be looking at examples of larger systems, were OpenSolver is used as a software component.

3.1 Introduction

OpenSolver evolved from the DICE (DIstributed Constraint Environment) system, which we discuss in Chapter 9. DICE itself started as an implementation of the coordination-based distributed constraint solver proposed by Monfroy and Arbab [Mon00a, AM00]. The original DICE system is described in [Zoe03b]. It is a framework for distributed branch-and-propagate tree search, whose functionality is determined by ***plug-ins*** for domain types, domain reduction functions,

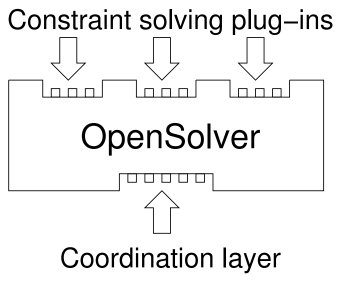


Figure 3.1: ***Plug-ins*** determine the actual functionality and appearance of an OpenSolver instance

and branching and traversal strategies. Basically, in DICE every plug-in resides in its own process, and in [Zoe03a] we proposed an optimization that allows an arbitrary distribution of the plug-ins over a set of cooperating solvers. On the one hand, OpenSolver implements the solvers of the proposed optimization.

On the other hand, many of our experiments do not require distributed solving, and we also wanted to implement an efficient sequential constraint solver. Moreover, we wanted to be able to use the same plug-ins in DICE and in this sequential solver. Therefore we decided to develop a single application, and tailor it towards reuse as a software component in several environments. To a large extent this is realized by the ***coordination layer*** plug-in, which forms the interface between OpenSolver and its environment. One plug-in configures OpenSolver as a component solver of DICE, and another plug-in configures it as a stand-alone solver.

In addition to these two roles, the coordination layer made it very easy to implement the time-out mechanism that forms the basis of the ***parallel constraint solver*** described in Chapter 8. We also use it to implement ***nested search***. This technique entails that the functionality of a domain reduction function involves a limited branch-and-propagate tree search. We use an almost autonomous OpenSolver instance for such a DRF, and this instance interfaces with another OpenSolver through a special coordination layer plug-in. Nested search is the topic of Chapter 7.

The remainder of this chapter is organized as follows. In Section 3.2 we describe the different categories of plug-ins related to constraint solving. Section 3.3 describes the odd one out: the category of coordination layer plug-ins. In Section 3.4 we clarify some implementation aspects, including writing new plug-ins for OpenSolver.

3.2 Constraint Solving Plug-ins

The following categories of plug-ins implement constraint solving:

- ***domain types*** that implement the domains of variables,

```
⟨Configuration⟩  →  ⟨Statement⟩; {⟨Statement⟩;}
⟨Statement⟩      →  ⟨Keyword3⟩ ⟨Identifier⟩ IS ⟨Identifier⟩ ⟨Specifier⟩
                 |  ⟨Keyword2⟩ ⟨Identifier⟩ ⟨Specifier⟩
⟨Keyword3⟩       →  VARIABLE | AUX
⟨Keyword2⟩       →  DRF | SCHEDULER | ANNOTATION | TDINFO
                 |  FRONTIER | INTERNAL | EXPLORE | EXPAND
⟨Specifier⟩      →  "{"⟨String⟩"}"
```

Figure 3.2: Syntax of the OpenSolver configuration language

- ***reduction operators*** that modify these domains,
- ***schedulers*** of reduction operators,
- ***containers*** of nodes of the search tree,
- ***selectors*** that make a selection among the nodes stored in containers,
- ***annotations*** that decorate the nodes of the search tree with extra information, to be used by plug-ins in some of the other categories,
- ***evaluators*** of nodes of the search tree; these determine whether a node of the search tree is a solution, failure, or internal node.

An OpenSolver instance is configured through a script in a simple language that has a statement for each of these categories. Program 3.1 is an example of such a script, related to one the experiments in Chapter 5.

Figure 3.2 defines the syntax of OpenSolver configuration scripts, where $\{\ldots\}$ should be read as "zero or more instances of the enclosed," and where "{" and "}" denote the curly bracket symbols. Each statement consists of a keyword for one of the plug-in categories, plus an ***identifier-specifier pair***. The identifier designates a particular plug-in in the category of the statement, and the specifier string is used to initialize an instance of this plug-in. For the purpose of this mechanism, every plug-in, in any category should be able to initialize itself from a specifier string. These specifier strings can be arbitrarily complex. For example, in Chapter 7 we use a plug-in that is an almost autonomous OpenSolver instance. Comparable to procedure definitions in imperative programming languages, the specifier string for this plug-in contains a full solver configuration. At the other extreme, when given the empty string as a specifier, the plug-in for domain type $\mathcal{F}$, the set of all floating-point intervals, yields a representation for the domain $[-\infty, \infty] = \mathbb{R}$.

The `VARIABLE` and `DRF` statements introduce variables and their domains, and reduction operators that operate on them. In addition to the identifier-specifier

```
VARIABLE x IS IntegerInterval {1..100000};
VARIABLE y IS IntegerInterval {1..100000};
VARIABLE z IS IntegerInterval {1..100000};
VARIABLE obj IS IntegerInterval {};
AUX aux_x3 IS IntegerInterval {};
AUX aux_y2 IS IntegerInterval {};
AUX aux_z3 IS IntegerInterval {};
AUX aux_x1y1 IS IntegerInterval {};
DRF IIARule { aux_x3^1 * (1) = x^3 };
DRF IIARule { aux_y2^1 * (1) = y^2 };
DRF IIARule { aux_z3^1 * (1) = z^3 };
DRF IIARule { aux_x1y1^1 * (1) = y * x };
DRF IIARule { x^3 * (1) = aux_x3 };
DRF IIARule { y^2 * (1) = aux_y2 };
DRF IIARule { z^3 * (1) = aux_z3 };
DRF IIARule { y^1 * (x) = aux_x1y1 };
DRF IIARule { x^1 * (y) = aux_x1y1 };
DRF IIARule { aux_x3^1 * (1) = -1*aux_y2 + 1*aux_z3 };
DRF IIARule { aux_y2^1 * (1) = -1*aux_x3 + 1*aux_z3 };
DRF IIARule { aux_z3^1 * (-1) = -1*aux_x3 + -1*aux_y2 };
DRF IIARule { obj^1 * (1) = 2*aux_x1y1 + -1*z };
DRF IIARule { aux_x1y1^1 * (-2) = -1*obj + -1*z };
DRF IIARule { z^1 * (1) = -1*obj + 2*aux_x1y1 };
DRF Optimize { +obj };
DRF RoundRobin { 0, x, y, z, obj };
SCHEDULER ChangeScheduler { schedule =
   { 1,2,9,4,0,2,10,5,0,1,11,6,3,12,13,7,8,3,14,15 }
};
```

Program 3.1: Example of an OpenSolver configuration script pairs of the other statements, the `VARIABLE` statement uses an extra identifier that is interpreted as the variable's name. The `AUX` keyword is a variant of `VARIABLE` for introducing auxiliary variables. These two statements are the only ones that ***add*** plug-in instances.

The other statements ***replace*** plug-in instances, for which a default is readily available. The `SCHEDULER` keyword is used for replacing the scheduler of reduction operators. `FRONTIER` and `INTERNAL` replace the containers for storing sets of nodes of the search tree, and `EXPLORE` and `EXPAND` replace the selectors operating on them. The `ANNOTATION` statement specifies what information, by means of an annotation plug-in, is attached to nodes of the search tree. The default is to use no annotations. The statement for introducing a node evaluator is `TDINFO`, for ***termination detection information***. This reflects OpenSolver's origin as a distributed system: establishing that distributed constraint propagation has finished is then a matter of distributed termination detection. Determining the nature of a node of the search tree is naturally combined with detecting the termination of constraint propagation.

In the remainder of this section, we discuss the different categories of constraint solving plug-ins.

3.2.1 Variable Domain Types

This category of plug-ins corresponds directly to the variable domain types discussed in Section 2.2.4. Plug-ins exist for the four standard domain types $\mathcal{B}$, $\mathcal{Z}$, $\mathcal{I}$, and $\mathcal{F}$ that were introduced there, and as is the case for all categories of plug-ins, new domain type plug-ins can be added to an OpenSolver installation. This is described in Section 3.4. In Chapter 6 we discuss special-purpose domain types that are introduced for solving one specific kind of combinatorial problems.

Just like in object-oriented programming an object is an instance of a class, in OpenSolver a variable domain is an instance of a variable domain type. As we shall see in Section 3.4, the plug-ins, and hence the individual domain types are actually classes, with a common base class for each category. For now it suffices to realize that being objects, the domains of variables have a state, on which a number of operations are defined, and that these operations are implemented by means of member functions (we use the C++ terminology, in other object-oriented languages member functions are called ***methods***).

In the OpenSolver input language, variables are introduced with the following statement.

$$\texttt{VARIABLE}\ \langle \textit{Identifier} \rangle\ \texttt{IS}\ \langle \textit{Identifier} \rangle\ \langle \textit{Specifier} \rangle$$

The first identifier gives the variable a ***name***, and the second identifier designates the plug-in that will be used to implement the domains that are associated with the variable, during the solving process. The specifier is a character string that represents the initial domain. It will be used to create a domain for the variable in the root of the search tree.

Specifier strings are interpreted by a constructor for the class that implements a plug-in. For a set T and a domain type $\mathcal{T} \subseteq \mathcal{P}(T)$, such a constructor implements a partial function $f : \mathcal{P}(T) \to \mathcal{T}$ having $f(D) = \mathcal{T}(D)$. For example, the plug-in `RealInterval` implements the standard type $\mathcal{F} \subset \mathcal{P}(\mathbb{R})$, the set of all floating-point intervals. For the specifier string we can use any interval where the bounds have a finite decimal representation. This is the case for $\frac{1}{10}$, but no floating-point representation exists for $\frac{1}{10}$, so

```
VARIABLE x1 IS RealInterval {[-0.1, 0.1]};
```

will create the smallest floating-point interval that properly contains $[-\frac{1}{10}, \frac{1}{10}]$.

For variable domain types, the operations on the state include the following.

- An operation to ***clone*** the domain. OpenSolver is a ***copying-based*** constraint solver (see Section 4.2), and the clone operation implements the copying on the level of the variable domains.

- An operation to ***split*** the domain into a number of subdomains. The member function for this operation has an integer argument that can be used to specify a particular method for generating these subdomains, for example enumeration, or bisection (see Figure 4.2 on page 70). The interpretation of this argument, and the value selection strategies that it encodes are specific to the variable domain types. Applying the split operation on the domain of a single variable is the primary method of branching, but more complex branching strategies, typically involving more than a single variable, are also supported.

- A member function that gives an indication of the ***size*** of the domain. This is a non-negative integer, where 0 means that the domain is empty, i.e., a failure has been deduced. The value 1 indicates that the domain is a singleton set, and values greater than 1 are an indication that the domain can be split into a number of subdomains. In principle, the domain sizes determine the nature (solution, failure, or internal) of the nodes of the search tree, but as we shall see below, this can be overridden by a node evaluator plug-in.

Instead of `VARIABLE`, the keyword `AUX` can be used to introduce a variable. The syntax is the same, and the only effect is that a flag is set, to mark the variable as auxiliary. Node evaluator plug-ins use this information to implement the notion of auxiliary variables introduced in Section 2.2.5. For auxiliary variables, all domains except the empty set are final domains. They are not considered in distinguishing solutions from internal nodes of the search tree. Therefore we do not need to branch on auxiliary variables.

3.2.2 Reduction Operators

Plug-ins in the reduction operator category implement the domain reduction functions and domain branching functions of Sections 2.3.1 and 2.3.2. A hybrid form is used for optimization. In the OpenSolver configuration language, the statement for introducing a reduction operator is

$$\texttt{DRF}\ \langle \textit{Identifier} \rangle\ \langle \textit{Specifier} \rangle$$

The specifier string typically contains the list of variables that the operator applies to, and some further specification of the operation that it performs on these variables. For example,

```
DRF DDNEQ { x1 - x2 <> 2 };
```

enforces the constraint $x_1 - x_2 \neq 2$ on two discrete domain variables x_1 and x_2.

State

Reduction operator instances have a state that holds at least an internal representation of the information extracted from the specifier string. Contrary to variable domains, the state of a reduction operator is global. It applies to all nodes of the search tree. The only information about a reduction operator that is stored per node, is a flag indicating whether the operator is active or not. Reduction operators can signal to the scheduler that controls their application that they have become redundant in a certain branch of the search tree. Schedulers (see below) may use this information to avoid unnecessary application of such operators. In principle, the design of OpenSolver also allows that new, redundant reduction operators (and auxiliary variables) are added during the solving process, to be active in particular parts of the search tree only, but these facilities are not currently exploited.

Interface

Three member functions constitute the basic interface of reduction operators and the rest of the system:

- a function that reports the names of the variables that the reduction operator applies to; this information is typically extracted from the specifier string when the reduction operator is created,
- the ***propagation function***, which is called during the constraint propagation phase, and
- the ***termination function***, which is called upon termination of constraint propagation, during the branching stage.

Both the propagation function and the termination function take as an argument an array of pointers, whose type is the abstract base class for variable domain types. Through these pointers, the functions can reduce the domains of the variables. The OpenSolver framework takes care that the arrays of pointers correspond directly to the variable names reported by the first of the above three member functions. Also, like the domain reduction functions that they implement, reduction operators have an input scheme and an output scheme: the set of variables that trigger their application, and the set of variables that they can modify, respectively. In Section 3.4 we see how these are implemented.

The propagation function and the termination function must supply the solver with information about variables that they change. The minimum requirement is that they set a flag for every change, but a more elaborate protocol is possible. For example it could be useful to set different flags for modifying a bound or an internal value of a domain. Per variable, a reduction operator can specify what modifications it is interested in. The reduction operators that modify a variable and the reduction operators that depend on this variable must use the same protocol for signaling such modifications. It is up to the scheduler that applies the reduction operators to exploit this information, though. An example of the use of this facility is discussed in Section 4.2.

Interaction with Domain Types

In Algorithm 2.1 domain reduction is realized by intersecting variable domains with the outcome of the domain reduction functions. In OpenSolver this is not implemented in such a clean way. All domain type plug-ins implement the intersection, but sometimes it is more efficient to use a different modification of the domains. A reduction operator can then perform modifications that are specific to the domain type that the operator is defined for. In other words, the operators make ***assumptions*** about the types of the domains that they operate on. Such assumptions are implemented by ***type casting*** (see also Section 3.4), as a result of which, member functions for domain specific modifications become available.

As an illustration, the `DDNEQ` operator of the above example operates on finite domains variables, and will typecast the argument domains to objects of the class that implements this domain type. If one of the argument domains has size 1, it can now retrieve the integer value that it contains. Instead of having to construct a domain for intersecting the other argument domain with, it will call a member function specific to the finite domains implementation that allows individual values to be removed.

The Termination Function

Normally, the termination function is used for branching. The creation of subproblems is implemented by this function creating subdomains for one or several of the variables. Subdomains can be created in any way that suits the branching strategy, but in most cases the plug-ins rely on the basic splitting methods provided by the domain types.

The termination and propagation functions can cooperate to implement optimization. If after termination of constraint propagation we have not deduced a failure, while none of the variable domains can be split any further, the node of the search tree is considered to be a (possibly suboptimal) solution. The termination function can then record some information about this suboptimal solution, such as a new bound for the outcome of an objective function. The objective function can be evaluated by regular constraint propagation, and the propagation function can then enforce this new bound as a dynamic constraint on the variable that holds the outcome of the objective function.

Classification of Reduction Operators

Depending on the use of the propagation and termination functions, three kinds of reduction operators[1] are distinguished.

- ***propagation operators***, these are reduction operators that do not modify their state, and whose termination function does not create any subdomains. Propagation operators are active only during constraint propagation, and through their propagation functions, they implement the DRFs of Definition 2.3.1 on page 19.
- ***branching operators***, these are reduction operators that do not modify their state, and where the propagation function does not modify the variable domains. Through their termination functions they implement the domain branching function of Definition 2.3.3 on page 23. Branching operators are active only during the branching stage.
- ***optimization operators***, these are reduction operators where the termination function creates no subdomains, but modifies the state, and where the result computed by the propagation function depends on this state. They are active in both the propagation stage and the branching stage of the solving algorithm.

3.2.3 Schedulers

The application of the reduction operators is controlled by plug-ins in the scheduler category. In principle, there are two schedulers involved: one for the propagation stage, which applies the propagation functions of the reduction operators, and one for the termination stage, which applies the termination functions. In

[1]The keyword `DRF` is a misnomer because it refers specifically to propagation operators, while it is also used to introduce branching operators and optimization operators.

practice, there is no need for elaborate scheduling mechanisms in the termination stage, and the latter scheduler is currently fixed to apply the termination functions once, in sequence.

In contrast, the scheduler for the propagation stage implements the constraint propagation algorithm, and is of great influence on the efficiency of the solver. The statement for modifying the propagation scheduler is

$$\texttt{SCHEDULER}\ \langle\mathit{Identifier}\rangle\ \langle\mathit{Specifier}\rangle$$

Contrary to the `VARIABLE` and `DRF` statements, which always extend the solver configuration, this statement ***replaces*** the current propagation scheduler. Currently, it is not possible to use different schedulers in different nodes of the search tree. In a distributed setting, however, each solver has its own scheduler, and depending on the constraints assigned to a solver, it may make sense to use different plug-ins here.

Schedulers have a state per node of the search tree. This state can be used to store the bookkeeping of reduction operators that still need to be applied, like the set G of Algorithm 2.1. In this sense, schedulers are similar to variable domains, and they also provide an operation for cloning the state of the scheduler. Cloning the scheduler state is interesting when branching commences before constraint propagation has reached a fixed point, and some operators are still scheduled for application.

The primary member function of a scheduler plug-in runs the constraint propagation algorithm. In addition, a scheduler provides to the framework member functions for scheduling individual variables and reduction operators. These are called when reduction operators are introduced, and when changes to variables are made outside the control of the scheduler, for example when creating new nodes of the search tree by splitting the domain of a variable.

The scheduler has access to two data structures: one that contains the problem structure (the `PStruct`) and one that contains information about the problem that is specific to a node of the search tree (the `WPStruct`, for ***world*** problem structure). The `PStruct` gives access to the reduction operators, and contains the dependencies between variables and reduction operators. The `WPStruct` is used mainly to keep track of which reduction operators are still active: through this data structure OpenSolver allows a scheduler to deactivate a reduction operator. If it signals after application that it will not be able to achieve further reduction, it can be deactivated in the present branch of the search tree. A scheduler plug-in can choose to use this facility or not. Reference counting, and copy-on-change are used to maintain this information. As a result, actually switching off reduction operators may involve a considerable memory overhead (bounded by the number of reduction operators times the size of the search frontier) for storing new versions of the bitmap of active operators.

Algorithm 2.1 computes a fixed point of the domain reduction functions in its input set F. In contrast, a scheduler plug-in need not run to completion. It may

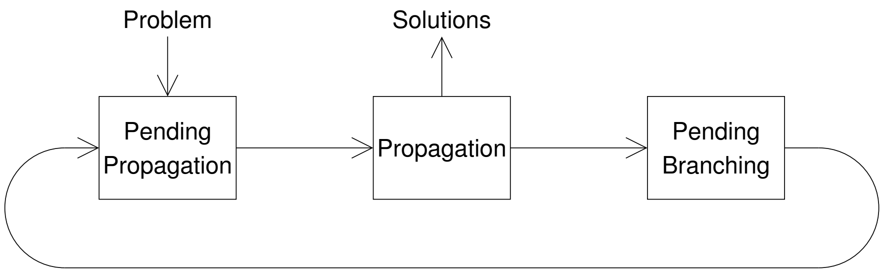


Figure 3.3: Three sets of nodes constitute the solver state

stop executing the constraint propagation algorithm at any point, but it needs to signal to the solver whether it wants to be reactivated later, to continue the computation of the fixed point. If this is not the case, the solver will consider that the propagation stage has finished. Otherwise it will activate the scheduler again before starting the branching stage. This can be useful even in a stand-alone constraint solver: as in the model of Monfroy and Arbab [AM00], multiple nodes of the search tree can be subject to constraint propagation, and each of these nodes can be expanded by branching. If schedulers run to completion, constraint propagation in these nodes is executed in sequence. Examples of CSPs exist where in some nodes of the search tree, constraint propagation takes much longer than in others. For such problems, one node could block the progress of search in the other nodes. By running a scheduler only for a fixed amount of time, and then passing control to other nodes before resuming the computation of the fixed point, we can benefit from concurrency in the search without having to resort to distributed processing.

3.2.4 Containers

The state of the OpenSolver search algorithm consists of three sets of nodes of the search tree (Figure 3.3). Each of these sets is implemented by a container plug-in.

- The ***search frontier***, containing unexplored nodes that are pending constraint propagation. Both the original CSP, and the subproblems that are the result of applying a branching operator enter the algorithm in this set.
- A set of nodes that are subject to ***constraint propagation***.
- A set of ***internal nodes*** where constraint propagation has terminated without deducing a failure or a solution. In these nodes, the search tree can be expanded by applying a branching operator.

In what follows, the ***state of the solver*** usually refers to these three sets.

Being set implementations, containers have member functions for adding and removing nodes of the search tree, and for iterating over their contents. Implementing sets of nodes of the search tree in specific, containers can be aware of some properties of the nodes that they contain, and iterate over these nodes accordingly. Such properties are implemented by ***annotations*** (discussed below). We will see examples of actual container implementations in Sections 4.1.2 and 4.3.

The containers for the sets of nodes that are pending propagation and branching can be modified. The syntax is, respectively,

`FRONTIER` ⟨*Identifier*⟩ ⟨*Specifier*⟩

`INTERNAL` ⟨*Identifier*⟩ ⟨*Specifier*⟩

Like the `SCHEDULER` command, these commands replace the currently active plug-in instances. For the set of nodes that are subject to propagation, it is important that all nodes can be removed from the container efficiently, so here an implementation based on a linked list is used. There seems to be little use for other alternatives here, so this particular container is currently fixed, and cannot be changed in the configuration language.

3.2.5 Selectors

Selectors are used to identify the nodes that are transfered between the three sets of Figure 3.3. All nodes where constraint propagation has terminated are moved to the rightmost set automatically, so selectors are used only for transferring nodes from the search frontier, and for selecting the nodes that will be expanded by branching. The respective commands are the following.

`EXPLORE` ⟨*Identifier*⟩ ⟨*Specifier*⟩

`EXPAND` ⟨*Identifier*⟩ ⟨*Specifier*⟩

These commands override the currently active selectors.

Apart from the plug-in machinery, selectors offer a single operation to the OpenSolver framework. The member function for this operation takes as an argument an array of containers, and returns an array of elements of these containers. So in OpenSolver, the selectors can inspect the entire state, i.e., all three sets of nodes of the search tree of Figure 3.3. It is the responsibility of the programmer of the plug-in, and of the application or person who writes the solver configuration to ensure that nodes are selected from the correct set. For example, if OpenSolver consults the selector for identifying the nodes that must be expanded by branching, and a node from the search frontier is selected, a run-time error will occur once OpenSolver discovers that this node cannot be removed from the container of nodes that are pending branching. We see selectors at work in Section 4.3.

3.2.6 Node Evaluators

The `TDINFO` command changes the OpenSolver node evaluator:

`TDINFO` ⟨*Identifier*⟩ ⟨*Specifier*⟩

The purpose of a node evaluator[2] is to determine the ***nature*** of a node of the search tree, solution, failure, or internal node, once constraint propagation has finished. In a distributed setting, this information is collected when the coordination-layer plug-ins of the cooperating solvers try to establish termination of distributed constraint propagation. For this reason the keyword refers to the information that is collected during termination detection.

There is basically one way to establish that a node is a ***failure***: at least one of the variable domains reports a size 0. Therefore the main purpose of a node evaluator is to distinguish between ***internal nodes*** and ***solution nodes***, the latter corresponding to subproblems that are in solved form. This implements the test $D_1 \in \mathcal{A}_1, \ldots, D_n \in \mathcal{A}_n$, for an ECSP of the form (2.2), as we discussed in Section 2.2.5. To this end, node evaluator plug-ins provide a member function that takes as an argument the array of domains of a node. In Section 4.5 we use a node evaluator to calculate solved forms of a limited precision, for variables whose domains are floating-point intervals.

After establishing the nature of a node, the plug-in instance is cloned, and attached to the node itself, in order that the nature of the node is made known to the branching operators. The termination functions of such operators can then verify properties that are hard to establish by means of constraint propagation, and may decide to fail a node yet, after it was characterized as a solution by the branch-and-propagate tree search.

In addition to establishing the nature of a node of the search tree, node evaluators may access the annotations, thus allowing extra information to be attached to a node before the branching stage commences. We will see how this facility is used in Section 4.3.

3.2.7 Annotations

Annotations are used to ***decorate*** nodes of the search tree with additional information, to be maintained and referenced by plug-ins in the other categories. The annotation of a node is set by the following command.

`ANNOTATION` ⟨*Identifier*⟩ ⟨*Specifier*⟩

In addition to the basic plug-in machinery, the base class for annotation implementations requires only that annotations can be cloned. Since annotations exist

[2]ILOG Solver also has a `NodeEvaluator` class, which implements related, but different functionality [Per99].

only for the convenience of other plug-ins, OpenSolver makes no further assumption about the functionality that they offer. This is entirely a matter of subtyping, and type casting by the plug-ins that use the annotations. We see specific uses of annotations in Sections 4.1 and 4.3, and in Chapter 8.

3.2.8 Putting it All Together

Before we move on to the coordination-layer plug-in, it is good to take a step back and discuss the relation and interaction between the plug-ins in the categories that we just introduced.

Generally, branch-and-propagate constraint solving starts with constraint propagation. This involves the domain type, reduction operator, and scheduler plug-ins. The ***domain type*** plug-ins provide representations for the domains of the variables, and the ***reduction operators*** inspect and modify the domains to enforce the constraints. Note that the concept of a constraint is absent in the system.

Before the start of the branch-and-propagate search, the system asks the reduction operators for the names of the variables that they want to be applied to. These names are compared to the variable names introduced by the `VARIABLE` and `AUX` statements, and the relation between the variables and the reduction operators is laid down in the `PStruct`. The actual application of the reduction operators is controlled by the ***scheduler*** plug-in. It is responsible for applying the reduction operators to the right variables, as specified in the `PStruct`. The scheduler plug-in applies only the propagation functions of the reduction operators, so during the constraint propagation stage, only two of the three kinds of reduction operators, namely propagation operators and optimization operators are active.

At some point, the scheduler plug-in will return control to the system, and notify that constraint propagation has finished. It now becomes important to realize that we have been working in a particular node of the search tree. At the start of the solving process, this is the root node. These nodes are data structures of the framework, but they may have been decorated with extra information in the form of an ***annotation*** plug-in.

When constraint propagation finishes in a node of the search tree, the system applies the ***node evaluator*** plug-in to this node. In any case, a node evaluator has to determine (typically by inspecting the sizes of the variable domains) whether a node is a solution, failure, or internal node of the search tree, but it may gather further information about the node, which can then be stored in its annotation.

Nodes that have been characterized as failures (dead-end leaves in the search tree) are least interesting from the perspective of the search process: these are basically just discarded. Solutions are slightly more interesting, but before a node gets the solution treatment, which may actually mean the end of the solving process, all reduction operators are applied once more. This time, instead of the propagation functions, the termination functions of the reduction operators are applied. Termination functions are used for branching, but that concerns internal nodes of the search tree only. For solutions they can perform some last-minute tests, and decide to characterize a node as a failure yet, but they can also record some information about the solution, such as a new bound for a criterion variable in the state of the reduction operator.

Internal nodes of the search tree are the most important for the search process. They are stored in the rightmost of the three sets of Figure 3.3. Exactly how they are stored, and in what order they can be retrieved again, is determined by the ***container*** plug-ins that implement these sets. On several occasions, two ***selector*** plug-ins will examine the state of the solver (the three sets of Figure 3.3). One of these plug-ins will make a (possibly empty) selection among the internal nodes in the rightmost set. These nodes are subjected to branching, which entails that the termination functions of all reduction operators are applied in sequence. Typically, most reduction operators are propagation operators, and have inactive termination functions, and there is exactly one reduction operator that is a branching operator. This operator's termination function will create subdomains for some (typically one) of the variables. The system will generate a new node for each of these subdomains, cloning the domains of the other variables. The original internal node can now be discarded, and the new nodes are stored in the set of nodes that await constraint propagation.

The second selector plug-in selects nodes from this latter set. These nodes are moved to the middle set of Figure 3.3, where they will be subjected to constraint propagation, as we described at the beginning of this section. In case of an all-solution search or an optimization problem, the constraint solving process ends when all three sets of nodes are empty. This completes our overview of the solving process, and the roles of the different plug-ins therein. The solving process is under tight control of the coordination layer plug-in, which we will discuss next.

3.3 The Coordination Layer Plug-in

Recall that a major design goal was that OpenSolver can be used as a software component in several solver cooperation schemes. This is realized through the coordination layer plug-in. Every OpenSolver instance has exactly one plug-in in this category installed. This plug-in controls the branch-and-propagate tree search, and through it, the solver can exchange information with its environment. OpenSolver is almost a full application, but it has to be complemented by a coordination layer plug-in to be able to function. Even when it is used as a stand-alone constraint solver, this plug-in is responsible for aspects such as

- all I/O, in particular providing a solver configuration in the language of Figure 3.2, and dispatching solutions,

- whether we search for one or all solutions, or whether we just want to count them.

The coordination layer plug-in is similar to the constraint solving plug-ins in the sense that it is activated using an identifier-specifier pair, but because its presence is required for OpenSolver to function, this pair is part of the shell command used to run the OpenSolver, for example,

```
opensolver -c SeqFileIO 8queens.inp
```

starts OpenSolver as a UNIX process, using the `SeqFileIO` coordination layer plug-in. This coordinates it as a stand-alone, sequential constraint solver that reads the configuration script from a file, whose name is read from the specifier string. The `-c` option is followed by the identifier-specifier pair for the coordination layer plug-in. In the case of `SeqFileIO` the specifier string is a filename. If it should include spaces (which is not the case for `SeqFileIO`), it has to be enclosed in quotes.

The interaction between OpenSolver and its coordination layer plug-in is via a command loop. After activating the plug-in, OpenSolver continually asks it what to do next. One of the first commands that are usually issued is the following.

specifier. This tells OpenSolver that a configuration specification is available. After receiving this command, OpenSolver asks the coordination layer plug-in for a pointer to a location where this specifier is stored, in ASCII byte format in the language of Figure 3.2. After processing the specifier string, OpenSolver notifies the coordination layer that the memory it occupies can be deallocated.

In total, the current version uses 20 commands that fall roughly in two categories: controlling the solving process, and interaction with the environment. Below we describe those commands that are essential for understanding the OpenSolver architecture, and its use as a software component in the other chapters.

3.3.1 Controlling the Solving Process

The nodes of the search tree reside in a data structure called the ***world database***, which is essentially an array of slots that can each hold a single node. When nodes are deallocated they leave a vacant slot, which can be reused when a new node must be stored. If no vacant slot is available, new slots are created. The primary purpose of the world database is to provide a uniform node identification scheme when several OpenSolvers participate in a distributed constraint propagation algorithm. In that case, the world database of each solver contains an array of slots for each of the participating solvers. Every node has an owner: the solver that created it, and all solvers access the data structures for that node through the same slot, in the array for the solver that owns it. Now a node can uniquely be identified by a tuple consisting of the number of the solver that owns it, and the index of the slot it occupies.

In a distributed setting, the three sets of nodes of Figure 3.3 are distributed over the cooperating solvers. All solvers maintain the three sets, but a single node can appear only in the state of one solver, i.e., the global state is distributed over the cooperating solvers. While only one solver marks a node as pending propagation, pending splitting, or being subject to propagation, all solvers may have data structures for the node, containing the local domains and information on active reduction operators.

We discuss the commands that control the solving process roughly in the order in which they occur during a regular branch-and-propagate tree search.

schedule propagation. Receiving this command, OpenSolver activates the propagation selector. This will identify a (possibly empty) subset of the set of nodes that are pending propagation. The nodes in this subset will be transfered to the set of nodes that are subject to constraint propagation.

propagation. For each node in the latter set, run the scheduler of DRFs to apply constraint propagation. When the scheduler finishes, it signals whether constraint propagation has terminated, and a flag on the node is set accordingly.

When a node is scheduled for propagation by the propagation selector, this is signaled to the coordination layer. From that point on, the coordination layer will monitor the progress of constraint propagation for the node, through the flag that we just mentioned. This monitoring takes place for all nodes that the coordination layer knows to be subject to constraint propagation, after each **propagation** command. It may seem counterintuitive that this is the responsibility of the coordination layer, but this is necessary in case several OpenSolver instances participate in a distributed constraint propagation algorithm, such as the one that is described in Chapter 9. In a distributed setting, the information on termination of constraint propagation is local: it applies only to the set of DRFs known to the solver that issued the **propagation** command, and the cooperating solvers have to combine this information in their coordination layers to obtain a global view.

As a part of a distributed termination detection algorithm, or in a stand-alone solver after having been informed that constraint propagation has terminated, the coordination layer may inquire about the status of a node by creating a new node evaluator, and asking the solver to apply it.

evaluate. Receiving this command, OpenSolver asks the coordination layer for a pointer to a node evaluator, and applies it to the node of the search tree for which the command is issued.

The next time OpenSolver asks the coordination layer for a command after **evaluate** was issued, the coordination layer knows that the node evaluator it created has been applied to the node of the search tree that it was interested in at that time. The node evaluator now contains information on whether the node is a solution, failure, or internal node, regarding the part of problem known to the present solver. In a distributed constraint propagation algorithm, this information must now be combined with the view of the other participating solvers. In a stand alone solver, the global status of the node is known immediately, and can be relayed to the solver with the following command.

termination. This command informs the solver that constraint propagation has terminated in a particular node of the search tree. In the case of distributed constraint propagation, this information is global. A node evaluator is provided by the coordination layer to indicate the status of the node.

For solutions, the solver runs the scheduler of reduction operators for the termination stage. This allows the optimization operators to update their state according to the current solution (for example, set a new bound). For failures, the coordination layer is informed that it can start the deallocation process for the node. Internal nodes are moved to the container of nodes that are pending branching.

In a distributed setting, the cooperating solvers must negotiate which solver is allowed to branch on an internal node. Only the coordination layer plug-in of the identified solver should issue the **termination** command for the node. For this purpose, a node evaluator can access information on whether a solver is configured to split a certain variable, so this decision can be made as a part of termination detection. For example, in a distributed fail-first policy, as a part of establishing termination of constraint propagation, the cooperating solvers will likely be circulating a token (see Section 9.2.3). Before forwarding the token, the solvers will issue **evaluate** commands. Through the node evaluator they can inquire about the sizes of the variable domains, and thus search for the smallest domain as a part of termination detection. Similarly, the node evaluator can determine whether a solver is able to branch on this variable, so the token is annotated with the id of the variable with the smallest domain found so far, the size of this domain, and if the token has already visited a solver that is able to split the domain of the variable, the id of this solver.

The following command initiates the actual branching.

schedule branching. This command activates the selector of nodes that are pending branching. This yields a subset of the nodes in the rightmost set of Figure 3.3. In each of these nodes, the scheduler of termination functions is run to actually generate subproblems. The nodes for these subproblems are added to the set of nodes that are pending propagation. The coordination layer is informed that it can start the deallocation of the parent nodes.

In a stand-alone solver, the process of deallocating a node of the search tree simply consists of issuing the following command to the solver.

forget world. This command tells the solver to deallocate the data structures for a particular node of the search tree, and to free the slot it occupies in the world database. Also any reference to it from within the containers that implement the sets of Figure 3.3 is removed.

In the distributed case, deallocating a node is less straightforward, though. The reason is that for processing nodes that descend from it, we may still need to clone some of the domains. This is done the first time that a solver learns about a particular node, as a part of constraint propagation, but this may well be after the solver that created the node has initiated the deallocation of the parent. Therefore deallocating a node involves counting the number of descendants that have been created. Only when this matches the total number of generated branches, the parent node can be deallocated. This tally is kept in the solver, and can be verified by the coordination layer.

more work? After receiving this command the solver reports to the coordination layer whether the sets of nodes that are pending branching and pending propagation are empty or not. In the former case, the traversal of the search tree has finished. In the latter case, the coordination layer can make the solver continue the search by issuing more **schedule branching** and **schedule propagation** commands.

The following command sequence coordinates a basic branch-and-propagate tree search by a stand-alone solver, where propagation runs in a single node, the scheduler of DRFs runs to completion, and internal nodes are split immediately.

specifier
schedule propagation
propagation
evaluate
termination
schedule splitting
forget world
more work?
quit

} repeated until the solver answers negative to the **more work?** question, or the coordination layer decides to break out of the loop, for example if the solver responds to the **termination** command with a notification that it has accepted a solution.

3.3.2 Interaction with the Environment

The **specifier** command, which we discussed at the beginning of this section, falls in the category of commands for regulating the interaction of an OpenSolver instance and its environment. The configuration specifications that it passes to the solver typically come from a file, or have been submitted by another program such as a calculator front-end. The coordination layer plug-in knows where to get these specifications, which varies for different situations where OpenSolver is used as a component solver.

In addition to the **specifier** command, and some commands for generating textual representations of solutions, and nodes of the search tree in general, the following commands are of importance for the way OpenSolver is used as a component solver in the rest of this thesis.

flush. The solver supplies to the coordination layer plug-in a configuration for each of the nodes of the search tree stored in the sets of Figure 3.3.

clear WDB. The solver empties the world database, i.e., it forgets what it was doing, and enters the initial state.

A node is essentially defined by the domains of the variables, and the reduction operators that are active. Like all plug-ins, variables and reduction operators can generate a textual representation of themselves, in the language of Figure 3.2, by which they can be re-created in another solver. This facility is used by the **flush** command. The **flush** command is typically followed by **clear WDB**.

When a new variable is introduced, OpenSolver will ask the coordination layer plug-in if the variable is meant to be exported or not. In a distributed constraint propagation algorithm, exported variables are those that appear in two or more solvers. Changes to the domains of such variables must be communicated. This is initiated by the following command.

pending sends. The solver responds to this command with the identifiers of the variables that have been marked for export, and whose domains have been modified since the last **pending sends** command was issued.

The following command takes care of the actual communication:

export. The solver supplies a pointer to the data structure that represents the domain of a given variable, in a given node of the search tree.

The **pending sends** and **export** commands could be combined for the purpose of distributed constraint propagation, but they have been left separated for the use of exported variables in the operator for nested search, discussed in Chapter 7. After the solver has responded to an **export** command, the coordination layer can actually send the modified domain to another solver. This message has to be tagged with the node of the search tree, to which the information applies. Because of the world database, this tag need only contain the two integers that uniquely identify a node during its lifetime.

The default mechanism for communicating domain updates is the text-based representation that is used for the **flush** command. This has the benefit of being machine independent. For example, we do not have to worry if integers are stored as big-endians or little-endians. To avoid the overhead of string manipulation, a coordination layer plug-in can make assumptions about the variable domain type plug-ins that are used. For example, it can rely on the assumption that all domains that it will ever export, will be able to write themselves to a binary representation. This may of course give rise to problems in a heterogeneous computing environment, but when the environment is homogeneous with respect to, say, integer representation, the string manipulation can easily be avoided. We have not experimented with this, but in a shared memory environment, it may even be feasible to use pointers as representations of the domains, through which they can then be accessed directly.

When receiving an incoming domain update, the coordination layer reconstructs the variable domain. Using the default mechanism, this involves interpreting an identifier-specifier pair. The resulting domain is then passed to the solver via the following command:

update. The solver computes the intersection of the domain of a variable in a particular node of the search tree, and a domain supplied by the coordination layer. The solver assumes that the domains are instances of the same plug-in. If the intersection is smaller than the original domain, the intersection is used as the new domain for the variable. The change is made known to the scheduler of reduction operators for the propagation phase, and the variable is marked as changed for the purpose of the **pending sends** command.

We conclude the description of the coordination layer by clarifying its name.

3.3.3 Coordination

In computer science, ***coordination*** refers to the orchestration of the interaction among the various active entities involved in a software system [Arb98]. As Gelernter and Carriero observed, coordination is a ubiquitous aspect of computing: even the simplest programs interact with their users to exchange input and output [GC92]. Furthermore, on the level of programming languages, the sequential execution of program statements can be seen as a particular form of coordination of computing entities.

While coordination can be recognized in all software systems, it is of particular relevance for ***concurrent systems***, where several interdependent computations overlap in time. Examples of such systems are

- ***parallel*** systems, where concurrency is introduced for reducing the turn-around time of a computation by distributing the workload over several hardware processors, and

- ***distributed*** systems, whose state is distributed over several computing entities that execute concurrently.

Coordination languages are languages for programming the interaction between the active entities of concurrent systems. Such languages are based on a particular ***coordination model***, i.e., a set of assumptions on how these entities interact. Examples of coordination languages, models, and architectures are:

- Linda [CG89], a coordination language based on a coordination model where processes communicate by injecting and consuming tuples from a shared tuple space.

- The Manifold [Arb] language, which implements the Idealized Worker Idealized Manager model, and depends on channel-based communication between processes (see also Chapter 9).

- The Discrete Time ToolBus [BK98], a coordination architecture for the integration of software components (tools). The integration is specified through a script that describes all possible interactions between the tools, and the coordination model supports explicit specification of the timing behavior of systems.

- Reo [Arb02], a channel-based coordination model, wherein complex coordinators, called connectors, are compositionally built out of simpler ones (see also Chapter 8).

Sometimes a distinction is made between ***endogenous*** and ***exogenous*** coordination. The former means that coordination is realized through operations ***within*** the entity that is being coordinated. The OpenSolver command loop, described in Section 3.3.1, is an example of endogenous coordination. Conversely, exogenous coordination is coordination from ***without***: the constructions that regulate the interaction are outside the interacting entities, and the computation code is separated from the coordination code. In this classification, Linda is an endogenous coordination language, and Manifold, ToolBus, and Reo are based on exogenous coordination models.

Although the field originated in the area of parallel and distributed computing, coordination now also manifests itself as an approach to component-based software engineering. From this point of view, a software component is an autonomous system with well defined behavior that has its own thread of control. Systems that are built from such components will be concurrent systems, and their composition requires a form of exogenous coordination. ToolBus and Reo are designed specifically for component-based software engineering.

Coordination and the Coordination Layer

We already pointed out that the OpenSolver command loop can be seen as a form of endogenous coordination, but the term “coordination layer” is primarily meant to emphasize that this software layer makes OpenSolver amenable for various forms of exogenous coordination.

3.4 Implementation

OpenSolver is an object-oriented application with an extendable class hierarchy. Defining the basic structure of a branch-and-propagate constraint solver, OpenSolver can be categorized as an object-oriented ***framework*** [Deu83, JF88]. It employs the typical inverse control mechanism: in principle, OpenSolver calls member functions on objects of the classes that configure it, not the other way around. This mechanism is also known as the ***Hollywood principle***: “Don’t call us, we’ll call you.” [DGS03]

The classes that implement the problem-specific aspects of the solving process are called “plug-ins” for two reasons:

- to avoid using the word ***component***, emphasizing that OpenSolver has not been implemented using any form of component-based software engineering, and
- to emphasize that in addition to being an object-oriented framework, OpenSolver is also a stand-alone application that can be configured for different tasks and environments.

For programming new plug-ins, OpenSolver can further be categorized as a ***white-box*** framework: the programmer needs to understand the implementation of the framework in order to be able to program for it. For some categories of plug-ins, this is limited to understanding the class interface, but especially for programming a coordination layer plug-in, it must be known how OpenSolver reacts to the various commands that can be issued.

As an example of an extendable class in OpenSolver, Program 3.2 shows part of the definition of the abstract class for reduction operator plug-ins.

- Member functions `Compute` and `Termination` are the propagation function and termination function, respectively. Their `arguments` parameter passes an array of pointers, pointing to the domains of the variables that the reduction operator applies to.
- Member functions `Mask` and `Activate` implement protocols for communicating changes to variable domains, as discussed at the end of Section 3.2.2. They also define an operator’s input scheme.

- Member functions `CanReduce` and `CanSplit` characterize an operator as a propagation operator, branching operator, or optimization operator, and define its output scheme.
- The member function `Idempotent` is used to tell a scheduler plug-in that the propagation function is idempotent. Recall that a function f is called idempotent if $f(f(x)) = f(x)$, for all x in the domain of f. If an idempotent DRF modifies the domain of a variable in its input scheme, this DRF need not be scheduled again.

```
class ReductionOperator : public OpenSolverPlugIn
{
public:
   static ReductionOperator *Factory(
         CoordinationLayer *coordinationLayer,
         const char *type,
         const char *spec);
   virtual ~ReductionOperator( );
   virtual const std::vector<char*> &VariableNames( ) const =0;
   virtual int Compute( Domain **arguments, unsigned int *changes,
                        bool *deactivate );
   virtual int Termination( TDInfo *tdInfo, Domain **arguments,
                            Annotation *annotation, bool *deactivate );
   virtual bool CanSplit( int i ); // i==-1: do you want to be called
                                   //        on termination?
                                   // i>=0:  can you split argument i?
   virtual bool CanReduce( int i );// i==-1: do you want to be called
                                   //        during propagation?
                                   // i>=0:  can you reduce argument i?
   virtual unsigned int Mask( int i );
                                   // mask for local_changes[i]
   virtual bool Activate( int i, unsigned int c );
                                   // If Mask[i] does not match,
                                   // do you want to be activated if
                                   // variable i has been modified
                                   // such that local_changes[i] == c?
   virtual bool Idempotent() const;
};
```

Program 3.2: Part of the definition of the abstract class for reduction operators

While as an open-ended, extendable system, OpenSolver is a white-box framework, it aims at providing a ***black-box*** framework for composing branch-and-propagate constraint solvers. Plug-ins can be combined with minimal knowledge of the implementation of the system and the plug-ins. The intention is to establish a set of atomic plug-ins, from which many different solvers can be composed. New plug-ins are added only when the desired functionality cannot be realized by composition of the readily available facilities, or when this cannot be done efficiently. OpenSolver is approached as a black-box framework through the configuration language of Figure 3.2.

Plug-in System

The identifiers for Plug-ins that are available in an OpenSolver installation can be used in the configuration language of Figure 3.2. This is realized by a static member function per plug-in category that contains a large conditional statement to couple the identifiers to the constructors of the classes that implement the plug-ins. This is the first member function in Program 3.2. For example, the code on page 40 would pass the identifier `RealInterval` and the specifier following it to this function. If a plug-in class for `RealInterval` is available, an object of this class is created by calling the constructor for this class, passing the specifier string `[-0.1, 0.1]` as an argument.

Currently, adding a plug-in to an OpenSolver installation involves modifying the actual C++ code for the static member function, by adding a branch to the conditional statement. For example, for `RealInterval`, the following lines were added:

```
#include "RealInterval.h"
...
else if ( !strcmp( type, "RealInterval" ) )
   res = new RealInterval( spec, &syntax_error );
```

Also, the makefile must be modified in order that the new class is compiled and linked, and the application must be rebuilt to take these modification into account. All of these tasks can easily be automated, and a plug-in system based on a source code distribution is straightforward.

Ideally we would not have to relink, or modify code for adding plug-ins. This can be realized by dynamic linking of the object code for plug-ins. A potential problem here is that current C++ compilers do not adhere to a single standard for encoding class member names (name mangling). There are ways around this problem, for example we could provide a C interface. An interesting option is to implement this interface by a general-purpose class in each of the plug-in categories, whose only purpose is to take on board C code that has been compiled separately.

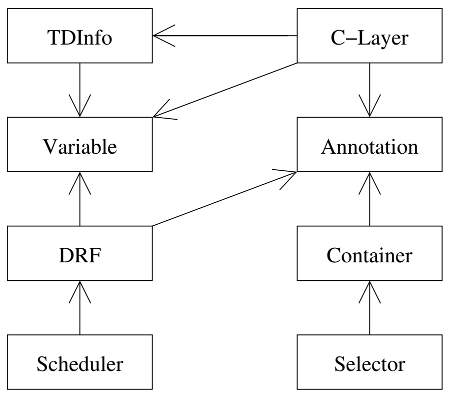


Figure 3.4: Category-dependencies, or assumptions, implemented by type casting. Arrows are drawn from the class that makes the assumption to the class whose objects are being cast

Type Casting

In Section 3.2.2 we already mentioned that it is very common for a plug-in to make assumptions about the plug-ins with which it works in conjunction. The OpenSolver framework accesses all plug-in through the abstract classes for their categories. From this point of view, every variable domain is an object of class `Domain`. However, a reduction operator for finite domains will assume that the variable domains to which it is applied are actually instances of a subclass of `Domain` that provides specific operations, such as deleting integer values. Such assumptions are implemented by ***type casting***, and the interaction between reduction operators and variable domains is just one of many situations where this happens. Figure 3.4 gives an overview of all cases were plug-ins make assumptions about other plug-ins.

It is the responsibility of the user or, more likely, the software that generates a solver configuration in the language of Figure 3.2, to ensure that the correct plug-ins are used. Violating an assumption leads to undefined behavior. It would be easy to implement a type checking mechanism that gives a proper run-time error in case incompatible plug-ins are used. Because checking the types is a potentially costly operation, we could limit this check to the root node of the search tree.

Lines of Code

To give an idea of the amount of code that is involved, OpenSolver itself, without any plug-ins consists of roughly 5,000 lines of C++ code. The plug-ins used for the experiments in this thesis comprise some 17,500 lines, including a few hundreds of lines of lex and yacc specifications, and 3,500 lines of peripheral programs and scripts were used.

3.4.1 Software Composition

It is tempting, but wrong to characterize the composition of branch-and-propagate constraint solvers, as supported by OpenSolver, as component-based software engineering. This would suggest that we aim for reuse of the plug-ins, while instead, we aim for reuse of the design of the solver. This is the case for object-oriented frameworks in general. In the first place, most plug-ins are of an inherent simplicity that would not make them valuable components. The challenge in designing a configurable constraint solver is in the definition of the interfaces, rather than in programming, for example, atomic arithmetic constraints. Secondly, instead of incorporating the more complex operators as third party components, OpenSolver promotes the composition of such operators from basic facilities, if possible. In the next chapters we will see several examples of this. Apart from this, external code can likely always be taken on board by wrapping it up as a plug-in. In the case of an autonomous application, OpenSolver could communicate with this application through a plug-in that acts as a ***proxy***, although we have made no particular provisions for this form of software composition.

In contrast, OpenSolver itself forms a versatile software component. Through the coordination layer plug-in it can be adapted to various computing environments. Because it is a stand-alone application instead of a library, it poses no restrictions on the programming languages used in these environments. In addition, the configuration language allows for external manipulation both of solvers and CSPs. We will see examples of the use of OpenSolver as a software component in Chapters 7, 8, and 9.

We are convinced that modules or classes are the right units of composition for realizing a branch-and-propagate constraint solver. As we will argue in Chapter 9, using coordination languages here involves too much overhead. Modules implementing ***abstract data types*** might allow for an easier implementation of a plug-in system, because they avoid the problem with name mangling that we discussed on page 59. They may also be a bit more efficient because some overhead is involved with calling virtual functions. However, we used C++ classes because of the language support for defining interfaces. Although we did not perform experiments, we expect that with modern compilers, the overhead for calling virtual functions is limited, and the implementation of a plug-in system based on dynamic loading was not a priority in our experimental setup. However, for a commercial system, C modules, with some rigorous scheme to enforce compliance with interfaces for the different categories of components, may be preferable.

3.4.2 Comparison with Other Systems

Several other object-oriented approaches to constraint programming exist. Most of them have extendable class hierarchies with inverse control, and can therefore be classified as object-oriented frameworks as well. To put our work into context, we compare OpenSolver to the following systems.

ILOG Solver. A commercially available C++ library for constraint programming. Some of its features are support for integer, floating-point, and set variables, an extensive collection of built-in constraints, and support for both tree search and local search [Ilo01]. The class hierarchies for constraints and search procedures can be extended. ILOG Solver is part of the ILOG Optimization suite, in which it can cooperate with the CPLEX mathematical programming engine.

Koalog Constraint Solver (KCS). A commercially available JAVA library for constraint solving on Boolean, integer and set domains, supporting tree search and local search [KoaA, KoaB]. The library can be extended with new constraints, search strategies, and solvers.

Elisa. A C++ library that offers a framework for integrating solvers in applications [Eli04]. The facilities offered by the 1.0.3 version of the system are focused on solving constraint on the reals through branch-and-propagate tree search, and include a large collection of consistency algorithms for such constraints. In addition, the Elisa class hierarchy can be extended with respect to many aspects of constraint solving, including domain types, constraints, reduction operators, and local search. Elisa is distributed under the GNU Lesser General Public License.

Disolver. A C++ library that offers a constraint-based optimization engine, with support for parallel and distributed (DisCSP) search [Ham05]. New constraints, and tree search and local search procedures can be defined by users. Disolver is reported to have been used for solving large industrial problems.

Figaro. A C++ library for finite domains constraint solving [HMN99]. To our knowledge, Figaro is the only system that is configurable with respect to state restoration policy (see Section 4.2).

EasyLocal++. A C++ library that provides a framework for realizing local search algorithms [DGS03].

Localizer++. A C++ library for local search [MVH01]. It is intended to make the facilities that are usually found in modeling languages (such as Localizer [MVH00]) available inside a mainstream programming language, to facilitate the integration in larger software projects. The library is extendable with respect to constraints, invariants (a modeling tool), and search strategies.

Despite the extendable class hierarchies, ILOG Solver, KCS, Disolver, and Localizer++ are primarily object-oriented ***toolkits***, meaning that some specific functionality (in this case, constraint solving) is made available to the developers who use the toolkit. Instead of a constraint solving toolkit, OpenSolver is a configurable branch-and-propagate tree search engine, implementing only the core algorithms of such toolkits. As such, the aspects of constraint solving that are configurable in OpenSolver are at a lower level than those that are configurable in toolkits. For example, new constraints can be added to the toolkits, while the notion of a constraint does not exist in OpenSolver: it only knows about reduction operators. Also the toolkits typically provide high-level modeling facilities such as arrays. In our case, these would be provided by a modeling environment that uses OpenSolver as its solving engine.

Being a configurable branch-and-propagate tree search engine, OpenSolver gives control over aspects of constraint solving that have hard-wired solutions in toolkits. Especially, toolkits typically fix the implementation of the domain types and the constraint propagation algorithms. This makes a constraint solver that uses OpenSolver as its solving engine a much more flexible system. Because it aims at black-box composition of constraint solvers through a configuration language, altering some aspects of the solving engine of an OpenSolver-based system should only rarely involve C++ programming. Instead, such a system could offer a menu of available options, which the user can combine at will. In this respect, OpenSolver is quite similar to the Elisa library, which is also highly extendable and configurable. To reflect that the system is a white-box framework that aims at black-box composition of solvers, the Elisa distribution contains both a programmer's manual and a user's manual.

Figaro is configurable on one very low-level aspect of a constraint solving (tree search) algorithm, namely the state restoration policy [CHN01]. In OpenSolver, the state restoration policy is fixed (see Section 4.2), and in retrospect, this limits its applicability for search tasks that rely on a specific, and different policy (see Section 4.4). However, the state restoration policy is closely related to the variable domain type implementation, and making both aspects configurable is not straightforward, and the impact should carefully be assessed.

Finally, the approach of EasyLocal++ is also similar to ours, in the sense that it is a framework, aiming at reuse of the basic solving algorithm structures. Apart from implementing a different solving algorithm (local search instead of tree search), EasyLocal++ is a white-box framework, where composition of a local search solver always involves C++ programming.

While similarities exists with each of these systems, OpenSolver is unique in providing a highly configurable and versatile solving engine, that is also an autonomous application. Aiming for black-box composition has led to an inherently linguistic approach, where every plug-in needs to be able to interpret and generate textual specifiers. This gives possibilities for external manipulation of CSPs and solver configurations that we have not found in any other system. An API would be a valuable extension, and we already use a rudimentary form of such an interface in Chapter 7, but this would not change the autonomous nature of the system. Building a system around OpenSolver is primarily a matter of ***exogenous coordination***. We will see examples, and discuss further possibilities of this approach in Chapters 8 and 9.

3.5 Summary

In this chapter we introduced OpenSolver, an open-ended constraint solver that is based on branch-and-propagate search. We discussed it from two perspectives:

Composing constraint solvers. This is a matter of black-box composition, by selecting a combination of plug-ins in seven categories that correspond directly or indirectly to the aspects of branch-and-propagate search identified in the summary of the previous chapter. OpenSolver is an autonomous application. By means of a plug-in in a special, eighth, category for its ***coordination layer***, it can be adopted to different software environments.

Implementation. OpenSolver is an object-oriented framework: it provides only basic mechanisms and data structures. The plug-ins, which are implemented as subclasses of abstract classes of the framework, determine the functionality. Because the coordination layer plug-in has tight control over the solving process, this facilitates several forms of exogenous coordination, allowing that OpenSolver is used as a software component for composing constraint solvers beyond a sequential branch-and-propagate tree search.

If we return to the concluding remarks at the end of the previous chapter, we now have an implementation of the model of constraint solving described there, with some additional features such as the coordination layer. Our next goal is to evaluate this implementation, from both the perspective of efficiency and the perspective of composing constraint solvers, on a number of standard and more advanced constraint solving techniques. This is done in the following six chapters, where Chapter 7 is a turning point, because there we move from composing constraint solvers within the OpenSolver framework to composing constraint solvers from several OpenSolver instances. At that point, our attention shifts from the seven categories of constraint solving plug-ins to the coordination layer mechanism.

Chapter 4

Applications

This is the first of three chapters that demonstrate how OpenSolver can be configured for specific application domains. In this chapter, we introduce the facilities for constraint solving on finite domains, Booleans, and real-valued variables, as well as some domain-independent plug-ins for constraint propagation and search. These are basic facilities that are available in many other systems. Together with the previous chapter, this forms a description of OpenSolver as a basic constraint solver. Of the other two chapters in this “applications” series, Chapter 5 deals with constraints on integer interval variables. These are also available in many other systems, and could therefore have been included in this basic facilities chapter, but because of the lengthy analysis, a separate chapter was devoted to these constraints.

The research question that underlies this chapter is whether the framework that we introduced and implemented in the two previous chapters is suitable for composing five particular solving techniques that would normally be hardwired in solvers: limited discrepancy search, best-first search, no-good recording, non-chronological backtracking, and applying domain reduction functions for decomposed arithmetic constraints in an order that respects their hierarchical relationships. These techniques are discussed in sections 4.1.2, 4.3, 4.4, and 4.5.

4.1 General-Purpose Facilities

4.1.1 Constraint Propagation

Three ***scheduler*** plug-ins are currently available in OpenSolver. These are general-purpose facilities for constraint propagation that can be used with any set of reduction operators. The schedulers apply the propagation functions of reduction operator plug-ins, so they control only the execution of propagation operators and optimization operators. Branching operators are left untouched.

A Basic Scheduler

The `BasicScheduler` plug-in implements the following round robin scheduling strategy, which also forms the basis for the AC-1 algorithm for computing arc consistency (see, e.g., [Dec03]): apply all operators in sequence, and keep doing so until a full sequence passes in which no variable domains are modified. This strategy can be expressed as an instantiation of Algorithm 2.1, having

$$update(F, D, D') := \begin{cases} F & \text{if } D' \neq D \\ \emptyset & \text{otherwise} \end{cases}$$

for the *select* function, we would need to introduce an extra variable that maintains the sequence number of the last operator that has been applied.

The plug-in does not use the specifier string of the `SCHEDULER` statement, so the default scheduler is replaced by the basic scheduler using the following command in the OpenSolver input language:

```
SCHEDULER BasicScheduler { };
```

Variable-Based Scheduling

The variable-based scheduler maintains a ***queue*** of modified variables. It keeps removing variables off the front of the queue, and for each variable that is removed, all reduction operators that have the variable in their input scheme are applied in sequence, in the order in which they are introduced in the solver configuration script. If the application of an operator modifies the domain of a variable, this variable is enqueued, unless it already is in the queue. By default, an operator is deactivated if it signals that it has become redundant in the current branch of the search tree. As explained in Section 3.2.3, this can have significant costs in terms of storage, and the variable-based scheduler can be made to ignore this information as follows:

```
SCHEDULER VariableScheduler { ignore };
```

The variable-based scheduler implements the scheduling strategy described in the ILOG Solver 5.1 reference manual. A potential disadvantage of this scheduler is that if a reduction operator depends on more than one, say two, variables, and both these variables have been enqueued, the DRF will be applied twice also if no relevant changes are made in between dequeueing the two variables.

Operator-Based Scheduling

In correspondence with Algorithm 2.1, the default scheduler plug-in explicitly maintains a set of reduction operators that still need to be applied. This set is implemented as an array, containing a bit per operator that is used to mark the operator as being scheduled for application. Like the basic scheduler introduced above, it ***cycles*** through the operators, applying those that are marked, and resetting their marks. If a variable is modified, the operators that have this variable in their input scheme are marked, but the operator-based scheduler will not reschedule idempotent DRFs that have modified the domain of a variable in their input scheme.

The operator-based scheduler maintains a ***counter*** of operators that still need to be applied. In principle, it keeps cycling through the sequence of operators until this counter reaches zero, but many aspects of this scheduler's operation can be configured through the definition of a schedule, as explained below.

Defining a Schedule

A ***schedule*** specifies the order in which the operators are considered for application, as an alternative for cycling through the sequence. This is not the same as the order in which they are applied: actual application of a reduction operator also depends on if it has been marked. In principle, a schedule is a list of (zero-based) operator indices. These indices refer to the sequence of DRF statements in a configuration script. The scheduler traverses the list, and at the end it terminates, reporting to OpenSolver whether a fixed point was reached or not. In the latter case, the node of the search tree where the scheduler was applied remains in the set of nodes that are subject to constraint propagation, and the scheduler will be called again for this node, at a later stage.

Parts of a schedule can be enclosed in brackets, to indicate a fixed point computation. When entering a pair of brackets during the execution of a schedule, the scheduler will keep executing this subschedule until a common fixed point is reached for the operators[1] that are pointed to from within the brackets. Typically, the entire schedule is enclosed in brackets, to indicate that the scheduler does not return control to OpenSolver before a fixed point it reached, but this facility can also be used to group several operators, and to implement priority schemes.

Two kinds of brackets can be used, for two different ways of performing the computation of the fixed point.

- Curly brackets specify that the scheduler ***cycles*** through the enclosed sequence. For example, the following statement specifies that the scheduler considers the first five operators in sequence, but does not continue with operators 5 through 9 until a fixed point of operators 10 through 14 has been reached.

```
    SCHEDULER ChangeScheduler {
       schedule = {0,1,2,3,4,{10,11,12,13,14},5,6,7,8,9} };
```

[1]Scheduler plug-ins coordinate the computation of a fixed point of the domains extensions of DRFs (see Section 2.3.1) implemented by the propagation functions of reduction operator plug-ins. Where this does not lead to confusion, we will sometimes refer to such fixed points as fixed points of a set of reduction operators.

This fixed point is computed by cycling through the sequence 10,11,12,13,14, when considering to apply these operators. Likewise, after considering to apply operator 9, the scheduler returns to the beginning of the top-level schedule and considers to apply operator 0.

- Parentheses specify that instead of cycling, each time that a change is made, the scheduler ***restarts***, and returns to the beginning of the sequence that they enclose. This is used for priority schemes, e.g., do not execute computation-intensive operators when less computation-intensive operators are scheduled for execution. Suppose that we have three groups of operators, with indices 0-4, 5-9, and 10-14, having increasing computational costs. The following schedule specifies that we do not execute the expensive operators before a fixed point of the easy operators has been reached, and as soon as an expensive operator modifies the domain of a variable, we first compute the fixed points of the less computation-intensive functions again.

```
SCHEDULER ChangeScheduler {
   schedule = (({0,1,2,3,4},5,6,7,8,9),10,11,12,13,14)
};
```

In addition, the top-level schedule can be enclosed in square brackets. This specifies that the schedule is executed once, as if no brackets were used at all. Likely, this does not lead to a fixed point, but the scheduler will still signal that constraint propagation has terminated. This is useful when we want to enforce a limit on the number of applications of very expensive reduction operators.

Like the variable-based scheduler, the specifier for the `ChangeScheduler` plug-in can be prefixed with the keyword `ignore` to save the memory costs of taking into account that certain DRFs have become redundant. In case both `ignore` and `schedule` are used, these are separated by a comma. The full syntax of the `ChangeScheduler` specifier language is given in Figure 4.1.

The operator-based scheduler allows for the composition of constraint propagation algorithms, similar to the approach proposed in [GM03]. It implements two of the three composition operators proposed there: sequence and closure. The third, decoupling, entails that several operators are evaluated independently on the same domains, and that the results are combined by intersection. The language of Figure 4.1 and the implementation of the operator could well be adapted to support this third mode of composition, but we did not need it for our experiments. Moreover, it is unlikely that the decoupling could be implemented efficiently for OpenSolver domains in general, because it would require making working copies of domains during constraint propagation, which is an expensive operation.

$$
\begin{array}{lcl}
\langle \textit{specifier} \rangle & \rightarrow & \epsilon \\
 & | & \texttt{ignore} \\
 & | & \langle \textit{schedule} \rangle \\
 & | & \texttt{ignore},\ \langle \textit{schedule} \rangle \\
\langle \textit{schedule} \rangle & \rightarrow & \texttt{schedule} = \langle \textit{subschedule} \rangle \\
 & | & \texttt{schedule} = [\ \langle \textit{sequence} \rangle\ ] \\
\langle \textit{subschedule} \rangle & \rightarrow & (\ \langle \textit{sequence} \rangle\ ) \\
 & | & \text{“\{”} \langle \textit{sequence} \rangle \text{“\}”} \\
\langle \textit{sequence} \rangle & \rightarrow & \langle \textit{schedule\_step} \rangle\ \{,\ \langle \textit{schedule\_step} \rangle\ \} \\
\langle \textit{schedule\_step} \rangle & \rightarrow & \langle \textit{Integer} \rangle \\
 & | & \langle \textit{subschedule} \rangle
\end{array}
$$

Figure 4.1: Syntax of the specifier for the DRF-based scheduler

An Assembly Language

Something that actually can go wrong with schedules is that if an index is omitted, the corresponding operator will not be executed, and a fixed point cannot be reached. In most scenarios, this leads to an infinite loop in the constraint propagation phase for the root of the search tree. This situation can easily be avoided by having the scheduler verify that all indices are present in the schedule. If not, a run-time error can be produced, or the scheduler could fall back on a default schedule for the omitted operators. More interesting than a possible solution is the fact that such situations may occur. Erroneous schedules are one example, the inter-category dependencies that we discussed in Section 3.4 are another.

Because configuring OpenSolver is error-prone, in most cases this is done by other programs. Therefore, it makes sense to consider the language of Figure 3.2 an ***assembly language***, now used to configure our abstract branch-and-propagate tree search engine instead of a CPU. Assembler is usually generated by compilers that bridge the semantic gap to a higher-level programming language, and likewise our configuration language is usually produced by peripheral programs that complement OpenSolver to form a constraint solver with a proper user interface. As an example, for the experiments reported in Chapter 5 we used a two-stage translation of arithmetic constraints into OpenSolver configuration files. The programs involved in this translation are responsible for generating correct schedules for the operator-based scheduler. Drawing the analogy with assembly languages further, for a coherent set of plug-ins it could be considered to develop a compiler for a modeling language such as OPL [VH99].

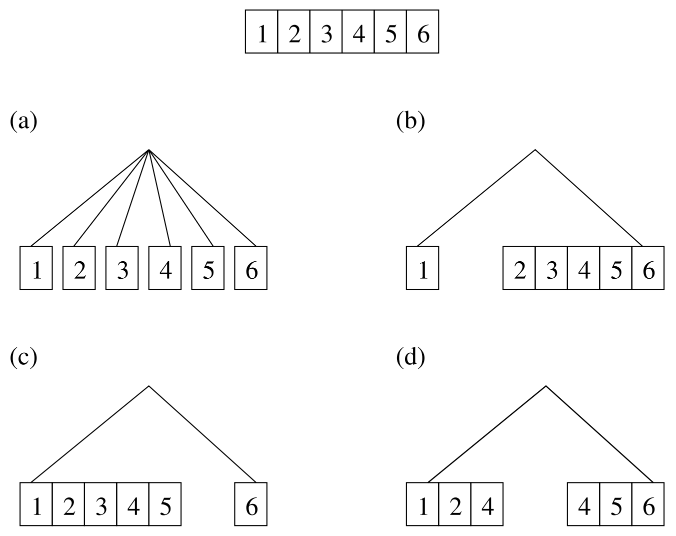


Figure 4.2: Four value selection strategies supported by `DiscreteDomain`: (a) enumeration, (b) left-enumeration, (c) right-enumeration, and (d) bisection

4.1.2 Search

Basic Strategies

Recall from Section 2.3.2 that search involves branching and traversal. Basic branching strategies can be defined by a combination of a variable selection strategy and a value selection strategy. To start with the latter, as we discussed in Section 3.2.1, the variable domain type plug-ins implement basic value selection strategies. For example, in our finite domains implementation, domains can be split in several different ways. These are illustrated in Figure 4.2, for the example domain $\{1, 2, 3, 4, 5, 6\}$.

Two reduction operator plug-ins, `FailFirst` and `RoundRobin` complement the basic value selection strategies of the domain types with a variable selection strategy to form complete branching strategies.

Fail-First. The `FailFirst` plug-in implements the fail-first variable selection strategy, discussed in Section 2.3.2. The specifier for this plug-in lists the variables that the strategy is applied to, preceded by an (integer) indication of the desired value selection strategy, for example:

```
DRF FailFirst { 0, x1, x2, ... };
```

This specifies that from the listed variables `x1`, `x2`, ..., we select a variable that reports the smallest domain size greater than one. If several such variables exist, the first one in the list is selected. The integer value 0 specifies how to create the subdomains for this variable, but interpretation of this value depends on the domain type of the selected variable. For example, if the selected variable is of type `DiscreteDomain` (see Section 4.2), then 0 specifies that the subdomains are generated through enumeration, 1 specifies left-enumeration, 2 specifies right-enumeration, etc. As another example, for `RealInterval` variables (see Section 4.5), only bisection has been implemented, but the integer argument is used to specify the order in which the subdomains are generated. This allows for easy switching between leftmost-first and rightmost-first traversal.

The `FailFirst` plug-in also implements a strategy that is sometimes referred to as ***fail-last***: select the variable with the largest domain. In either mode, as an alternative, the search for the variable with the desired domain size can be started from the middle of the list outwards. The same effect could be achieved by a permutation of the list of variables. Fail last, and the alternative search for the smallest or largest domain are specified by prefixing the specifier string (with a letter `l` and/or `m`, respectively).

Round Robin. The `RoundRobin` plug-in tries to branch on all variables in turn. Starting from a sequential traversal of the variables in the specifier string, it selects the variable with domain size greater than one that has least recently been selected. Therefore, at every node of the search tree we need to remember the index of the variable that was split to create it. For this purpose we use an annotation with a single integer.

Program 4.1 shows a typical configuration for round robin search. The first line installs the integer annotation. The value 0 in its specifier string is the initial value, which applies to the root node of the search tree. In this case, it indicates that the first variable in the `RoundRobin` specifier string, `x1`, is to be split first. If the `IntegerAnnotation` plug-in instance is not present, `RoundRobin` will just select the first variable that can be split, resulting in a chronological variable selection strategy. The specifier string for the `RoundRobin` plug-in itself is similar to that for `FailFirst`, discussed above, so again the leading zero identifies a specific, but domain type dependent value selection strategy.

```
ANNOTATION IntegerAnnotation { 0 };
VARIABLE x1 IS ...
VARIABLE x2 IS ...
...
DRF ...
...
DRF RoundRobin { 0, x1, x2, ... };
```

Program 4.1: Skeleton of a configuration for search based on a round robin variable selection strategy

The ***default traversal strategy*** is to maintain the search frontier as a stack, resulting in a ***depth-first*** search. Constraint propagation runs to completion, in a single node of the search tree. Below we will discuss an alternative traversal strategy called limited discrepancy search, and a variant of this strategy that was implemented for OpenSolver.

Limited Discrepancy Search

Limited discrepancy search (LDS, [HG95]), can be used when a good heuristic, in the form of a value selection strategy, is available to guide the search. The idea is that the heuristic will make only a few mistakes when assigning values to variables. When reaching the first node of the search tree that is a failure (the leftmost path in the tree), LDS first tries all alternatives that make exactly one different decision. In a binary search tree, this corresponds to all paths that follow the right branch in exactly one internal node, and the left branch everywhere else. If these new attempts all fail, LDS continues by trying two deviations from the leftmost path, and so on, gradually increasing the number of deviations, or ***discrepancy***, until all alternatives have been explored.

LDS can be effective for single-solution search, and for optimization schemes, such as branch-and-bound (see Section 5.9.2), where it may find better suboptimal solutions and achieve stronger pruning than a regular depth-first search. For a purely combinatorial all-solution search it will not improve on any other traversal strategy.

The straightforward implementation of LDS in OpenSolver uses an integer annotation to record the discrepancy of each node of the search tree. Just like the `RoundRobin` branching operator annotates every node of the search tree with the index of the least recently selected variable, the branching operator for LDS search could maintain the discrepancy annotation. Complemented with a container plug-in that keeps the nodes in the search frontier sorted on the basis of their (integer valued) annotation, we can explore the nodes with the smallest discrepancy first.

The problem with this implementation is that while exploring the set of nodes that have discrepancy n (called the n-th ***wave***), we will also be generating nodes of a higher discrepancy ($n+1$, and higher values in case of a non-binary value selection strategy). The size of the search frontier that we accumulate before starting the next wave is exponential in the size of the problem. To see this, consider the search tree for a problem with n binary variables. After processing all nodes of discrepancy $d-1$, the search frontier consists of those nodes of discrepancy d that are right branches. Let r_d denote the number of such nodes. In the worst case (no pruning), the tree consists of $2^{n+1}-1$ nodes, 2^n-1 of which are right branches. With n binary variables, there are $n+1$ possible discrepancy values, so if all nodes were evenly distributed over the discrepancies, there would be $(2^n-1)/(n+1)$ nodes of a given discrepancy. This is not the case: there is, for example, only one node of the highest discrepancy $n+1$, so $(2^n-1)/(n+1)$

is a lower bound for the largest r_d in $r_0, \ldots, d_n$. This fraction is exponential in n, and as a result, even though in practice constraint propagation will prune a large part of the search space, the size of the search frontier is bounded only by an exponential function, and this implementation is too memory intensive.

Memory-Bounded LDS

The root of the problem with LDS is that OpenSolver explicitly maintains the search frontier. The problem is even more severe because OpenSolver is a ***copying-based*** solver (see also the next section) where the nodes in the search frontier are full copies of their parents, with minor modifications. LDS was originally formulated as an ***iterative*** algorithm: for increasing discrepancy values, a search procedure is called that performs a depth-first search, pruning all nodes that exceed the discrepancy value. This iterative algorithm does not suffer from the memory overhead, but a large amount of work is potentially repeated: for a purely combinatorial problem, the last iteration of an exhaustive search is a full depth-first exploration.

However, LDS is used primarily for constrained optimization problems. In this case, if we want to spend only a limited time on searching, and a trust-worthy value selection strategy exists, iterative LDS was demonstrated to find better suboptimal solutions than backtracking [HG95]. We have not found any results in the literature indicating that iterative LDS outperforms chronological backtracking for a full best-solution search.

There are several ways to implement an iterative scheme in OpenSolver, but all of them are slightly artificial. One that we tried is based on nested search (see Chapter 7): a branching operator enumerates increasing allowed discrepancy values. For each value, by means of nested search we solve a full CSP in which a constraint has been posted that actively prunes away the nodes with discrepancy values that exceed the current maximum. For this implementation, some special-purpose reduction operators were needed to maintain the discrepancy information in regular CSP variables, in order that it becomes amenable to constraint propagation.

The experiment for which we used LDS (see Section 7.5.4) involves a full best-solution search for an optimization problem. While good solutions were found early compared to depth-first search, the stronger pruning did not outweigh the duplicate work, and overall performance did not improve. Therefore, instead of the iterative implementation we used a variant of the straightforward implementation discussed above: store the entire search frontier, and explore the nodes with the smallest discrepancy annotation first. If, as a result of branching, the size of the search frontier exceeds a certain threshold, switch to depth-first search to clean up the search frontier. Only when the size of the search frontier drops below a second, lower threshold, resume the discrepancy-based traversal.

For this scheme, which we refer to as ***memory-bounded*** LDS, the following facilities are needed:

- an annotation plug-in `LDSAnnotation` that allows us to maintain both the depth and the discrepancy of a node in the search tree,
- a branching operator `Discrepancy` to annotate the nodes,
- a container `MBLDSStack` that switches between the two modes of traversal.

Program 4.2 shows a typical configuration file for memory-bounded LDS, using fail-first as a variable selection strategy. The specifier string for the `MBLDSStack`

```
FRONTIER MBLDSStack { 1024, 10240 };
ANNOTATION LDSAnnotation { 0, 0 };
VARIABLE x1 IS ...
VARIABLE x2 IS ...
...
DRF ...
...
DRF Discrepancy { FailFirst { 0, x1, x2, ... } };
```

Program 4.2: Skeleton of a configuration for memory-bounded LDS search

container plug-in consists of the threshold sizes that trigger switching traversal modes. In the current implementation, these are numbers of stored nodes. The actual amount of memory occupied by a node depends on many factors, notably the number of variables. Therefore an implementation of the container that measures actual memory usage would be preferable.

Memory-bounded LDS is not as robust as iterative LDS, because potentially it could be doing a depth-first traversal most of the time. For our experiments, this did not happen, and we were able to exploit the advantages of LDS for a full best-solution search, in a copying-based solver.

Adapters

The specifier string for the `Discrepancy` plug-in contains an identifier-specifier pair for another branching operator. Internally, it actually creates this other plug-in instance, for which it serves as a wrapper. The branching is performed by the inner operator. `Discrepancy` only annotates the resulting nodes with the correct depth and discrepancy information.

Plug-ins like `Discrepancy` are called ***adapters***. They are used to make minor modifications to the functionality of other plug-ins, usually in the same category, or to combine the functionality of several such plug-ins. We will see more examples of adapters in this thesis.

4.2 Finite Domains

Many combinatorial problems are naturally expressed as a CSP using $\mathcal{Z}$, the set of all finite sets of integers, as a variable domain type. The variables of such CSPs are usually called ***finite domains*** variables, because of the nature of the domains in $\mathcal{Z}$, but a more important property is that a representation exists for all possible subsets of a variable's original domain.

Plug-ins

The OpenSolver plug-in for finite domains is called `DiscreteDomain`. It is activated as follows:

`VARIABLE` *identifier* `IS DiscreteDomain` { *specifier* }`;`

where *identifier* is the name of the variable, and *specifier* is a sequence of integer ranges.

The following constraints are available for `DiscreteDomain`:

- The binary disequality constraint $x - y \neq c$, where c is an integer constant. It is implemented by a plug-in `DDNEQ` that works exactly as explained in Example 2.3.2 on page 19. Its use is demonstrated by Program 4.3 on page 77.
- The constraint $\langle x, y \rangle \in T$, where T is some set of allowable tuples. It is implemented by the `BinaryConstraint` plug-in. The following example shows its use.

```
DRF BinaryConstraint { <q1,q2> IN {
        <1,3>,<1,4>,<2,4>,<3,1>,<4,1>,<4,2> } };
```

The latter plug-in uses only a very crude algorithm and data structures for verifying that values in one domain are supported, through the list of allowed tuples, by a value in the other domain. Values that are not supported are removed from the domains. State-of-the-art algorithms for enforcing arc consistency try to minimize the number of ***support checks***, and will be more efficient than our implementation. See notably the work of Van Dongen, e.g., [vD02].

As we mentioned at the end of Section 3.2.2, a set of reduction operators can maintain a protocol to distinguish different kinds of modifications of domains. This is used by the reduction operators for finite domains variables: `DDNEQ` and `BinaryConstraint` distinguish between the following events:

- changing the bounds of a domain
- deleting a value that is not a bound
- reducing a domain to a singleton set

DDNEQ can make a change only if the value of one of the variables that it is applied to is fixed, so its application is triggered only if a domain becomes a singleton set. BinaryConstraint is triggered by all three events, and the first event, a change of the domain bounds, could be used to trigger a reduction operator that links a DiscreteDomain variable to a variable whose domain has an interval representation.

Protocols regarding modifications to variable domains can be implemented in two ways (see also Program 3.2 on page 58):

- By setting bits in an unsigned integer array element for each argument of a reduction operator. A complementary bitmask per argument is used to see if a particular operator needs to be scheduled after a domain has been modified.

- If the bitmask does not match (e.g., because it is set to all-zeros), a second check is performed to see if a DRF wants to be scheduled for a particular value of the unsigned integer. This can be used for a different encoding scheme, using an enumeration of possible cases, when the 32 bits offered by the bitmask are insufficient to encode all changes that we are interested in.

For the finite domains reduction operators we used the first implementation.

Efficiency of OpenSolver

The unavoidable benchmark problem for finite domains constraint solving is the ***n*-queens problem**: place n queens on an $n \times n$ chess board in such a way that they do not attack each other. This can be formulated as a CSP as follows:

$$\begin{aligned}&\langle\ q_i \neq q_j,\ q_i - q_j \neq j - i,\ q_i - q_j \neq i - j,\ \text{for } 1 \leq i < j \leq n\ ;\\ &\quad q_1, \ldots, q_n \in \{1, \ldots, n\}\ \rangle\end{aligned}$$

Any solution to the n-queens problem will have exactly one queen per column (and row) of the chess board. This constraint is inherent to our CSP formulation, which uses a variable per column, indicating the position of the queen in that column. The constraints of the CSP state that no two queens can be on the same row or diagonal, in either direction.

Other CSP formulations for the n-queens problem exist, notably one where the above $\frac{3}{2}n(n+1)$ constraints are replaced by 3 all_different constraints, one for the row constraints, and one for both diagonals. The all_different constraint entails that different values are assigned to all variables that the constraint applies to. It can be applied to an arbitrary number of variables, and specialized algorithms, beyond regular constraint propagation exist for enforcing it. Constraints that have these properties are sometimes called ***global constraints***.

Because the specialized algorithms used for processing global constraints run counter to the compositional approach that we experiment with, no global constraints have been implemented for OpenSolver. There are, however, no limitations for doing so, and plug-ins for global constraints would make valuable additions. Typically, global constraint processing algorithms require that additional information about the CSP is maintained during constraint propagation and search. Because the state of reduction operator plug-ins is global, this information needs to be maintained elsewhere, for example in auxiliary variables of a special-purpose domain type (see Chapter 6).

While other CSP formulations of the n-queens problem exists, because of its scalability and simplicity the above formulation is perfectly suited for comparing the efficiency of the basic machinery of different solvers. For example we do not have to worry about the efficiency of the implementation of the all_different constraint. In Table 4.1 we report the results of a comparison of OpenSolver with ILOG Solver 5.1, on a SUN E450. We compare user time, as reported by the GNU/Linux `time` command, for counting all solutions. The configurations are similar to that of Program 4.3, and use the variable-based scheduler, which resembles the scheduling procedure that is described in the ILOG Solver manual.

```
VARIABLE q1 IS DiscreteDomain {1..4};
VARIABLE q2 IS DiscreteDomain {1..4};
VARIABLE q3 IS DiscreteDomain {1..4};
VARIABLE q4 IS DiscreteDomain {1..4};
DRF DDNEQ { q1-q2 <> 0 }; DRF DDNEQ { q1-q2 <>-1 };
                             DRF DDNEQ { q1-q2 <> 1 };
DRF DDNEQ { q1-q3 <> 0 }; DRF DDNEQ { q1-q3 <>-2 };
                             DRF DDNEQ { q1-q3 <> 2 };
...
DRF DDNEQ { q3-q4 <> 0 }; DRF DDNEQ { q3-q4 <>-1 };
                             DRF DDNEQ { q3-q4 <> 1 };
DRF FailFirst { 0, q1, q2, q3, q4 };
SCHEDULER VariableScheduler { };
```

Program 4.3: Configuration for solving the 4-queens problem

Both solvers report the same number of solutions, failures and internal nodes, from which we can conclude that the computations are comparable. The results indicate that the efficiency of the basic procedures in OpenSolver is realistic in the sense that it is comparable to that of a successful commercial solver.

				time (sec.)	
n	solutions	failures	internal	ILOG Solver	OpenSolver
4	2	4	5	0.02	0.01
8	92	292	383	0.05	0.05
10	724	4,992	5,715	0.72	0.76
12	14,200	101,882	116,081	17.61	17.16
14	365,596	2830,370	3,195,965	572.31	512.97
15	2,279,184	16,263,952	18,543,135	4757.66	3056.49

Table 4.1: A comparison of OpenSolver and ILOG Solver for n-queens

State Restoration Policy

An important aspect of a finite domains constraint solver implementation is the construction of the data structures, notably the variable domains, for the node of the search tree where search continues. This is usually referred to as the ***state restoration policy***. While hybrid methods exist, the main options are [Sch99]:

Copying When the search tree is expanded by branching, the data structures that define the current node are copied for all new nodes. These copies are then modified to construct subproblems. At potentially high memory costs, every node of the search frontier is immediately available for further exploration.

Trailing Only the current node of the traversal is maintained, but all changes (deletions of values) to the domains of variables leading up to this node are registered. Backtracking is implemented by undoing changes to reach an internal node of the search tree, from which search can progress along an alternative branch. Trailing is the predominant method used in current constraint solvers.

Recomputation Instead of unwinding a trail of changes, with a recomputation state restoration policy, the internal nodes are reconstructed from a shallower internal node by repeating a part of the traversal of the search tree. Iterative schemes such as LDS, which we discussed in the previous section, can be seen as a form of recomputation. Other forms of recomputation also exist that represent internal nodes in the search frontier by the branching decisions that need to be made to arrive at that node.

OpenSolver is copying-based, so the search frontier is maintained explicitly and the full data structures are available in every node. For finite domains this is the most memory-intensive option, but it can be justified because OpenSolver is a general-purpose solver. Interval computations, for example, use very modest data structures, and for that domain type, copying is a logical choice.

		ILOG Solver		DiscreteDomain	
n	nodes	time (sec.)	memory	time (sec.)	memory
100	508,426	528.84	12M	523.53	5992K
500	364,754	3643.97	111M	4210.80	106M
1000	996	819.93	418M	1241.24	453M

Table 4.2: Memory consumption for large n-queens instances (first-solution)

To have an idea about the memory efficiency of our solver, we ran a number of larger instances of the n-queens problem. Because of the large search space and many solutions, we did a first-solution search on these instances. For search we still use the fail-first variable selection strategy, but now starting the search for the variable with the smallest domain from the middle of the chessboard outwards. In the ECL^{i}PS^e tutorial [CHS$^+$03], this strategy is shown to give better performance for this problem. Because of the quadratic number of DRFs we let the variable-based scheduler ignore the information on redundant DRFs. Table 4.2 shows the results of these experiments. Reported memory usage is the resident set size, reported by the `top` command.

ILOG Solver uses a combination of trailing and recomputation [Per99]. These results suggest that also regarding memory consumption, in spite of using copying as a state restoration policy, OpenSolver configured as a finite domains constraint solver is fairly efficient. The difference in running times for these large instances is partly due to using input files, which have to be parsed. ILOG Solver is a library, so no I/O is involved with setting up the large number of reduction operators involved in these instances. Another, more important factor is ignoring the redundancy of reduction operators. Probably in ILOG solver this information is subject to recomputation as well, or a more efficient representation is used, for example one based on instantiated variables. The latter solution fits the specified scheduling mechanism well, and would be easy to incorporate in OpenSolver.

The default implementation of finite domains uses a bitmap, that is copied during the cloning operation. As an experiment, we implemented a plug-in `DDTrail` that uses an alternative representation, where each variable domain is represented by a linked list of deleted values. Copying a domain as a part of branching consists only of replicating a pointer to the last element of this list. The list then becomes a tree, and different changes can be made in the different branches. Reference counts are used to manage the deallocation of the trails of deleted values. During constraint propagation, upon the first membership test, the bitmap is restored to avoid having to traverse the linked list to the root of the tree many times. The idea was to exploit some of the benefits of the trailing state restoration policy in a copying-based solver. This will probably work for problems with large numbers of large finite domains variables, to which little changes are made between different nodes of the search tree (i.e., many needless copies of large bitmaps). For our experiments, this was not the case.

4.3 Best-First Search: the Knight's Tour

Having introduced finite domains, and the basic facilities for search, we now study a slightly more complex combinatorial problem that requires an advanced search strategy for efficient solving. The purpose of this section is to demonstrate how such search strategies can be composed from a selection of OpenSolver plug-ins. Being of limited practical relevance yet, the problem is to move a knight piece around an $n \times n$ chess board, in such a way that all positions on the board are visited exactly once.

The problem can be formulated as a CSP as follows: with each of the $n \times n$ locations that constitute the tour we associate a variable, whose possible values are the actual positions on the board, numbered $1..n \times n$. The variables for all $n \times n - 1$ consecutive steps are constrained such that the positions that they indicate must be reachable through a knight's move, and all variables must assume different values. As an example, for a 3×3 board, where the positions are numbered

7	8	9
4	5	6
1	2	3

and for which there obviously is no solution, the CSP is

$$\begin{aligned}\langle \langle x_1, x_2\rangle, \langle x_2, x_3\rangle, \ldots, \langle x_8, x_9\rangle \in \{&\langle 1,8\rangle, \langle 1,6\rangle, \langle 4,9\rangle, \langle 4,3\rangle, \langle 7,6\rangle, \langle 7,2\rangle, \langle 2,7\rangle,\\ &\langle 2,9\rangle, \langle 8,3\rangle, \langle 3,4\rangle, \langle 3,8\rangle, \langle 6,7\rangle, \langle 9,2\rangle, \langle 9,4\rangle\},\\ \mathsf{all\_different}&(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9)\ ;\\ x_1, x_2, \ldots, x_9 &\in \{1, \ldots, 9\}\ \rangle\end{aligned}$$

.

OpenSolver can easily be configured for solving this CSP:

- for the $n^2 - 1$ "knight's move" constraints, whose definition involves an enumeration of all positions that are linked by such a move, we can use the `BinaryConstraint` plug-in, and

- for the all_different constraint we can use $\frac{1}{2}n^2(n^2+1)$ disequalities, as we did for n-queens.

The search space is quite large though, and doing a basic fail-first search seems already intractable for $n = 8$ (but a better implementation of all_different could improve the situation).

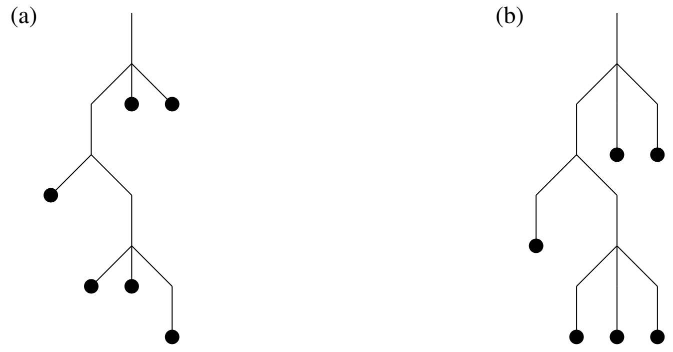


Figure 4.3: Search trees for (a) regular branch-and-propagate search and (b) our implementation of best-first search, vertical branches representing constraint propagation

Fortunately, a very good heuristic exists, which makes the problem easy to solve. It is said to have been discovered in 1823, by H.C. von Warnsdorf[2]. This heuristic dictates that for the next location of the tour, we always choose the position that has the smallest number of possibilities for moving on. It is very easy to exploit this heuristic in a program written specifically for solving the knight's tour problem. Alternatively, we could write an OpenSolver branching operator that knows about the problem, and generates subdomains for the variable holding the next location of the tour accordingly: a singleton set with the selected position, and a second subdomain containing all other alternatives. This results in a dedicated solver as well, but being based on OpenSolver, we benefit from readily available facilities like the finite domains implementation and search.

Instead of a problem-specific solution, we can aim at a more generalized approach, and try to formulate the heuristic in terms of aspects of our model of constraint solving. Observe that with enumeration value selection, the descendants of an internal node of the search tree correspond to the different alternatives for the next location of the tour. After constraint propagation, the nodes that comply with the heuristic will have the smallest total size of the variable domains. Based on this observation, we can implement the heuristic as follows.

- After branching, perform constraint propagation in all descendant nodes.
- Proceed by expanding a node for which the sum of all domain sizes is minimal.

Compared to the default, where the traversal depends on a selection from the set of nodes that are pending constraint propagation, we now make the selection from the set of nodes that are pending branching. Figure 4.3 illustrates the difference in the resulting search trees.

[2]Our source of information on this problem is the website `http://www.delphiforfun.org/`

Implementation

The search strategy that we just described is implemented in OpenSolver by three plug-ins:

- a node evaluator `AnnotateSize` that annotates the nodes of the search tree with the sum of the domain sizes,
- a container `AnnotationOrderedStack` that always has a node with the smallest integer annotation on top,
- a selector `RestrictiveBranching` that selects a single node from the "pending branching" (Figure 3.3) set only if the set of nodes that are pending propagation becomes empty.

These three plug-ins are activated by including the four lines of Program 4.4 in the configuration script. The `ANNOTATION` statement is there just to provide an initial annotation for `AnnotateSize` to use. Note that `AnnotateSize` is another example of an adapter, this one in the node evaluator category. Internally, before annotating the node, it applies another node evaluator to determine the nature (solution, failure, or internal) of a node. Here we use the `CanonicalDomains` node evaluator, which tests for ***canonical domains*** to distinguish solutions. `CanonicalDomains` is also the ***default node evaluator***, and therefore it normally does not occur in configuration scripts. We will see an alternative to the default in Section 4.5.

```
TDINFO AnnotateSize { CanonicalDomains { } };
ANNOTATION IntegerAnnotation { 0 };
INTERNAL AnnotationOrderedStack { };
EXPAND RestrictiveBranching{ };
```

Program 4.4: Activating the plug-ins that implement Warnsdorf's heuristic

Using `RestrictiveBranching` to select the nodes where the search tree is expanded by branching, the set of nodes that are pending propagation is emptied before new branches are created. The selected node is then split into a number of subproblems. Constraint propagation is applied in each of these nodes, and if this does not lead to a failure, the nodes are added to the "pending branching set." Only after all descendants have been processed, and the "pending propagation" set is empty again, a new node is selected for branching. Because branching reduces the domains, with `AnnotateSize` and `AnnotationOrderedStack` this new

30	33	26	41	16	19	24	21
27	42	29	32	25	22	15	18
34	31	40	43	46	17	20	23
39	28	53	62	57	48	45	14
54	35	56	47	44	63	58	49
3	38	61	52	59	8	13	10
36	55	2	5	64	11	50	7
1	4	37	60	51	6	9	12

Figure 4.4: Knight's tour for $n = 8$

selection will be one of the most recent additions. Globally, the search is depth-first, but at every level, constraint propagation is applied in all siblings. The resulting search tree is illustrated in Figure 4.3(b).

Warnsdorf's heuristic is a form of ***best-first search***, where a measure of quality is associated with the nodes. In this case, this measure is evaluated after constraint propagation, but this is not typical. Using the new search strategy, a solution to the knight's tour problem is found without backtracking, for all instances that we tried. Figure 4.4 shows a solution for the 8×8 problem. It is found in less than a second on our test machine, and 219 nodes are visited in the process. For $n = 18$, the search is still backtrack-free, and 1394 nodes are visited in 59 seconds. In each of these nodes, a fixed point of 52,649 DRFs is computed, 323 of which are instances of `BinaryConstraint`. Table 4.3 shows some additional information.

We should remark that our scheme is not a fully accurate implementation of Warnsdorf's heuristic. In each of the candidate nodes, constraint propagation may reduce the domains of variables that are more than a single step away, which could lead to different choices. Since the desired effect is obtained, we

		nDRF			time	memory
n	nvar	$\neq$	K.M.	nodes	(sec.)	(bytes)
8	64	2,016	63	219	0.400	3584K
10	100	4,950	99	378	1.760	8932K
12	144	10,296	143	574	5.310	20M
14	196	19,110	195	810	13.120	45M
16	256	32,640	255	1,108	29.240	89M
18	324	52,326	323	1,394	58.870	174M

Table 4.3: Statistics; K.M. is the number of "knight's move" constraints

have not investigated if this occurs in practice. Although the search is backtrack-free, a large number of nodes are visited and stored. The time spent to visit the nodes is justified because we want to evaluate the heuristic using constraint propagation. A more efficient solver can be obtained by writing a dedicated branching operator, as we discussed at the beginning of this section. Storing the nodes could be avoided by writing a generic branching operator that internally performs a round of constraint propagation to evaluate the alternatives, and then generates two branches: one for the alternative that evaluates best, and one for all other alternatives. Such a branching operator would be parameterized by a set of propagation operators, and possibly a node evaluator. It would be quite similar to the operator for nested search, discussed in Chapter 7.

It would be interesting to investigate the effect of Warnsdorf's heuristic on other constraint satisfaction problems, for example on randomly generated problems. Perhaps this could help us to identify the properties that a CSP must possess in order for the heuristic to work. The generic implementation described in this section facilitates such experiments. To our knowledge, the heuristic is not normally available in other general-purpose constraint solvers.

4.4 Satisfiability of Propositional Formulas

In this section we configure OpenSolver as a solver for checking the satisfiability of propositional formulas in conjunctive normal form (CNF). This problem is usually referred to as the SAT problem, and solvers for it are called SAT solvers.

A propositional formula in CNF is a conjunction of ***clauses***, where a clause is a disjunction of ***literals***, and literals are propositional variables (***positive*** literals), or their negations (***negative*** literals). An example is the formula

$$(x \vee y \vee \neg z) \wedge (\neg x \vee z) \wedge (y \vee z).$$

Possible values for the variables are true and false. A clause evaluates to true if at least one of its literals evaluates to true, where a negation of a propositional variable evaluates to true if the value of the variable is false. A formula is called ***satisfiable*** if there exists an assignment of values to variables for which all clauses evaluate to true.

Almost all complete solvers for the SAT problem descend from an algorithm known as DPLL [DLL62], which can be explained as a systematic exploration of all possible assignments, plus the following inference techniques:

- ***Unit propagation*** If a clause contains only a single literal, we can deduce that this literal must have the value true (the clause is said to be ***resolved***).
- The ***pure literal rule*** If a variable occurs only as one form of literal, i.e., the negation of this literal does not occur in any unresolved clause, then we can set the variable such that the literal evaluates to true.

If a literal is set to true, either as part of the exploration or as a result of inference, DPLL removes all clauses in which it occurs, and removes its negation from the remaining clauses. Once we reach an empty formula, we have established satisfiability of the original formula.

The straightforward way to configure OpenSolver as a SAT solver uses a propagation operator per clause that enforces the constraint that at least one of the variables occurring as positive literals must have the value true, or at least one of the variables that occur as a negative literal must have the value false. This corresponds to the unit propagation step of the DPLL algorithm, except that OpenSolver does not directly support the modification of constraints. However, variables whose domains have been reduced to singleton sets do not trigger any further reduction, and the reduction operators for clauses that evaluate to true can deactivate themselves in any branch of the search tree. Implementation of the pure literal rule is not so straightforward, but the SAT community seems to agree that the application of this rule does not pay off (see, e.g., [ZM02]).

Two plug-ins implement the configuration that we just outlined. Domain type Bool supports Boolean variables, and propagation operator Clause implements the constraint that at least one literal of a clause evaluates to true. The code below demonstrates the use of these plug-ins for the example problem at the beginning of this section. We could have made the Clause plug-in operate on finite domains, but for experimenting with SAT specific heuristics that are discussed below, we used a dedicated variable domain type.

```
VARIABLE x IS Bool { 0,1 };
VARIABLE y IS Bool { 0,1 };
VARIABLE z IS Bool { 0,1 };
DRF Clause { x,y;z };
DRF Clause { z;x };
DRF Clause { y,z; };
DRF FailFirst { 0,x,y,z };
```

In their basic form, the Bool and Clause plug-ins consist of 400 lines of code, and from this perspective, little effort is required to configure OpenSolver as a SAT solver that is comparable to the DPLL algorithm. However, the DPLL algorithm is only a very basic solver, and contemporary solvers can handle vastly larger sets of problems. Complete solvers for the SAT problem, such as Chaff [MMZ+01] and GRASP [MSS96] all descend from the DPLL algorithm, but augment it with the following techniques:

- a good variable and value selection strategy,
- clause learning, and
- non-chronological backtracking.

Variable and value selection strategies for the SAT problem can be implemented in OpenSolver. As an example, we implemented the DLIS (Dynamic Largest Individual Sum) heuristic [Sil99], which entails that the search tree is always expanded by assigning the value `true` to the literal that occurs in the largest number of unresolved clauses. We implemented this heuristic by decorating each domain of type `Bool`, through the specifier string, with both the number of clauses in which it appears as a positive literal, and the number of clauses in which it appears as a negative literal. When a clause is resolved, it will ask the OpenSolver scheduler to be deactivated. At the same time, it will tell the `Bool` instances to which it applies to decrease one of these two counters, depending on the sign of the variable in that particular clause. When splitting a variable of type `Bool`, it will instantiate itself to `false` in the node that is added to the search frontier last, if the counter of negative occurrences is greater than the counter of positive occurrences, and to `true` otherwise. By default, the search frontier is managed as a stack, and the most recently added alternative is explored first. As the final element of our implementation of DLIS, uninstantiated `Bool` variables report as their size 2, plus the largest of their two clause counters, and we select a variable that reports the largest size (fail-last, see 4.1.2).

Clause learning is based on the observation that each time a failure is deduced, we can derive a new clause that explicitly prevents the combination of assignments that has lead to that particular node in the search tree. Not all assignments are relevant to this failure though, and a careful bookkeeping of the changes that trigger clause resolution will allow us to isolate exactly the literals that represent the contradicting assumptions made during the search. A negation of the conjunction of these literals is itself a clause. Such a clause, or in general, a constraint that corresponds to a deduced failure is also called a ***no-good***, and maintaining these constraints is known as ***no-good recording***. No-goods carry redundant information that is hidden too deep in the problem for the solver to exploit. As such it makes sense to add a no-good to the problem that we are trying to solve, because if it had been present in the first place, the current failure would have been prevented. Moreover, because only a fraction of all assignments that lead to the failure are actually causing it, the same failure could occur again in another part of the search tree, and adding a no-good will prevent this.

Maintaining an explanation for the assignments of truth values is straightforward. This was implemented inside the `Bool` plug-in by building alongside the search tree, for each variable, a tree of data structures with leaves for branching decisions, and internal nodes for resolved clauses, linking to the nodes for the assignments that trigger the resolution. The `Clause` plug-in was made aware of this data structure, and modified to use it for deriving a no-good upon deduction of a failure. Implementation of clause learning is impeded, though, by the lack of facilities for adding DRFs during the search. This is not a design decision, it simply has not been implemented. As a work-around, we can store the learned clauses in the domain of a special-purpose variable. All `Clause` instances can access this variable, and a dedicated operator, which applies to all Boolean variables, would then actually enforce the learned constraints. A problem with this work-around is that the scheduling of the DRFs for the learned clauses then becomes the responsibility of this last `Clause` instance, and is separated from the regular scheduling mechanism.

The conflicting assumptions that lead to a failure can be stored as a no-good, for more powerful pruning in other parts of the search tree, but analyzing the conflict may also allow us to skip a part of the search tree that would normally be explored by backtracking from a failure. This is the case if the most recent assignment to a variable involved in the conflict occurs at a shallower level in the search tree than the level directly above the conflict. Techniques for analyzing, and exploiting conflicts to "jump" to a higher level in the search tree are known as ***non-chronological backtracking***, or ***look-back*** techniques. Conversely, constraint propagation can be explained as looking forward to future branching decisions). A comprehensive overview of non-chronological backtracking techniques is given in [Dec03].

While heuristics and clause learning can be implemented, the design of OpenSolver is unsuited for non-chronological backtracking. The fact that the search frontier is stored explicitly makes this difficult: the nodes that would be skipped by a backjump of more than a single level have already been created. The hierarchical information that is needed to identify the nodes that can be skipped after a backjump can be maintained in annotations, but even so there is no mechanism for discarding these nodes, and they have to be pruned away by propagation of a no-good for the conflict. Mechanisms for backjumping can be added, but the copying state-restoration policy used in OpenSolver is probably the worst alternative for a proper implementation of backjumping, because expensive copying operations are involved in creating the nodes that may be discarded later.

The clause learning scheme that we outlined above did not lead to a significant speedup of SAT solving. It would be interesting to investigate if more advanced conflict analysis techniques that may deduce more powerful no-goods, and the implementation of proper backjumping mechanisms would allow us to approach the performance of modern SAT solvers. For the latter we would also need to experiment with a trailing state restoration policy, which does not suffer from the overhead of unnecessary copying.

4.5 Real Numbers

In this section we demonstrate how OpenSolver is configured for solving constraints on the reals. The facilities for these constraints are a modification of those for arithmetic constraints on integer intervals, which are analyzed in more detail in Chapter 5. Where the same notions apply, we refer to the definitions in that chapter.

The model of constraint solving that underlies this thesis, and which we described in Chapter 2, applies to combinatorial problems, where we search for combinations of elements from the domains of the variables that satisfy all constraints. In principle these domains are finite sets, but constraints on real valued variables can be handled by considering finitely many intervals of real numbers. For this purpose, in Section 2.2.4 we introduced domain type $\mathcal{F}$, containing all intervals of reals, of which the bounds are floating-point numbers.

Here we limit ourselves to ***arithmetic constraints*** that can be written as

$$p \ op \ c$$

where p is a polynomial, c is a constant, and $op \in \{=, \leq, \geq\}$ (see Section 5.3). The way that many constraint systems handle such constraints is by decomposing them into ***atomic constraints***. For arithmetic constraints on the reals, a suitable set of atomic constraints is the following:

- addition $x + y = z$
- multiplication $x \cdot y = z$
- exponentiation $x = y^n$, for $n > 1$,
- equality $x = y$,
- disequality $x \leq y$

During the search, hull consistency (see Section 2.2.4) is then maintained for the decomposed problem. This involves the introduction of new variables. It should be noted that in general, hull consistency for the decomposed problem is weaker than hull consistency for the original problem: let $P_{decomp} = \langle \mathcal{C}' \ ; \ x_1 \in D_1, \ldots, x_n \in D_n, x_{n+1} \in D_{n+1}, \ldots, x_{n+m} \in D_{n+m} \rangle$ be the decomposition of problem $P = \langle \mathcal{C} \ ; \ x_1 \in D_1, \ldots, x_n \in D_n \rangle$. Hull consistency of P_{decomp} does not imply hull consistency of P. The following example demonstrates this property. A detailed analysis is given in [CDR99].

4.5.1. EXAMPLE. The constraints $y = x^3$, $x + y = 0$ form a decomposition of $x + x^3 = 0$. The CSP $\langle y = x^3, \ x + y = 0 \ ; \ x \in [-1, 1], y \in [-1, 1] \rangle$ is hull consistent[3], but $\langle x + x^3 = 0 \ ; \ x \in [-1, 1] \rangle$ is not. $\Box$

Constraint propagation is usually implemented by inverting the atomic constraints to isolate all variable occurrences. These inversions define ***projections*** of the domains of the other variables on the domains of the isolated variables. For example, for $x \cdot y = z$ we have

- $x = z/y$, if $y \neq 0$

[3]assuming $[-1, 1] \in \mathcal{F}$, and using $x + y = 0$ as a shorthand for $x + y = c$, with $c \in \{0\}$.

- $y = z/x$, if $x \neq 0$
- $z = x \cdot y$

The projections follow from interval extensions of the arithmetic operations. Interval extensions form the basis of ***interval arithmetic***, which is due to Moore [Moo66]. An ***interval extension*** of a function $f : \mathbb{R}^n \rightarrow \mathbb{R}$ is a mapping $F : \mathcal{F}^n \rightarrow \mathcal{F}$, such that for all $\langle D_1, \ldots, D_n \rangle \in \mathcal{F}^n$ and $\langle d_1, \ldots, d_n \rangle \in D_1 \times \ldots \times D_n$, $f(d_1, \ldots, d_n) \in F(D_1, \ldots, D_n)$. For example, ignoring bounds $+/-\infty$ and using an infix notation, the interval extension of the subtraction can be defined as follows

$$[a, b] - [c, d] = \mathsf{hull}([a - d, b - c])$$

When applied to the domains of the variables in the right-hand side of the inversions in the above example, the interval extensions of the division and multiplication operations yield a set of values for the isolated variable that do not violate the constraint. The domains of these variables are then intersected by the smallest $\mathcal{F}$ interval that contains this set.

For expressions that involve more than a single operator, an interval extension can be constructed from the syntactical form of the expression as follows.

- Every constant is replaced by the smallest $\mathcal{F}$ interval that contains it,
- every variable is replaced by an interval variable, and
- every operator is replaced by the interval extension of that operator.

This is called the ***natural interval extension*** of the expression.

A problem with the natural interval extension is that multiple occurrences of the same variable are treated as if they were separate variables. Consider for example the natural interval expression of $x - x^3$, applied to the interval $[-1, 1]$. The interval extension of the exponentiation applied to this interval yields the interval $[-1, 1]$, the set of all third powers of values in the original interval. When subtracted from the interval for the other occurrence of x, we get $[-2, 2]$, but this interval is wider than the interval of possible outcomes of $x - x^3$. In fact, it is the interval of all possible outcomes of $x - y^3$, with $x, y \in [-1, 1]$. This is called the ***dependency problem*** of the natural interval extension. It lies at the heart of the discrepancy between hull consistency for a compound constraint, and hull consistency of the decomposition, illustrated by Example 4.5.1. The decomposition into atomic constraints entails that the natural interval extension is used. Alternative interval extensions exists, having properties that may be preferable to those of the natural interval extension, in some situations. A discussion of these is outside the scope of this thesis, and the reader is referred to the extensive literature on constraints on the reals, where [CDR99] is a good starting point.

Implementation

Domain type $\mathcal{F}$ is implemented by a plug-in called `RealInterval`, of which we have already seen an example on page 40. The `RIARule` (Real Interval Arithmetic Rule) plug-in implements constraint propagation. The syntax is

$$\texttt{DRF RIARule } \{x\texttt{\^{}}n\texttt{*}(m)\ op\ p\}\texttt{;}$$

where x is a variable, n is an integer, m is a monomial, p is a polynomial with integer coefficients, and $op \in \{\texttt{=}, \texttt{<=}\}$. The plug-in instance will evaluate the natural interval extension of the expression $\sqrt[n]{p/m}$, and update the domain of x with the hull of the resulting interval, according to operator op.

Recall from Section 2.2.4 that the hull of a set of reals is the smallest floating-point interval that contains the set. The plug-ins `RealInterval` and `RIARule` are implemented using the ***gaol*** library [Gou], which supports interval arithmetic based on floating-point intervals. Computing with floating-point numbers entails that potentially, a rounding error is made. In gaol, the hull of the outcome of interval arithmetic operations is calculated by ***outward rounding***, i.e., the lower bound is rounded to the greatest floating-point number smaller than, or equal to the actual value for the lower bound, and the upper bound is rounded to the smallest floating-point number greater than, or equal to the actual value for upper bound.

The `RIARule` plug-in can compute more complex expressions than needed for the inversions of the atomic constraints. Formally, as soon as we allow more than a single interval arithmetic operation per projection function, more than a single rounding error can be made, and it becomes unclear what level of consistency we are computing. For example, we could allow arbitrary linear constrains as atomic constraints, as we do for the integer case in Section 5.7. Consider then the constraint $x+y+z = w$. When we evaluate $\mathsf{hull}(D_x+D_y+D_z)$, the hull of the interval that contains all possible sums of an element from D_x, D_y, and D_z each, we have three options for which two intervals to add first. Because floating-point addition is non-associative, we would be computing the hull of a decomposition that has a new variable added for either $x + y$, $x + z$, or $y + z$. Because of the accumulated rounding errors, this interval can be larger than the proper hull of $D_x + D_y + D_z$. What is worse, different inversions likely correspond to different decompositions, and the level of consistency is no longer clearly defined. Even though there is no reason to expect that this will be a problem in practice, we therefore prefer to use `RIARule` only for inversions of ***atomic*** constraints.

It is interesting to compare our approach with the HC4 algorithm [BGGP99], which employs a single reduction operator for a compound constraint. Internally, this operator decomposes the constraint into atomic constraints, and enforces hull consistency for this decomposition. Besides not having to introduce new variables for the decomposition, the HC4 operator is very efficient because it applies the inversions of the atomic constraints in a sequence that respects their hierarchical relationship. For example, if the constraint $2x = z - y^2$ is decomposed into

$$\begin{array}{ccc} & t_1 = t_2 & \\ t_1 = 2x & & t_2 = z - t_3 \\ & & \quad t_3 = y^2 \end{array}$$

then the HC4 algorithm first updates the domains of the (internal) variables t_3, t_2, and t_1 in a ***forward evaluation phase***. Then it enforces the top level constraint $t_1 = t_2$, and traverses the decomposition in the opposite direction in a ***backward propagation phase*** to update the domain of t_3, and of the original variables x, y, and z.

Because we associate a reduction operator with every inversion of a constraint, in our approach we can also exploit these dependencies, but without having to implement a specialized, and somewhat "heavy-weight" reduction operator like HC4. Instead, we use the programmable scheduler of Section 4.1.1 to ensure that first the inversions of the forward evaluation phase are applied, and then the inversions of the backward propagation phase. The schedule for the operator-based scheduler can easily be generated automatically, along with the decomposition. This way, HC4-like functionality can be ***composed*** from readily available facilities. A disadvantage is that the decomposition has to be made explicit, but we can characterize the variables that are introduced as ***auxiliary*** variables. This way, these variables do not influence the search process, and only imply some memory overhead. The scheme discussed here is demonstrated in Section 5.9.2 in the context of arithmetic constraints on integer interval variables.

Precision

The last two plug-ins that are part of the facilities for constraints on the reals are a node evaluator and a branching operator that reduce the precision of the intervals in the solved forms. For domain type $\mathcal{F}$, the default node evaluator `CanonicalDomains`, which we encountered briefly in Section 4.3, implements an ECSP

$$\langle \mathcal{C} \ ; \ x_1 \in D_1, \ldots, x_n \in D_n \ ; \ \mathcal{D}_1, \ldots, \mathcal{D}_n \ ; \ \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle,$$

having

$$\mathcal{A}_i = \lfloor \mathcal{F} \rfloor = \{[a, b] \mid a, b \in \mathbb{F}, \ a \leq b, \ \neg\exists c \in \mathbb{F} \ a < c < b\}$$

i.e., branching continues until the domains are canonical intervals. Sometimes, we are interested in less precise solved forms. The `Precision` node evaluator plug-in allows us to specify that a precision of $\epsilon > 0$ suffices:

$$\mathcal{A}_i = \lfloor \mathcal{F} \rfloor \cup \{[a, b] \mid a, b \in \mathbb{F}, \ 0 \leq b - a < \epsilon\}$$

A node evaluator only characterizes a node of the search tree as a solution, failure, or internal node. If a node is characterized as an internal node, it will be subject to branching. We want to prevent that a variable is selected for branching, whose domain already has the required precision. There are several ways in which we can realize this:

- Through the specifier string, parameterize `RealInterval` instances with the required precision. When asked for their size, an instance representing $[a, b]$ will report 1 iff $0 \leq b - a < \epsilon$.
- Alternatively, we can use an adapter for each plug-in, that overrides the size reported by the plug-in, with one based on the width of the interval that it represents. This would look something like

  ```
  VARIABLE x IS LimitedPrecisionRealInterval { 1.0e-8,
        RealInterval { [-1.0, 1.0] } };
  ```

- Make it the responsibility of the branching operator. This operator would then have to know it is dealing with `RealInterval` instances, in order that it can inquire about the width of the interval they represent. However, this means that we would have to re-implement the variable selection strategies offered by the existing branching operators for the specific case of `RealInterval` variables.

In Section 7.3.2 we see a situation where the same `RealInterval` instance (or actually, a clone of it) is used in two cooperating solvers that use different precisions. This is all but prevented by the first two of the above alternatives, and for this reason, we chose to make it the responsibility of the branching operator yet. We can avoid re-implementing the general-purpose variable section strategies by using an adapter:

```
DRF LimitedPrecision { 1.0e-8, RoundRobin{ 0, x1, x2, ... } };
```

Internally, the branching function of the adapter passes the array of domain pointers to the branching function of the inner reduction operator, but before doing so, it inspects the width of all domains. The pointers to those domains that already have the required precision are replaced by a pointer to a dummy domain of size 1, and will not be split.

This concludes the discussion of the facilities for constraints on the reals. We will see an example application in Section 7.3.2.

4.6 Conclusions

In this chapter we described the plug-ins for solving constraints on domain types $\mathcal{Z}$, $\mathcal{B}$, and $\mathcal{F}$, and we introduced some basic facilities for constraint propagation and search. These facilities allowed us to investigate how several existing solving strategies can be realized in OpenSolver through composition. Our conclusions are the following.

- An implementation of iterative limited discrepancy search inside OpenSolver is possible, but slightly artificial in the sense that some facilities are not used as intended.[4] A non-iterative variant of limited discrepancy search can be realized by an adapter that annotates the nodes of the search tree with their discrepancy values, and a container that keeps the search frontier sorted on their annotations. To limit memory usage, we switch between LDS and depth-first traversal, depending on the size of the search frontier.

- Best first-search can be realized by annotating the candidate nodes for continuing the exploration with a measure of the likelihood that they contain a solution, according to some heuristic. We demonstrated this for Warnsdorf's heuristic for solving the knight's tour problem. In this case the likelihood is evaluated after constraint propagation in the candidate nodes, which involves a limited amount of breadth-first traversal. The actual evaluation is done by a node evaluator plug-in. The traversal strategy is realized by a combination of a container and a selector plug-in.

- Of the techniques used in SAT solving, clause learning (no-good recording) can be implemented, but storing the full search frontier and using a copying state restoration policy makes OpenSolver unsuited for non-chronological backtracking.

- One of the standard scheduler plug-ins supports programmable schedules. This allows us to optimize constraint propagation by taking into account knowledge about how the reduction operators interact, and about their computational complexity. As an example, if reduction operators implement a decomposition of a polynomial constraint into atomic arithmetic constraints, we can apply them in an order that respects their hierarchical relationship. Normally this is hard-wired in heavy-weight reduction operators that implement algorithms like HC4 of Granvilliers et al. [BGGP99] In OpenSolver, the same effect can be realized through composition.

While the efficiency of some plug-ins can be optimized further, we have shown that OpenSolver is not inherently less efficient than systems that are used in practice. In particular, we compared performance with that of ILOG Solver on the n-queens problem, a benchmark that is well suited for testing the basic machinery of solvers.

Having discussed the implementation of domain types $\mathcal{Z}$, $\mathcal{B}$, and $\mathcal{F}$, in the next chapter we turn our attention to the one remaining standard domain type of Chapter 2: that of the integer intervals.

[4]The coordination layer mechanism offers further possibilities for implementing iterative schemes, though.

Chapter 5

An Analysis of Arithmetic Constraints on Integer Intervals

In this chapter we demonstrate how OpenSolver can be configured for solving arithmetic constraints on variables with integer interval domains. We study a number of approaches to implement constraint propagation for these constraints. To describe them we introduce integer interval arithmetic. Each approach is explained using appropriate proof rules that reduce the variable domains.

Our goal is to determine which approach can be expected to give the best performance. To this end, we compare them on a set of benchmark problems. For the most promising approach we provide results that characterize the effect of constraint propagation.

5.1 Introduction

5.1.1 Motivation

The subject of arithmetic constraints on reals has attracted a great deal of attention in the literature. In contrast, arithmetic constraints on integer intervals have not been studied even though they are supported in a number of constraint programming systems. In fact, constraint propagation for them is present in ECL^{i}PS^e, SICStus Prolog, GNU Prolog, ILOG Solver and undoubtedly most of the systems that support constraint propagation for linear constraints on integer intervals. Yet, in contrast to the case of linear constraints — see notably [HS03] — we did not encounter in the literature any analysis of this form of constraint propagation.

In this chapter we study these constraints in a systematic way. It turns out that in contrast to linear constraints on integer intervals there are a number of natural approaches to constraint propagation for these constraints.

It could be argued that since integer arithmetic is a special case of real arithmetic, specialized constraint propagation methods for integer arithmetic constraints are not needed. Indeed, a constraint satisfaction problem (CSP) involving arithmetic constraints on integer variables can be solved using any known method for constraints on the reals, with additional constraints ensuring that the variables assume only integer values. This was suggested in [BO97] and implemented for example in RealPaver [Gra04b]. However, a dedicated study and implementation of the integer case is beneficial for a number of reasons.

- In some cases the knowledge that we are dealing with integers yields a stronger propagation than the approach through the propagation for arithmetic constraints on the reals.

- The 'indirect' approach through the reals is based on floating point numbers, which are of limited precision. With a library like GNU MP (Multiple Precision, [Gra04a]) arbitrary precision floating point numbers can be used. However, for each problem the precision has to be chosen separately.

  In contrast, for integer variables, we can use arbitrary length integers. These are limited only by available memory, and do not involve setting any parameters, making this approach more flexible and natural.

- Since arithmetic constraints on integer intervals are supported in a number of constraint programming systems, it is natural to investigate in a systematic way various approaches to their implementation. The direct approaches based on the integers are amenable for a clear theoretical analysis. In particular, in Section 5.8 and Subsection 5.9.1 we provide the characterization results that clarify the effect of constraint propagation for the approach that emerged in our studies as the fastest.

An example that supports the first argument is the constraint $x \cdot y = z$, where $-3 \leq x \leq 3$, $-1 \leq y \leq 1$, and $1 \leq z \leq 2$. When all variables are integers, there are no solutions having $x = 3$ or $x = -3$, and the constraint propagation methods that we consider here will actually remove these values from the domain of x. However, if these variables are considered to be reals, these values may not be removed, and solving the integer problem through constraint propagation methods for constraints on the reals may lead to a larger search space.

As an indication that integer representation is not entirely a theoretical issue, consider the following benchmark from [BO97]. Find n integers $x_1, \dots, x_n$, $1 \leq x_i \leq n$, verifying the conditions

$$\sum_{i=1}^{n} x_i = \sum_{i=1}^{n} i, \quad \prod_{i=1}^{n} x_i = \prod_{i=1}^{n} i, \quad x_1 \leq x_2 \leq \ldots \leq x_n.$$

For $n = 10$ the initial maximum value of the left-hand side expression of the second constraint equals 10^{10}, which exceeds 2^{32}, the number of values that can be represented as 32-bit integers. For $n = 16$, there is already no signed integer representation of this bound in 64 bits. A small program written specifically for this benchmark indicates that on our machines, hardware integer operations are slightly faster than floating-point operations, and using arbitrary length integers costs less than a factor 4.

5.1.2 Outline of the Chapter

In the next section we define arithmetic constraints, and we relate our analysis to the model of constraint solving of Section 2.3. The unifying tool in our analysis is integer interval arithmetic that is modeled after the real interval arithmetic, see e.g., [HJvE01]. There are, however, essential differences since we deal with integers instead of reals. For example, multiplication of two integer intervals does not need to be an integer interval. In Section 5.3 we introduce the integer interval arithmetic and establish the basic results. Then in Section 5.4 we show that using integer interval arithmetic we can define succinctly the well-known constraint propagation for linear constraints on integer intervals.

The next three sections, 5.5, 5.6 and 5.7, form the main part of the chapter. We introduce there three approaches to constraint propagation for arithmetic constraints on integer intervals. They differ in the way the constraints are treated: either they are left intact, or the multiple occurrences of variables are eliminated, or the constraints are decomposed into a set of atomic constraints.

Then in Section 5.8 we characterize the effect of constraint propagation for the last approach. In Section 5.9 we discuss in detail our implementation of the alternative approaches, and in Section 5.10 we describe the experiments that were performed to compare them. They indicate that an optimized version of the third approach is superior to the other approaches. Finally, in Section 5.11 we provide the conclusions.

This chapter is based on joint work with Krzysztof Apt. Preliminary results were reported in [Apt03] and [AZ04].

5.2 Preliminaries

5.2.1 Arithmetic Constraints

To define the arithmetic constraints use the alphabet that comprises

- variables,
- two constants, 0 and 1,
- the unary minus function symbol '$-$',
- three binary function symbols, '$+$','$-$'and '$\cdot$', all written in the infix notation.

By an ***arithmetic expression*** we mean a term formed in this alphabet and by an ***arithmetic constraint*** a formula of the form

$$s \; op \; t,$$

where s and t are arithmetic expressions and $op \in \{<, \leq, =, \neq, \geq, >\}$. For example

$$x^5 \cdot y^2 \cdot z^4 + 3x \cdot y^3 \cdot z^5 \leq 10 + 4x^4 \cdot y^6 \cdot z^2 - y^2 \cdot x^5 \cdot z^4 \tag{5.1}$$

is an arithmetic constraint. Here x^5 is an abbreviation for $x \cdot x \cdot x \cdot x \cdot x$ and similarly with the other expressions. If '$\cdot$' is not used in an arithmetic constraint, we call it a ***linear constraint***.

By an ***extended arithmetic expression*** we mean a term formed in the above alphabet extended by the unary function symbols '$.^n$' and '$\sqrt[n]{.}$' for each $n \geq 1$ and the binary function symbol '/' written in the infix notation. For example

$$\sqrt[3]{(y^2 \cdot z^4)/(x^2 \cdot u^5)} \tag{5.2}$$

is an extended arithmetic expression. Here, unlike in (5.1), x^5 is a term obtained by applying the function symbol '$.^5$' to the variable x. The extended arithmetic expressions will be used only to define constraint propagation for the arithmetic constraints.

Fix now some arbitrary linear ordering $\prec$ on the variables of the language. By a ***monomial*** we mean an integer or a term of the form

$$a \cdot x_1^{n_1} \cdot \ldots \cdot x_k^{n_k}$$

where $k > 0$, $x_1, \ldots, x_k$ are different variables ordered w.r.t. $\prec$, and a is a non-zero integer and $n_1, \ldots, n_k$ are positive integers. We call then $x_1^{n_1} \cdot \ldots \cdot x_k^{n_k}$ the ***power product*** of this monomial.

Next, by a ***polynomial*** we mean a term of the form

$$\Sigma_{i=1}^n m_i,$$

where $n > 0$, at most one monomial m_i is an integer, and the power products of the monomials $m_1, \ldots, m_n$ are pairwise different. Finally, by a ***polynomial constraint*** we mean an arithmetic constraint of the form $s \; op \; b$, where s is a polynomial with no monomial being an integer, $op \in \{<, \leq, =, \neq, \geq, >\}$, and b is an integer. It is clear that by means of appropriate transformation rules we can transform any arithmetic constraint to a polynomial constraint. For example, assuming the ordering $x \prec y \prec z$ on the variables, the arithmetic constraint (5.1) can be transformed to the polynomial constraint

$$2x^5 \cdot y^2 \cdot z^4 - 4x^4 \cdot y^6 \cdot z^2 + 3x \cdot y^3 \cdot z^5 \leq 10$$

So, without loss of generality, from now on we shall limit our attention to the polynomial constraints.

5.2.2 Constraint Solving

In this chapter, the arithmetic constraints are interpreted over elements of domain type $\mathcal{I}$. Recall from Section 2.2.4 that the domains in $\mathcal{I}$ are ***integer intervals***, or ***intervals*** for short, having the form

$$[a..b]$$

where a and b are integers; $[a..b]$ denotes the set of all integers between a and b, including a and b. If $a > b$, we call $[a..b]$ the ***empty interval*** and denote it by $\emptyset$. Further, by a ***range*** we mean an expression of the form

$$x \in I$$

where x is a variable and I is an interval.

As final domains we will be using $\lfloor \mathcal{I} \rfloor$, containing all domains with a single integer. In Sections 5.6 and 5.7 we will be rewriting CSPs to eliminate multiple occurrences of variables, or to decompose constraints into atomic constraints. This rewriting involves the introduction of new variables, and for these we will consider all domains in $\mathcal{I} - \{\emptyset\}$ as final domains. In other words, the new variables introduced by rewriting constraints are ***auxiliary variables***. As a result, branching never takes place on these variables. This is justified in Section 5.6. With this information, an explicit reference to ECSPs is not necessary. Also, all approaches to solving arithmetic constraints considered in this chapter have the property that if domains are singleton sets and the corresponding assignment of values to variables is not a solution, a failure is deduced, so all solved forms correspond to CSP solutions.

Finally, to conform our notation to that of [Apt03] and [AZ04], in this chapter we describe constraint propagation by means of proof rules that act on CSPs and preserve equivalence. An interested reader can consult [Apt98] or [Apt03] for a precise explanation of this approach to describing constraint propagation. In general it allows the description of transformations of CSPs beyond those of Section 2.3, but here we will consider only rules of the form

$$\frac{\langle \mathcal{C} \ ; \ x_1 \in D_1, \ldots, x_n \in D_n \rangle}{\langle \mathcal{C}' \ ; \ x_1 \in D'_1, \ldots, x_n \in D'_n \rangle}$$

that modify the domain of a single variable x_j, and have $D'_i = D_i$ for $i \neq j$, and $\mathcal{C}' = \mathcal{C}[D'_1, \ldots, D'_n]$, the new domain projected on the constraints. In presence of integer interval domains, such rules correspond to domain reduction functions of signature

$$f : \mathcal{I}^n \to \mathcal{I}$$

with input scheme $\langle 1, \ldots, n \rangle$ and output scheme $\langle j \rangle$, having

$$f(D_1, \ldots, D_n) = D'_j.$$

A CSP that is closed under application of such rules corresponds then to a common fixed point of (the domains extensions of) the corresponding DRFs.

5.3 Integer Set Arithmetic

To reason about the arithmetic constraints we employ a generalization of the arithmetic operations to the sets of integers. Here and elsewhere $\mathbb{Z}$ denotes the set of all integers.

5.3.1 Definitions

For X, Y sets of integers we define the following operations:

- addition:
$$X + Y := \{x + y \mid x \in X, y \in Y\},$$
- subtraction:
$$X - Y := \{x - y \mid x \in X, y \in Y\},$$
- multiplication:
$$X \cdot Y := \{x \cdot y \mid x \in X, y \in Y\},$$
- division:
$$X/Y := \{u \in \mathbb{Z} \mid \exists x \in X \exists y \in Y \; u \cdot y = x\},$$
- exponentiation:
for each natural number $n > 0$,
$$X^n := \{x^n \mid x \in X\},$$
- root extraction:
for each natural number $n > 0$,
$$\sqrt[n]{X} := \{x \in \mathbb{Z} \mid x^n \in X\}.$$

All the operations except division are defined in the expected way. We shall return to it at the end of Section 5.7. At the moment it suffices to note the division operation is defined for all sets of integers, including $Y = \emptyset$ and $Y = \{0\}$. This division operation corresponds to the following division operation on the sets of reals introduced in [Rat96]:

$$X \oslash Y := \{u \in \mathbb{R} \mid \exists x \in X \exists y \in Y \; u \cdot y = x\}.$$

For an integer or a real number a and $op \in \{+, -, \cdot, /, \oslash\}$ we identify $a \, op \, X$ with $\{a\} \, op \, X$ and $X \, op \, a$ with $X \, op \, \{a\}$.

To present the rules we are interested in we shall also use the addition and division operations on the sets of real numbers. Addition is defined in the same way as for the sets of integers, and division is defined above. In [HJvE01] it is explained how to implement these operations on, possibly unbounded, real intervals.

Further, given a set A of integers or reals, we define

$$\begin{aligned} {}^{\leq}A &:= \{x \in \mathbb{Z} \mid \exists a \in A\; x \leq a\}, \\ {}^{\geq}A &:= \{x \in \mathbb{Z} \mid \exists a \in A\; x \geq a\}. \end{aligned}$$

When limiting our attention to intervals of integers the following simple observation is of importance.

5.3.1. NOTE. For X, Y integer intervals and a an integer the following holds:

- $X \cap Y$, $X + Y, X - Y$ are integer intervals.
- $X/\{a\}$ is an integer interval.
- $X \cdot Y$ does not have to be an integer interval, even if $X = \{a\}$ or $Y = \{a\}$.
- X/Y does not have to be an integer interval.
- For each $n > 1$ $\;X^n$ does not have to be an integer interval.
- For odd $n > 1$ $\;\sqrt[n]{X}$ is an integer interval.
- For even $n > 1$ $\;\sqrt[n]{X}$ is an integer interval or a disjoint union of two integer intervals. □

For example we have

$$\begin{aligned} [2..4] + [3..8] &= [5..12], \\ [3..7] - [1..8] &= [-5..6], \\ [3..3] \cdot [1..2] &= \{3, 6\}, \\ [3..5]/[-1..2] &= \{-5, -4, -3, 2, 3, 4, 5\}, \\ [-3..5]/[-1..2] &= \mathbb{Z}, \\ [1..2]^2 &= \{1, 4\}, \\ \sqrt[3]{[-30..100]} &= [-3..4], \\ \sqrt[2]{[-100..9]} &= [-3..3], \\ \sqrt[2]{[1..9]} &= [-3..-1] \cup [1..3]. \end{aligned}$$

To deal with the problem that non-interval domains can be produced by some of the operations we introduce the following operation on the sets of integers:

$$\mathsf{int}(X) := \mathcal{I}(X)$$

It is the integer interval counterpart of the hull operation on floating-point intervals, introduced in Section 2.2.4, and has the property that

$$\mathsf{int}(X) = \begin{cases} \text{smallest integer interval containing } X & \text{if } X \text{ is finite,} \\ \mathbb{Z} & \text{otherwise.} \end{cases}$$

For example $\mathsf{int}([3..5]/[-1..2]) = [-5..5]$ and $\mathsf{int}([-3..5]/[-1..2]) = \mathbb{Z}$.

5.3.2 Implementation

To define constraint propagation for the arithmetic constraints on integer intervals we shall use the integer set arithmetic, mainly limited to the integer intervals. This brings us to the discussion of how to implement the introduced operations on the integer intervals. Since we are only interested in maintaining the property that the sets remain integer intervals or the set of integers $\mathbb{Z}$ we shall clarify how to implement the intersection, addition, subtraction and root extraction operations of the integer intervals and the $\mathsf{int}(.)$ closure of the multiplication, division and exponentiation operations on the integer intervals. The case when one of the intervals is empty is easy to deal with. So we assume that we deal with non-empty intervals $[a..b]$ and $[c..d]$, i.e., $a \leq b$ and $c \leq d$.

Intersection, addition and subtraction. It is easy to see that

$$[a..b] \cap [c..d] = [\mathsf{max}(a,c)..\mathsf{min}(b,d)],$$

$$[a..b] + [c..d] = [a + c \;..\; b + d],$$

$$[a..b] - [c..d] = [a - d \;..\; b - c].$$

So the interval intersection, addition, and subtraction are straightforward to implement.

Root extraction. The outcome of the root extraction operator applied to an integer interval will be an integer interval or a disjoint union of two integer intervals. We shall explain in Section 5.5 why it is advantageous not to apply $\mathsf{int}(.)$ to the outcome. This operator can be implemented by means of the following case analysis.

Case 1. Suppose n is odd. Then

$$\sqrt[n]{[a..b]} = [\lceil \sqrt[n]{a} \rceil \;..\; \left\lfloor \sqrt[n]{b} \right\rfloor].$$

Case 2. Suppose n is even and $b < 0$. Then

$$\sqrt[n]{[a..b]} = \emptyset.$$

Case 3. Suppose n is even and $b \geq 0$. Then

$$\sqrt[n]{[a..b]} = [-\left\lfloor |\sqrt[n]{b}| \right\rfloor \;..\; -\left\lceil |\sqrt[n]{a^+}| \right\rceil] \cup [\left\lceil |\sqrt[n]{a^+}| \right\rceil \;..\; \left\lfloor |\sqrt[n]{b}| \right\rfloor]$$

where $a^+ := \mathsf{max}(0, a)$.

Multiplication. For the remaining operations we only need to explain how to implement the $\mathsf{int}(.)$ closure of the outcome. First note that

$$\mathsf{int}([a..b] \cdot [c..d]) = [\mathsf{min}(A)..\mathsf{max}(A)],$$

where $A = \{a \cdot c, a \cdot d, b \cdot c, b \cdot d\}$.

Using an appropriate case analysis we can actually compute the bounds of $\mathsf{int}([a..b] \cdot [c..d])$ directly in terms of the bounds of the constituent intervals.

Division. In contrast, the $\mathsf{int}(.)$ closure of the interval division is not so straightforward to compute. The reason is that, as we shall see in a moment, we cannot express the result in terms of some simple operations on the interval bounds.

Consider non-empty integer intervals $[a..b]$ and $[c..d]$. In analyzing the outcome of $\mathsf{int}([a..b]/[c..d])$ we distinguish the following cases.

Case 1. Suppose $0 \in [a..b]$ and $0 \in [c..d]$.

Then by definition $\mathsf{int}([a..b]/[c..d]) = \mathbb{Z}$. For example,

$$\mathsf{int}([-1..100]/[-2..8]) = \mathbb{Z}.$$

Case 2. Suppose $0 \notin [a..b]$ and $c = d = 0$.

Then by definition $\mathsf{int}([a..b]/[c..d]) = \emptyset$. For example,

$$\mathsf{int}([10..100]/[0..0]) = \emptyset.$$

Case 3. Suppose $0 \notin [a..b]$ and $c < 0$ and $0 < d$.

It is easy to see that then

$$\mathsf{int}([a..b]/[c..d]) = [-e..e],$$

where $e = \mathsf{max}(|a|, |b|)$. For example,

$$\mathsf{int}([-100.. - 10]/[-2..5]) = [-100..100].$$

Case 4. Suppose $0 \notin [a..b]$ and either $c = 0$ and $d \neq 0$ or $c \neq 0$ and $d = 0$.

Then $\mathsf{int}([a..b]/[c..d]) = \mathsf{int}([a..b]/([c..d] - \{0\}))$. For example

$$\mathsf{int}([1..100]/[-7..0]) = \mathsf{int}([1..100]/[-7.. - 1]).$$

This allows us to reduce this case to Case 5 below.

Case 5. Suppose $0 \notin [c..d]$.

This is the only case when we need to compute $\mathsf{int}([a..b]/[c..d])$ indirectly. First, observe that we have

$$\mathsf{int}([a..b]/[c..d]) \subseteq [\lceil \mathsf{min}(A) \rceil .. \lfloor \mathsf{max}(A) \rfloor],$$

where $A = \{a/c, a/d, b/c, b/d\}$.

However, the equality does not need to hold here. Indeed, note for example that $\mathsf{int}([155..161]/[9..11]) = [16..16]$, whereas for $A = \{155/9, 155/11, 161/9, 161/11\}$ we have $\lceil \mathsf{min}(A) \rceil = 15$ and $\lfloor \mathsf{max}(A) \rfloor = 17$. The problem is that the value 16 is obtained by dividing 160 by 10 and none of these two values is an interval bound.

This complication can be solved by preprocessing the interval $[c..d]$ so that its bounds are actual divisors of an element of $[a..b]$. First, we look for the least $c' \in [c..d]$ such that $\exists x \in [a..b]\ \exists u \in \mathbb{Z}\ u \cdot c' = x$. Using a case analysis, the latter property can be established without search. Suppose for example that $a > 0$ and $c > 0$. In this case, if $c' \cdot \lfloor b/c' \rfloor \geq a$, then c' has the required property. Similarly, we look for the largest $d' \in [c..d]$ for which an analogous condition holds. Now $\mathsf{int}([a..b]/[c..d]) = [\lceil \mathsf{min}(A) \rceil .. \lfloor \mathsf{max}(A) \rfloor]$, where $A = \{a/c', a/d', b/c', b/d'\}$.

In view of the auxiliary computation in case $0 \notin [c..d]$, we shall introduce in Section 5.9 a modified division operation with a more direct implementation.

Exponentiation. The $\mathsf{int}(.)$ closure of the interval exponentiation is straightforward to implement by distinguishing the following cases.

Case 1. Suppose n is odd. Then

$$\mathsf{int}([a..b]^n) = [a^n..\, b^n].$$

Case 2. Suppose n is even and $0 \leq a$. Then

$$\mathsf{int}([a..b]^n) = [a^n..\, b^n].$$

Case 3. Suppose n is even and $b \leq 0$. Then

$$\mathsf{int}([a..b]^n) = [b^n..\, a^n].$$

Case 4. Suppose n is even and $a < 0$ and $0 < b$. Then

$$\mathsf{int}([a..b]^n) = [0..\mathsf{max}(a^n, b^n)].$$

5.3.3 Correctness Lemma

Given now an extended arithmetic expression s each variable of which ranges over an integer interval, we define $\mathsf{int}(s)$ as the integer interval or the set $\mathbb{Z}$ obtained by systematically replacing each function symbol by the application of the $\mathsf{int}(.)$ operation to the corresponding integer set operation. For example, for the extended arithmetic expression $s := \sqrt[3]{(y^2 \cdot z^4)/(x^2 \cdot u^5)}$ of (5.2) we have

$$\mathsf{int}(s) = \mathsf{int}(\sqrt[3]{\mathsf{int}(\mathsf{int}(Y^2) \cdot \mathsf{int}(Z^4))/\mathsf{int}(\mathsf{int}(X^2) \cdot \mathsf{int}(U^5))}),$$

where we assume that x ranges over X, etc.

The discussion in the previous subsection shows how to compute $\mathsf{int}(s)$ given an extended arithmetic expression s and the integer interval domains of its variables.

The following lemma is crucial for our considerations. It is a counterpart of the so-called 'Fundamental Theorem of Interval Arithmetic' established in [Moo66]. Because we deal here with the integer domains an additional assumption is needed to establish the desired conclusion.

5.3.2. Lemma (Correctness). *Let s be an extended arithmetic expression with the variables $x_1, \ldots, x_n$. Assume that each variable x_i of s ranges over an integer interval X_i. Choose $a_i \in X_i$ for $i \in [1..n]$ and denote by $s(a_1, \ldots, a_n)$ the result of replacing in s each occurrence of a variable x_i by a_i.*

Suppose that each subexpression of $s(a_1, \ldots, a_n)$ evaluates to an integer. Then the result of evaluating $s(a_1, \ldots, a_n)$ is an element of $\mathsf{int}(s)$.

Proof. The proof follows by a straightforward induction on the structure of s. $\square$

5.4 An Intermezzo: Constraint Propagation for Linear Constraints

Even though we focus here on arithmetic constraints on integer intervals, it is helpful to realize that the integer interval arithmetic is also useful to define in a succinct way the well-known rules for constraint propagation for linear constraints (studied in detail in [HS03]). To this end consider first a constraint $\Sigma_{i=1}^{n} a_i \cdot x_i = b$, where $n \geq 0$, $a_1, \ldots, a_n$ are non-zero integers, $x_1, \ldots, x_n$ are different variables, and b is an integer. To reason about it we can use the following rule parametrized by $j \in [1..n]$:

LINEAR EQUALITY

$$\frac{\langle \Sigma_{i=1}^{n} a_i \cdot x_i = b \ ; \ x_1 \in D_1, \ldots, x_n \in D_n \rangle}{\langle \Sigma_{i=1}^{n} a_i \cdot x_i = b \ ; \ x_1 \in D'_1, \ldots, x_n \in D'_n \rangle}$$

where

- for $i \neq j$
$$D'_i := D_i,$$
- $$D'_j := D_j \cap \mathsf{int}\Big((b - \Sigma_{i \in [1..n]-\{j\}} a_i \cdot x_i)/a_j\Big).$$

Note that by virtue of Note 5.3.1

$$D'_j = D_j \cap (b - \Sigma_{i\in[1..n]-\{j\}}\mathsf{int}(a_i \cdot D_i))/a_j.$$

To see that this rule preserves equivalence, first note that taking the intersection implies $D'_j \subseteq D_j$, i.e., the domain is not extended by application of the rule. Further, suppose that for some $d_1 \in D_1, \ldots, d_n \in D_n$ we have $\Sigma_{i=1}^n a_i \cdot d_i = b$. Then for $j \in [1..n]$ we have

$$d_j = (b - \Sigma_{i\in[1..n]-\{j\}} a_i \cdot d_i)/a_j$$

which by the Correctness Lemma 5.3.2 implies that

$$d_j \in \mathsf{int}\Big((b - \Sigma_{i\in[1..n]-\{j\}} a_i \cdot x_i)/a_j\Big),$$

i.e., $d_j \in D'_j$.

Next, consider a constraint $\Sigma_{i=1}^n a_i \cdot x_i \leq b$, where $a_1, \ldots, a_n, x_1, \ldots, x_n$ and b are as above. To reason about it we can use the following rule parametrized by $j \in [1..n]$:

LINEAR INEQUALITY

$$\frac{\langle \Sigma_{i=1}^n a_i \cdot x_i \leq b \ ; \ x_1 \in D_1, \ldots, x_n \in D_n\rangle}{\langle \Sigma_{i=1}^n a_i \cdot x_i \leq b \ ; \ x_1 \in D'_1, \ldots, x_n \in D'_n\rangle}$$

where

- for $i \neq j$
$$D'_i := D_i,$$
-
$$D'_j := D_j \cap ({}^{\leq}\mathsf{int}(b - \Sigma_{i\in[1..n]-\{j\}} a_i \cdot x_i)/a_j)$$

To see that this rule preserves equivalence, first note that $D'_j \subseteq D_j$. Further, suppose that for some $d_1 \in D_1, \ldots, d_n \in D_n$ we have $\Sigma_{i=1}^n a_i \cdot d_i \leq b$. Then $a_j \cdot d_j \leq b - \Sigma_{i\in[1..n]-\{j\}} a_i \cdot d_i$. By the Correctness Lemma 5.3.2

$$b - \Sigma_{i\in[1..n]-\{j\}} a_i \cdot d_i \in \mathsf{int}(b - \Sigma_{i\in[1..n]-\{j\}} a_i \cdot x_i),$$

so by definition

$$a_j \cdot d_j \in^{\leq} \mathsf{int}(b - \Sigma_{i\in[1..n]-\{j\}} a_i \cdot x_i)$$

and consequently

$$d_j \in^{\leq} \mathsf{int}(b - \Sigma_{i\in[1..n]-\{j\}} a_i \cdot x_i)/a_j$$

This implies that $d_j \in D'_j$.

5.5 Constraint Propagation: First Approach

We now move on to a discussion of constraint propagation for the arithmetic constraints on integer intervals. The following example illustrates our first approach. Consider the constraint

$$x^3y - x \leq 40$$

and the ranges $x \in [1..100]$ and $y \in [1..100]$. We can rewrite it as

$$x \leq \left\lfloor \sqrt[3]{(40 + x)/y} \right\rfloor \tag{5.3}$$

since x assumes integer values. The maximum value that the expression on the right-hand side can take is $\lfloor \sqrt[3]{140} \rfloor$, so we conclude $x \leq 5$. By reusing (5.3), now with the information that $x \in [1..5]$, we conclude that the maximum value the expression on the right-hand side of (5.3) can take is actually $\lfloor \sqrt[3]{45} \rfloor$, from which it follows that $x \leq 3$.

In the case of y we can isolate it by rewriting the original constraint as $y \leq 40/x^3 + 1/x^2$ from which it follows that $y \leq 41$, since by assumption $x \geq 1$. So we could reduce the domain of x to $[1..3]$ and the domain of y to $[1..41]$. This interval reduction is optimal, since $x = 1, y = 41$ and $x = 3, y = 1$ are both solutions to the original constraint $x^3y - x \leq 40$.

More formally, we consider a polynomial constraint $\Sigma_{i=1}^{m} m_i = b$ where $m > 0$, no monomial m_i is an integer, the power products of the monomials are pairwise different and b is an integer. Suppose that $x_1, \ldots, x_n$ are its variables ordered w.r.t. $\prec$.

Select a non-integer monomial m_l and assume it is of the form

$$a \cdot y_1^{n_1} \cdot \ldots \cdot y_k^{n_k},$$

where $k > 0$, $y_1, \ldots, y_k$ are different variables ordered w.r.t. $\prec$, a is a non-zero integer and $n_1, \ldots, n_k$ are positive integers. So each y_i variable equals to some variable in $\{x_1, \ldots, x_n\}$. Suppose that y_p equals to x_j. We introduce the following proof rule:

POLYNOMIAL EQUALITY

$$\frac{\langle \Sigma_{i=1}^{n} m_i = b \;;\; x_1 \in D_1, \ldots, x_n \in D_n \rangle}{\langle \Sigma_{i=1}^{n} m_i = b \;;\; x_1 \in D'_1, \ldots, x_n \in D'_n \rangle}$$

where

- for $i \neq j$

$$D'_i := D_i,$$

- $$D'_j := \mathsf{int}\left(D_j \cap \sqrt[n_p]{\mathsf{int}\left((b - \Sigma_{i\in[1..m]-\{l\}}m_i)/s\right)}\right)$$

  and
  $$s := a \cdot y_1^{n_1} \cdot \ldots \cdot y_{p-1}^{n_{p-1}} \cdot y_{p+1}^{n_{p+1}} \ldots \cdot y_k^{n_k}.$$

To see that this rule preserves equivalence, first note that taking the intersection implies $D'_j \subseteq D_j$, i.e., the domain is not extended by application of the rule. Next, choose some $d_1 \in D_1, \ldots, d_n \in D_n$. To simplify the notation, given an extended arithmetic expression t denote by t' the result of evaluating t after each occurrence of a variable x_i is replaced by d_i.

Suppose that $\Sigma_{i=1}^m m'_i = b$. Then

$$d_j^{n_p} \cdot s' = b - \Sigma_{i\in[1..m]-\{l\}}m'_i,$$

so by the Correctness Lemma 5.3.2 applied to $b - \Sigma_{i\in[1..m]-\{l\}}m'_i$ and to s

$$d_j^{n_p} \in \mathsf{int}(b - \Sigma_{i\in[1..m]-\{l\}}m_i)/\mathsf{int}(s).$$

Hence
$$d_j \in \sqrt[n_p]{\mathsf{int}(b - \Sigma_{i\in[1..m]-\{l\}}m_i)/\mathsf{int}(s)}$$

and consequently

$$d_j \in \mathsf{int}\left(D_j \cap \sqrt[n_p]{\mathsf{int}\left((b - \Sigma_{i\in[1..m]-\{l\}}m_i)/s\right)}\right)$$

i.e., $d_j \in D'_j$.

Note that we do not apply $\mathsf{int}(.)$ to the outcome of the root extraction operation. For even n_p this means that the second operand of the intersection can be a union of two intervals, instead of a single interval. To see why this is desirable, consider the constraint $x^2 - y = 0$ in the presence of ranges $x \in [0..10]$, $y \in [25..100]$. Using the $\mathsf{int}(.)$ closure of the root extraction we would not be able to update the lower bound of x to 5.

Next, consider a polynomial constraint $\Sigma_{i=1}^m m_i \leq b$. Below we adopt the assumptions and notation used when defining the *POLYNOMIAL EQUALITY* rule. To formulate the appropriate rule we stipulate that for the extended arithmetic expressions s and t

$$\mathsf{int}((^{\leq}s)/t) \ := \ ^{\geq}Q \cap {}^{\leq}Q,$$

with $Q = (^{\leq}\mathsf{int}(s))/\mathsf{int}(t)$.

To reason about this constraint we use the following rule:

POLYNOMIAL INEQUALITY

$$\frac{\langle \Sigma_{i=1}^{n} m_i \leq b \ ; \ x_1 \in D_1, \ldots, x_n \in D_n \rangle}{\langle \Sigma_{i=1}^{n} m_i \leq b \ ; \ x_1 \in D'_1, \ldots, x_n \in D'_n \rangle}$$

where

- for $i \neq j$
$$D'_i := D_i,$$
-
$$D'_j := \mathsf{int}\left(D_j \cap \sqrt[n_p]{\mathsf{int}\left(^{\leq}(b - \Sigma_{i \in [1..m]-\{l\}} m_i)/s \right)} \right)$$

To prove that this rule preserves equivalence, first note that $D'_j \subseteq D_j$. Next, choose some $d_1 \in D_1, \ldots, d_n \in D_n$. As above given an extended arithmetic expression t we denote by t' the result of evaluating t when each occurrence of a variable x_i in t is replaced by d_i.

Suppose that $\Sigma_{i=1}^{m} m'_i \leq b$. Then

$$d_j^{n_p} \cdot s' \leq b - \Sigma_{i \in [1..m]-\{l\}} m'_i.$$

By the Correctness Lemma 5.3.2

$$b - \Sigma_{i \in [1..m]-\{l\}} m'_i \in \mathsf{int}(b - \Sigma_{i \in [1..m]-\{l\}} m_i),$$

so by definition

$$d_j^{n_p} \cdot s' \in^{\leq} \mathsf{int}(b - \Sigma_{i \in [1..m]-\{l\}} m_i).$$

Hence by the definition of the division operation on the sets of integers

$$d_j^{n_p} \in^{\leq} \mathsf{int}(b - \Sigma_{i \in [1..m]-\{l\}} m_i)/\mathsf{int}(s)$$

Consequently

$$d_j \in \sqrt[n_p]{^{\leq}\mathsf{int}(b - \Sigma_{i \in [1..m]-\{l\}} m_i)/\mathsf{int}(s)}$$

This implies that $d_j \in D'_j$.

Note that the set $^{\leq}\mathsf{int}(b - \Sigma_{i \in [1..m]-\{l\}} m_i)$ appearing in the definition of D'_j is not an interval. So to properly implement this rule we need to extend the implementation of the division operation discussed in Subsection 5.3.2 to the case when the numerator is an extended interval. Our implementation takes care of this.

In an optimized version of this approach we simplify the fractions of two polynomials by splitting the division over addition and subtraction and by dividing out common powers of variables and greatest common divisors of the constant factors. Subsequently, fractions whose denominators have identical power products are added. We used this optimization in the initial example by simplifying

$(40{+}x)/x^3$ to $40/x^3{+}1/x^2$. The reader may check that without this simplification step we can only deduce that $y \leq 43$.

To provide details of this optimization, given two monomials s and t, we denote by

$$[s/t]$$

the result of performing this simplification operation on s and t. For example, $[(2 \cdot x^3 \cdot y)/(4 \cdot x^2)]$ equals $(x \cdot y)/2$, whereas $[(4 \cdot x^3 \cdot y)/(2 \cdot y^2)$ equals $(2 \cdot x^3)/y$.

In this approach we assume that the domains of the variables $y_1, \ldots, y_{p-1}, y_{p+1}, \ldots, y_n$ of m_l do not contain 0. (One can easily show that this restriction is necessary here). For a monomial s involving variables ranging over the integer intervals that do not contain 0, the set $\mathsf{int}(s)$ either contains only positive numbers or only negative numbers. In the first case we write $\mathsf{sign}(s) = +$ and in the second case we write $\mathsf{sign}(s) = -$.

The new domain of the variable x_j in the *POLYNOMIAL INEQUALITY* rule is defined using two sequences $m'_0...m'_n$ and $s'_0...s'_n$ of extended arithmetic expressions such that

$$m'_0/s'_0 = [b/s] \text{ and } m'_i/s'_i = -[m_i/s] \text{ for } i \in [1..m].$$

Let $S := \{s'_i \mid i \in [0..m]{-}\{l\}\}$ and for an extended arithmetic expression $t \in S$ let $I_t := \{i \in [0..m] - \{l\} \mid s'_i = t\}$. We denote then by p_t the polynomial $\sum_{i \in I_t} m'_i$. The new domains are then defined by

$$D'_j := \mathsf{int}\left(D_j \cap \sqrt[n_p]{^{\leq}\mathsf{int}\left(\Sigma_{t\in S}\ p_t \oslash t\right)}\ \right)$$

if $\mathsf{sign}(s) = +$, and by

$$D'_j := \mathsf{int}\left(D_j \cap \sqrt[n_p]{^{\geq}\mathsf{int}\left(\Sigma_{t\in S}\ p_t \oslash t\right)}\ \right)$$

if $\mathsf{sign}(s) = -$. Here the $\mathsf{int}(s)$ notation used in the Correctness Lemma 5.3.2 is extended to expressions involving the division operator $\oslash$ on real intervals in the obvious way. We define the $\mathsf{int}(.)$ operator applied to a bounded set of real numbers, as produced by the division and addition operators in the above two expressions for D'_j, to denote the smallest interval of real numbers containing that set.

5.6 Constraint Propagation: Second Approach

In this approach we limit our attention to a special type of polynomial constraints, namely the ones of the form $s\ op\ b$, where s is a polynomial in which each variable occurs ***at most once*** and where b is an integer. We call such a constraint a ***simple polynomial constraint***. By introducing variables that are equated with appropriate monomials we can rewrite any polynomial constraint into a sequence of simple polynomial constraints. We apply then to the simple polynomial constraints the rules introduced in the previous section.

To see that the restriction to simple polynomial constraints can make a difference consider the constraint

$$100x \cdot y - 10y \cdot z = 212$$

in presence of the ranges $x, y, z \in [1..9]$. We rewrite it into the sequence

$$u = x \cdot y,\ v = y \cdot z,\ 100u - 10v = 212,$$

where u, v are new variables, each with the domain $[1..81]$.

It is easy to check that the *POLYNOMIAL EQUALITY* rule introduced in the previous section does not yield any domain reduction when applied to the original constraint $100x \cdot y - 10y \cdot z = 212$. In presence of the discussed optimization the domain of x gets reduced to $[1..3]$.

However, if we repeatedly apply the *POLYNOMIAL EQUALITY* rule to the simple polynomial constraint $100u - 10v = 212$, we eventually reduce the domain of u to the empty set (since this constraint has no integer solution in the ranges $u, v \in [1..81]$) and consequently can conclude that the original constraint $100x \cdot y - 10y \cdot z = 212$ has no solution in the ranges $x, y, z \in [1..9]$, without performing any search.

The integer interval domains of the introduced variables are fully determined by the integer interval domains of the original variables. For this reason, no branching on these variables is needed: when the domains of the original variables are reduced to singleton sets, the domains of the introduced variables will be singleton sets as well. It is not efficient to branch on the new variables either. Consider for example that we have problem variables $x, y \in [1..10]$, with a variable $u \in [1..100]$ introduced to represent their product $u = x \cdot y$. If the domain of u is bisected, two branches are created in which the domains of the problem variables x and y have an overlap after propagation of the constraint $u = x \cdot y$:

left branch:	right branch:
$u \in [1..50]$	$u \in [51..100]$
$x \in [1..10]$	$x \in [6..10]$
$y \in [1..10]$	$y \in [6..10]$

This overlap can lead to a larger search space, so branching on the introduced variables should be avoided. For the search process, such variables can be considered ***auxiliary variables***, as introduced in Section 2.2.5.

5.7 Constraint Propagation: Third Approach

In this approach we focus on a small set of 'atomic' arithmetic constraints. We call an arithmetic constraint ***atomic*** if it is in one of the following two forms:

- a linear constraint,
- $x \cdot y = z$.

It is easy to see that using appropriate transformation rules involving auxiliary variables we can transform any arithmetic constraint into a sequence of atomic arithmetic constraints. In this transformation, as in the second approach, the auxiliary variables are equated with monomials so we can easily compute their domains.

The transformation to atomic constraints can strengthen the reduction. Consider for example the constraint

$$u \cdot x \cdot y + 1 = v \cdot x \cdot y$$

and ranges $u \in [1..2]$, $v \in [3..4]$, and $x, y \in [1..4]$. The first approach without optimization and the second approach cannot find a solution without search. If, as a first step in transforming this constraint into a linear constraint, we introduce an auxiliary variable w to replace $x \cdot y$, we are effectively solving the constraint

$$u \cdot w + 1 = v \cdot w$$

with the additional range $w \in [1..16]$, resulting in only one duplicate occurrence of a variable instead of two. With variable w introduced (or using the optimized version of the first approach) constraint propagation alone finds the solution $u = 2$, $v = 3$, $x = 1$, $y = 1$.

We explained already in Section 5.4 how to reason about linear constraints. (We omitted there the treatment of the disequalities which is routine.) Next, we focus on the reasoning for the multiplication constraint $x \cdot y = z$ in presence of the non-empty ranges $x \in D_x$, $y \in D_y$ and $z \in D_z$. To this end we introduce the following three domain reduction rules:

MULTIPLICATION 1

$$\frac{\langle x \cdot y = z \ ; \ x \in D_x, y \in D_y, z \in D_z \rangle}{\langle x \cdot y = z \ ; \ x \in D_x, y \in D_y, z \in D_z \cap \mathsf{int}(D_x \cdot D_y) \rangle}$$

MULTIPLICATION 2

$$\frac{\langle x \cdot y = z \ ; \ x \in D_x, y \in D_y, z \in D_z \rangle}{\langle x \cdot y = z \ ; \ x \in D_x \cap \mathsf{int}(D_z / D_y), y \in D_y, z \in D_z \rangle}$$

MULTIPLICATION 3

$$\frac{\langle x \cdot y = z \ ; \ x \in D_x, y \in D_y, z \in D_z \rangle}{\langle x \cdot y = z \ ; \ x \in D_x, y \in D_y \cap \mathsf{int}(D_z/D_x), z \in D_z \rangle}$$

The way we defined the multiplication and the division of the integer intervals ensures that the *MULTIPLICATION* rules *1,2* and *3* are equivalence preserving. Consider for example the *MULTIPLICATION 2* rule. Take some $a \in D_x, b \in D_y$ and $c \in D_z$ such that $a \cdot b = c$. Then $a \in \{x \in \mathbb{Z} \mid \exists z \in D_z \exists y \in D_y \ x \cdot y = z\}$, so $a \in D_z/D_y$ and a fortiori $a \in \mathsf{int}(D_z/D_y)$. Consequently $a \in D_x \cap \mathsf{int}(D_z/D_y)$. Because we also have $(D_x \cap \mathsf{int}(D_z/D_y)) \subseteq D_x$, this shows that the *MULTIPLICATION 2* rule is equivalence preserving.

The following example from [Apt03] shows an interaction between all three *MULTIPLICATION* rules.

5.7.1. Example. Consider the CSP

$$\langle x \cdot y = z \ ; \ x \in [1..20], y \in [9..11], z \in [155..161] \rangle. \tag{5.4}$$

To facilitate the reading we underline the modified domains. An application of the *MULTIPLICATION 2* rule yields

$$\langle x \cdot y = z \ ; \ x \in \underline{[16..16]}, y \in [9..11], z \in [155..161] \rangle$$

since, as already noted in Subsection 5.3.2, $[155..161]/[9..11]) = [16..16]$, and $[1..20] \cap \mathsf{int}([16..16]) = [16..16]$. Applying now the *MULTIPLICATION 3* rule we obtain

$$\langle x \cdot y = z \ ; \ x \in [16..16], y \in \underline{[10..10]}, z \in [155..161] \rangle$$

since $[155..161]/[16..16] = [10..10]$ and $[9..11] \cap \mathsf{int}([10..10]) = [10..10]$. Next, by the application of the *MULTIPLICATION 1* rule we obtain

$$\langle x \cdot y = z \ ; \ x \in [16..16], y \in [10..10], z \in \underline{[160..160]} \rangle$$

since $[16..16] \cdot [10..10] = [160..160]$ and $[155..161] \cap \mathsf{int}([160..160]) = [160..160]$.

So using all three multiplication rules we could solve the CSP (5.4). □

Now let us clarify why we did not define the division of the sets of integers Z and Y by

$$Z/Y := \{z/y \in \mathbb{Z} \mid y \in Y, z \in Z, y \neq 0\}.$$

The reason is that in that case for any set of integers Z we would have $Z/\{0\} = \emptyset$. Consequently, if we adopted this definition of the division of the integer intervals, the resulting *MULTIPLICATION 2* and *3* rules would not be equivalence preserving anymore. Indeed, consider the CSP

$$\langle x \cdot y = z \ ; \ x \in [-2..1], y \in [0..0], z \in [-8..10] \rangle.$$

Then we would have $[-8..10]/[0..0] = \emptyset$ and consequently by the *MULTIPLICATION 2* rule we could conclude

$$\langle x \cdot y = z \ ; \ x \in \emptyset, y \in [0..0], z \in [-8..10]\rangle.$$

So we reached an inconsistent CSP while the original CSP is consistent.

In the remainder of the chapter we will also consider variants of this third approach that allow squaring and exponentiation as atomic constraints. For this purpose we explain the reasoning for the constraint $x = y^n$ in presence of the non-empty ranges $x \in D_x$ and $y \in D_y$, and for $n > 1$. To this end we introduce the following two rules in which to maintain the property that the domains are intervals we use the $\mathsf{int}(.)$ operation of Section 5.3:

EXPONENTIATION

$$\frac{\langle x = y^n \ ; \ x \in D_x, y \in D_y\rangle}{\langle x = y^n \ ; \ x \in D_x \cap \mathsf{int}(D_y^n), y \in D_y\rangle}$$

ROOT EXTRACTION

$$\frac{\langle x = y^n \ ; \ x \in D_x, y \in D_y\rangle}{\langle x = y^n \ ; \ x \in D_x, y \in \mathsf{int}(D_y \cap \sqrt[n]{D_x})\rangle}$$

To prove that these rules are equivalence preserving suppose that for some $a \in D_x$ and $b \in D_y$ we have $a = b^n$. Then $a \in D_y^n$, so $a \in \mathsf{int}(D_y^n)$ and consequently $a \in D_x \cap \mathsf{int}(D_y^n)$. Also $b \in \sqrt[n]{D_x}$, so $b \in D_y \cap \sqrt[n]{D_x}$, and consequently $b \in \mathsf{int}(D_y \cap \sqrt[n]{D_x})$. The set intersection operation prevents the extension of the domains, as usual.

5.8 A Characterization of the *MULTIPLICATION* Rules

It is useful to reflect on the effect of the proof rules used to achieve constraint propagation. In this section, by way of example, we focus on the *MULTIPLICATION* rules and characterize their effect using the notion of bounds consistency of [VHSD98]. Let us recall first the definition that we adopt here to the multiplication constraint. Given an integer interval $[l..h]$ we denote by $[l, h]$ the corresponding real interval.

5.8.1. Definition. The CSP $\langle x \cdot y = z \ ; \ x \in [l_x..h_x], y \in [l_y..h_y], z \in [l_z..h_z]\rangle$ is called ***bounds consistent*** if

- $\forall a \in \{l_x, h_x\} \ \exists b \in [l_y, h_y] \ \exists c \in [l_z, h_z] \ a \cdot b = c$,

- $\forall b \in \{l_y, h_y\}\ \exists a \in [l_x, h_x]\ \exists c \in [l_z, h_z]\ a \cdot b = c$,
- $\forall c \in \{l_z, h_z\}\ \exists a \in [l_x, h_x]\ \exists b \in [l_y, h_y]\ a \cdot b = c$. □

Then we have the following result.

5.8.2. Theorem (Bounds consistency). *Suppose a CSP* $\langle x \cdot y = z\ ;\ x \in D_x, y \in D_y, z \in D_z\rangle$ *with integer interval domains is bounds consistent. Then it is closed under the applications of the MULTIPLICATION 1, 2 and 3 rules.*

Proof. See the Appendix. □

The converse of the above result does not hold. Here is an example. Consider the CSP

$$\langle x \cdot y = z\ ;\ x \in [-2..1], y \in [-3..10], z \in [8..10]\rangle.$$

It is not bounds consistent, since for $y = -3$ no real values $a \in [-2, 1]$ and $c \in [8, 10]$ exist such that $a \cdot (-3) = c$. Indeed, it is easy to check that

$$\{y \in \mathbb{R} \mid \exists x \in [-2, 1]\ \exists z \in [8, 10]\ x \cdot y = z\} = (-\infty, -4] \cup [8, \infty).$$

However, this CSP is closed (see Section 5.2.2) under the applications of the *MULTIPLICATION 1*, *2* and *3* rules since

- $[8..10] \subseteq \mathsf{int}([-2..1] \cdot [-3..10])$, as $\mathsf{int}([-2..1] \cdot [-3..10]) = [-20..10]$,
- $[-2..1] \subseteq \mathsf{int}([8..10]/[-3..10])$ as $\mathsf{int}([8..10]/[-3..10]) = [-10..10]$, and
- $[-3..10] \subseteq \mathsf{int}([8..10]/[-2..1])$ as $\mathsf{int}([8..10]/[-2..1]) = [-10..10]$.

The following result clarifies that this example identifies the only cause of discrepancy between the closure under the *MULTIPLICATION* rules and bound consistency. Here, given an integer interval $D := [l..h]$ we define

$$\langle D\rangle := \{x \in \mathbb{Z} \mid l < x < h\}.$$

5.8.3. Theorem (Bounds consistency 2). *Consider a CSP* $\phi := \langle x \cdot y = z\ ;\ x \in D_x, y \in D_y, z \in D_z\rangle$ *with non-empty integer interval domains and such that*

$$0 \in \langle D_x\rangle \cap \langle D_y\rangle \textit{ implies } 0 \in D_z. \tag{5.5}$$

Suppose ϕ *is closed under the applications of the MULTIPLICATION 1, 2, and 3 rules. Then it is bounds consistent.*

Proof. See the Appendix. □

Let us mention here that to deal with the constraint $x \cdot y = z$ in [SS01] similar rules to our *MULTIPLICATION* rules were proposed. These rules were defined without the use of interval arithmetic. The rules for the variables x and y are different and more complex (also from an implementation point of view) than our *MULTIPLICATION* rules 2 and 3. As a result they achieve bounds consistency for the constraint $x \cdot y = z$ for arbitrary integer interval domains.

5.9 Implementation Details

5.9.1 Weak Division

We already mentioned in Section 5.3 that the division operation on the intervals does not admit an efficient implementation. The reason is that the $\mathsf{int}(.)$ closure of the interval division $[a..b]/[c..d]$ requires an auxiliary computation in case when $0 \notin [c..d]$. The preprocessing of $[c..d]$ becomes impractical for small intervals $[a..b]$, and large $[c..d]$, occurring for example for the constraint $\prod_{i=1}^{n} x_i = \prod_{i=1}^{n} i$, of the benchmark problem mentioned in Subsection 5.1.1. To remedy this problem we have used in our implementation another division operation. We call it ***weak division*** since it yields a larger set (and so is 'weaker'). This operation is defined as follows:

$$[a..b] : [c..d] := \begin{cases} [\lceil \mathsf{min}(A) \rceil \, .. \, \lfloor \mathsf{max}(A) \rfloor] & \text{if } 0 \notin [c..d], \text{ or} \\ & \quad 0 \notin [a..b] \text{ and } 0 \in \{c, d\} \text{ and } c < d, \\ [a..b]/[c..d] & \text{otherwise} \end{cases}$$

where $A = \{a/c', a/d', b/c', b/d'\}$, and $[c'..d'] = [c..d] - \{0\}$.

Then $\mathsf{int}([a..b] : [c..d])$ can be computed by a straightforward case analysis already used for $\mathsf{int}([a..b]/[c..d])$ but now without any auxiliary computation.

In particular, in our implementation we used the following counterparts of the *MULTIPLICATION* rules *2* and *3*:

MULTIPLICATION 2w

$$\frac{\langle x \cdot y = z \; ; \; x \in D_x, y \in D_y, z \in D_z \rangle}{\langle x \cdot y = z \; ; \; x \in D_x \cap \mathsf{int}(D_z : D_y), y \in D_y, z \in D_z \rangle}$$

MULTIPLICATION 3w

$$\frac{\langle x \cdot y = z \; ; \; x \in D_x, y \in D_y, z \in D_z \rangle}{\langle x \cdot y = z \; ; \; x \in D_x, y \in D_y \cap \mathsf{int}(D_z : D_x), z \in D_z \rangle}$$

In the assumed framework based on constraint propagation and tree search, all domains become eventually singletons or empty sets. It can easily be verified that both division operations are then equal, i.e., $[a..b] : [c..d] = [a..b]/[c..d]$, for $a \geq b$ and $c \geq d$. For this reason, we can safely replace any of the reduction rules introduced in this chapter, notably *POLYNOMIAL EQUALITY*, *POLYNOMIAL INEQUALITY*, and *MULTIPLICATION 2* and *3*, by their counterparts based on the weak division. For the *MULTIPLICATION* rules specifically, the following theorem states that both sets of rules actually achieve the same constraint propagation.

5.9.1. Theorem (*MULTIPLICATION*). *A CSP $\langle x \cdot y = z \; ; \; x \in D_x, y \in D_y, z \in D_z \rangle$ with integer interval domains is closed under the applications of the MULTIPLICATION 1, 2 and 3 rules iff it is closed under the applications of the MULTIPLICATION 1, 2w and 3w rules.*

Proof. See the Appendix. □

Let us clarify now the relation between the *MULTIPLICATION* rules and the corresponding rules based on real interval arithmetic coupled with the rounding of the resulting real intervals inwards to the largest integer intervals. The CSP $\langle x \cdot y = z \; ; \; x \in [-3..3], y \in [-1..1], z \in [1..2] \rangle$, which we already discussed in the introduction, shows that these approaches yield different results. Indeed, using the *MULTIPLICATION* rule *2* we can reduce the domain of x to $[-2..2]$, while the second approach yields no reduction.

On the other hand, the applications of the *MULTIPLICATION* rules *2w* and *3w* to $\langle x \cdot y = z \; ; \; x \in D_x, y \in D_y, z \in D_z \rangle$ such that $\mathsf{int}(D_z : D_x) \neq \mathsf{int}(D_z / D_x)$ and $\mathsf{int}(D_z : D_y) \neq \mathsf{int}(D_z / D_y)$ (so in cases when the use of the weak interval division differs from the use of the interval division) do coincide with the just discussed approach based on real interval arithmetic and inward rounding. This is a consequence of the way the multiplication and division of the real intervals are defined, see [HJvE01]. We did not implement these instances of the *MULTIPLICATION* rules *2w* and *3w* through a detour via the rules for real intervals for the reasons explained in the introduction.

5.9.2 Implementation

Constraint Propagation

Integer intervals in OpenSolver are implemented by the `IntegerInterval` domain type plug-in. This plug-in, and the interval arithmetic operations on it are built using the `mpz` type of the GNU MP library [Gra04a], which supports arbitrary precision (or rather arbitrary length) integers. Domains of type `IntegerInterval` consist of an indication of the type of the set (bounded, unbounded, left/right-bounded, or empty), and the appropriate number (0, 1, or 2) of bounds.

Left-bounded and right-bounded sets have the respective forms $\{x \in \mathbb{Z} \mid x > l\}$ and $\{x \in \mathbb{Z} \mid x < h\}$, which are not integer intervals. Therefore, instead of $\mathcal{I}$, `IntegerInterval` is a (rather crude) implementation of the domain type containing all sets $[l..h]$, with $l, h \in Z \cup \{-\infty, \infty\}$, where Z is a finite subset of $\mathbb{Z}$ containing all integers that can be represented on a particular machine, using type `mpz`.

The reduction rules are implemented by a plug-in `IIARule` (Integer Interval Arithmetic Rule). Its specifier string has the form

$$x_j^{n_p} \cdot (s) \; op \; q,$$

where $op \in \{\leq, =\}$, s is a monomial, and q is a polynomial. When the domain of a variable in s or p is modified, `IIARule` will set the domain D_j of x_j to

$$\mathsf{int}(D_j \cap \sqrt[np]{\mathsf{int}(q/s)})$$

if op is the symbol $=$, or to

$$\mathsf{int}(D_j \cap \sqrt[np]{\mathsf{int}({}^{\leq}(q/s))})$$

if op is the symbol $\leq$. With q set to $b - \Sigma_{i \in [1..m]-\{l\}} m_i$ this implements the *POLYNOMIAL EQUALITY* and *POLYNOMIAL INEQUALITY* rules of Section 5.5, of which all other rules are instances.

As an example of its use in a solver configuration, the following three operators implement the constraint $x^3y - x \leq 40$.

```
DRF IIARule { x^3 * (1*y) <= 1*x + 40 };
DRF IIARule {!y^1 * (1*x^3) <= 1*x + 40 };
DRF IIARule { x^1 * (-1) <= -1*x^3*y + 40 };
```

The `!` prefix in the second specifier string activates the optimization described at the end of Section 5.5, which entails that common power products in s and q are eliminated. This cannot be implemented as a preprocessing stage, because `IIARule` needs to know the full monomial s, so that it can select the appropriate case for $\mathsf{sign}(s)$.

Scheduling Reduction Operators

For the second and third approach, we make use of the scheduling facilities of the operator-based scheduler, described in Section 4.1.1. We distinguish ***user constraints*** from the constraints that are introduced to define the values of auxiliary variables. Before considering for execution a DRF f that is part of the implementation of a user constraint, it is ensured that all auxiliary variables that f relies on are updated. For this purpose, the indices of the DRFs that update these variables precede the index of f in the schedule. If f can change the value of an auxiliary variable, its index is followed by the indices of the DRFs that propagate back these changes to the variables that define the value of this auxiliary variable.

As an example, the following operators (prefixed by a sequence number that is not part of the configuration) enforce the constraint $100x \cdot y - 10y \cdot z = 212$.

```
0.  DRF IIARule { aux_xy^1 * (1) = x*y };
1.  DRF IIARule { aux_yz^1 * (1) = y*z };
2.  DRF IIARule { x^1 * (y) = aux_xy };
3.  DRF IIARule { y^1 * (x) = aux_xy };
4.  DRF IIARule { y^1 * (z) = aux_yz };
5.  DRF IIARule { z^1 * (y) = aux_yz };
6.  DRF IIARule { aux_xy^1 * (100) = 10*aux_yz + 212 };
7.  DRF IIARule { aux_yz^1 * (-10) = -100*aux_xy + 212 };
```

The operators with sequence number 6 and 7 correspond to the user constraint. Operators 1, 2, and 3 constrain the auxiliary variable `aux_xy` to be equal to $x \cdot y$, and 0, 4, and 5 do the same for `aux_yz` and $y \cdot z$. The following schedule is generated along with the decomposition.

```
SCHEDULER ChangeScheduler { schedule = { 1,6,2,3,0,7,4,5 } };
```

It specifies that in principle, operators 6 and 7 are applied in sequence, but operator 1 is considered before operator 6 in order that `aux_xy`, which appears in the right-hand side expression for operator 6, is updated before operator 6 is applied. If this modifies `aux_yz`, operators 2 and 3 will propagate this modification back to x and y before operator 7 is applied. The example is artificial because without other constraints on x and y, the "problem" could be solved entirely on the auxiliary variables, and evaluation and back propagation would only have to be applied once.

For the third approach, there can be hierarchical dependencies between auxiliary variables. Much like the HC4 algorithm of [BGGP99] (see also Section 4.5), the generated schedule specifies a bottom-up traversal of this hierarchy in a forward evaluation phase and a top-down traversal in a backward propagation phase before and after applying a DRF of a user constraint, respectively. In the forward evaluation phase, the DRFs that are executed correspond to the *MULTIPLICATION 1* and *EXPONENTIATION* rules. The DRFs of the backward propagation phase correspond to the *MULTIPLICATION 2* and *3*, and *ROOT EXTRACTION* rules. It is easy to construct examples showing that the use of hierarchical schedules can be beneficial compared to cycling through the rules.

Optimization

One of our benchmark problems is an ***optimization*** problem, where we want to find the assignment of values to decision variables that yields the optimal value for an objective function. Our approach to optimization problems is to introduce a variable for the outcome of an objective function, which can then be evaluated by constraint propagation. An optimization operator (a particular form of reduction operator, discussed in Section 3.2.2) then monitors this variable. It records the best value seen for any solution, and applies the dynamic constraint that new solutions must improve on this value.

For integer objective functions this is implemented by the `Optimize` reduction operator:

```
DRF Optimize { -x };
```

Its specifier string is the name of an `IntegerInterval` variable, prefixed with `-` for minimization, or `+` for maximization. If the objective function is composed of arithmetic operations, it can be evaluated using the `IIARule` reduction operator.

This yields a particular form of ***branch-and-bound*** search (see for example [Dec03]). Branch-and-bound algorithms maintain an estimation for the outcome of the objective function in the subtree that is currently being explored. This estimation is a ***bound*** for the outcome of the objective function: a lower bound for minimization, and an upper bound for optimization. Based on this estimation, it may be possible to conclude that a particular branch of the search tree will never be able to improve on the current best solution. Such subtrees can then be pruned away. In our case the estimation is an interval that is guaranteed to contain the outcome of the objective function for any solutions that descend from the current node of the search tree.

Approaches

The proposed approaches were implemented by first rewriting arithmetic constraints to polynomial constraints, and then to a sequence of DRFs that correspond with the rules of the approach used. We considered the following methods:

1a the first approach, discussed in Section 5.5;

1b the optimization of the first approach discussed at the end of Section 5.5 that involves dividing out common powers of variables;

2a the second approach, discussed in Section 5.6. The conversion to simple polynomial constraints is implemented by introducing an auxiliary variable for every nonlinear monomial. This procedure may introduce more auxiliary variables than necessary;

2b an optimized version of approach 2a, where we stop introducing auxiliary variables as soon as the constraints contain no more duplicate occurrences of variables;

3a the third approach, discussed in Section 5.7, allowing only linear constraints and multiplication as atomic constraints;

3b idem, but also allowing $x = y^2$ as an atomic constraint;

3c idem, allowing $x = y^n$ for all $n > 1$ as an atomic constraint.

Approaches 2 and 3 involve an extra rewrite step, where the auxiliary variables are introduced. The resulting CSP is then rewritten according to approach 1a. During the first rewrite step the hierarchical relations between the auxiliary variables are recorded and the schedules are generated as a part of the second rewrite step. For approaches 2b and 3 the question of which auxiliary variables to introduce is an optimization problem in itself. Some choices result in more auxiliary variables than others. We have not treated this issue as an optimization problem but relied on heuristics. For this reason we have to consider the possibility that performance for these approaches can be further improved because in our experiments we used a suboptimal rewriting. The rewrite steps are executed by an external program, called `pcrewrite` (polynomial constraint rewrite), that could serve as the interface between OpenSolver and a calculator front-end.

5.10 Experiments

5.10.1 Problems

In our experiments we used the following benchmarks.

Cubes. The problem is to find all natural numbers $n \leq 100000$ that are a sum of four different cubes, for example

$$1^3 + 2^3 + 3^3 + 4^3 = 100.$$

This problem is modeled as follows[1]:

$$\begin{array}{l}\langle 1 \leq x_1,\ x_1 \leq x_2 - 1,\ x_2 \leq x_3 - 1,\ x_3 \leq x_4 - 1,\ x_4 \leq n, \\ \ x_1^3 + x_2^3 + x_3^3 + x_4^3 = n;\ n \in [1..100000],\ x_1, x_2, x_3, x_4 \in \mathbb{Z}\rangle\end{array}$$

Opt. We are interested in finding a solution to the constraint $x^3 + y^2 = z^3$ in the integer interval [1..100000] for which the value of $2x \cdot y - z$ is maximal.

Program 3.1 on page 38 shows the OpenSolver configuration script for solving this problem according to approach 2a, which in this case is identical to that for approach 3c.

Fractions. This problem is taken from [SS02]: find distinct nonzero digits such that the following equation holds:

$$\frac{A}{BC} + \frac{D}{EF} + \frac{G}{HI} = 1$$

There is a variable for each letter. The initial domains are [1..9]. To avoid symmetric solutions an ordering is imposed:

$$\frac{A}{BC} \geq \frac{D}{EF} \geq \frac{G}{HI}$$

[1]Note that because the inequality constraints update only one bound, this works only because the domain type implemented by `IntegerInterval` supports domains of the form $\{x \in Z | x \geq l\}$ and $\{x \in Z \mid x \leq h\}$, with Z a finite subset of $\mathbb{Z}$. For using domain type $\mathcal{I}$ we would have to provide initial bounds for x_1, x_2, x_3, and x_4.

Also two redundant constraints are added:

$$3\frac{A}{BC} \geq 1 \qquad \text{and} \qquad 3\frac{G}{HI} \leq 1$$

Because division is not present in our arithmetic expressions, the above constraints are multiplied by the denominators of the fractions to obtain arithmetic constraints. Two representations for this problem were studied:

- *fractions1* in which five constraints are used: one equality and four inequalities for the ordering and the redundant constraints,
- *fractions2*, used in [SS02], in which three auxiliary variables, BC, EF and HI, are introduced to simplify the arithmetic constraints: $BC = 10B + C$, $EF = 10E + F$, and $HI = 10H + I$.

Additionally, in both representations, 36 disequalities $A \neq B$, $A \neq C$, ..., $H \neq I$ are used.

Kyoto. The problem (from [DS95]) is to find the number n such that the alphanumeric equation

$$\begin{array}{cccccc} & \mathrm{K} & \mathrm{Y} & \mathrm{O} & \mathrm{T} & \mathrm{O} \\ & \mathrm{K} & \mathrm{Y} & \mathrm{O} & \mathrm{T} & \mathrm{O} \\ + & \mathrm{K} & \mathrm{Y} & \mathrm{O} & \mathrm{T} & \mathrm{O} \\ \hline & \mathrm{T} & \mathrm{O} & \mathrm{K} & \mathrm{Y} & \mathrm{O} \end{array}$$

has a solution in the base-n number system. Our representation uses a variable for each letter and one variable for the base number. The variables K and T may not be zero. There is one large constraint for the addition, 6 disequalities $K \neq Y$ ... $T \neq O$ and four constraints stating that the individual digits K, Y, O, T, are smaller than the base number. To spend some CPU time, we searched base numbers 2..100.

Sumprod. This is the problem cited in Subsection 5.1.1, for $n = 14$. We use the following representation:

$$\begin{aligned} \langle & x_1 + \ldots + x_n = c_1 + \ldots + c_n, \\ & x_1 \cdot \ldots \cdot x_n = c_1 \cdot \ldots \cdot c_n, \\ & x_1 \leq x_2, x_2 \leq x_3, \ldots, x_{n-1} \leq x_n \ ; \\ & x_1, \ldots, x_n \in [1..n], \\ & c_1 \in \{1\}, c_2 \in \{2\}, \ldots, c_n \in \{n\} \rangle \end{aligned}$$

For $n = 14$, the value of the expression $\prod_{i=1}^{n} i$ equals 14!, which exceeds 2^{32}, and to avoid problems with the input of large numbers, we used bound variables $c_1, \ldots, c_n$ and constraint propagation to evaluate it.

5.10.2 Results

Tables 5.1 and 5.2 compare the proposed approaches on the problems defined in the previous subsection. We used a chronological variable selection strategy and a bisection branching, and in all experiments we searched for all solutions, traversing the entire search tree by means of depth-first leftmost-first chronological backtracking. The first two columns of table 5.1 list the number of variables and the DRFs that were used. Column nodes lists the size of the search tree, including failures and solutions. The next two columns list the number of times that a DRF was executed, and the percentage of these activations that the domain of a variable was actually modified. For the *opt* problem, the DRF that implements the optimization is not counted, and its activation is not taken into account. The elapsed times in the last column are the minimum times (in seconds) recorded for 5 runs on a 1200 MHz Athlon CPU.

Table 5.2 lists measured numbers of basic interval operations. Note that for approach 1b, there are two versions of the division and addition operations: one for integer intervals, and one for intervals of reals of which the bounds are rational numbers (marked $\mathcal{Q}$). Columns multI and multF list the numbers of multiplications of two integer intervals, and of an integer interval and an integer factor, respectively. These are different operations in our implementation.

For the *cubes*, *opt*, and *sumprod* problems, the constraints are already in simple form, so approaches 1a, 1b and 2b are identical. For *cubes* and *opt* all nonlinear terms involve a single multiplication or exponentiation, so for these experiments also approaches 2a and 3c are the same. For both versions of the *fractions* problem, and for *sumprod*, no exponentiations are used, so versions a, b, and c of approach 3 are identical.

The results of these experiments clearly show the disadvantage of implementing exponentiation by means of multiplication: the search space grows because we increase the number of variable occurrences and lose the information that it is the same number that is being multiplied. For *opt* and approach 3a, the run did not complete within reasonable time and was aborted.

Columns E and I of table 5.1 compare the propagation achieved by our approaches with two other systems, respectively ECL^{i}PS^e Version 5.6 [WNS97] using the ic library, and ILOG Solver 5.1 [Ilo01] using type ILOINT. For this purpose we ran the test problems without search, and compared the results of constraint propagation. A mark '=' means that the computed domains are the same, '+' that our approach achieved stronger propagation than the solver that we compare with, and '-' that propagation is weaker. For *cubes*, ECL^{i}PS^e computes the same domains as those computed according to approach 3b, so here the reduction is stronger than for 3a, but weaker than for the other approaches. For *opt* ECL^{i}PS^e and ILOG Solver compute the same domains. These domains are narrower than those computed according to approaches 3a and 3b, but the other approaches achieve stronger reduction. In all other cases except for *kyoto*

	nvar	nDRF	nodes	activated	%eff.	elapsed (sec.)	E	I
cubes								
1,2b	5	14	169,755	1,876,192	9.52	9.60	+	=
2a,3c	9	22	169,755	2,237,590	16.28	6.46	+	=
3a	13	34	206,405	3,011,749	20.02	8.46	-	-
3b	13	34	178,781	2,895,717	20.62	8.61	=	-
opt								
1,2b	4	7	115,469	5,187,002	42.16	20.97	+	+
2a,3c	8	15	115,469	9,800,017	60.00	21.47	+	+
3a	10	21	?	?	?	?	-	-
3b	10	21	5,065,195	156,906,444	46.49	406.39	-	-
fractions1								
1a	9	154	11,289	1,193,579	3.65	16.00	=	=
1b	9	154	7,879	734,980	3.45	17.99	=	=
2a	37	210	11,289	1,410,436	23.27	4.95	=	=
2b	32	200	11,289	1,385,933	21.65	5.41	=	=
3	43	208	11,131	1,426,204	27.76	5.07	=	=
fractions2								
1a	12	105	2,449	270,843	9.72	0.58	=	=
1b	12	105	989	94,894	9.12	0.55	=	=
2a	20	121	2,449	350,390	22.19	0.48	=	=
2b	15	111	2,449	301,865	17.50	0.46	=	=
3	22	123	1,525	293,051	27.33	0.41	=	=
kyoto								
1a	5	37	87,085	3,299,814	6.09	21.80	=	=
1b	5	37	87,085	3,288,461	5.94	46.12	+	+
2a	13	53	87,085	3,781,514	23.02	10.73	=	=
2b	12	51	87,085	3,622,461	21.45	11.06	=	=
3a	16	60	87,087	4,276,066	26.70	10.25	=	=
3b	16	60	87,085	4,275,957	26.70	10.34	=	=
3c	16	59	87,085	3,746,532	23.26	9.33	=	=
sumprod								
1,2b	28	82	230,233	10,910,441	7.91	112.25	=	=
2a	30	86	230,233	9,196,772	9.39	89.37	=	=
3	54	134	55,385	3,078,649	18.01	26.57	=	=

Table 5.1: Statistics and comparison with other solvers

	root	exp	div	multI	multF	sum	total
cubes							
1,2b	1,182	4,224	0	0	4,756	4,245	14,408
2a,3c	180	181	0	0	4,756	4,245	9,363
3a	0	0	589	438	4,927	4,363	10,317
3b	192	198	384	198	4,842	4,305	10,121
opt							
1,2b	2,299	4,599	1,443	1,444	11,064	5,187	26,037
2a,3c	1,636	1,538	2,150	738	8,138	4,445	18,645
3a	?	?	?	?	?	?	?
3b	21,066	18,106	54,172	18,285	106,652	57,470	275,751
fractions1							
1a	0	0	868	28,916	14,238	13,444	57,466
1b	0	0	51	11,892	8,010	6,727	29,584
			1,550 $\mathcal{Q}$			1,355 $\mathcal{Q}$	
2a	0	0	734	933	4,736	4,669	11,071
2b	0	0	776	1,509	5,292	5,147	12,725
3	0	0	693	339	4,835	4,769	10,636
fractions2							
1a	0	0	142	690	304	212	1,348
1b	0	0	19	127	59	26	344
			65 $\mathcal{Q}$			49 $\mathcal{Q}$	
2a	0	0	124	149	138	94	505
2b	0	0	124	206	210	118	658
3	0	0	114	46	142	101	403
kyoto							
1a	735	11,041	1,963	13,853	10,853	13,946	52,390
1b	735	8,146	218	8,955	12,516	10,592	48,749
			4,310 $\mathcal{Q}$			3,277 $\mathcal{Q}$	
2a	383	759	1,591	484	5,324	7,504	16,044
2b	383	759	1,597	1,360	5,756	8,008	17,863
3a	0	0	1,991	578	5,324	7,505	15,398
3b	< 0.5	< 0.5	1,990	578	5,324	7,504	15,397
3c	1	1	1,554	484	5,324	7,504	14,868
sumprod							
1,2b	0	0	4,032	100,791	85,419	149,479	339,721
2a	0	0	2,186	27,948	81,728	149,479	261,340
3	0	0	609	205	25,799	46,960	73,573

Table 5.2: Measured numbers (thousands) of interval operations

and approach 1b the results of all three solvers are the same.

For the fractions puzzle, the symbolic manipulation of approach 1b reduces the search tree by a factor 0.70 for the first representation, and by a factor 0.40 for the second. However, this reduction is not reflected in the timings. For *fractions1* the elapsed time even increases. The reason is that computing the domain updates involves adding intervals of real numbers. The arithmetic operations on such intervals are more expensive than their counterparts on integer intervals, because the bounds have to be maintained as rational numbers. Arithmetic operations on rational numbers are more expensive because they involve the computation of greatest common divisors. For *kyoto* the symbolic manipulation did not reduce the size of the search tree, so the effect is even more severe.

In general, the introduction of auxiliary variables leads to a reduction of the number of interval operations compared to approach 1a. The reason is that auxiliary variables prevent the evaluation of subexpressions that did not change. This effect is strongest for *fractions1*, where the main constraint contains a large number of different power products. Without auxiliary variables all power products are evaluated for every *POLYNOMIAL EQUALITY* rule defined by this constraint, even those power products the variable domains of which did not change. With auxiliary variables the intervals for such unmodified terms are available immediately, which leads to a significant reduction of the number of interval multiplications. For *sumprod*, the difference between approaches 1a and 2a is a bit artificial, because the operations that are saved involve the computation of the constant term $c_1 \cdot \ldots \cdot c_n$. A comparable number of interval additions can be saved if we introduce a variable for the constant term $c_1 + \ldots + c_n$. If we add these variables to the CSP all variants of approaches 1 and 2 are essentially the same.

The effect that stronger reduction is achieved as a result of introducing auxiliary variables, mentioned in Section 5.7, is seen for both representations of the *fractions* benchmark, and prominently for *sumprod*. In the latter case, this effect depends on a decomposition of the term $\prod_{i=1}^{n} x_i$ as $x_1 \cdot (x_2 \cdot (\ldots \cdot (x_{n-1} \cdot x_n) \ldots))$, with an auxiliary variable per pair of brackets. The decomposition then matches the chronological ordering used to select the variable for branching. If the ordering is reversed, the number of nodes is equal to that of the other approaches. The effect described in Section 5.6 is not demonstrated by these experiments.

If we do not consider the symbolic manipulation of approach 1b, then approach 3c leads to the smallest total number of interval operations in all cases, but the scheduling mechanism discussed in Section 5.9 is essential for a consistent good performance. If for example the schedule is omitted for *opt*, the number of interval operations almost triples, and performance of approach 2a and 3c is then much worse than that of approach 1a.

The total numbers of interval operations in table 5.2 do not fully explain all differences in elapsed times. One of the reasons is that different interval operations have different costs. Also some overhead is involved in applying a DRF, so if the number of applications differs significantly for two experiments, this influences the elapsed times as well (*opt*, 1a, 2a, *fractions2*, 2a, 2b). The elapsed times are not the only measure that is subject to implementation details. For example, we implemented division by a constant interval $[-1..-1]$ as multiplication by a constant, which is more efficient in our implementation. Such decisions are also reflected in the numbers reported in table 5.2.

5.11 Conclusions

In this chapter we discussed a number of approaches to constraint propagation for arithmetic constraints on integer intervals. To assess them we implemented them using the OpenSolver framework, and compared their performance on a number of benchmark problems. We can conclude that:

- Implementation of exponentiation by multiplication gives weak reduction. In our third approach $x = y^n$ should be used as an atomic constraint.

- The optimization of the first approach, where common powers of variables are divided out, can significantly reduce the size of the search tree, but the resulting reduction steps rely heavily on the division and addition of rational numbers. These operations are more expensive than their integer counterparts, because they involve the computation of greatest common divisors. As a result, our implementation of this approach was inefficient.

- Introducing auxiliary variables can be beneficial in two ways: it may strengthen the propagation, as discussed in Sections 5.6 and 5.7, and it may prevent the evaluation of subexpressions the variable domains of which did not change.

- As a result, given a proper scheduling of the rules, the second and third approach perform better than the first approach without the optimization, in terms of numbers of interval operations. Actual performance depends on many implementation aspects. However for our test problems the performance of variants 2a, 2b and 3c does not differ much, except for one case where the decomposition of a single multiplication of all variables significantly reduced the size of the search tree.

Because of the inherent simplicity of the reduction rules and the potential additional reduction of the search tree, approach 3c is our method of choice. We decompose polynomial constraints into multiplication, exponentiation, and linear constraints. A hierarchical scheduling of the resulting reduction rules is essential for improving the performance of approach 1a. As we noted at the end of Section 5.9.2, it may be possible to improve the performance further by treating the decomposition into atomic constraints as an optimization problem, minimizing the number of auxiliary variables.

Given that approach 1b can achieve a significant reduction of the search tree, it would be interesting to combine it with approach 3c. Depending on the effect of the symbolic manipulation, a selection of the optimized rules that enforce a particular constraint according to approach 1b could be used as redundant rules. In this case, the internal computations need not be precise, and we could maintain the rational bounds as floating-point numbers, thus avoiding the expensive computation of greatest common divisors.

Note that our characterization of the third approach is limited to version 3a. A characterization of the linear equality constraints can be found, for example, in [Apt03], but the atomic constraint $x = y^n$ is not covered. Also a proper characterization of the first and second approach may help us formalize the improved reduction observed in Sections 5.6 and 5.7. Because of the lengthy proofs involved, we have left this as an opportunity for future work.

We would like to point out that the operators studied in this chapter are similar to those for enforcing hull consistency, which we discussed in Section 4.5 for floating-point intervals. In Chapter 7 we will implement a stronger notion of consistency called box consistency, and apply this both to floating-point intervals and integer intervals. The operators for enforcing box consistency will be composed from the facilities introduced here and in Section 4.5, plus a generic operator for nested search.

So far we have only seen examples of constraint solvers on a single domain type. In the next chapter we will study a hybrid solver, where some of the facilities introduced here, namely those for optimization, are combined with reduction operators on special-purpose domain types.

Chapter 6
Job-Shop Scheduling in OpenSolver

As a case study, we demonstrate how OpenSolver can be configured as a solver for the ***job-shop scheduling problem*** (JSSP). For this purpose we will introduce a small number of dedicated plug-ins. Because we can rely on existing facilities for search and optimization, building this specialized solver involves only a modest implementation effort. Two of the new plug-ins are variable domain types, and this particular OpenSolver configuration demonstrates a technique that we refer to as ***constraining special-purpose domain types***. We will conclude the chapter with a discussion of the pros and cons of this technique. Also, JSSP is considered to be a non-trivial problem that is representative for many scheduling problems that occur in practice. As such, this case study demonstrates that our approach leads to solvers that have a relevance beyond puzzle-type problems such as the ones we used in the previous chapters.

6.1 Introduction

The tools that are available to model a combinatorial problem as a CSP differ for various constraint solvers. The basic machinery typically includes:

- finite domain and interval representations for the domains of integer variables, and a floating-point interval representation for real numbers,
- arithmetic constraints on numerical variables,
- global constraints, such as the all_different constraint.

Depending on the problem that we want to model, these facilities may or may not be fully adequate to construct a CSP.

In an open-ended constraint solver, we have the possibility to add new facilities. If a problem is hard to model, it may be easier to add a few special-purpose facilities, such as a new constraint. This may lead to a CSP that is much closer to the original problem than a CSP that uses only the built-in primitives. In a library-based system, like ILOG Solver, the user can write new constraints and goals to guide the search by writing new subclasses of base classes provided by the library. In logic programming systems, like ECL^{i}PS^e, such facilities can be written in the host language, usually Prolog. Here we will be using the readily available facilities for search and optimization of OpenSolver, and complement these with plug-ins for special purpose domain types and reduction operators for the job-shop scheduling problem.

The chapter is structured as follows: In Section 6.2 we introduce the job-shop scheduling problem, and describe an algorithm for solving it. In Section 6.3 we detail the implementation of this algorithm in OpenSolver. We conclude in Section 6.4 with a discussion of our approach from a software engineering perspective. An evaluation of our implementation on a set of benchmark problems is postponed until the next chapter, where we compare it with an alternative implementation based on nested search.

6.2 The Job-Shop Scheduling Problem

A JSSP instance consists of a set of ***activities***, and a number of ***machines*** (in general, ***resources***). An activity is characterized by the machine that it must be processed on, and by a ***processing time***, which specifies for how long the machine is needed. JSSP is a non-preemptive scheduling problem, which means that activities cannot be interrupted. They acquire the machines for their full processing time. Activities are grouped in ***jobs***, where all activities of a job have to be executed in a specified order. The problem is to find for each activity an interval in which it can be executed on the specified machine, such that no two activities require the same machine simultaneously (the ***capacity constraint***: for JSSP, the machines have a capacity of one activity), and such that the ***precedence constraints*** inside the jobs are respected. An optimal schedule minimizes the completion time of the activities that finish last.

The table in Figure 6.1(a) specifies an example JSSP consisting of three jobs, each having three activities that require three different machines. Each row of the table specifies the numbers of the machine needed for the three activities, and between parentheses the processing time of the activity. An optimal schedule for this JSSP instance is depicted in Figure 6.1(b). The three bars represent the machines, with the activities drawn on them. Black areas correspond to machines being idle.

Algorithm 6.1 is a basic JSSP solver, due to Baptiste, Le Pape, and Nuijten [BLPN01]. It is a branch-and-propagate algorithm, where branching determines the relative order of the activities, expanding a partial schedule until all activities have been ordered, and constraint propagation verifies that the current partial schedule does not violate any precedence or capacity constraints. Each activity

A_i has the following data associated with it:

- its ***release date***, or earliest possible starting time, denoted r_i,
- its ***deadline***, or latest possible completion time, denoted d_i, and
- its processing time, denoted p_i.

From these follow:

- the ***earliest possible completion time***, denoted ect_i, and
- the ***latest possible starting time*** of the activity, denoted lst_i.

The rule for step 1 of the algorithm is to select the machine that is the ***critical resource***. Criticality is measured by comparing supply and demand for the resources. Supply is the time window given by the earliest release date, and the latest deadline among all activities that require the machine. Demand is their total processing time. A machine with the smallest difference between these two quantities, which is called the resource's ***slack time***, is selected. The rule for step 2 is to select the activity with the earliest release date. The latest starting time is used for breaking ties.

Constraint propagation in step 3 of the algorithm narrows the time windows for the activities by increasing release dates and decreasing deadlines to enforce the precedence and capacity constraints. In [BLPN01] a number of propagation techniques are presented for these constraints. From this collection we used the following techniques:

- For two consecutive activities A_i and A_j of a job, we ensure that $d_i \leq lst_j$, and $ect_i \leq r_j$ to enforce the precedence constraint.
- The disjunctive constraint. For every pair of activities A_i and A_j that require the same machine we know that either A_i precedes A_j, or A_j precedes A_i. Therefore, if we find that $ect_j > lst_i$, we know that A_j cannot precede A_i, and we can propagate the reverse constraint by enforcing $d_i \leq lst_j$, and $ect_i \leq r_j$, and similarly for the case that $ect_i > lst_j$.
- The ***edge finding*** algorithm, implementing further pruning for the capacity constraint by identifying activities that must execute first, or last, in a given set of activities. We implemented the variant of the algorithm described in [BLPN01]. Its time complexity is quadratic in the number of activities that require the same resource.

Edge finding was introduced by Carlier and Pinson [CP89], and provided a breakthrough in job-shop scheduling because it allowed that for the first time, the famous benchmark problem FT10 (also known as MT10) was solved.

(a)

	A1	A2	A3
J1:	2(5)	1(2)	3(3)
J2:	2(4)	3(4)	1(1)
J3:	1(3)	3(3)	2(4)

(b)

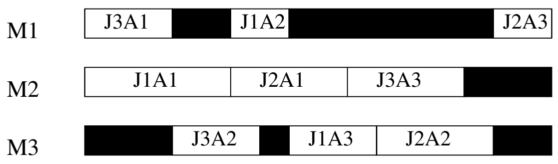


Figure 6.1: a 3 × 3 JSSP instance and a minimal schedule for it

---

1. Select a machine for which the activities are not fully ordered
2. Select an activity to execute first among the unordered activities of that machine. ***Post the corresponding precedence constraints***. Keep the the other activities as alternatives to be tried upon backtracking.
3. Verify feasibility of the partial schedule by means of constraint propagation. Backtrack upon failure.
4. Iterate step 2 until all activities on the selected machine are ordered.
5. Iterate step 1 until all activities on all resources are ordered.

Algorithm 6.1: Basic algorithm for solving the job-shop scheduling problem

6.3 JSSP in OpenSolver

In this section we describe the plug-ins that were developed to implement algorithm 6.1 as an OpenSolver configuration. First, two special-purpose variable domain types were introduced.

`Activity`

These are data structures consisting of three integers, that hold the release date, deadline, and processing time of an activity. Branching on these variables will enumerate candidate starting times, or candidate completion times, similar to left/right-enumeration, which is illustrated in Figure 4.2(b) and (c) for finite domains. Algorithm 6.1 branches only on the order, and not on the actual timing of the activities, though, but this facility will be used in the solver of Section 7.5.4. The size reported by an `Activity` domain is one plus the width of its time window minus its processing time, which is the actual number of possibilities for scheduling the activity.

The command for introducing an activity A_i, with (initial) release date and deadline r_i and d_i and processing time p_i is

```
AUX $A_i$ IS Activity { $r_i$, $p_i$, $d_i$ };
```

Here the keyword `AUX` is used instead of `VARIABLE` to mark an activity as an auxiliary variable.

`Ranking`

OpenSolver does not directly support posting and retracting constraints, as specified in step 2 of Algorithm 6.1. Instead, we introduced a domain type `Ranking` for exploring alternative assignments of a machine to activities. A value for a variable of type `Ranking` is essentially a permutation of the numbers $0 \ldots n_a - 1$ that specifies a particular order in which n_a activities that require the same machine are executed. The domain of such a variable is a set of permutations. It is implemented as a data structure consisting of

- an array of length n_a, containing (initially in that order) the different indices 0 through $n_a - 1$.
- an integer n_o, indicating that the first n_o entries of the array have been ordered. The remaining $n_a - n_o$ entries are considered to be unordered.
- an index $i \in [0..n_a - n_o - 1]$, indicating a specific entry in the unordered part of the array. This is the next candidate for expanding the ordered part.

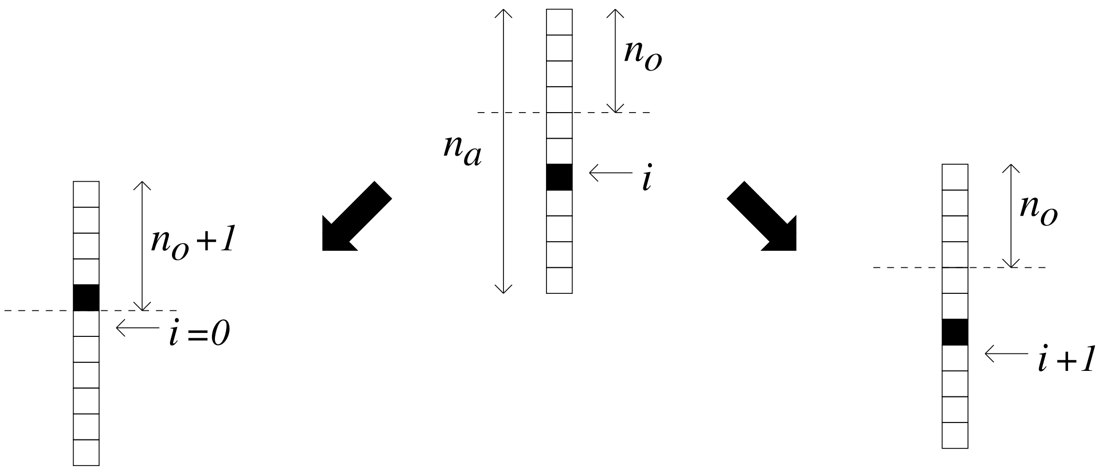


Figure 6.2: Branching on a variable of domain type `Ranking`

The search tree is expanded by splitting variables of domain type `Ranking`. this is depicted in Figure 6.2. In the left branch, the element indicated by the index i is added to the ordered set, and i is set to point to the first element of the new unordered set. In the right branch, the ordered set is unchanged, and i points to the next candidate in the unordered set. In other words, the domain in the left branch corresponds to all permutations where the number pointed to by i is the next element, and the right branch corresponds to the set of all permutations where this number is ***not*** the next element. If $i > n_a - n_o - 2$ then the `Size()` method of a `Ranking` domain will return 1, indicating that the permutation is fixed. Otherwise a value greater than 1 is returned (detailed below).

The code for introducing a `Ranking` variable for machine m_j is the following.

`VARIABLE` m_j `IS Ranking` $\{ n_a \}$;

A number of reduction operators implement the interaction between logical variables of domain types `Activity` and `Ranking`:

`Precedes`

This propagation operator enforces the precedence constraint on two activities, as described in the previous section. The operation is similar to enforcing bounds consistency for the linear inequality constraint on integer intervals (see Chapter 5), but now operating on `Activity` domains. This operator is applied to all consecutive pairs of activities $A_i,\ A_j$ of the same job:

`DRF Precedes` $\{ A_i,\ A_j \}$;

`RankActivities`

This operator applies to a variable of domain type `Ranking`, plus n_a `Activity` variables. It is imposed on the activities that require the same machine. On the activities whose indices are in the ordered part of the `Ranking` data structure, the corresponding precedence constraints are enforced. A precedence constraint is also enforced on all possible combinations of the last ordered activity and an unordered activity.

The syntax for introducing the DRF that enforces the constraint that activities $A_{i_1}, \ldots, A_{i_{n_a}}$ are processed on the machine with `Ranking` variable m_j is the following.

DRF RankActivities $\{ m_j,\ A_{i_1}, \ldots,\ A_{i_{n_a}} \}$;

We would like to remark that `RankActivities`, and some of the other reduction operators used in this chapter, have all the properties of a ***global constraint***, as described in Section 4.2. Indeed, as we already discussed there, global constraints could be implemented in this way, and special-purpose domain types can be used to store any information that needs to be maintained during constraint propagation or search.

The `Precedes` and `RankActivities` operators already provide the necessary ingredients for a JSSP solver that is sound and complete. From this point of view, the other plug-ins are an optimization:

`Disjunctive`

The `Disjunctive` plug-in implements the constraint that two activities that require the same machine cannot overlap in time. It is applied to all pairs of activities $A_i,\ A_j$ that require the same machine:

DRF Disjunctive $\{ A_i,\ A_j \}$;

`EdgeFinding`

This plug-in implements the edge finding algorithm. Its specifier string lists the n_a `Activity` variables that require the same machine:

DRF EdgeFinding $\{ A_{i_1}, \ldots,\ A_{i_{n_a}} \}$;

`DecorateRanking`

This is a branching operator that serves only to ***decorate*** a variable of domain type `Ranking` with the information that is needed to implement the variable selection strategy described in Section 6.2. Because no subdomains are created by the branching operator, we call it a ***pseudo branching operator***. Like `RankActivities`, it applies to a variable of domain type `Ranking`, and a sequence of n_a activities requiring the same machine. For the unordered activities, it calculates the difference between the size of the available time window, and the total processing time, as described in the previous section. `Ranking` variables that have not been subject to branching report this difference plus one as the size of the domain. In combination with a regular fail-first variable selection strategy, this ensures that activities on critical resources are tried first in branching.

In addition to this, the unordered part of the array of Figure 6.2 is sorted according to increasing release date and latest starting time. When splitting a `Ranking` domain, this results in the value section strategy for step 2 of Algorithm 6.1. The specifier string for the `DecorateRanking` plug-in is identical to that for the `RankActivities` plug-in:

`DRF DecorateRanking{` $m_j,\ A_{i_1}, \ldots,\ A_{i_{n_a}}$ `};`

Algorithm 6.1 computes feasible schedules instead of minimal schedules. Optimization can be realized by adding an extra step 6, which backtracks after a solution, and constrains subsequent solutions to have a shorter schedule length than the current solution. The last solution found is then the optimal schedule. The resulting branch-and-bound search is implemented by introducing an activity `makespan` having processing time 0, which is scheduled to start after the last activity of each job. This approach is described in [VHPP00]. Via another special-purpose operator `BoundActivity`, the domain of an integer interval variable is constrained to range from the release date to the deadline of this activity. The `Optimize` operator, introduced in Section 5.9.2, constrains the length of the schedule to decrease for subsequent solutions.

Program 6.1 shows an example configuration file for a JSSP instance. Such files are generated from JSSP specifications by a small preprocessor program. The initial release date for the `makespan` activity is set to the maximum processing time among all jobs and all sets of activities that are assigned to the same machine. Its initial deadline is the sum of all activity processing times. All variables except the instances of `Ranking` are auxiliary variables. This means that for any feasible ordering that is found, the time windows for the individual activities may be wider than their processing times. To avoid having to search actively for the minimum makespan of a given ordering, which can easily be achieved by letting all activities start on their release dates, we use an additional operator `FixMakespan`. For all (suboptimal) solutions encountered during the search, this pseudo branching operator collapses the time window for the `makespan` activity such that it can only be scheduled at its release date. The minimal schedule for a feasible permutation of activities follows. We implemented a variant of the `Precedence` and `Disjunctive` operators that apply to any number of activities. We found that this gives slightly better performance than the binary operators described above. A schedule of reduction operators, in the language of Figure 4.1, is generated to coordinate constraint propagation such that application of the expensive edge finding operators is postponed until a fixed point of the other operators is reached. This fixed point is recomputed if edge finding reduces the time window of an activity.

Instead of branch-and-bound, the solver described in [BLPN01] performs a bisection search for the minimal schedule. We compare both optimization algorithms on a set of benchmark problems in Section 7.5.1, where we use nested search to implement the latter alternative.

6.4 Discussion

Constraining Special-Purpose Domain Types

The scheduling facilities discussed here were developed primarily for the purpose of testing OpenSolver on benchmarks of arguable relevance, but we believe that the technique of constraining special-purpose data structures, like the `Ranking` and `Activity` variables, is interesting in itself. On the one hand, it illustrates the use of OpenSolver as an abstract branch-and-prune tree search engine, that can be configured in different ways for different tasks. Again, the effort of developing these plug-ins is modest compared to developing a JSSP solver from scratch. While these plug-ins have little relevance outside the specific application of scheduling, the framework allows for a seamless integration with existing facilities, notably for search, optimization, and parallel processing.

On the other hand, scheduling is an example of a combinatorial problem for which an efficient translation to regular constraint programming primitives is not straightforward. For this reason, other platforms have built-in facilities (for example, the OPL `Resource` data type [VH99]) for scheduling as well (implementing permutation-based JSSP solving on top of standard modeling facilities is explored in [Zho97]). In general, these “heavy-weight” application-specific domain types allow us to write constraint programs that are very close to the original problem, and are hence more easily verified to be correct. A specific advantage of our approach is that these facilities are not hard-wired in the system. We expect that the technique can be applied to other combinatorial (optimization) problems, for which a direct translation to regular constraints is not straightforward.

A disadvantage of our approach is that heavy-weight domain types like `Ranking` do not lend themselves naturally for domain reduction by means of constraint propagation. For implementing Algorithm 6.1 this is not problematic, but it impedes the implementation of several possible improvements of this algorithm. For example, if the disjunctive constraint implies that one activity always precedes another in a particular branch of the search tree, we cannot reduce the domain of a `Ranking` variable accordingly. Permutations that violate this deduced constraint will be generated over and over again (trashing), and have to be refuted by propagation of the disjunctive constraint. This may lead to substantially larger search trees.

```
AUX J0A0 IS Activity {0,5,29};
AUX J0A1 IS Activity {0,2,29};
AUX J0A2 IS Activity {0,3,29};
AUX J1A0 IS Activity {0,4,29};
AUX J1A1 IS Activity {0,4,29};
AUX J1A2 IS Activity {0,1,29};
AUX J2A0 IS Activity {0,3,29};
AUX J2A1 IS Activity {0,3,29};
AUX J2A2 IS Activity {0,4,29};
AUX makespan IS Activity {13,0,29};
AUX imakespan IS IntegerInterval { };
VARIABLE M0 IS Ranking { 3 };
VARIABLE M1 IS Ranking { 3 };
VARIABLE M2 IS Ranking { 3 };
DRF RankActivities { M0, J0A1, J1A2, J2A0 };
DRF RankActivities { M1, J0A0, J1A0, J2A2 };
DRF RankActivities { M2, J0A2, J1A1, J2A1 };
DRF Precedes { J0A0,J0A1,J0A2, makespan };
DRF Precedes { J1A0,J1A1,J1A2, makespan };
DRF Precedes { J2A0,J2A1,J2A2, makespan };
DRF Disjunctive { J0A1,J1A2,J2A0};
DRF Disjunctive { J0A0,J1A0,J2A2};
DRF Disjunctive { J0A2,J1A1,J2A1};
DRF EdgeFinding{ J0A1, J1A2, J2A0 };
DRF EdgeFinding{ J0A0, J1A0, J2A2 };
DRF EdgeFinding{ J0A2, J1A1, J2A1 };
DRF DecorateRanking { M0, J0A1, J1A2, J2A0 };
DRF DecorateRanking { M1, J0A0, J1A0, J2A2 };
DRF DecorateRanking { M2, J0A2, J1A1, J2A1 };
DRF BoundActivity { makespan, imakespan };
DRF FixMakespan { imakespan, imakespan };
DRF Optimize { -imakespan };
DRF FailFirst { 0, M0, M1, M2 };
SCHEDULER ChangeScheduler { schedule =
   ( { 0, 1, 2, 3, 4, 5, 6, 7, 8, 12, 13, 14, 15, 16, 17, 18 },
     9, 10, 11 ) };
```

Program 6.1: OpenSolver configuration for the JSSP instance of Figure 6.1

One way to avoid this problem would be to “virtually” reduce the domain of `Ranking` variables by recording this kind of information in the domain data structures, and preventing that the erroneous permutations are generated. This way, more and more constraint solving functionality is pushed into the basic value selection strategy offered by the domain type. This is all the more interesting from a solver cooperation perspective, but other CSP formulations are better suited for this kind of propagation. One option would be to use a Boolean variable for every pair of activities on the same machine, and to use so-called ***reified constraints*** to enforce the precedence relations that these variables encode. A reified constraint is a constraint of the form $b \leftrightarrow C$, with b a Boolean variable and C a constraint [SS02]. It reads “b if and only if C,” which entails that if b is false, the negation of C is enforced, and if C becomes redundant or falsified, the domain of b is reduced accordingly. Reified constraints are typically used to express logical connectives between constraints, such as $C \leftrightarrow C'$.

Search

Another issue is the flexibility of our search strategy. Languages like OPL and SALSA [LC02] offer richer facilities for specifying search procedures. To illustrate the limitations of the current set of plug-ins, consider the variable selection strategy of Algorithm 6.1. Once a machine is selected, all tasks that require it are scheduled before another machine is considered. This was implemented by manipulating the size of the variable domain that is reported by `Ranking` instances. When a domain of type `Ranking` is split, as depicted in Figure 6.2, the resulting subdomains will always report size 2. Domains that have never been split report a size greater than 2 that reflects the total slack time of the resource. As a result, a fail-first variable selection strategy will prefer variables that have been split before, which leads to the required strategy.

A more elegant implementation of the variable selection strategy would be to compose it from two basic strategies. Suppose that in addition to fail-first (FF) we have at our disposal a strategy R that always selects the variable that has been selected most recently. A SALSA expression for our variable selection strategy would then be

$$(\mathsf{R}^\star \diamond \mathsf{FF})^\star \diamond \mathsf{cont}$$

This specifies that we keep applying R until no subdomains are generated by splitting the most recently split variable. This happens at the beginning of the search. In that case we apply fail-first. This composite procedure is repeated until a leaf of the search tree is reached, where instead of terminating the search, we continue the exploration.

A similar composition can be achieved in OpenSolver through the adapter mechanism. We can implement a branching operator `CompositeBranching` that accommodates two, or any number of branching operators. When one of these internal branching operators does not yield any subdomains, it applies the next.

The following code would then result in the variable selection strategy for job-shop scheduling:

```
DRF CompositeBranching {
   DRF RepeatBranching { 0, M0, M1, M2 };
   DRF AnnotateVariableSelection { FailFirst { 0, M0, M1, M2 } };
};
```

where `RepeatBranching` branches on the most recently selected variable. The required variable index can be maintained using an annotation, like the `RoundRobin` plug-in does (see Section 4.1.2). Here we assume that `FailFirst` is modified to maintain this annotation through an adapter `AnnotateVariableSelection`.

Using this composition, we do not have to set the size of `Ranking` subdomains to two. If, as an experiment, we would like to reconsider the choice of the resource each time we extend the partial schedule with a new activity, we would just have to replace the above composite branching strategy with `FailFirst`. Currently we would have to modify and recompile the `Ranking` plug-in for this experiment. In OPL, Algorithm 6.1 is realized by nesting the value selection in a `while` statement, that prevents a new `Resource` to be selected while the current resource has not been completely ranked. For our experiment we would simply have to remove this `while` statement (see [VHPP00]).

For computing the slack times, and sorting the unordered part of the arrays of Figure 6.2, there are no good alternatives for using pseudo-branching operators. If we want to make this the responsibility of the branching operator, it would need to be aware of the relation between the ranking variables and the activities. This is possible, but would result in special-purpose branching operators, while currently, the problem-specific details are hidden in the pseudo branching operator `DecorateRanking`. Also we could consider to implement facilities for programmable value selection strategies, as supported by SALSA `Choice` specifications. However, it should be realized that the OpenSolver configuration language is a lower-level language than SALSA and OPL. As we discussed in Section 4.1.1, it should be seen as an assembly language, and consequently we have many options for realizing specific techniques. For the present problem, the pseudo branching operators seem a good solution. However, for a coherent set of plug-ins, languages that are closer to OPL and SALSA could be implemented on top of OpenSolver, as a compiler that generates configuration specifications.

6.5 Concluding Remarks

In this chapter we have demonstrated how OpenSolver can be configured as a basic solver for the job shop scheduling problem. This involves a technique that we refer to as constraining special-purpose domain types, which entails that new domain types and reduction operator plug-ins are added when problems cannot be modeled efficiently with the facilities that are readily available. Through its open-ended nature, OpenSolver is particularly suited for this technique, and it demonstrates its use as an abstract branch-and-propagate tree search engine.

The job-shop scheduling problem is an interesting test case because it is computationally expensive, and because it is considered to be representative for many scheduling problems that occur in practice. An evaluation of the efficiency of the solver that we described here is postponed until Section 7.5.1 in the next chapter, where we compare it with an alternative optimization scheme on a set of benchmark problems.

Chapter 7

Applications of Nested Search

Nested search entails that a limited branch-and-propagate tree search is performed during constraint propagation. In this chapter we propose a generic reduction operator for nested search, and investigate the extent to which it can be used to express a number of well-known techniques, from different application domains, for improving the efficiency of constraint solving. Generalizing solving techniques has several advantages. From a modeling perspective, the technique extends easily to other application domains, and from a software engineering perspective, it avoids duplicate code with only small variations for different applications.

This is the first of three chapters that demonstrate the use of OpenSolver as a software component. In this case OpenSolver implements the generic reduction operator for nested search. Consequently, we gain rich facilities for expressing the nested search, at the cost of an overhead for using a general-purpose constraint solver for very specific search problems. We demonstrate that despite this overhead, our approach leads to a viable implementation of the techniques that we are interested in.

7.1 Introduction

Constraint propagation is usually implemented as the repeated application of reduction operators that enforce some form of local consistency, such as arc consistency, or an approximation thereof. In this context 'local' means that only individual constraints, applying to a small subset of the variables are considered when removing values from the domains of the variables. For arithmetic constraints, these individual constraints are usually the result of a decomposition of complex constraints into atomic constraints, for which the resulting form of consistency is weaker than for the original constraints.

Sometimes the efficiency of constraint solving can be improved by enforcing a stronger, less local form of consistency. Operators that enforce such a form of consistency typically still update the domain of a single variable, but more than a single constraint is considered when removing values. This is achieved by actively ***trying*** different subdomains for the variable that we want to reduce, and then ***verifying*** by means of constraint propagation that this does not violate the combined constraints. The resulting ***trial-and-refutation*** mechanism can be seen as a limited form of branch-and-propagate search, where branching is on the domain of a single variable only.

We will look at two examples of such operators: enforcing ***box consistency*** for arithmetic constraints, and removing (***shaving*** off) unfeasible activity starting times and completion times in scheduling problems. Box consistency can be explained as to consider all atomic constraints in the decomposition of a single user constraint, and is therefore a stronger notion of consistency than what is achieved for the decomposition alone. Shaving considers the full set of capacity and precedence constraints in a scheduling problem, when trying to refute possible values for the variable that it is applied to. We will show that both operators can be expressed as applications of a generic operator for nested search. A third application is ***optimization*** by means of a bisection search in the range for the outcome of an objective function.

Instead of implementing dedicated operators for each of these techniques, our approach entails that these operators are ***composed*** from a limited set of basic facilities, which includes the generic operator for nested search. This is an advantage in hybrid solvers, supporting multiple domain types, where we want to avoid that techniques like shaving are available only for a subset of the domain types. It is of even greater importance for open-ended solvers, where the set of domain types can be extended. A disadvantage is that a generic operator is likely not as efficient as a dedicated implementation. We provide the results of experiments that show that despite a general-purpose constraint solver is used for the nested search, we still obtain a workable implementation.

The rest of this chapter is structured as follows. The reduction operator for nested search is defined in Section 7.2. In Section 7.3 we show how it can be used to define optimization, box consistency, and shaving. Section 7.4 details the implementation of the operator, and Section 7.5 describes the experiments.

7.2 An Operator for Nested Search

In this section we propose a generic operator for nested search. It is presented as a domain reduction function that, like all other DRFs, is used in the context of a branch-and-propagate search for a solved form of an ECSP. The DRF is evaluated during the constraint propagation phase, and may yield a smaller domain for one of the variables of the ECSP. Since we are defining a ***generic*** operator, the DRF for this operator is ***parameterized*** with some extra information. This extra information ***instantiates*** the DRF to perform a particular reduction step.

Inner and Outer ECSP

In this case, the information that parameterizes the DRF is another ECSP. So unlike the DRFs that we encountered so far, the DRF for nested search involves more than one ECSP:

1. The ECSP whose domains are reduced by the DRF. For regular DRFs, this is the only ECSP that we need to consider. Here we will call this ECSP the ***outer*** ECSP.

2. The ECSP that parameterizes the DRF. We will call this ECSP the ***parameter*** ECSP.

3. The parameter ECSP is combined with the domains that the DRF is evaluated for (detailed below). This results in a third ECSP that we will call the ***inner*** ECSP.

Evaluation of the DRF for nested search involves solving the inner ECSP by means of a branch-and-propagate search. The term ***nested search*** refers to this branch-and-propagate search on the inner ECSP, and emphasizes that it occurs as a single domain reduction step in the encompassing branch-and-propagate search on the outer ECSP.

The inner ECSP is a modified version of the parameter ECSP. It is obtained by replacing the domains of certain variables with the domains that the DRF is evaluated for. We introduce the following notation to describe this modification.

7.2.1. Definition. Let P be an ECSP with variables $x_1, \dots, x_m$ and corresponding domain types $\mathcal{T}_1, \dots, \mathcal{T}_m$. For $D_1 \in \mathcal{T}_1, \dots, D_n \in \mathcal{T}_n$, and $n \leq m$, let $(P, \langle D_1, \dots, D_n \rangle)$ denote the ECSP obtained by replacing in P the domains of the first n variables $x_1, \dots, x_n$ with $D_1, \dots, D_n$, and projecting the new sequence of domains on the constraints. □

7.2.2. Example. For

$$P := \langle \mathcal{C}_P \ ; \ x, y, z \in \{0,1,2\} \ ; \ D_x, D_y, D_z \in \mathcal{Z} \ ; \ \mathcal{A}_x, \mathcal{A}_y, \mathcal{A}_z \rangle,$$

$(P, \langle \{0,1\}, \{1,2\} \rangle)$ denotes the ECSP obtained by replacing the domain of x in P with $\{0,1\}$, and the domain of y in P with $\{1,2\}$:

$$\begin{aligned}(P, \langle \{0,1\}, \{1,2\} \rangle) = \langle \mathcal{C}'_P \ ; \ & x \in \{0,1\}, y \in \{1,2\}, z \in \{0,1,2\} \\ ; \ & D_x, D_y, D_z \in \mathcal{Z} \\ ; \ & \mathcal{A}_x, \mathcal{A}_y, \mathcal{A}_z \rangle\end{aligned}$$

where $\mathcal{C}'_P$ is $\mathcal{C}_P[\{0,1\}, \{1,2\}, \{0,1,2\}]$, the projection of the new domains on the constraints of P. The domains of x and y in P, and their sets of final domains are irrelevant for the purpose of this notation. The projection is needed only to maintain the property that constraints are subsets of Cartesian products of domains. □

Definition of the Operator

The operator for nested search is now defined as follows.

7.2.3. Definition. Given

- an ECSP P with at least n variables, and
- a variable x_j with $1 \leq j \leq n$,

let $\mathcal{T}_1, \ldots, \mathcal{T}_n$ be the domain types of the first n variables in P. We define the function

$$f_{P,x_j} : \mathcal{T}_1 \times \ldots \times \mathcal{T}_n \to \mathcal{T}_j$$

as follows:

$$f_{P,x_j}(D_1, \ldots, D_n) = \begin{cases} D'_j & \text{if } (P, \langle D_1, \ldots, D_n \rangle) \text{ is consistent} \\ \emptyset & \text{otherwise} \end{cases}$$

where if $P' := (P, \langle D_1, \ldots, D_n \rangle)$ is consistent, D'_j is the domain of x_j in a γ solved form of P', for some notion of local consistency γ. □

To elucidate this definition, note that

- P and P' are the argument ECSP and inner ECSP, respectively. f_{P,x_j} is a domain reduction function for an outer ECSP Q that has variables $x_1, \ldots, x_n$ and the corresponding domain types $\mathcal{T}_1, \ldots, \mathcal{T}_n$ in common with P. For our applications, P can always be defined such that $x_1, \ldots, x_n$ is a subsequence of the variables of Q.
- If P' is consistent, there may exist more than a single γ solved form of P'. In this case, the definition is not specific about which γ solved form delivers the outcome of the function. We will comment on this after the example below.

Further, recall from Definition 2.2.14 on page 16 that a γ solved form of an ECSP is a subproblem that is γ consistent, and whose domains are elements of their respective sets of final domains. γ refers to some notion of local consistency, e.g., $\gamma = \text{arc}$ for arc consistency.

Operationally, the evaluation of f_{P,x_j} on a given sequence of argument domains $D_1, \ldots, D_n$ consists of the following three steps (see also Figure 7.1).

1. Construction of the inner ECSP $(P, \langle D_1, \ldots, D_n \rangle)$
2. Branch-and-propagate search on the inner ECSP. This is the actual nested search.

$$\begin{tikzcd}
 & f_{P,x_j}(D_1,\dots,D_n) = \arrow[dr, bend left] & & & \\
1. & & \text{construct } P' := (P, \langle D_1,\dots,D_n \rangle) \arrow[d] & & \\
2. & & \text{solve } P' \arrow[dl] \arrow[dr] & & \\
3. & \text{consistent} \arrow[d] & & \text{inconsistent} \arrow[r] & \emptyset \\
 & \gamma \text{ solved form} \ \langle \mathcal{C}_P \ ; \ \dots, x_j \in D'_j, \dots \ ; \ \dots \ ; \ \dots \rangle \arrow[rrr] & & & D'_j
\end{tikzcd}$$

Figure 7.1: Evaluation of the DRF for nested search

3. If step 2 determines that the inner ECSP is inconsistent, $f_{P,x_j}(D_1,\dots,D_n)$ evaluates to $\emptyset$. Otherwise, $f_{P,x_j}(D_1,\dots,D_n)$ evaluates to the domain of variable x_j in the γ solved form that is found in step 2.

In two of the three application considered in the next section, the inner ECSP has exactly one decision variable, and in all applications, the inner ECSP has one or more auxiliary variables. As a result, the branch-and-propagate search of step 2 is limited in the sense that it is not an exhaustive exploration of all possible combinations of canonical domains. Search takes place on a subset of the variables only.

An Example

We will see three examples of specific uses of the DRF of Definition 7.2.3 in the next section. Just to illustrate the interaction between the inner ECSP, the DRF, and the outer ECSP, we present the following (contrived) example.

7.2.4. Example. Let Q be the ECSP

$$Q := \langle \mathcal{C}_Q \ ; \ w, x, y \in \{0,1,2\} \ ; \ D_w, D_x, D_y \in \mathcal{Z} \ ; \ \mathcal{A}_w, \mathcal{A}_{Q,x}, \mathcal{A}_{Q,y} \rangle$$

Consider the instance $f_{P,y}$ of the DRF for nested search that is parameterized by the ECSP

$$P := \langle x < y, \ y \neq z \ ; \ x, y \in \mathbb{Z}, z \in \{0,1,2\} \ ; \ D_x, D_y, D_z \in \mathcal{Z} \ ; \ \mathcal{A}_{P,x}, \mathcal{A}_{P,y}, \mathcal{A}_z \rangle$$

having sets of final domains $\mathcal{A}_{P,x} = \lfloor \mathcal{Z} \rfloor$, $\mathcal{A}_{P,y} = \mathcal{Z} - \{\emptyset\}$, and $\mathcal{A}_z = \lfloor \mathcal{Z} \rfloor$, i.e., x and z are decision variables, and y is an auxiliary variable in P.

Q and P share variables x and y, so as a DRF on Q, $f_{P,y}$ has the signature

$$\mathcal{T}_x \times \mathcal{T}_y \rightarrow \mathcal{T}_y$$

with $\mathcal{T}_x = \mathcal{Z}$ and $\mathcal{T}_y = \mathcal{Z}$.

If we evaluate this function for the domains of x and y in Q, i.e., we compute $f_{P,y}(\{0,1,2\},\{0,1,2\})$, we first construct the inner ECSP

$$P' = (P, \langle \{0,1,2\}, \{0,1,2\} \rangle).$$

This is the first step of Figure 7.1, and in this case we have

$$P' = \langle x < y,\ y \neq z\ ;\ x,y,z \in \{0,1,2\}\ ;\ D_x, D_y, D_z \in \mathcal{Z}\ ;\ \mathcal{A}_{P,x}, \mathcal{A}_{P,y}, \mathcal{A}_z \rangle$$

This ECSP is consistent, so suppose that the branch-and-propagate search of step 2 finds the arc solved form

$$\langle x < y,\ y \neq z\ ;\ x = 0, y \in \{1,2\}, z = 0\ ;\ D_x, D_y, D_z \in \mathcal{Z}\ ;\ \mathcal{A}_{P,x}, \mathcal{A}_{P,y}, \mathcal{A}_z \rangle,$$

which we also encountered in Example 2.2.15 on page 16. Now as step 3 we select the domain of y in this arc solved form as the outcome of the function evaluation, and we have

$$f_{P,y}(\{0,1,2\},\{0,1,2\}) = \{1,2\}$$

In a branch-and-propagate search on Q, this result is then used as a new domain for y, and Q is transformed into the following ECSP.

$$\langle \mathcal{C}'_Q\ ;\ w,x \in \{0,1,2\}, y \in \{1,2\}\ ;\ D_w, D_x, D_y \in \mathcal{Z}\ ;\ \mathcal{A}_w, \mathcal{A}_{Q,x}, \mathcal{A}_{Q,y} \rangle,$$

with $\mathcal{C}'_Q := \mathcal{C}_Q[\{0,1,2\},\{0,1,2\},\{1,2\}]$. □

One way in which this example is contrived is that the relation between P and Q is not clear. In our applications, P is constructed so that applying f_{P,x_j} will remove values for which the constraints in $\mathcal{C}_Q$ cannot be satisfied.

As we mentioned after Definition 7.2.3, the functionality of the operator depends on the solved form that is found by the branch-and-propagate search in step 2 of Figure 7.1. If in Example 7.2.4, we had found another arc solved form, for example one having $D_x = \{0\}$, $D_y = \{1\}$, and $D_z = \{2\}$, then $f_{P,y}(\{0,1,2\},\{0,1,2\})$ would have evaluated to $\{1\}$ instead of $\{1,2\}$.

In two of the three applications discussed in the next section, we require a specific kind of branch-and-propagate search in step 2 of Figure 7.1, namely the one based on a depth-first, leftmost-first traversal strategy. This traversal strategy will lead to a specific solved form, which then fully determines the functionality of the DRF. We will use a a superscript L to indicate this requirement: f^L_{P,x_j}.

Because the output variable x_j is also in the input scheme of f_{P,x_j}, and because $f_{P,x_j}(D_1,\ldots,D_n)$ evaluates to the domain of x_j in a solved form of the inner ECSP, by Definition 2.2.14 on page 16 we have $f_{P,x_j}(D_1,\ldots,D_n) \subseteq D_j$. Returning to the discussion of DRF properties on page 21, this leads to inflationary updates of ECSPs, which ensures that instantiations of the operator for nested search will not cause non-termination of generic iteration algorithms. However, as we shall see in the next section, the operators are not necessarily monotonic, or, as was already demonstrated by Example 7.2.4, equivalence preserving.

7.3 Applications

In our applications, we will not use the operator for nested search to find a particular value for the output variable x_j, but only to update one of its ***bounds***. However, without special care, the domain of x_j may be reduced to a singleton set during search. This happens if x_j is a decision variable of the inner ECSP, but also if its value is uniquely determined by the decision variables of the inner ECSP. Therefore we will use a ***copy*** of the output variable during the search. This copy is a regular CSP variable that is added to the parameter ECSP, and serves to update one of the bounds of the output variable. For this we use the regular inequality constraints.

The initial domain of the copy variable is set to that of the output variable. We cannot use an equality constraint here, because that would still reduce the domain of the output variable to a singleton set, once a solved form is found. Therefore we introduce the following constraint.

7.3.1. Definition. Given two variables x and y with respective domains D_x and D_y of the same domain type, let the constraint $x := y$ denote the subset $(D_x \cap D_y) \times D_y$ of $D_x \times D_y$.

This constraint can be thought of as an assignment operator: modifications of the domain of y are propagated to the domain of x, but not the other way around. Our first application below will demonstrate its use.

Of the three example applications discussed in this section, optimization by means of a bisection search involves inner ECSPs that have the most similarity with the ECSPs that we have encountered so far: they represent purely combinatorial problems to which two auxiliary variables are added for optimization. Therefore we discuss this application first.

7.3.1 Optimization

As an alternative to branch-and-bound, in optimization we can perform a bisection search in the range of an objective function that is defined on the variables of a combinatorial problem. The following search procedure is adapted from [BLPN01], where it is applied to job-shop scheduling problems. Suppose we want to minimize an integer objective function that evaluates in the range $[l..h]$. Assuming that a solution to the combinatorial problem exists (this is often the case for constrained optimization problems, and certainly for job-shop scheduling), we can determine the minimum as follows:

1. Split the domain $[l..h]$ for the outcome of the objective function into two halves $[l..m]$ and $[m+1..h]$, with $m := \lfloor \frac{1}{2}(l+h) \rfloor$, and first try to solve the combinatorial problem with the domain for the outcome of the objective function set to $[l..m]$.

2. If the combinatorial problem has a solution for which the objective function evaluates to a value $v \in [l..m]$, then if v equals l this is the minimum, and we are done. If $v > l$, restart the search at step 1, now using $[l..v]$ as the range for the objective function.

3. If no solution exists for which the objective function evaluates in $[l..m]$, restart the search at step 1, now using $[m + 1..h]$ as the range for the objective function.

We now describe this optimization scheme as an application of our operator for nested search. Let

$$R := \langle \mathcal{C} \ ; \ x_1 \in D_1, \ldots, x_n \in D_n \ ; \ \mathcal{T}_1, \ldots, \mathcal{T}_n \ ; \ \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle$$

be the ECSP for a combinatorial problem. In what follows we consider that we want to ***minimize*** the outcome of an ***integer*** objective function g on variables $x_1, \ldots, x_n$.

The Parameter ECSP

We will construct a parameter ECSP P such that evaluating $f_{P,c}([l..h])$ verifies that a solved form of R exists, for which $g(x_1, \ldots, x_n)$ falls within the range $[l..h]$. If it exists, $f_{P,c}([l..h])$ evaluates to $[l..v]$ where v is the outcome of g for the particular solved form found during the evaluation of $f_{P,c}$. If it does not exist, $f_{P,c}([l..h])$ evaluates to $\emptyset$. R is the basis of the parameter ECSP P, but two variables and three constraints are added.

First we add a variable c' for the outcome of the objective function, and constrain it accordingly:

$$c' = g(x_1, \ldots, x_n).$$

Assuming that R contains no auxiliary variables, the value of c' is fixed when g is evaluated for a solved form of R. Since in general, a solved form for R will not yield the minimal outcome of g, we can only use this value as an upper bound, and we add a second variable c, and constrain it to be less than, or equal to c.

$$c \leq c'$$

Now c will be the output variable of the DRF for nested search, and c' is its copy for performing the search, as we described just before Section 7.3.1. All that is needed now is a third constraint that gives c' its initial domain:

$$c' := c$$

Assuming that c and c' do not occur in R, the parameter ECSP now becomes

$$P := \langle \mathcal{C}_P \ ; \ \underline{c \in \mathbb{Z}, c' \in \mathbb{Z}}, \ x_1 \in D_1, \ldots \ ; \ \underline{\mathcal{T}_c, \mathcal{T}_{c'}}, \ \mathcal{T}_1, \ldots, \ ; \ \underline{\mathcal{A}_c, \mathcal{A}_{c'}}, \ \mathcal{A}_1, \ldots \rangle$$

where the underlined elements are the additions to the combinatorial problem R, and where

- $\mathcal{C}_P := \mathcal{C} \cup \underline{\{\ c' = g(x_1, \ldots, x_n),\ c \leq c',\ c' := c\ \}}$,
- $\mathcal{T}_c$ and $\mathcal{T}_{c'}$ both equal $\mathcal{I}$ (g is an integer function), and
- $\mathcal{A}_c$ and $\mathcal{A}_{c'}$ equal $\mathcal{I} - \{\emptyset\}$.

The sets of final domains $\mathcal{A}_c$ and $\mathcal{A}_{c'}$ render c and c' auxiliary variables of P: search is on the variables of R only.

The Domain Reduction Function $f_{P,c}$

The ECSP P gives rise to the following instance of the generic operator for nested search:

$$f_{P,c} : \mathcal{I} \rightarrow \mathcal{I}$$

When $f_{P,c}$ is evaluated for an integer interval domain $[l..h]$, as step 1 of Figure 7.1, the inner ECSP is constructed by substituting the domain of c in P with $[l..h]$. Then, as step 2 of Figure 7.1, branch-and-propagate search is performed on P'. Propagation of the constraint $c' := c$ will set the domain of c' equal to $[l..h]$ initially, but during the search, the domain of c' will be modified further. According to Definition 7.3.1 these modifications will not propagate back to the domain of c, but through the constraint $c \leq c'$ the upper bound for c is modified yet. However, these modifications do not affect the outcome of $f_{P,c}([l..h])$ until a solved form is found. In such a solved form, the domain of c' has likely been reduced to a singleton set $\{v\}$ because c' is tied to the decision variables by the constraint $c' = g(x_1, \ldots, x_n)$. Through the constraint $c \leq c'$, the domain of c in this solved form equals $[l..v]$, which is the outcome of $f_{P,c}([l..h])$. When no solved form of P' exists, $f_{P,c}([l..h])$ evaluates to the empty set.

Optimization as Branch-and-Propagate Search

With $f_{P,c}$ the optimization scheme can be described as a branch-and-propagate search on an outer ECSP

$$Q := \langle C_Q\ ;\ c \in [l..h]\ ;\ D_c \in \mathcal{Z}\ ;\ \mathcal{A}_c = \lfloor \mathcal{Z} \rfloor \rangle$$

This outer ECSP contains a single decision variable c, with initial domain $[l..h]$, where l and h are trivial lower bounds for g that follow from the domains of $x_1, \ldots, x_n$ in R. C_Q can be thought of as the constraint that there exists a solved form of R for which g evaluates to c. Nodes of the search tree are created by bisection of the domain of c, and in the constraint propagation phase $f_{P,c}$ is applied once to the domain of c. If we perform a depth-first, leftmost-first traversal, the first solution is guaranteed to contain the minimum value of the outcome of g for any solved form of R.

7.3.2. Example. Let R be an ECSP with variables $x_1, \ldots, x_n$, and let g be an integer function on $x_1, \ldots, x_n$. Suppose that for all possible combinations of values allowed by the domains of these variables in R, g evaluates in the range 0..100. Suppose further that R is consistent, and that the minimum value of g for any solved form of R is 23. The search for this minimum proceeds as follows.

Initially, in the outer ECSP Q we have $c \in [0..100]$. As the initial constraint propagation phase we evaluate $f_{P,c}([0..100])$. As a part of this evaluation, in step 1 of Figure 7.1, we transform

$$P := \langle \mathcal{C}_P \ ; \ c \in \mathbb{Z}, c' \in \mathbb{Z}, \ldots \ ; \ \ldots \ ; \ \ldots \rangle$$

into

$$P' := \langle \mathcal{C}_P \ ; \ c \in [0..100], c' \in \mathbb{Z}, \ldots \ ; \ \ldots \ ; \ \ldots \rangle.$$

Now in step 2 of Figure 7.1, we search for a solved form of P'. Through propagation of the constraint $c' := c$ in $\mathcal{C}_P$, the domain of c' is immediately changed from $\mathbb{Z}$ to $[0..100]$. As the search progresses, the domain of c' undergoes further changes, and the upper bound for c is updated accordingly. Suppose that the nested search in step 2 of Figure 7.1 finds a solved form for which g evaluates to 36, i.e., this yields a suboptimal value for the objective function g. This solved form looks like this:

$$\langle \mathcal{C}_P \ ; \ c \in [0..36], c' \in \{36\}, \ldots \ ; \ \ldots \ ; \ \ldots \rangle,$$

the domain of c' is fixed, but c only has its upper bound modified, and $f_{P,c}([0..100])$ evaluates to $[0..36]$.

In the branch-and-propagate search on the outer ECSP Q, the value of $f_{P,c}([0..100])$ is used as the new domain for c. This is not yet a singleton set, and we proceed by branching on c, yielding subdomains $[0..18]$ and $[19..36]$. Because we do a depth-first leftmost-first search on Q, we continue the search in the $c \in [0..18]$ branch. It will turn out that no solution to P exists for which $g(X)$ lies in this range, i.e., $f_{P,c}([0..18]) = \emptyset$, which voids the domain of c in this branch. Then search proceeds in the $c \in [18..36]$ branch, where $f_{P,c}([18..36])$ yields a tighter upper bound for c, and so on, until finally the domain of c has been narrowed to $\{23\}$. Because the search is depth-first leftmost-first, this is guaranteed to be the minimum. $\square$

Discussion

It is important that during the constraint propagation phase, the domain reduction function $f_{P,c}$ is applied only once, to verify that a solution for the current domain of the criterion variable exists, and to update the upper bound of this domain for the actual solution that is found. If unless we deduce a failure, we keep iterating the function until it makes no more modifications to the domain of the criterion variable, we will apply it at least once more. This second application may find the same solved form that caused the initial reduction, in which case we just duplicate the work, but because we start the nested search with a different range for the objective function, we may well find a solved form that implies a tighter bound. In the latter case, we achieve a further reduction, and the operator is applied again. This way we embark on a limited branch-and-bound search in the constraint propagation phase, that could lead all the way to the minimum. Because the bisection search is proposed as an ***alternative*** to branch-and-bound, this behavior is undesirable.

Note that $f_{P,c}$ is not a monotonic function. In Example 7.3.2, $f_{P,c}([0..100])$ evaluates to $[0..36]$ because the objective function g evaluates to 36 for the particular solved form found $c \in [0..100]$. However, it is possible that for a narrower domain for c, say $[0..99]$, the search on the inner ECSP leads to a solved form with a higher outcome of the objective function, say 37. While $[0..99] \subseteq [0..100]$, we then have $f_{P,c}([0..99]) \not\subseteq f_{P,c}([0..100])$, which entails that the transformation is non-monotonic with respect to the subproblem relation. Returning to the discussion of DRF properties on page 21, this implies that generic iteration is no longer guaranteed to terminate in the least common fixed point of the DRFs involved. For this application, this is not a problem, because the operator is applied only once.

Further, note that our proposed optimization scheme does not lead to a fully accurate implementation of the procedure described at the beginning of this section. As we described it there, the combinatorial problem is solved in left branches only. Right branches are guaranteed to contain a solution if the left branch fails, and these are split again immediately. In Section 7.5.1 we see how the same effect can be achieved with nested search in a branch-and-propagate setting.

Finally, we did not specify the level of local consistency that the nested search is to be based on. Apart from the requirement that ***assignments*** violating the constraints $\mathcal{C}$ in R should be filtered out, we only require that the constraint $c' := c$ is bounds consistent, and that the domain of c' is voided if it does not contain the outcome of g, for a solved form of R. All further propagation helps to speed up the solving process. If the constraint $c' = g(x_1, \ldots, x_n)$ propagates back to the domains of $x_1, \ldots, x_n$, the nested search accelerates. As is demonstrated in Example 7.3.2, the search on the outer ECSP also benefits from bounds consistency of $c \leq c'$.

7.3.2 Box Consistency

In Section 4.5 we introduced hull consistency, which is an approximation of arc consistency used for arithmetic constraints and floating-point interval domains. Box consistency [BMVH94] is another such approximation. It was introduced to avoid decomposing constraints, and as such it partly avoids the dependency problem that we also discussed in Section 4.5. Before we can define box consistency, and demonstrate how it is enforced using our generic operator for nested search, we first need to recall the notion of an interval extension of a constraint.

An ***interval extension*** of a constraint $C \subset \mathbb{R}^n$ is a relation $\boldsymbol{C} \subseteq \mathcal{F}^n$ such that for all $D := \langle D_1, \ldots, D_n \rangle \in \mathcal{F}^n$, $D \in \boldsymbol{C}$ if there exists a tuple $\langle d_1, \ldots, d_n \rangle \in D_1 \times \ldots \times D_n$ for which $\langle d_1, \ldots, d_n \rangle \in C$.

7.3.3. Example. An interval extension $\boldsymbol{C}_{\text{eq}} \subset \mathcal{F} \times \mathcal{F}$ of the equality constraint is

$$\langle D_1, D_2 \rangle \in \boldsymbol{C}_{\text{eq}} \text{ iff } D_1 \cap D_2 \neq \emptyset.$$

As an example of an interval extension of a particular class of constraints, we can define interval extensions for the other relational symbols $\leq$ and $\geq$ as well, and modify the definition of the natural interval extension of Section 4.5 to include not only arithmetic expressions, but also arithmetic constraints. □

Interval extensions of constraints are called ***interval constraints***.

Now a constraint $C \subset \mathbb{R}^n$ on variables $x_1, \ldots, x_n$ with associated domains $D_1, \ldots, D_n \in \mathcal{F}$ is said to be ***box consistent*** if for all $1 \leq j \leq n$

$$D_j = \mathsf{hull}(D_j \cap \{r \in \mathbb{R} \mid \langle D_1, \ldots, D_{j-1}, \mathsf{hull}(\{r\}), D_{j+1}, \ldots, D_n \rangle \in \boldsymbol{C}\})$$

where $\boldsymbol{C}$ is an interval extension of C.

In [BGGP99] a notion of box consistency is defined that supports using different interval extensions for different occurrences of variables, and that also captures a number of other, alternative definitions. In what follows, we always use the natural interval extension for $\boldsymbol{C}$. We will further limit ourselves to polynomial ***equalities***, for which we use the interval extension proposed in Example 7.3.3.

7.3.4. Example.

- The constraint $x^3 + x = 0$ on $x \in \mathsf{hull}([-1, 1])$ is not box consistent: the domain of x properly contains $\mathsf{hull}(\{-1\})$, which is not in the interval extension of the constraint $x^3 + x = 0$.
- The constraint $x^3 + x = 0$ on $x \in \mathsf{hull}(\{0\})$ is box consistent. □

Given a compound constraint, enforcing box consistency for this constraint may yield narrower domains than enforcing hull consistency for the decomposition of the constraint [CDR99]. This can be seen by comparing Example 7.3.4 and Example 4.5.1. However, the accuracy of the condition

$$\langle D_1, \ldots, D_{j-1}, \mathsf{hull}(\{r\}), D_{j+1}, \ldots, D_n \rangle \in \boldsymbol{C}$$

is still subject to the dependency problem. Therefore it will not achieve hull consistency for the compound constraint in general. In other words, box consistency is weaker than hull consistency, but stronger than hull consistency for the decomposed constraint.

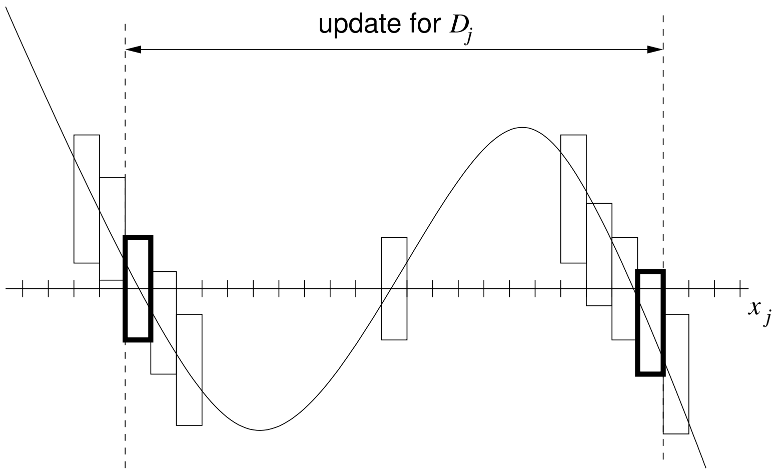


Figure 7.2: The new domain for D_j is bounded by the leftmost and rightmost canonical intervals that satisfy the unary interval constraint $F(D_j) \cap \{0\} \neq \emptyset$

Enforcing Box Consistency

The idea behind enforcing box consistency is to fix, for every variable x_j that a constraint applies to, the domains of the other variables to interval constants. The interval extension of the constraint with all but one variable replaced by an interval constant is then a ***unary interval constraint***, and we can remove those subintervals from D_j that do not satisfy it. Because we will take the intersection of the remaining domain with D_j, and compute the hull of this intersection to be able to represent it as an $\mathcal{F}$ interval again, we only need to know the leftmost and the rightmost canonical interval that are a subset of D_j, and for which the unary interval constraint holds. We can then intersect D_j with the hull of the union of these canonical intervals. This is illustrated in Figure 7.2 for a constraint $f(x_1, \ldots, x_n) = 0$. The marks on the x_j axis are the floating-point numbers in $\mathbb{F}$, which delimit the canonical intervals. The boxes drawn along the curve represent the ranges for the outcome of f for a particular canonical interval in the domain of x_j, and in presence of the current domains of the other variables.

In [HMD97] a very general algorithm for enforcing box consistency is given. In this algorithm, procedures `LeftNarrow` and `RightNarrow` search in the domain D_j for the leftmost and rightmost canonical intervals $\mathsf{hull}(\{r_l\})$ and $\mathsf{hull}(\{r_r\})$ that satisfy the unary interval constraint described above, and update D_j accordingly. Both procedures can be described as instances of the generic operator for nested search. Left narrowing for variable x_j of a constraint $C \subset \mathbb{R}^n$ on variables $x_1, \ldots, x_n$ is realized by

$$f^L_{P,x_j} : \mathcal{F}^n \to \mathcal{F} \tag{7.1}$$

where P is the following ECSP.

$$\begin{aligned}\langle\ & \{x'_j/x_j\}C,\ x'_j := x_j,\ x_j \geq x'_j \\ & ;\ x_1,\dots,x_n \in \mathbb{R},\ x'_j \in \mathbb{R} \\ & ;\ D_1,\dots,D_n \in \mathcal{F},\ D_{x'_j} \in \mathcal{F} \\ & ;\ \mathcal{A}_1,\dots,\mathcal{A}_n = \mathcal{F} - \{\emptyset\},\ \mathcal{A}_{x'_j} = \lfloor\mathcal{F}\rfloor\rangle\end{aligned}$$

In this parameter ECSP, $\{x'_j/x_j\}C$ is the constraint that we want to enforce box consistency for, with all occurrences of x_j replaced by x'_j, the copy of x_j introduced for performing the search. Through its set of final domains $\mathcal{A}_{x'_j} = \lfloor\mathcal{F}\rfloor = \{\mathsf{hull}(\{r\}) \mid r \in \mathbb{R}\}$, x'_j is the only decision variable in P. All other variables, including x_j have $\mathcal{F} - \{\emptyset\}$ as their set of final domains. They are auxiliary variables in the inner ECSP, and no branching needs to be performed on their domains to reach a solved form.

When f^L_{P,x_j} is evaluated for a sequence of intervals $D_1,\dots,D_n$, in step 1 of Figure 7.1 the domains of $x_1,\dots,x_n$ in the parameter ECSP P are replaced by $D_1,\dots,D_n$. This yields the inner ECSP P'. During step 2, propagation of the constraint $x'_j := x_j$ ensures that the domain of x'_j is equal to that of x_j initially. Because no branching takes place on the domains of $x_1,\dots,x_n$, $\{x'_j/x_j\}C$ now effectively has become a unary interval constraint on the domain of x'_j. If P' is consistent, and we perform a depth-first, leftmost-first search for a solved form, as specified by the superscript L, then the domain of x'_j in this solved form is the leftmost canonical interval in D_j that satisfies the unary interval constraint obtained by replacing in C the domains of $x_1,\dots,x_{j-1},x_{j+1},\dots,x_n$ with $D_1,\dots,D_{j-1},D_{j+1},\dots,D_n$, and by taking the natural interval extension. Propagation of the constraint $x_j \geq x'_j$ updates the lower bound for x_j accordingly, and in step 3 of Figure 7.1, f^L_{P,x_j} evaluates to D_j with the lower bound set to that of the leftmost canonical interval that satisfies the unary interval constraint.

Coupled with analogous operators for right narrowing, and for the other variables that participate in the constraint, f^L_{P,x_j} enforces box consistency for constraint C. Intuitively, narrowing domains of variables can only move the leftmost and rightmost canonical intervals in the domain of these, and other variables inwards. This can be used to demonstrate that operators for enforcing box consistency are in fact monotonic functions, so the order in which they are applied by an iteration algorithm is irrelevant for the outcome of their combined computation.

For evaluating the unary interval constraint we can use the facilities described in Section 4.5. This way the operators for enforcing box consistency are ***composed*** from the generic reduction operator for nested search, and the facilities for enforcing hull consistency for a decomposition of an arithmetic constraint into atomic constraints.

7.3.3 Shaving

Shaving is a constraint propagation technique used for solving scheduling problems. We refer to the description of this technique in [MS96], and use job-shop scheduling as an example.

Recall from Chapter 6 that a job-shop scheduling problem (JSSP) instance consists of a set of ***activities*** and a number of ***machines***. An activity is characterized by the machine that it must be processed on, and by a ***processing time***. Activities are grouped in ***jobs***, and all activities of a job have to be executed in a specified order. The problem is to find for each activity an interval in which it can be executed on the specified machine, such that no two activities require the same machine simultaneously, and such that the precedence constraints inside the jobs are respected. An optimal schedule minimizes the ***makespan*** of the schedule, being the completion time of the activities that finish last.

A possible CSP formulation of the JSSP contains an integer interval variable for the starting time of each activity. The lower bound for the starting time of an activity is called the ***release date***, and the upper bound plus the processing time of the activity is called the ***deadline***. Here we will consider the procedure for updating release dates. The procedure for deadlines is analogous.

The shaving technique entails that starting with the release date, we see what happens if we fix the activity to start at that time. After experimentally fixing the starting time of an activity we apply constraint propagation. If propagation of the fixed starting time leads to a failure, we can safely remove this candidate starting time from the domain of the variable, and we proceed by trying the next possible starting time, and so on, until we encounter a starting time that does not lead to a failure. This is then the new release date for the activity. Shaving can be explained as a leftmost-first search in the domain of a single variable, and as such it can be expressed as an application of the generic operator for nested search.

Let

$$Q := \langle \mathcal{C}_Q \ ; \ x_1 \in D_n, \ldots, x_n \in D_n \ ; \ \mathcal{T}_1, \ldots, \mathcal{T}_n \ ; \ \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle$$

be an ECSP for a job-shop scheduling problem, and assume that x_j, with $1 \leq j \leq n$ is the variable for the starting time of an activity A. Shaving the starting time of A can be expressed as the domain update

$$D_j := f^L_{P,x_j}(D_1, \ldots, D_n)$$

where P is the ECSP

$$\begin{aligned}\langle \ & \{x'_j/x_j\}\mathcal{C}_Q, \ x'_j := x_j, \ x_j \geq x'_j \\ & ; \ x_1 \in \mathcal{T}_1^\top, \ldots, x_n \in \mathcal{T}_n^\top, \ x'_j \in \mathcal{T}_j^\top \\ & ; \ \mathcal{T}_1, \ldots, \mathcal{T}_n, \ \mathcal{T}_{x'_j} = \mathcal{T}_j \\ & ; \ \mathcal{A}_1 = \mathcal{T}_1 - \{\emptyset\}, \ldots, \mathcal{A}_n = \mathcal{T}_n - \{\emptyset\}, \ \mathcal{A}_{x'_j} = \lfloor \mathcal{T}_j \rfloor \ \rangle.\end{aligned}$$

In this case, the parameter ECSP P is a full copy of the outer ECSP Q, with a single variable x'_j added. This variable is a copy of the variable whose starting time we want to update. The set of constraints in the parameter ECSP, $\{x'_j/x_j\}\mathcal{C}_Q$, is the set of constraints of the outer ECSP, with every occurrence of x_j replaced by x'_j. All variables of the outer ECSP are auxiliary variables in the inner ECSP, and search takes place only on the "copy" variable x'_j. It is coupled to the original x_j in the usual way, through constraints $x'_j := x_j$ and $x_j \geq x'_j$. The domain of the variables $x_1, \ldots, x_n$ in the inner ECSP are irrelevant, and we set them here to the largest elements of the corresponding domain types $\mathcal{T}_1^\top, \ldots, \mathcal{T}_n^\top$. These domains are replaced by their counterparts in the outer ECSP when the DRF is evaluated.

7.3.5. Example. The example JSSP of Figure 6.1, on page 132 consists of three jobs, each having three activities that require three different machines. To implement shaving for this problem, we need 18 operators: one for each of the 9 activity starting times, and one for each of the completion times. Consider the operator for one of the starting times. The nested search finds the smallest value for this starting time that has the property that if the activity is actually scheduled to start at that time, regular constraint propagation on the full problem, involving all 17 other starting times and completion times, does not lead to a failure. In the global CSP, all earlier starting time are removed from the domain of the variable. □

The extent to which infeasible starting times are removed depends on the level of consistency that we enforce during the nested search. In [BLPN01] it is suggested that we use the level of consistency enforced by the ***edge finding*** algorithm. The precedence constraints inside the jobs propagate modifications to the other machines, during the nested search. Also the constraint $x_j \geq x'_j$ must be enforced in order that the infeasible starting times are actually removed.

A depth-first leftmost-first search combined with bisection or enumeration branching on the domain of x'_j will correctly update the lower bound of x_j, but in [MS96] a different branching scheme is described for step 2 of Figure 7.1. This alternative scheme entails that we first try to shave off a single value, and double the size of interval to shave off until an interval is found that allows for a feasible schedule. Then we search for the lower bound in this interval by regular bisection.

Several different notions of shaving exists. In [VHPP00] a different form is implemented to demonstrate nested search in OPL. Here the nested search does not directly modify release dates and deadlines, but only the ranking of the activities. Constraint propagation verifies whether individual activities can be ranked first among the set of unranked activities on a machine. If this fails, constraints are added to ensure that at least one of the other unranked activities is ranked before that particular activity.

The locality of our shaving operation is in between that of the other two applications. Box consistency performs nested search on a single variable, and

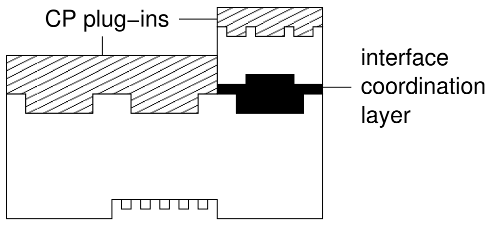


Figure 7.3: Almost autonomous OpenSolver instances implement the nested search

propagates only a single constraint. For optimization, the nested search is on all variables, propagating all constraints. Like box consistency, nested search for shaving branches only on one variable, but consistency checking involves all constraints. Using the same reasoning as for box consistency, we can demonstrate that DRFs for shaving are monotonic functions.

7.4 Implementation

The plug-in that implements the operator for nested search is an almost[1] autonomous OpenSolver instance, acting as a reduction operator. A special coordination layer plug-in forms the interface between the solver that uses the nested search, and the solver that performs the nested search. This is illustrated in Figure 7.3. A benefit of this implementation is that all facilities of the OpenSolver framework are immediately available for nested search.

Programs 7.1 and 7.2 show examples of how the operator is used. The plug-in name is `NestedSearch`, and its specifier string consists of the following:

- The name of a Boolean variable, whose only purpose is that its domain will be voided if we find that the inner ECSP is inconsistent.
- The names of the variables that the operator applies to (the input variables). The output variable, whose domain is updated by the reduction operator, is identified by a prefix `&`.
- Between curly brackets, an OpenSolver configuration for the parameter ECSP, in the language of Figure 3.2 on page 37. All input variables must appear in this configuration, but their domains are irrelevant. These will be provided by the coordination layer plug-in each time the operator is applied.

The Boolean variable was introduced because the operator cannot make any assumptions about the domains of the other variables, and hence has no uniform way of voiding their domains. It is a deviation from the specification in

[1]There is a single thread of control throughout.

Section 7.2, but this is no fundamental difference. Also, more than a single variable can be marked as an output variable. This is a fundamental difference, but we have not found a use for this facility yet, and taking it into account would complicate the formal description of the operator further.

When it is activated during constraint propagation, the `NestedSearch` operator will pass the domains of the input variables to the interface coordination layer, and run the local OpenSolver instance. The solver configuration in the specifier string is parsed the first time that the operator is applied. The interface coordination layer keeps a copy of the root node of the search tree, in order that the specifier string need only be parsed once. On each application of the reduction operator, a fresh copy of this cached root node is made, which is updated with the current domains of the input variables. The interface coordination layer issues commands for a regular first-solution search on this modified root node. When a solution node is found, an **export** command (see Section 3.3.2) is given for the output variable. The solver will respond to this command via a call-back function provided by the coordination layer. In this case, the call-back function updates the domain of the output variable. If no solution is found, the domain of the Boolean variable is voided. After the first-solution search, the interface coordination layer issues the **clear WDB** command to reset the solver.

7.5 Experiments

In this section we describe the experiments that we performed for the applications discussed in Section 7.3.

7.5.1 Optimization: Job-Shop Scheduling

We tested the optimization technique described in Section 7.3.1 on the job-shop scheduling problem, using the plug-ins that were introduced in Chapter 6.

Program 7.1 shows an OpenSolver configuration for the approach of Section 7.3.1. The `NestedSearch` operator is applied through an adapter `Idempotent DRF`. This forces the scheduler of reduction operators to treat it as an idempotent domain reduction function. The operator-based scheduler (see Section 4.1.1), which is the default, takes idempotency of DRFs into account to avoid unnecessary applications of an operator. In this case, it prevents that the operator is applied more than once in the same node of the search tree, for the reasons discussed at the end of Section 7.3.1. Also, because all right branches are guaranteed to contain a feasible schedule, an adapter `PropagateLeft` is used to hide changes that are made by the `FailFirst` branching operator to variable `bound` in right branches. Hiding means suppressing the protocol for communicating domain changes, discussed in Section 3.2.2, thus preventing needless activation of the nested search in these branches.

```
VARIABLE bound IS IntegerInterval {1119..5110};
AUX b IS Bool {0,1};
DRF IdempotentDRF { NestedSearch { b, &bound, {
   AUX bound IS IntegerInterval {};
   AUX imakespan IS IntegerInterval {};
   VARIABLE makespan Is Activity {0,0,5110};
   ...
   code for the JSSP, where makespan executes
   after the last activity of every job.
   ...
   DRF BoundActivity { makespan, imakespan };
   DRF IIARule { imakespan^1 * (1) = bound };
   DRF IIARule { bound^1 * (1) <= imakespan };
} } };
DRF PropagateLeft { FailFirst { 0, bound } };
```

Program 7.1: OpenSolver code for bisection search for the minimal makespan of a JSSP

The indented code is a configuration for solving a job-shop scheduling problem, where the length of the schedule is constrained not to exceed the upper bound of the domain for the integer variable `bound`. It is similar to Program 6.1 on page 138. The `IIARule` operators effectively enforce the constraints `imakespan` := `bound` and `bound` $\leq$ `imakespan`. The parameter 0 in the specifier for the `FailFirst` plug-in specifies a bisection where the left branch is generated last. By default, the search frontier is maintained as a stack, so this results in a depth-first, leftmost-first exploration.

Table 7.1 compares the bisection optimization algorithm that we implemented by nested search with regular branch-and-bound, as described in Chapter 6, on the ten 10×10 JSSP instances of [AC91], which are also used in [BLPN01]. For each instance we report the number of nodes visited, and the user time in seconds, as reported by the GNU/Linux `time` command on a 1200 MHz Athlon PC. For the bisection search, the number of nodes applies to the nested search on the JSSP only. The search in the domain of the criterion variable is not taken into account. Either algorithm outperforms the other in half the cases, but the total running time for all ten cases is much better for the bisection search, which seems to suggest that it is more robust. While an evaluation of these approaches to optimization is beyond the scope of this section, it shows that the bisection search is a useful tool. Having implemented it using OpenSolver as a software component increases its value as a building block for solvers, because we can now

	branch-and-bound		bisection		
instance	nodes	time (sec.)	nodes	time (sec.)	opt
ft10	238,045	38.87	321,878	61.38	930
abz5	385,699	44.84	586,315	67.23	1234
abz6	57,277	6.28	114,487	12.57	943
la19	319,547	40.80	228,283	34.47	842
la20	106,785	13.78	139,146	18.26	902
orb01	7,331,421	643.40	63,688	10.91	1059
orb02	642,645	63.93	200,350	28.86	888
orb03	14,031,873	2186.06	3,657,439	563.12	1005
orb04	179,037	39.12	278,507	71.62	1005
orb05	4,461,777	621.77	98,268	14.54	887

Table 7.1: A Comparison of Branch-and-Bound and a Bisection Search for the optimum on ten 10×10 JSSP instances

combine it with other facilities, such as memory bounded LDS.

Taking the differences in clock speeds into account, our results for bisection search on these ten benchmark problems do not match the results reported in [BLPN01]. For some of the instances, performance is better, but for the majority of them it is worse. More important, though, is that the instances rank differently, which indicates that we have not been able to reproduce exactly the same heuristics.

As a further indication that our technology leads to competitive constraint solvers, Koalog Constraint Solver (KCS, [KoaA]) is reported to solve the ft10 benchmark, which is also known under the name MT10, in 11 minutes, using 293,000 backtracks. KCS is a commercially available Java library for solving combinatorial optimization problems using constraint programming or local search, and is used in industry.

7.5.2 Box Consistency

The implementation of box consistency for constraints on the reals was tested on the ***Broyden banded functions***, a benchmark that is often used to demonstrate the advantage of box consistency over hull consistency, for example in [BMVH94]. The problem is to find the zeros of the functions

$$f_i(x_1, ..., x_n) = x_i(2 + 5x_i^2) + 1 - \sum_{j \in J_i} x_j(1 + x_j) \quad (1 \leq i \leq n), \tag{7.2}$$

where $J_i = \{j \mid j \neq i, \mathsf{max}(1, i-5) \leq j \leq \mathsf{min}(n, i+1)\}$, and $x_1, \ldots, x_n \in [-1, 1]$.

Every function f_i depends on the 2 to 7 variables in the set $J_i \cup \{x_i\}$, and for every variable that a function depends on, two reduction operators are generated: one for the left narrowing, and one for the right narrowing. Program 7.2 shows the operator that implements left narrowing for argument x_1 of the function f_3. Variable `lx1` corresponds to x'_j used in Section 7.3.2. It is linked to `x1` by the first two `RIARule` operators. Auxiliary variables `x1`,...,`x4` are the input variables, and they are given their domains each time the operator is applied. The other `RIARule` operators evaluate f_3, using `fx1`,...,`fx4` to store intermediate results. The interval for the outcome of f_3 is intersected with that of the variable `zero`, which contains only the value 0, so effectively this implements a ***generate-and-test*** search for the canonical interval in the domain of x_1 that contains the leftmost zero of f_3.

```
DRF NestedSearch { b, &x1, x2, x3, x4, {
   AUX x1 IS RealInterval {};
   AUX x2 IS RealInterval {};
   AUX x3 IS RealInterval {};
   AUX x4 IS RealInterval {};
   VARIABLE lx1 IS RealInterval {};
   AUX fx1 IS RealInterval {};
   AUX fx2 IS RealInterval {};
   AUX fx3 IS RealInterval {};
   AUX fx4 IS RealInterval {};
   AUX zero IS RealInterval { 0 };
   DRF RIARule { x1^1 * (-1) <= -1*lx1 };
   DRF RIARule { lx1^1 * (1) = x1 };
   DRF RIARule { fx1^1 * (1) = 1 + lx1 };
   DRF RIARule { fx2^1 * (1) = 1 + x2 };
   DRF RIARule { fx3^1 * (1) = 2 + 5*x3^2 };
   DRF RIARule { fx4^1 * (1) = 1 + x4 };
   DRF RIARule { zero^1 * (1) =  x3 * fx3 + 1
                             - lx1 * fx1 - x2 * fx2 - x4 * fx4 };
   DRF RoundRobin { 0, lx1 };
} };
```

Program 7.2: Left-narrowing for argument x_1 of $f_3(x_1, x_2, x_3, x_4)$ of the Broyden banded functions

This is a slight deviation from the method outlined in Section 7.3.2. We use a decomposition, but we do not enforce hull consistency for it. We only use the decomposition to verify the unary interval constraint. We have this option because the `RIARule` operators implement the individual projections of constraints. Since we do not aim at computing hull consistency of the decomposition, and therefore use only one projection per constraint, we do not need to limit ourselves to atomic constraints, as we discussed in Section 4.5. In fact, the only reason for using a decomposition is that `RIARule` does not support brackets in the specifier string, while we want to use the syntax of formula (7.2) as a basis for the interval extension. This form appears to be less sensitive to the dependency problem than the form

$$2x_i + 5x_i^3 + 1 - \sum_{j \in J_i} (x_j + x_j^2)$$

The efficiency of the evaluation could probably be improved with a schedule for the operator-based scheduler that applies the operators only once.

In our opinion, being able to use `RIARule` both for computing hull consistency and for evaluating interval extensions of functions is a definite advantage of the design decision to separate the individual projections of constraints. The `RIARule` and `IIARule` plug-in are versatile tools for composing constraint solvers, while in combination with the programmable scheduler of Section 4.1.1, such solvers are not inherently less efficient than specialized algorithms such as HC4 for computing hull consistency.

The following results demonstrate the (well known) effect of computing box consistency for the Broyden banded functions benchmark: computation time increases linearly with the problem size. The target precision is 1.0e-8, but inside the nested search we split down to machine precision. This is realized by including in the top-level configuration two commands to replace the standard node evaluator, and to prevent branching on `RealInterval` variables whose domain is of the required precision:

```
TDINFO Precision { 1.0e-8 };
DRF LimitedPrecision { 1.0e-8, RoundRobin { 0, x1, x2, ... } };
```

This way, the system of equations is solved by propagation alone. The reported numbers are elapsed times in seconds, for problem instances specified by n.

$n =$	10	20	40	80	160	320
	2.536	6.210	13.503	28.467	58.097	118.926

We have described only a basic implementation of box consistency, and there is much room for improvement. For example, we do not compute hull consistency for the decomposition used inside the nested search, but it may actually be worthwhile to do so, and propagate back the information that the functions should evaluate to zero, or in general, that the unary interval constraints should hold. Also, because the consistency check is implemented by constraint propagation, it should be easy to add propagators for the Newton reduction step described in [BMVH94]. It should be investigated in how far our approach supports the use of other interval extensions than the natural interval extension.

The code of Program 7.2 is currently generated by a program written specifically for this benchmark. We still need to extend the OpenSolver preprocessor for arithmetic constraints with an option to generate code for enforcing box consistency. The preprocessor could then also generate the reduction operators corresponding to the Newton reduction step.

7.5.3 Box Consistency for Arithmetic Constraints on the Integers

Box consistency can also be enforced for arithmetic constraints on integer variables. In that case, a constraint $C \subset \mathbb{Z}^n$ on variables $x_1, \ldots, x_n$ with associated domains $D_1, \ldots, D_n \in \mathcal{I}$ is called box consistent if for all $1 \leq j \leq n$

$$D_j = int(D_j \cap \{i \in \mathbb{Z} \mid \langle D_1, \ldots, D_{j-1}, \{i\}, D_{j+1}, \ldots, D_n \rangle \in \boldsymbol{C}\})$$

where $\boldsymbol{C}$ is an interval extension of C. To illustrate that this is a useful application, consider the constraint

$$x^2y^2 - 4x^2y + 4x^2 - 4xy^2 + 16xy - 16x + 4y^2 - 16y + 16 = 4$$

and ranges $x, y \in [0..10^5]$. When we solve this equation by means of decomposition into atomic constraints, according to the approach proposed in Section 5.7, the 8 solutions are found in approximately 14 sec. in a search tree of 40255 nodes. When the code for the decomposed constraint is packed in four different `NestedSearch` operators, for left and right narrowing of x and y, the search tree is reduced to 39 nodes, and exploration takes less than 4 seconds.

7.5.4 Shaving

Shaving appears to be efficient only for larger problems. For small instances, like the ones we used in Section 7.5.1, the effort spent on shaving outweighs the benefit of the reduced search space. A similar experience is reported in [Zho97]. In this reference, the shaving, or ***bound trimming*** is on the domains of the variables that determine the processing order of the activities, so this technique is comparable to that of [VHPP00] mentioned at the end of Section 7.3.3.

An example of a larger problem instance, for which shaving is essential, is **swv01**. The optimum for this instance was first found by Perron [Per99], using a combination of limited discrepancy search, shaving, and parallel search. As we discussed in Section 4.1.2, LDS in OpenSolver is memory-intensive, and for this reason we used ***memory bounded LDS***, which we introduced in that same section.

Using memory bounded LDS and the implementation of shaving described in Section 7.3.3, we were able to prove optimality for swv01 in approximately 100 hours on a 2000 MHz Athlon XP processor. Without shaving, by that time the current best solution is nowhere near the optimum, and finding it within an acceptable multiple of the already long running time seems unlikely.

7.6 Discussion and Concluding Remarks

We have shown that in a branch-and-propagate tree search solver, three powerful pruning techniques from optimization, analysis of nonlinear functions, and constraint-based scheduling can be expressed as applications of a generic operator for nested search. In our implementation, each instance of this operator is itself an almost autonomous OpenSolver instance. This has the advantage that all facilities of the framework are available for specifying the nested search. A disadvantage is the overhead of using a general-purpose solver for very specific search problems, and dedicated implementations of box consistency and shaving will likely always be more efficient.

An evaluation of the efficiency of our implementation of box consistency and shaving is currently missing. Because this also depends on, for example, the interval arithmetic library that we use, this would require the implementation of dedicated operators for these techniques. There are still opportunities to improve the efficiency of our solution, though. For example, we plan to extend the set of commands of Section 3.3 with a command for first-solution search, to bypass the command loop for a longer period. Often, however, strong consistency notions like box consistency and shaving determine whether a problem can be solved or not. In such cases, being able to experiment with enforcing strong consistency notions is of greater importance than the actual efficiency.

Our approach promotes the composition of constraint solvers as low-level cooperations of basic solvers. As we discussed, this has further advantages: it avoids duplicate code in the solver implementation, and techniques carry over to other domains than those for which they were originally conceived. From this point of view it is desirable to have a small set of basic operators and combinators, that can be used to realize a wide range of constraint solving techniques. It seems reasonable that compositionality comes at the cost of some computational overhead for the framework.

There is an analogy between our operator for nested search and procedures in procedural programming languages. The input and output variables can be seen as by-value and by-reference parameters, respectively, and the OpenSolver input for the local CSP can be seen as a procedure body block. It would be interesting to investigate this analogy further, perhaps to define some notion of a parameterized reduction operator. This way we may be able to avoid duplicate code for activating plug-ins, and ease the rewriting. To illustrate that this is a useful facility, for the Broyden banded function, the code for each function (as shown in Figure 7.2) is repeated up to 14 times, with minor differences for left or right narrowing, and for the particular variable that we want to update. The OpenSolver input for $n = 320$ contains over 120,000 lines, with roughly the same number of plug-in instances created.

If we apply the shaving mechanism to arithmetic constraints, we achieve a notion of consistency known as ***3B consistency*** [Lho93]. Also shaving itself can be nested, which results in a technique known as ***double shave*** [MS96]. Both these techniques can be expressed as further applications of the generic operator for nested search. Other facilities that could be implemented using nested search are the ***squash*** operator of ECL^{i}PS^e, and the ***absolve*** operator of the CLIP system [Hic01].

The operator defined in Section 7.2 performs a single solution search, and in our applications we use this for updating the bounds of interval variables. For finite domains variables, it is attractive to have a variant that searches for all solutions. We can then use enumeration instead of bisection, and use the union of the domains for each variable as the result of the reduction operator. A potential application is to enforce ***singleton arc consistency*** (SAC, [DB97]). This notion of local consistency entails that for every value in the domain of every variable, the CSP can be made be made arc consistent if that value is assigned to the variable. To implement SAC using the proposed all-solution variant of our operator for nested search, we would need one such operator per DRF variable. This operator then searches in the domain of its variable for arc solved forms of the ECSP obtained by taking the full outer CSP, and making all other variables auxiliary. The output of the operator is the union of these arc solved forms, projected on the variable that it is applied to.

It is also possible to define box consistency (and the other applications discussed here) in terms of this all-solution variant, but then all internal zeros would be lost because of the interval representation. Therefore the first-solution search is a more natural and efficient implementation of these techniques.

Chapter 8

A Component-Based Parallel Constraint Solver

This is the second of three chapters that demonstrate the use of OpenSolver as a software component. Here we present the design and implementation of a parallel constraint solver that is composed of several autonomous OpenSolver instances. A small amount of specialized software coordinates the cooperation of the component solvers, and in our presentation we focus on the coordination aspects of the parallel solver. Since the goal of parallel processing is to reduce the turn-around time of a computation by distributing the workload, it is important to achieve a good load balance, and to ensure that communication does not dominate computation. This is realized by a time-out mechanism, implemented in the coordination layer of the solvers. By means of experiments we investigate whether the time-out mechanism, and the component-based implementation enabled by it lead to efficient parallel solvers.

8.1 Introduction

The goal of parallel processing is to reduce the turn-around time of a computation by distributing the workload over several hardware processors. Because constraint solving is computation-intensive, it can benefit from parallel processing:

- When the best known heuristics allow that problem instances of certain dimensions (number of variables, domain sizes) can be solved within acceptable time, users will probably want to solve problem instances of larger dimensions, possibly resulting from more detailed models.

- If constraint solving is used to predict the effect of some decision made in the context of a real-world problem, while the outcome of one experiment determines the parameters of the next, then being able to solve problems faster allows that more scenarios can be explored.

The obvious way to parallelize constraint solving is to explore different parts of the search tree in parallel: even for small problems, the search tree is generally large (see for example Table 7.1 on page 162), and every node of the search tree is an ECSP in itself, and can in principle be processed independently of the nodes in other subtrees.

The efficiency of a parallel computation depends on two factors: ***load balancing*** and the ***communication overhead***. We propose to address both factors by equipping a branch-and-propagate solver with a ***time-out*** mechanism. When an ECSP can be solved before the elapse of a given time-out, the solver simply produces all solutions that it has found (or ***the*** solution that it has found, if we are not interested in all solutions). Otherwise it also produces some representation of the work that still needs to be done. For tree search, this is a collection of subproblems that must still be explored: the ***search frontier***. These subproblems are then re-distributed among a homogeneous set of solvers that run in parallel. The initial solver is part of this set, and each solver in the set may split its input into further subproblems, when its time-out elapses.

The time-out mechanism provides an implicit load balancing: when a solver is idle, and there are currently no subproblems available for it to work on, another solver is likely to produce new subproblems when its time-out elapses. The time-out mechanism also gives control over the communication / computation ratio: when communication dominates, we can increase the time-out value in order that the solvers spend more time searching in between exchanging subproblems. We expect to be able to tune the time-out value such that it is both sufficiently small to ensure that enough subproblems are available to keep all solvers busy, and sufficiently large to ensure that the overhead of communicating the subproblems is negligible. The idea of using time-outs is quite intuitive, but to our knowledge, its application to parallel search is novel.

Rather than a parallel algorithm, we present this scheme as a pattern for composing a parallel constraint solver from component solvers. The only requirement is that these components can publish their search frontiers. We believe that this requirement is modest compared to building a parallel constraint solver from scratch. In the particular case of OpenSolver, the coordination-layer facilitates that the time-out mechanism is implemented without modifying the solver proper. Our presentation of the scheme in Section 8.3 uses the notion of abstract behavior types, and the Reo coordination model. These are introduced in Section 8.2. Section 8.4 details the implementation, and in Section 8.5 we describe the experiments that were performed to test the parallel solver. Compared to parallelizing an existing constraint solver, the component-based approach has further benefits. These are discussed in Section 8.6, together with related work and directions for future research.

8.2 Coordination and Abstract Behavior Types

As we discussed in Section 3.3.3, coordination, as a field of study in computer science, provides a perspective on component-based software engineering. From this perspective, software systems are composed from interacting component system, whose computations overlap in time. Contrary to modules and objects, which are the units of composition in the classical software engineering paradigms of modular and object-oriented programming, an instance of a prospective software component has then at least one thread of control. For the purpose of composition, the component is a black box, and we can assume that it communicates with its environment through a set of ports.

A software system that complies with the above notion of a component can be specified conveniently by an ***abstract behavior type*** (ABT) [Arb02]. ABTs are reminiscent of ***abstract data types*** (ADTs) used in modular programming:

- ADTs hide the internals of data structures. Through ADTs, data structures are characterized by the operations that are defined on them. This is the only information that is relevant for modular composition of software.

- ABTs hide the implementation of component systems, and characterize them by their behavior. This is the only information that is relevant for the composition of software systems through exogenous coordination. Because we are dealing with concurrent systems, timing is an important aspect of the behavior of a component.

Before we can introduce ABTs we first need to recall the definition of timed data streams. This notion originates from the work of Jan Rutten on co-algebras, stream calculus, and notably a co-algebraic semantics for the Reo (see below) coordination model [AR02].

A ***stream*** over some set A is an infinite sequence of elements of A. Zero-based indices are used to denote the individual elements of a stream, e.g., $\alpha(0)$, $\alpha(1)$, $\alpha(2)$, ... denote the first, second, third, etc. elements of the stream α. Also $\alpha^{(k)}$ denotes the stream that is obtained by removing the first k values from stream α (so $\alpha(0)$ is the head of the stream, and $\alpha^{(1)}$ is its tail). Relational operators on streams apply pairwise to their respective elements, e.g., $\alpha < \beta$ means $\alpha(0) < \beta(0)$, $\alpha(1) < \beta(1)$, $\alpha(2) < \beta(2)$, ...

A ***timed data stream*** over some set D is a pair of streams $\langle \alpha, a \rangle$, consisting of a data stream α over D, and a time stream a over the set of positive real numbers, and having $a(i) < a(j)$, for $0 \leq i < j$. The interpretation of a timed data stream $\langle \alpha, a \rangle$ is that for all $i \geq 0$, the input/output of data item $\alpha(i)$ occurs at "time moment" $a(i)$.

An ***abstract behavior type*** is a (maximal) relation over timed data streams. Every timed data stream involved in an ABT is tagged either as its input or output. For an ABT R with one input timed data stream I and one output timed data stream O we use the infix notation $I\ R\ O$. Also for two such ABTs R_1 and R_2, let the composition $R_1 \circ R_2$ denote the relation

$$\begin{aligned}\{\ \langle\langle\alpha,a\rangle,\langle\beta,b\rangle\rangle \mid\ & \text{there exists a timed data stream } \langle\gamma,c\rangle \\ & \text{such that } \langle\alpha,a\rangle R_1\langle\gamma,c\rangle \text{ and } \langle\gamma,c\rangle R_2\langle\beta,b\rangle\ \}.\end{aligned}$$

ABTs specify only the black box behavior of components. For a model of their implementation, other specification methods are likely to be more appropriate, but that information is irrelevant for the coordination of the components.

Reo [Arb02, ABRS04] is a channel-based exogenous coordination model wherein complex coordinators, called *connectors* are compositionally built out of simpler ones. The simplest connectors in Reo are a set of *channels* with well defined behavior. In Section 8.3.2 we use Reo connectors to specify the coordination of our component solvers.

8.3 Specification

8.3.1 Component Solver

In this section we define an ABT for a constraint solver with the time-out mechanism. In Section 2.2.6 we defined constraint solving as the transformation of extended constraint satisfaction problems, so for formalizing the notion of a solver we need a domain, or universe, of ECSPs. Let U denote the set of all ECSPs, and let $\mathcal{U}$ denote the set of all finite subsets of U. Also, for $p \in U$ we define the following set

$$sol_\gamma(p) = \{p' \in U \mid p' \text{ is a } \gamma \text{ solved form of } p\}$$

Next we specify that a constraint solver transforms a problem into a set of mutually incomparable problems. Let D denote the data domain $U \cup \mathcal{U} \cup \{\tau\}$, where $\tau \notin U$ is an arbitrary data element that serves as a ***token***. In the following, let $\langle\alpha,a\rangle$ and $\langle\beta,b\rangle$ be timed data streams over D. Now the behavior of a ***basic solver*** is captured by the *BSol* ABT, defined as

$$\langle\alpha,a\rangle\ BSol\ \langle\beta,b\rangle \equiv\ a < b \wedge S(\alpha,\beta)$$

where S is a relation on U and $\mathcal{U}$, such that for all $p \in U$ and $R \in \mathcal{U}$, $S(p,R)$ iff

- every ECSP in R is a proper subproblem of p,
- no ECSP in R is a subproblem of another ECSP in R, and
- $sol_\gamma(p) = \bigcup_{r\in R} sol_\gamma(r)$, for some notion of consistency γ.

The *BSol* ABT formalizes the notion of an ***incomplete constraint solver*** of Section 2.2.6.

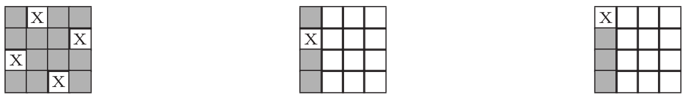
8.3.1. Example. An example of a software system that complies with the *BSol* ABT is a UNIX process that keeps reading (character encoded) ECSPs from standard input. Some time after reading each ECSP, and before reading the next, it produces on standard output one of the following (character encoded) set of ECSPs:

- the empty set if the ECSP that was read is inconsistent, or otherwise
- a set containing the first solved form found with branch-and-propagate tree search, plus an ECSP for every element of the search frontier of the branch-and-propagate algorithm.

This can be realized by repeated application of Algorithm 8.1, which is a modified version of Algorithm 2.2 on page 25.

Figure 8.1 shows the search tree for a possible execution of this algorithm for the four queens problem (see Section 4.2). We assume chronological variable selection, enumeration value selection, and depth-first leftmost-first traversal. The nodes in the figure depict the domains of the four variables as columns of four possible values (rows), where white fields are elements of the domains, and dark fields have been removed from the domains. Vertical edges denote constraint propagation, and diagonal edges denote branching. If, by propagation or branching, the domain of a variable becomes a singleton set, an X in the remaining field marks the position of the queen on the chess board.

The two leftmost leaves of the search tree are failures (they contain columns without white fields), and the algorithm backtracks twice to find the first solution in the third leaf from the left. At this point, the search frontier still contains two nodes, and the output of our process for an ECSP corresponding to the four queens problem is a set of three ECSPs, corresponding to the following configurations, a solution and two internal nodes:



□

The *Str* (streamer) ABT specifies that a stream of sets of problems, as produced by a basic solver, is transformed into a stream of problems, where the sequence of problems for each input set is delimited by a token:

$$\begin{aligned}\langle \alpha, a\rangle \; \mathit{Str} \; \langle \beta, b\rangle \equiv\; & a(0) = b(0) \\ & \wedge \beta(k) = \tau \\ & \wedge \alpha(0) = \{\beta(0), \ldots, \beta(k-1)\} \\ & \wedge \langle \alpha^{(1)}, a^{(1)}\rangle \; \mathit{Str} \; \langle \beta^{(k+1)}, b^{(k+1)}\rangle\end{aligned}$$

where for all $i \in \mathbb{N}$, $\alpha(i) \in \mathcal{U}$ and $\beta(i) \in U \cup \{\tau\}$, and k denotes $|\alpha(0)|$, the cardinality of the (finite) set of problems at the head of stream α. Now the

parameters: function *select*,
function *propagate*.

input: an ECSP $P := \langle \mathcal{C} \ ; \ x_1 \in D_1, \ldots, x_n \in D_n \ ; \ \mathcal{T}_1, \ldots, \mathcal{T}_n \ ; \ \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle$,
a domain branching function f,
a set R of domain reduction functions,

output: a set S of sequences of domains such that for all $\langle D'_1, \ldots, D'_n \rangle \in S$,

$$\langle \mathcal{C}[D'_1, \ldots, D'_n] \ ; \ x_1 \in D'_1, \ldots, X_n \in D'_n \ ; \ \mathcal{T}_1, \ldots, \mathcal{T}_n \ ; \ \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle$$

is a subproblem of P. If S is non-empty, at least one of these subproblems is also a γ solved form of P, where γ is the notion of consistency enforced for the constraints in $\mathcal{C}$ by *propagate* and R.

```
$F := \{\langle D_1, \ldots, D_n \rangle\}$
$S := \emptyset$
repeat
    $select\ D_w \in F$
    $F := F - \{D_w\}$
    $D'_w := propagate(D_w, R)$
    if $\neg failed(D'_w)$
    then
        if $final(D'_w)$
        then
            $S := S \cup \{D'_w\}$
        else
            $F := F \cup f(D'_w)$
        end
    end
until $F = \emptyset$ or $S \neq \emptyset$
$S := S \cup F$
```

Algorithm 8.1: A first-solution search version of Algorithm 2.2

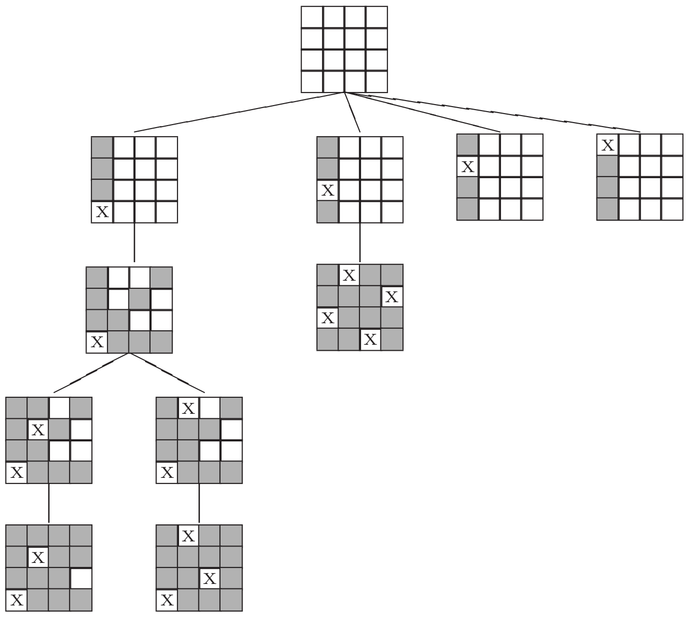


Figure 8.1: First-solution search for the four queens problem

behavior of a constraint solver component is captured by the *Sol* ABT, defined as

$$Sol = BSol \circ Str$$

Thanks to the *Str* ABT, we can reduce the data domain from $U \cup \mathcal{U} \cup \{\tau\}$ to $U \cup \{\tau\}$: systems that comply with the *Sol* ABT deal only with ECSPs and the token τ.

8.3.2. Example. Figure 8.2 shows the input timed data stream $\langle \alpha, a \rangle$ and output timed data stream $\langle \beta, b \rangle$ of a system that complies with the *Sol* ABT. Some time t_s after an ABT for the four queens problem appears on the input timed data stream, a solution, and two subproblems followed by the token τ appear on the output timed data stream. □

Unlike our example solver, existing complete constraint solvers do not usually produce subproblems other than solutions. The search frontier is inaccessible, and the token τ can be thought of as the notification "no" that a Prolog interpreter would produce to indicate that no (more) solutions have been found. Also, if we model such solvers using the *Sol* ABT, there is typically no upper bound on the time that elapses before solutions start to appear on the output timed data stream.

In contrast, the load-balancing solver component that we propose here stops searching for solutions after the elapse of a time-out t. At that moment, it generates a subproblem for every solution that it has found, plus one for every

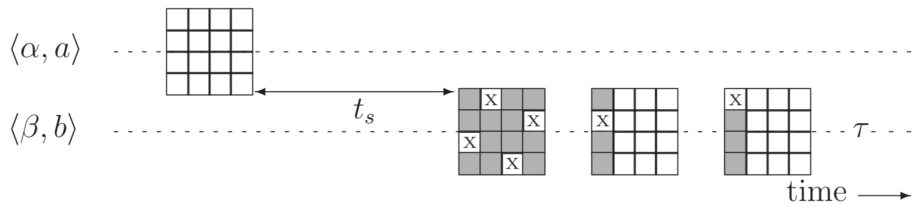


Figure 8.2: Input stream and output stream of a solver that complies with the *BSol* ABT

subproblem that must still be explored. For $t \in \mathbb{R}^+$, the Sol_t ABT specifies that there is an upper bound on the time needed for the solver to produce results.

$$\begin{aligned}\langle \alpha, a \rangle \; BSol_t \; \langle \beta, b \rangle \equiv \; & \langle \alpha, a \rangle \; BSol \; \langle \beta, b \rangle \\ & \wedge \forall i \in \mathbb{N} \cdot b(i) - a(i) < t + t_\epsilon\end{aligned}$$

$$Sol_t = BSol_t \circ Str$$

where $t_\epsilon \in \mathbb{R}^+$ is some extra time that allows the solver to finish what it is doing, after the time-out t has elapsed.

The Sol_t behavior can be realized trivially by removing the loop from Algorithm 8.1. For an input ECSP, the resulting solver then performs a single round of constraint propagation and splitting. However, in order to limit the amount of communication, for our application we want to make the solvers perform as much work as possible, within the given time-out period.

We can capture this additional requirement by adding the following condition to the $BSol_t$ ABT:

$$\forall i \in \mathbb{N} \cdot b(i) - a(i) \geq t \;\vee\; \beta(i) = sol_\gamma(\alpha(i))$$

This ensures that unless the search space has been explored exhaustively, the output is produced between time t and $t + t_\epsilon$.

Implementing this exact behavior is not straightforward, but an approximation of it can be realized by modifying the loop of Algorithm 8.1:

```
repeat
    ...
until $F = \emptyset$ or $S \neq \emptyset$
```

as follows.

```
$t_0$ := clock()
repeat
    ...
until $F = \emptyset$ or clock() $- t_0 \geq t$
```

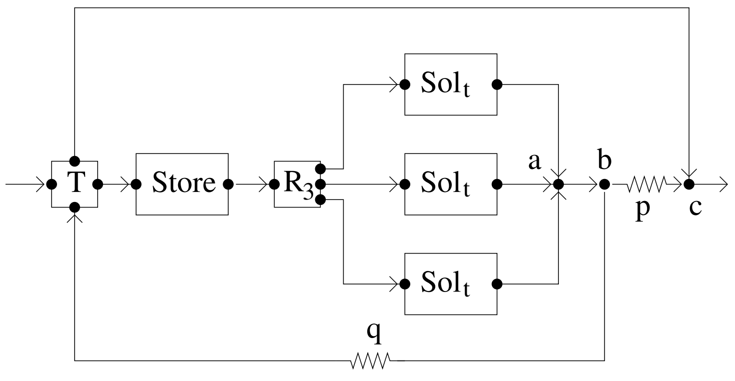


Figure 8.3: 3-way parallel solver

In this modified code t is a new parameter that specifies the time-out value, and clock is a function that returns the current time, or ***wall time***. Such a function is typically made available by the operating system. The resulting solver is an approximation of the required behavior in the sense that the operations inside the loop cannot be interrupted. Especially constraint propagation may take longer than the allowed overrun time t_ϵ. This is not a problem in practice.

8.3.2 Parallel Solver

Figure 8.3 shows a channel-based design for a (3-way) parallel solver. All channels in this design are synchronous[1]: read and write operations block until a matching operation is performed on the opposite channel end. The “resistors” depict Reo filters: synchronous channels that forward data items that match a certain pattern (set of allowable data items) and discard data items that do not match this pattern. At node b in Figure 8.3, all output of the solvers is replicated onto two filters. Channel bc filters out solutions. Its pattern (p) is

$$\texttt{Filter}(\{p \in P \mid p \in sol_\gamma(p)\}).$$

The channel from b to T discards all solutions. Its pattern (q) is

$$\texttt{Filter}(\{p \in P \mid p \notin sol_\gamma(p)\} \cup \{\tau\}).$$

The ABTs of the channels are specified in [Arb02].

Apart from the channels and the three load-balancing solvers Sol_t, there are three elements of the design that require further clarification: the special-purpose connector T, the 3-ary exclusive router R_3, and the Store. Because we focus on the component solvers instead of on the coordinating framework, we do not give full ABTs for the other elements of Figure 8.3, but only an intuitive description.

[1]The synchronicity of the communication is not an important aspect of the design.

The special purpose connector T implements ***termination detection***. Initially, it reads a problem from its left-hand side input port. All subproblems entering T, through either input port, are forwarded immediately through its right-hand side output port to the Store. Also T counts the number of problems forwarded to the Store, and the number of tokens τ received through its bottom port (from node b). While these numbers do not match, the parallel solver is busy, and T will accept new (sub)problems from its bottom input port (connected to node b) only. As soon as the number of problems is canceled out by the number of tokens, T sends a token τ through its top port (to node c), indicating that the parallel solver has finished working on its current problem. Then it returns to its initial state, and accepts a new problem from its left-hand side input port. It should be noted that termination detection for the parallel solver is much easier than termination detection for a distributed application in general. The latter case is discussed in Section 9.2.3.

Connector R_3 is a general-purpose 3-ary ***exclusive router***. It operates synchronously, and every data item on its input port is forwarded on exactly one of its output ports. If none of the channels connected to the output ports is able to forward a data item, the router blocks. If a data item can be forwarded on more than one output port, a non-deterministic choice is made. Construction of the exclusive router from Reo primitives is shown in [ABRS04].

The ***Store*** is a channel-like connector that is specific to this application. It buffers incoming problems, and examines them to determine the level of the corresponding node of the search tree. This information can be used to enforce a global traversal strategy. When R_3 is ready to accept data (i.e., when one of the load-balancing solvers has become idle) it forwards a problem according to this strategy. For example, it may forward a node of the deepest available level in an attempt to implement depth-first search globally. This effectively drains the Store. Forwarding a node of the shallowest available level implements breadth-first search, filling up the Store with more subproblems.

8.4 Implementation

To test the proposed implementation of parallel search, we equipped our OpenSolver constraint solver with the time-out mechanism, and developed a distributed program to combine several such solvers into a parallel constraint solver.

8.4.1 Component Solver

A special coordination layer plug-in `StreamingIO` has been developed that configures OpenSolver as a load-balancing solver, as specified in Section 8.3. When it is equipped with this plug-in, an OpenSolver instance keeps reading configuration specifications from its standard input. These specifications are sequences of

ASCII characters, in the language of Figure 3.2 (see Program 4.3 on page 77 for an example). The individual configuration specifications are delimited by brackets, and configure OpenSolver for solving a particular ECSP.

When a solver configuration has been read from standard input, the coordination layer plug-in instructs the solver to parse it, and starts the search for solutions. This coordinates the solver to perform a regular branch-and-propagate search, as explained at the end of Section 3.3.1. When the time-out elapses, or when the search frontier becomes empty, the `StreamingIO` plug-in stops issuing commands that drive the search for solutions. Instead it issues the **flush** and **clear WDB** commands of Section 3.3.2.

Every plug-in implements a method to write itself into a character string. When executing the command to flush the search tree, this method is called for all plug-ins that define a particular node of the search tree, notably the variable domains and the DRFs. These strings are then passed to the coordination layer. Normally this mechanism is used to produce the solved forms of an ECSP, but because we do not perform an exhaustive search, in this case it also produces the search frontier. This information is used by the `StreamingIO` coordination layer plug-in to construct new solver configurations that are written to standard output. After the flushing operation is complete, the coordination layer plug-in generates a character-encoded token τ, and proceeds by reading a new problem specification from standard input. Except for the token, the output of this coordination layer plug-in can directly be fed into another solver as a stream of solver configurations.

The component solvers are configured to perform a depth-first traversal of the search tree, but through an adapter, the branching operators are modified to annotate the nodes with their level in the search tree. These annotations appear in the solver configurations that are forwarded through the network, and can be interpreted by the process that implements the Store of Section 8.3.2 to impose a high-level traversal strategy on top of the depth-first traversal of the solvers.

As we discussed in Section 4.2, OpenSolver is based on copying, so the search frontier is maintained explicitly. This is a great convenience for publishing the search frontier, but we are convinced that our method extends to solvers that use trailing or recomputation. Especially when searching for all solutions, every node of the search tree must be generated eventually, so no extra work is involved if this is done for the current search frontier when the time-out elapses.

8.4.2 Parallel Solver

Depending on the complexity of the interaction, it may make sense to use a dedicated coordination language to orchestrate the interaction of the cooperating entities in a concurrent system. For example if the population of processes is highly dynamic, the Manifold coordination language [Arb96] may be a logical choice (see also Section 9.2.1). In this case, we implemented the coordination protocol of Section 8.3.2 as a master-slave distributed program coded in C using

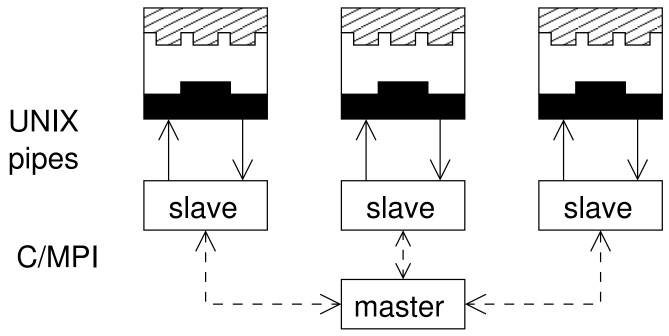


Figure 8.4: Software architecture of the parallel solver

the MPI message passing interface. Without the facilities for gathering statistics, the size of this "glue code" is just a little more than 600 lines. The slave processes fork a new UNIX process to start the component solvers, and a pair of pipes is connected to the standard input and output of these processes to facilitate the character-based implementation of the timed data streams.

The channels of the coordination model are implemented by directed send and receive MPI calls. Upon reception of a token τ, a new subproblem is sent to the solver that generated the token. For this purpose, the character-based encoding of the token contains the identity of this solver. Also the number of solutions counted for each subproblem is piggybacked on the token.

When reading from the pipe that is connected to the standard output of a solver, the slave processes perform some parsing to recognize the beginning of a new solver configuration. At this point, an entire problem is sent to the master process as a character string. The master process implements the distribution and gathering of the problems. Figure 8.4 illustrates this software architecture. In total, for an n-way parallel run, $2n+1$ user processes are running on n processors.

Note that the component solvers are still stand-alone applications that rely on character-based standard I/O only. Our primary goal was a performance evaluation of the time-out mechanism, and from that perspective, a master-slave implementation is acceptable. However, the channel-based design of Section 8.3.2 has many advantages over this rigid scheme. In particular, the decision where to send the next subproblem is now taken on the basis of solver output, whereas a true implementation of the exclusive router would be able to detect that a solver is idle when the channel connecting to that solver is ready to accept new data. This has the benefit of a better separation of concerns and of a reusable solution. The Manifold coordination language fully supports the design of Section 8.3.2.

8.5 Experiments

The parallel solver was tested on three combinatorial problems:

Queens An instance of the n-queens problem, as described in Section 4.2. Program 4.3 shows a solver configuration for $n = 4$. Instead of the variable based scheduler we used the default, operator-based scheduler. The results reported here are for $n = 15$, for which there are 2,279,184 solutions.

Sat An instance of the propositional satisfiability problem, described in Section 4.4. For these experiments we use the benchmark formula par16-2-c from the DIMACS test set[2]. This formula has 1392 clauses on 349 variables.

Coloring This is a graph coloring problem. In general, the problem is to find an assignment of colors to the vertices of a graph, such that two vertices that are connected by an edge have different colors. Here we verify that no 9 coloring exists for graph DSJC125.5, also from a DIMACS test set, having 125 nodes and 3891 edges. In our model we use a variable for every node, and a disequality constraint for every edge. The disequalities are implemented using the `DDNEQ` DRF plug-in of Section 4.2.

In all cases, we used a fail-first variable selection strategy, selecting a variable with the smallest remaining number of alternative values. As a second criterion for ***Coloring***, variables are ordered according to the degree of their corresponding nodes of the graph. In order to generate a large number of subproblems, we used an enumeration value selection strategy (see Figure 4.2 on page 70). The component solvers perform a depth-first traversal, but using the level annotation of the configurations generated by the solvers, the master switches between breadth-first and depth-first traversal, depending on the number of available subproblems. If this number is below a certain threshold value (512, for these experiments) priority is given to the shallowest available nodes. These are least likely to complete before the time-out, and can thus be expected to increase the number of problems available to the master, making it easier to keep all solvers busy. Also, when the full problem is first submitted to the first solver, this solver uses a very small time-out in order to generate work for the other solvers quickly.

The results reported below are for an all-solution search, and solutions are only counted, not stored or communicated. An all-solution search avoids the effect known as the ***speedup anomaly***, which entails that for a non-exhaustive search, part of the speedup is due to the different traversal of the search space. For example, consider that the search space is split into two subtrees, and that the root node of the second subtree happens to reduce to a solution. Parallel search on these two subtrees would find the solution almost immediately, resulting in a ***super-linear speedup*** over the sequential case where the other subtree

[2]available at `ftp://dimacs.rutgers.edu/pub/challenge`

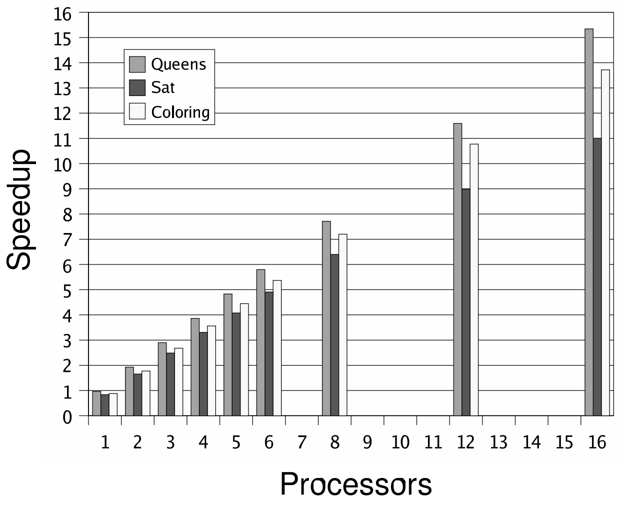


Figure 8.5: Speedup figures

is processed first. Because we are interested in evaluating the efficiency of our proposed parallelization of tree search, we tried to avoid demonstrating the speedup anomaly.

Table 8.1 shows the sequential and parallel runtimes (elapsed time) for our test problems, as well as the parallel efficiency, which is the actual speedup divided by the number of processors. As a further indication that our solver is a realistic implementation, depending on the search strategy, the standard example for 15-queens in ECL^{i}PS^e 5.5 [WNS97, CHS+03] completes in 900 - 1500 sec. on the same hardware. The speedup figures (sequential runtime divided by parallel runtime) are shown in Figure 8.5. All elapsed times shown are averages of 10 repeated runs on a Beowulf cluster built from 1200 MHz Athlon nodes. The entries for "parallel" runs on 1 processor are an indication of the overhead of the time-out mechanism. For ***Queens*** and ***Sat*** we used a time-out value of 3200ms. For ***Coloring*** we used 9600ms. These values were found to give good results in preparatory experiments, but performance did not seem overly sensitive to the actual time-out used. The master process always runs on the same node as one of the component solvers and its slave process, and competes with these processes for CPU time.

As can be seen from Figure 8.5, our parallel solver scales well. For ***Queens*** and ***Coloring***, the parallel efficiency remains practically constant for the numbers of processors that we have tested with, and the scalability can be expected to extend to higher numbers of processors. The difference in efficiency for these two series of runs, and for the ***Sat*** runs on lower numbers of processors can be explained by the different sizes of the problem representations, and their associated

	Processors				
	Seq	1	2	3	4
Queens	734.16	760.79	380.85	253.41	190.04
eff.		0.96	0.96	0.97	0.97
Sat	1541.12	1842.55	931.26	619.65	466.08
eff.		0.84	0.83	0.83	0.83
Coloring	419.29	475.92	236.50	156.47	117.70
eff.		0.88	0.89	0.89	0.89

	Processors				
	5	6	8	12	16
Queens	152.01	126.67	95.18	63.32	47.86
eff.	0.97	0.97	0.96	0.97	0.96
Sat	378.14	313.91	240.91	171.43	140.02
eff.	0.82	0.82	0.80	0.75	0.69
Coloring	94.31	78.11	58.23	38.92	30.56
eff.	0.89	0.89	0.90	0.90	0.86

Table 8.1: Elapsed times (sec.) and parallel efficiency

communication costs.

For ***Sat***, parallel efficiency drops after 8 processors. The reason is that because the variable domains are binary, the search frontiers are smaller than for the other two problems, and the master has difficulty keeping all solvers busy. Also the problem seems to have a less balanced search space: submitting a shallow subproblem to one of the solvers is less likely to generate new nodes than for ***Queens*** and ***Coloring***. We hope to remedy the problem of the binary search trees by using a special-purpose branching strategy plug-in, which instantiates several variables at the same time, thus generating larger search frontiers. However, this strategy will also generate assignments that would otherwise have been prevented by constraint propagation, so it is hard to predict the overall effect.

The ***Queens*** experiments have also been run overnight on several (mostly idle) workstations connected by a local area network. While a detailed analysis of these experiments has not been made, here too we saw good speedup and scalability. Our approach seems well suited for such an environment: because no solver will work longer than the specified time-out before sharing work with other solvers, the proposed implementation of parallel search will likely be insensitive to the existing load and heterogeneity of the hardware. Because good results were obtained on a cluster (distributed memory), the parallel solver can also be expected to perform well on shared memory machines.

8.6 Discussion

Perspectives and Limitations

As an alternative to implementing the time-out mechanism in the component solvers, we could move this mechanism into the software that coordinates them. It would be equally easy to modify a constraint solver to respond to some interrupt, and somehow an interrupt mechanism seems less alien to constraint solving than a time-out mechanism. In either case the solver must be able to publish the state of its search algorithm, for which we use a character-based encoding.

There are other advantages to enabling a solver to publish its search frontier. For instance, it allows user interaction in constraint solving, e.g., for computational steering, and supports a mechanism for checkpoints. When the set of subproblems held by the master process is saved to disk at regular intervals, and subproblems are not discarded until their results have been processed, the solver can restart from the last saved set of subproblems after, for example, a power failure has occurred. Also saving subproblems to disk may increase the applicability of limited discrepancy search in a copying-based solver (see Section 4.1.2), especially if the I/O can be performed in the background, and does not imply busy waiting.

Another possibility is to implement in-search transformations of CSPs outside the core solver. In OpenSolver, such transformations are currently limited to deactivating reduction operators that have become redundant. For other transformation techniques, such as adding redundant constraints, or symbolic rewriting of arithmetic constraints, it is not immediately clear how to incorporate these in the branch-and-propagate search in a uniform way. Moreover, extending OpenSolver to accommodate specialized transformation techniques may make it less efficient in those cases where these techniques are not needed. By applying transformations outside the branch-and-propagate search, we risk that constraint propagation is weaker than necessary, but because of the time-out mechanism, this inefficiency will not last long, and the overall effect may be a good compromise between ease of implementation and efficiency.

Our current implementation is not suited for optimization, because new bounds for a criterion variable are not communicated between solvers. When a new bound is discovered, many of the subproblems in the Store may never be able to improve on this bound, but they have to be processed nonetheless. What is worse, the new bound remains local to the subproblems of the ECSP in which it was found. After flushing its search frontier, a solver returns to its initial state, and forgets the bound, and only the solvers that pick up one of the generated subproblems will temporarily be able to use it. This can be remedied by adding two processes to the network of Figure 8.3. One process inspects the value of the criterion variable for outgoing solutions, and sends this value to the other process, which adds a reduction operator for enforcing the current best bound to any subproblem

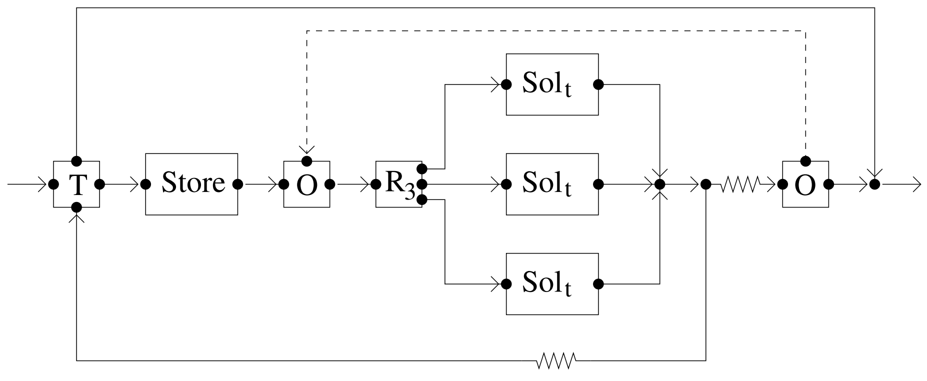


Figure 8.6: 3-way parallel solver with optimization

leaving the Store. This is illustrated in Figure 8.6.

Branch-and-bound optimization was studied from a coordination point of view in [Sta02], but in our work, the emphasis is on the component side rather than on the coordination framework, and on the demonstration of a realistic implementation. We do not expect that our component-based solution performs worse than other parallel implementations of branch-and-bound, but this should be verified by further experiments.

Constraint solving was used as an example application, but our method can probably be applied to other problems that involve tree search. This is not surprising, because for many such problems, there exists a more or less efficient encoding as a constraint satisfaction problem. However, some problems that involve tree search have special requirements. As an example, we have seen in Section 4.4 that specialized solvers for the SAT problem rely on so-called learning search algorithms, which derive new constraints during the traversal of the search tree. These constraints are redundant, but when they are made explicit they achieve a stronger pruning of the search tree. It is not directly clear how our method should be extended to facilitate learning solvers, and the complementing backjumping techniques.

Related Work

Other approaches to parallel constraint solving often use a scheme where the parallel solvers exchange nodes of the search tree only when one of them becomes idle, see for example [MS94, Per99, Sch00, Ham05]. For such schemes, solvers can potentially run for a long time without having to respond to a request for work from other solvers, but once a solver becomes idle, it may be more difficult to find another solver that is willing to share part of its search frontier. In contrast, our approach aims at having a large repository of work, assuming that the time-out can be tuned such that publishing the search frontier is relatively cheap. From a software engineering point of view it is simpler, and better suited for a component-based implementation, but from a user's point of view, our scheme is more complicated because it introduces a tuning factor. It may well be possible, though, to use a heuristic for tuning the time-out automatically during the computation. For example, the Store of Figure 8.3 could increase the time-out value while enough subproblems are available to keep all solver busy for some time.

In [HKS01] a shared-memory scheme is described where first the original CSP is split by assigning values to variables in a generate-and-test phase, until a large set of subproblems are available. These problems are then solved in a data-parallel way, using either a static or dynamic partitioning. We expect that scheme to be more sensitive to load imbalance because it is possible that most of the work is concentrated in only a few of the generated subproblems.

The approach of Disolver [Ham05] is unique in the sense that load balancing and bound sharing (in optimization) can be controlled through setting some pre-defined logical variables. In addition, several properties of the search process for which we use annotations are reflected in the domains of other pre-defined variables. Regular constraints can be now be used to activate or deactivate load balancing and bound sharing depending on properties of the search process. This way, adaptive cooperations between the parallel running solvers can be specified through constraints.

For all alternatives discussed here, a comparison of reported efficiency results is difficult, because the hardware platforms and software environments, and the benchmark problems used in each case are quite diverse. For example, the results in [Per99] for ILOG Solver apply to job-shop scheduling problems. Because these are optimization problems, the experiments are very sensitive to the traversal order, leading to speedup anomalies: for various experiments, the observed speedup ranges from 1.23 to 3.92 on two processors, and from 2.4 to 28.95 on four processors. The author mentions that running the parallel solver with one processor incurs an overhead of 2–3%, which could be comparable to the column for 1 processor in Table 8.1. This result is for a shared memory system, which may explain the low overhead compared to our approach.

Some of the results in the other references above do not suffer from the speedup anomaly, and just to indicate that despite its straightforward load balancing scheme our parallel solver is fairly efficient, Table 8.2 presents a comparison yet. For each system, the best and worst speedups among the presented set of experiments are listed. From [Sch00] we did not consider the optimization problems because of the speedup anomaly. For the same reason, from [HKS01] we cite only the results for inconsistent problems. Furthermore it should be noted that the speedup figures of [MS94] were not obtained by comparison with the best possible sequential run. They apply to the parallelized system, which is less efficient. Also [HKS01] concerns a shared memory implementation. The quoted results are for their best (dynamic) partitioning strategy only. The results reported for ECL^{i}PS^e in [MS94] and for the Mozart implementation of Oz in [Sch00] are for distributed

	Speedup			
CPUs	OpenSolver	[MS94]	[Sch00]	[HKS01]
2	1.65–1.93	1.78–1.99	1.74–1.85	
3	2.49–2.9		2.47–2.63	
4	3.31–3.86	3.19–3.90	2.92–3.30	3.74–3.96
5	4.08–4.83		3.12–3.51	
6	4.91–5.8		3.17–3.81	
8	6.4–7.71	4.60–7.56		6.32–7.44
12	8.99–11.59	5.37–10.79		
16	11.01–15.34			9.08–13.95

Table 8.2: Comparison of speedup figures

memory (networked) systems, like ours. The last two systems both use recomputation to regenerate nodes of the search tree that have been transferred from one machine to another. For Disolver, we did not find any results that do not suffer from the speedup anomaly.

8.7 Conclusions

We proposed an implementation of parallel tree search in constraint solving based on time-outs. Instead of a parallel algorithm, we presented and implemented the method as a protocol for the coordination of multiple instances of a component solver. After equipping a constraint solver with the time-out mechanism, some 600 lines of C/MPI code were sufficient to coordinate several of these component solvers to perform parallel search. Experiments showed that a good speedup is obtained on 2 to 16 CPUs, which indicates a good load balance. We conclude that:

- The time-out mechanism is an effective way to implement parallel search in constraint solving.
- Once a solver is able to publish its search frontier, building a parallel constraint solver becomes a matter of component-based software engineering.
- The OpenSolver plug-in mechanism made it very easy to meet this requirement.
- Separating computation and coordination, i.e., adding a protocol instead of implementing a parallel algorithm is a viable approach.

We also described how to implement parallel optimization. We do not expect that our component-based solution performs worse than other parallel implementations of branch-and-bound, but this should be verified by further experiments.

Chapter 9

Distributed Constraint Solving

The subject of this chapter is DICE (DIstributed Constraint Environment), a framework for distributed constraint solving. In addition to the design of the framework, and its implementation in the Manifold coordination language, we will discuss a number of extensions that were proposed to improve the efficiency of distributed constraint solving in DICE. OpenSolver was originally developed as a part of DICE, to realize these optimizations, and the material presented here clarifies some of the design decisions. It also demonstrates a third potential application of OpenSolver as a software component.

9.1 Introduction

Constraint propagation algorithms can essentially be characterized as a set of functions, plus a scheduler that coordinates their application. This supports the observation of Gelernter and Carriero that all useful programs consist of a combination of computation and ***coordination*** [GC92]. This observation was made in the context of coordination ***languages***. Although they originated in the area of parallel and distributed computing, coordination languages now also manifest themselves as a technology for realizing component-based software engineering.

As we have seen in the previous chapters, constraint solving comprises a collection of largely independent techniques. To solve a CSP efficiently, a constraint solver typically combines several procedures, heuristics, and even stand-alone solvers. This indicates that in addition to providing evidence for the "programming = computation + coordination" proposition, constraint solving might also benefit from the component-based approach that is facilitated by contemporary coordination languages.

The above observations suggest a coordination-based implementation of constraint solving, which was explored in two articles: [Mon00a], on a coordination-based constraint propagation algorithm, and [AM00], which complements this algorithm with facilities for search in a distributed setting, i.e., in absence of a centralized representation of CSPs. An additional benefit of the approach proposed in these articles is that because of the concurrent nature of the coordination language, the solver can be applied in situations that require distributed solving, or where it can be expected that parallel execution of domain reduction functions will reduce the turn-around time of solving.

An implementation of the distributed constraint propagation algorithm in the Manifold coordination language provided the proof of concept, and DICE [Zoe03b] combined both algorithms into a general purpose coordination-based constraint solver. Because of its distributed nature (every variable and reduction operator in DICE had its own thread, possibly running in its own process), constraint solving in DICE involved massive communication among concurrent threads, which made it quite inefficient compared to existing, sequential constraint solvers. For this reason, in [Zoe03a] we proposed an alternative implementation that allows an arbitrary distribution of variables and reduction operators over a set of cooperating solvers. The resulting system allows all configurations ranging from a fully distributed solver to a single sequential constraint solver. OpenSolver was intended to implement the cooperating solvers of this alternative implementation.

The remainder of this Chapter is organized as follows. Section 9.2 covers the original DICE system. It briefly introduces the Manifold coordination language, and recalls the coordination-based constraint solving algorithms of [Mon00a] and [AM00]. In Section 9.3 we describe the alternative implementation proposed in [Zoe03a], and in Section 9.4 we relate this proposed alternative to the current OpenSolver implementation. In Section 9.5 we evaluate the benefits of our approach, and discuss related work.

9.2 DICE

DICE (DIstributed Constraint Environment) is a framework for distributed constraint solving, implemented using the Manifold coordination language. A running system consists of a number of processes, that cooperate according to coordination protocols for constraint propagation, distributed termination detection, and search. These are described below in Sections 9.2.2–9.2.4. First we introduce Manifold and its coordination model.

9.2.1 Coordination Model and Language

Coordination languages offer language support for composing and controlling software architectures made of concurrently executing entities. In the Idealized Worker Idealized Manager (IWIM) model of coordination [Arb96], these entities are represented by ***processes***. In addition to processes, the basic concepts of IWIM are ***ports***, ***channels*** and ***events***. A process is a black box that exchanges ***units*** of information with the other processes in its environment through its input ports and output ports, by means of standard I/O primitives analogous to read and write. The interconnections between the ports of processes are made through directed channels. Independent of channels, there is an event mechanism for information exchange in IWIM. Events are broadcast by their sources, yielding ***event occurrences***. Processes can tune in to specific event sources, and react to event occurrences.

The IWIM view of a software system is a dynamic ensemble of interconnected processes. A process can be regarded as a worker process or a manager process. The responsibility of a worker process is to perform a (computational) task. The responsibility of a manager is to coordinate the communications among a set of worker processes. For this purpose, manager processes can create worker processes and make channel connections to their ports. A manager process may be considered a worker processes by another manager. At the bottom of this hierarchy there is always a layer of ***atomic workers***.

Manifold [Arb96, Arb] is a coordination language for writing program modules (coordinator processes) to manage complex, dynamically changing interconnections among sets of independent, concurrent, cooperating processes that comprise a single application. The conceptual model behind Manifold is based on IWIM. A Manifold application consists of a (potentially very large) number of processes running on a network of heterogeneous hosts, some of which may be parallel systems. Processes in the same application may be written in different programming languages and some of them may not know anything about Manifold, nor the fact that they are cooperating with other processes through Manifold in a concurrent application.

9.2.2 A Distributed Constraint Propagation Algorithm

In contrast to inherently sequential constraint propagation algorithms like Algorithm 2.1 on page 22, DICE implements the coordination-based chaotic iteration algorithm of [Mon00a]. In this algorithm, each CSP variable is represented by a process that maintains the domain of that variable. Also each domain reduction function is represented by a process that receives input from the processes corresponding to the CSP variables that the function applies to. Channel connections are made between the ports of Variable and DRF processes according to the structure of the CSP. The DRF processes have a buffer associated with each input port, which stores the domain last seen on that port. These buffers are initialized by having a Variable process send its domain each time a connection to a DRF process is made.

Figure 9.1(a) shows an example process network of this algorithm. Variable processes send ***reduction requests*** to DRF processes. Reduction requests contain the domain of the CSP variable. The DRF process uses this domain to update the buffer associated with the input port that delivers the reduction request. Then it applies the domain reduction function to the domains in the

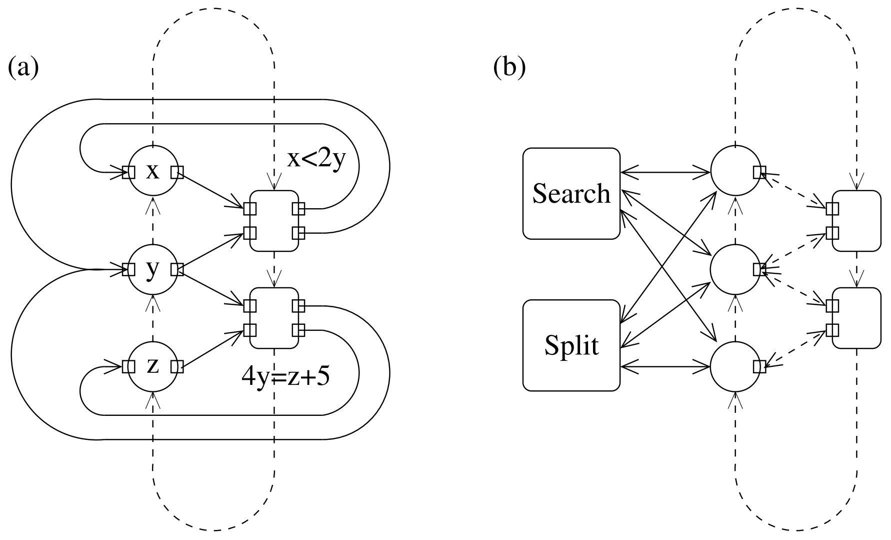


Figure 9.1: (a) An example process network of the distributed constraint propagation algorithm, (b) the propagation engine is coordinated from the outside to perform search

buffers[1]. This yields new domains for the output variables of the domain reduction function. These domains are dispatched through the output ports of the DRF process to the corresponding Variable processes as ***update commands***.

Upon receiving an update command, a Variable process computes the intersection of the domain held in the update command and the domain of the CSP variable held in its internal store. If this intersection is a proper subset of the current domain, the store is updated with the intersection, and the new domain of the CSP variable is dispatched through the output port of the Variable process as a reduction request. The reduction request is broadcast to all DRF processes that connect to this output port. If the intersection does not reduce the domain of the CSP variable, the update command has no effect.

In [Mon00a] it is argued that this distributed algorithm implements a restriction of the Generic Iteration Algorithm for Compound Domains of [Apt99]. This allows us to benefit from several properties that have been proven for that algorithm. One of these properties is that the algorithm is guaranteed to terminate if the domains are finite and the DRFs are inflationary. The latter is effectively ensured by having Variable processes compute intersections.

The elegance of the coordination-based algorithm is that neither the Variable processes nor the DRF processes have to know anything about the CSP that they are solving. The input and output schemes of DRFs, which are used in Algorithm 2.1 to maintain the set G of DRFs that still need to be applied, are

[1]In the implementation, application of the domain reduction function is postponed until no more reduction requests are immediately available on the input ports. Such details are omitted from this presentation.

laid down in the network of channels that conveys the domain updates. These schemes need only be accessed when a DRF is added to the network. However, the apparent simplicity of not having to maintain the set of pending DRFs comes at the cost of having to detect termination of a distributed computation.

9.2.3 Distributed Termination Detection

Although this is not strictly necessary, usually we do not want to consider expanding the search tree by branching on the domain of a variable before constraint propagation has finished. Therefore, we need to know when the propagation algorithm terminates. With a sequential algorithm, like Algorithm 2.1, this is easy: it terminates when the set of atomic reduction steps that still need to be applied becomes empty.

In the case of the distributed algorithm of the previous section, the conditions for concluding that constraint propagation has finished are more difficult to verify. The algorithm terminates when:

1. all Variable and DRF processes are idle, and

2. there are no pending update commands or reduction requests in the channels.

DICE employs the algorithm described in [Dij87] to detect these conditions. For the purpose of this algorithm, the processes of the constraint propagation algorithm are connected in a ring network. The dashed lines in Figure 9.1(a) show the extra channels for termination detection. All processes maintain a local counter of the number of update commands and reduction requests in the network. The ring network is used for circulating a token. This token is forwarded when the process that holds it becomes idle. When it returns to the process that created it, the token has accumulated the local counters of all process. Termination can be concluded only if this sum equals 0. Together with a black/white coloring of the processes and the token the algorithm ensures correctness in case of asynchronous channels. This corresponds to the Manifold communication model.

9.2.4 Search

DICE employs a scheme similar to that of [AM00], where the network of processes of the constraint propagation algorithm is performing work in several nodes of the search tree simultaneously. As a result, multiple tokens of the termination detection algorithm may be circulating on the ring network, one for every instance of the constraint propagation algorithm, and all administration inside the Variable and DRF processes for the purpose of the propagation and termination detection algorithms is per node of the search tree:

$$\begin{array}{rl}
\textit{Variable}:: & \\
v: \textit{World} \overset{m}{\rightarrow} \textit{Domain} & \\
\textit{color}: \textit{World} \overset{m}{\rightarrow} \{\text{black}, \text{white}\} & \\
\textit{n\_msg}: \textit{World} \overset{m}{\rightarrow} \mathbb{Z} &
\end{array}
\qquad
\begin{array}{rl}
\textit{DRF}:: & \\
I{:}\text{ARRAY } [1..n] \text{ OF } \textit{World} \overset{m}{\rightarrow} \textit{Domain} & \\
\textit{color}: \textit{World} \overset{m}{\rightarrow} \{\text{black}, \text{white}\} & \\
\textit{n\_msg}: \textit{World} \overset{m}{\rightarrow} \mathbb{Z} &
\end{array}$$

where n is the number of input ports of the DRF process, and *World* is a datatype whose elements serve as identifiers for nodes of the search tree. $A \overset{m}{\rightarrow} B$ denotes a ***map*** data structure, i.e., a set of tuples $\langle a, b \rangle \in A \times B$ in which every $a \in A$ occurs at most once. Maps v and $I[1], \ldots, I[n]$ hold the data for the propagation algorithm, and *color* and *n_msg* represent the state of the termination detection algorithm.

A partial order is defined on the elements of *World*, by which an ancestor node is ***compatible*** to its descendants, and a descendant is ***smaller*** than its parent. On several occasions, we look for information in the smallest compatible world of a world w. For example, the update commands of the propagation algorithm now consist of a world w, and a domain d. If the world w is not yet known to the Variable process, it intersects d with the domain d' of the CSP variable in the smallest compatible world of w. Only if $d' \cap d \subset d'$, the element $w \rightarrow d' \cap d$ is added to the map v of the Variable process.

The facilities offered by this administration per world are used by two new processes ***Split*** and ***Search***, which implement the branching strategy (involving variable selection and value selection) and traversal strategy, respectively. These processes have connections to all Variable processes (Figure 9.1(b)), and coordinate the network of the propagation algorithm to perform search.

The Split process is triggered when propagation finishes in a certain world, and may query Variable processes for their domains in that world. On the basis of this information, the Split process can then decide which variable to branch on (if any), and construct a set of new *World-Domain* pairs for that variable. The worlds of this set correspond to the subproblems created by the branching.

Upon receiving new worlds and corresponding domains from the Split process, a Variable process tells the Search process about these new worlds. This allows the Search process to maintain an administration of worlds where the constraint propagation algorithm still needs to be applied. The Search process coordinates the activities of the propagation network through the search tree, by issuing commands that start propagation in worlds that it knows about. In the current implementation of DICE, the Search process may consider starting propagation in a new world on two occasions: when propagation finishes in a certain world, and when a Variable process notifies it that new worlds have become available.

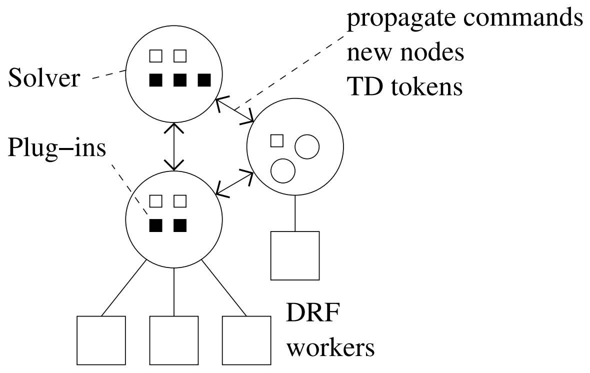


Figure 9.2: Example DICE network

9.3 Cooperating Solvers

The design of the previous section supports the cooperation of solvers on the level of reduction steps inside the branch-and-propagate search. Because of the very small grain size of the computational tasks that are typically performed in a reduction step, this has limited applicability. Therefore, in [Zoe03a], we proposed to adopt a more general scheme, which is comparable to that of [MR99], from a constraint propagation point of view. The basic process instance in this scheme is a ***solver***. A solver process can:

- maintain any number of variables,
- apply DRFs to variables that it maintains,
- branch on the domains of variables that it maintains in order to generate new nodes in the search tree, and
- start constraint propagation in nodes of the search tree that it knows about.

In Sections 9.3.1 and 9.3.2 we discuss some details and implications of this scheme, related to constraint propagation and search, respectively. In Section 9.3.3 we introduce DRF worker processes to support parallel search. Figure 9.2 shows an example network of solvers.

9.3.1 Grouping Variables and Reduction Operators

Solver processes can have a pair (input and output) of ports for each of the variables that they maintain. Channel connections can be made to these ports in order to connect variables in solver processes that correspond to the same CSP variable. Solvers modify the domains of CSP variables by computing a fixed point of their local DRFs. When a solver process modifies the domain of a variable that it maintains, and for this variable there exists an output port that has one or more channel connections, the new domain for that variable is sent through this output port as a ***propagate command***. Propagate commands are handled like the update commands and reduction requests of Section 9.2.2. Incoming propagate commands that reduce the domain of a CSP variable trigger the fixed point computation.

Compared to the design of Section 9.2.2, we can now combine several Variable processes into a single process. Also domain reduction functions that involve the corresponding CSP variables can be applied by this same process directly, without communication. Solvers can have local variables, for which no ports and channel connections exist. This way, communication takes place only for CSP variables that are shared by solvers. Based on the results reported in [MR99], we can expect that for sufficiently large problems, a partitioning of CSP variables and DRFs can be found for which efficiency can be gained from distributed execution.

The DRF processes of Section 9.2.2 are special cases of solvers that apply a single DRF. The Variable processes can be implemented as solvers that maintain a single CSP variable, and do not apply any DRFs. In the original design, Variable processes served to coordinate the activities of the DRF processes by forwarding update commands as reduction requests. In DICE, solvers can send each other updates of variable domains directly. Network topology is no longer centered around a set of processes whose main task is to forward updated variable domains. Failing to connect two solvers on a common CSP variable, however, may influence the level of consistency that is enforced by constraint propagation. Therefore we should provide a default topology that ensures the maximum level of consistency that can be achieved by the domain reduction functions.

9.3.2 Search by Cooperating Solvers

More than one branching strategy may be active in a network of solver processes. This has two potential applications:

- ***Complementary strategies*** that cooperate to implement a global strategy. Every solution occurs exactly once in the global search tree. The obvious example here is that different solvers branch on different variables.
- ***Competing strategies***, where there are different ways of arriving at the same solution. This can be useful when searching for a single solution. This is also sometimes called ***diversification***.

For complementary strategies, facilities must be provided that allow cooperating solvers to adhere to a common variable selection strategy. For example, if indeed we use a dedicated process for each variable, as in Section 9.2.2, it makes sense to let these processes control the branching of their variables. In that case, to implement fail-first, these processes must be able to determine among themselves which process holds the variable with the smallest domain. In the presence of competing strategies, in order to control the size of the search tree, we will probably want to prevent one strategy from splitting a subproblem generated by another strategy. For this purpose, nodes of the search tree generated according to different strategies must be distinguishable, and form separate subtrees of the global search tree.

New nodes that are generated inside a solver by application of a branching strategy can be handled in two ways:

- they can be stored locally, in a set of nodes that await constraint propagation, or
- they can be sent to another solver via dedicated ports and channels.

The purpose of the latter option is to allow one solver to coordinate the traversal of the search tree, in the case that more than one solver is able to generate new nodes. This solver then plays the role of the Search process of Section 9.2.4. Reports of new nodes created by another solver are treated by the receiving solver in the same way as new nodes created internally. If a node is labeled with the solver process and branching strategy that created it, it is always known what other solver needs to be instructed to start propagation in a particular node of the search tree.

Solver processes consult their traversal strategy plug-ins, if available, on two occasions: (1) when constraint propagation terminates in some node of the search tree, and (2) when new nodes become available in the solver (created internally, or reported by another solver). On these occasions propagation may be started in any node of the search tree that the solver knows about. A special instance of the termination detection algorithm is needed to detect termination of the global search, by counting the nodes of the search tree that await constraint propagation.

Compared to the scheme of Section 9.2.4, where the traversal is coordinated from outside the propagation network by the Search process, we now have the option to let this be handled by the solvers that are performing constraint propagation. From one point of view this can be regarded as mixing concerns that were separated in the IWIM design. From another point of view, it can be explained as a looser form of coordination: in principle propagation is performed as soon as a new node of the search tree becomes available. But to regulate the traffic in the network, the processes may choose to hold the messages in several nodes of the search tree, and release the messages in others, as bandwidth becomes available. Internally this is implemented by keeping a set of nodes that await propagation, and selecting nodes from this set according to a traversal strategy.

9.3.3 Parallel Search by Delegation

As a second extension to the design of Sections 9.2.2–9.2.4, solver processes are allowed to delegate the actual application of domain reduction functions and branching strategies (internally these have a common interface) to DRF worker processes in a master-slave fashion. Using this option, the solvers become IWIM managers themselves. The main reason for introducing this extra level of coordination is to provide support for parallel search. Constraint propagation may already be running for several nodes of the search tree simultaneously, but with only one process instance available for each solver, the network will be multiplexing the work in these different nodes, to a large extent. On the one hand, a pool of DRF workers increases the capacity of the network to actually handle the propagate commands in different nodes in parallel.

On the other hand, at the task granularity of a single reduction step, the communication overhead will generally outweigh the potential gain from exploiting parallelism, and there is little justification for doing this. This facility is useful only for computation-intensive reduction steps. The primary example would be a solver that autonomously explores a subtree, and splits the root node of this subtree into a set of nodes that contains a leaf node for every solution, plus several internal nodes for the part of the subtree that it has not yet explored.

When using DRF workers, for the purpose of the termination detection algorithm a solver is considered to be idle in a particular node of the search tree when (1) no commands (concerning any node) are immediately available on any of its input ports, (2) there is no need to compute the fixed point of the DRFs for that node, and (3) the solver is not expecting any results from DRF workers concerning that node of the search tree. Many options still exist for implementing this coordination pattern. In particular, we propose to use a pool of DRF workers per solver, and not to have DRF workers cache variable domains between two calls. This involves more communication than necessary, but this should not be a problem for the coordination of autonomous solvers, as suggested above.

9.4 Implementation

A full implementation of the DICE system, as described in Section 9.2, exists. It is implemented using Manifold, with atomic workers written in C++. The DICE implementation has a plug-in system comparable to that of OpenSolver, with only four categories: variable domain types, domain reduction functions, branching strategies, and traversal strategies. In the context of DICE, the plug-ins are called ***components***. A more detailed description can be found in [Zoe03b]. As an optimization, several DICE variables can be combined in a single process, and also an adapter-like domain reduction function exists that computes a common fixed point of a set of other DRFs. This gives some control over the amount of communication in the constraint propagation phase, but cannot prevent that data is exchanged for every node of the search tree. Given the size of the search tree even for problems that can be solved within a few seconds (see for example, Table 7.1 on page 162), this is still an obstacle for competitive performance.

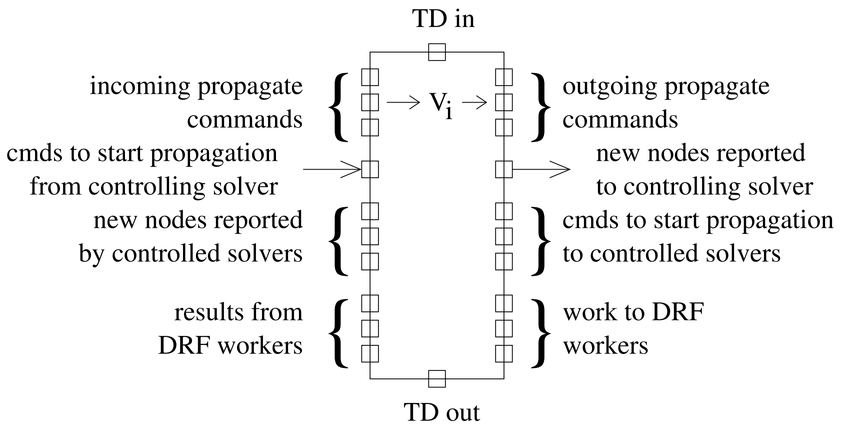

TD in
incoming propagate commands
V_i
outgoing propagate commands
cmds to start propagation from controlling solver
new nodes reported to controlling solver
new nodes reported by controlled solvers
cmds to start propagation to controlled solvers
results from DRF workers
work to DRF workers
TD out

Figure 9.3: Port footprint of a solver process

OpenSolver was intended to play the role of the cooperating solvers of Section 9.3. In this role it would be complemented with a coordination layer plug-in that offers the set of ports shown in Figure 9.3. These ports can then be connected by Manifold channels to form networks like that of Figure 9.2. This intended use has greatly influenced the design of OpenSolver, presented in Chapter 3. It explains several features that do not follow directly from the model of constraint solving of Section 2.3, notably

- the world database,
- maintaining a ***set*** of nodes that are subject to constraint propagation, and
- the coordination layer, and giving it responsibilities like the final confirmation that constraint propagation has terminated.

Although OpenSolver was designed to implement the cooperating solvers of the previous section, this system was never fully implemented, and some aspects of the design are still missing. The facilities described in Section 3.3.2 have been implemented only to the extent that nested search and parallel search are supported. Notably the commands **pending sends** and **update**, and the bookkeeping of variable changes for exporting modified domains have not been implemented. Furthermore, neither the set of commands of Section 3.3.2, nor the current implementation supports delegating the application of domain reduction functions to DRF worker processes, as proposed in Section 9.3.3.

Having said this, the design of OpenSolver does support an easy implementation of these features, and after implementing the distributed solver described in Section 9.2, we are confident that the design is well suited for distributed constraint propagation. The results in Chapter 8 also essentially prove the concept of parallel search by delegation, so technically, the distributed solver of the previous section is feasible, but a full implementation was not needed for our experiments.

9.5 Discussion

9.5.1 Benefits

From our experience with the DICE system we can conclude that the design of Sections 9.2.2–9.2.4 leads to an effective general-purpose distributed constraint solver. Given the efficiency of OpenSolver, which was demonstrated on several benchmark problems in the previous chapters, we can expect that the optimizations proposed in Section 9.3 lead to an efficient implementation of the system, allowing local processing where distribution is not required. Nevertheless, we should be careful in the assessment of the benefits of our approach. We discuss these on the basis of the following aspects: the role of coordination, the need for distributed constraint solving, and the contribution of a system that combines parallel and distributed processing.

The Role of Coordination

As we discussed in Section 9.1, constraint propagation through iteration of domain reduction functions can be seen as a form of exogenous coordination. This is made explicit in the IWIM design of the constraint propagation algorithm of [Mon00a], which we used in Section 9.2.2, where the processes that apply the DRFs are coordinated by a layer of processes corresponding to the variables of a CSP. Both kinds of processes are atomic workers in the sense that they are largely unaware of the environment that they operate in. The computation that they perform is determined by the network of channels that connects these processes. In this sense, the algorithm can be characterized as a coordination-based algorithm, but in our opinion, it is primarily a ***distributed*** algorithm. Its conformance to the IWIM coordination model is an advantage only in comparison with other distributed algorithms. For example in a distributed algorithm that relies on message passing, the processes would need to know the identification of the processes that they communicate with. Compared to such algorithms, the coordination-based approach is more elegant and flexible, and can more easily be verified to be correct.

Since we use a distributed algorithm for constraint propagation, we have to deal with distributed termination detection, and coordination models and languages could also be of importance here. In the current model and implementation, the automatic replication of messages through ports with multiple outgoing channels implies that for counting the number of messages in the network, which is inherent to many termination detection algorithms, the variables need to know how many DRFs they are connected to. This runs counter to the inherent simplicity of the constraint propagation algorithm proper. For example, we now have to temporarily halt a Variable processes before making or removing a channel connection to its output port, in order to correctly update the counter of its outgoing channels before it sends any new messages. Because termination detection is such a common problem, in our opinion, primitives for it would be valuable additions to a coordination language like Manifold.

Even without support for termination detection, using a proper coordination language like Manifold is convenient because it has rich facilities for process management, and for channel-based communication. However, it does not simplify composing constraint solvers from software components, as we anticipated at the beginning of this chapter. This has two reasons.

- As we discussed in Section 3.3.3, from a coordination point of view, a software component typically has its own thread of control and interacts with its environment through a set of ports. This model does not fit the majority of constraint solver building blocks that we categorized in Chapter 3, and wrapping them up as autonomous processes is artificial.

  Conversely, in those cases where we want to use an autonomous solver that fits the model of an IWIM process, a component-based framework like OpenSolver can communicate with it through a ***proxy***. Wrapping up autonomous solvers as objects through proxies is equally artificial, but at least it entails that the appropriate method of software composition is used for the majority of units of composition. In other words, do not distribute everything because in a few cases, this makes software composition easier.

- While some specific instances of these building blocks are of a complexity that calls for code re-use, most of them entail very simple data structures and algorithms. The effort of implementing a configurable constraint solver is in the definition of their interfaces, rather than in writing the code for these data structures and algorithms. Conformance to the IWIM model, or the use of a coordination language does not simplify this task.

In summary, if for some reason it is necessary to perform distributed constraint propagation, the algorithm of [Mon00a] provides an effective solution. Because of its conformance to the IWIM coordination model it has specific advantages over other solutions to distributed constraint propagation, and it is convenient to use a coordination language like Manifold to implement it.

The Need for Distributed Constraint Propagation

Because communication is involved with domain updates, distributed constraint propagation is communication-intensive. In general, communication will outweigh computation, and we can expect to be able to apply it efficiently only in the following cases:

- In case of very computation-intensive reduction operators we may achieve a reduction of turn-around time if the cooperating solvers run on parallel processors. The operator of Section 7.3.1 easily involves seconds of computation time per application, and would be a good example. For our application we need precisely one instance, but there probably exist situations where it would make sense to use multiple operators of comparable functionality.

- If the problem itself is distributed, while it is impossible or undesirable to gather all information in a single location, distributed constraint propagation is unavoidable if we want to perform a branch-and-propagate tree search.

In the first case, if constraint propagation is performed as a part of branch-and-propagate tree search, it will likely be more efficient yet to parallelize the search instead of the propagation. The latter case is known as the ***distributed constraint satisfaction problem*** (DisCSP). Our approach is definitely suitable for DisCSP solving.

Combining Parallel and Distributed Constraint Solving

The extensions proposed in Section 9.3.1 are a valuable addition to the design of Sections 9.2.2–9.2.4. It allows us to group variables and DRFs such that communication is limited to what is absolutely necessary for a given DisCSP situation. In addition, in Section 9.3.3, we proposed to increase the capacity of the constraint propagation network by delegating the actual application of domain reduction functions to DRF worker processes, with the particular goal to support parallel search. In retrospect, it does not seem likely that parallel search and distributed propagation need ever be combined, and there is no need for a system that supports both. The coordination-based approach of Chapter 8 is much simpler and more flexible, and the design of OpenSolver is complicated enough without the facilities for delegation.

Another justification for the delegation mechanism would be that it prevents that time-consuming constraint propagation in one node of the search tree prevents progress of the search in other subtrees, that are being explored simultaneously. In our opinion, the option to have an OpenSolver scheduler plug-in return control before a fixed point has been computed, as discussed in Section 3.2.3, is a better solution.

Finally, since we want to prevent that a branching strategy is applied to a node that is generated by another branching strategy, the competing branching strategies of Section 9.3.2 can also be implemented by having multiple solvers running concurrently. For optimization purposes, we can use the time-out mechanism of Chapter 8, and share bounds implied by new suboptimal solutions between multiple instances of loop networks as shown in Figure 8.6. If the number of available processors is smaller than the number of competing strategies, these loops should then contain only one solver. For competing strategies, the advantage of the interrupt mechanism discussed in Section 8.6 over the time-out mechanism is even greater, because it allows that new bounds are taken into account immediately, without having to wait for the elapse of a time-out.

9.5.2 Related Work

Above we argued that we should not distribute a constraint propagation algorithm based on generic iteration, just to provide for the case that one of the DRFs that we want to apply happens to be an autonomous process. BALI [Mon00b] is a system that supports exactly this mode of solver cooperation. Fixed point computation is just one of the cooperation patterns of BALI, and likely, BALI itself could be implemented as an instance of a (distributed) generic iteration algorithm. From this perspective, it is all a matter of selecting the right method of software composition for the units that we want to combine. A Manifold implementation of BALI is investigated in [AM98].

In our opinion, the coordination-based approach is beneficial only in comparison to other distributed constraint solvers. In particular, the distributed constraint propagation algorithm of [MR99] is mentioned in [Mon00a].

In branch-and-propagate tree search, even in the case that different parts of the search tree are explored in parallel, the expansion of the search tree by branching and the selection of the nodes for further exploration are inherently synchronous and sequential operations. In contrast, the ***asynchronous backtracking*** algorithm distributes the search itself. Different ***agents***, each maintaining their own variable, propose values for these variables to other agents, with whom they share a constraint. By exchanging such proposals and no-goods, the agents will eventually find a solution if it exists. An overview of asynchronous backtracking and related algorithms is given in [Yok01]. These algorithms still rely on distributed termination detection, but now it has to be established only once, to detect that consensus has been reached. The Disolver system [Ham05] is reported to support this kind of distributed search. It would be interesting to compare the performance of both approaches, asynchronous backtracking and that of Sections 9.2.2–9.2.4 with respect to communication, CPU time, turnaround time, and memory usage. To our knowledge, such a comparison has not been made.

Diversification, i.e., competing search strategies, in the context of distributed search are discussed in [RH05]. A variant of the DisCSP problem where constraints can also be retracted is known as the ***dynamic*** distributed constraint satisfaction problem (DynDCSP). An algorithm for maintaining arc consistency for this class of problems is given in [Rin05]. OpenSolver cannot deal with constraint retraction, and our approach to constraint solving in general is unsuited for solving of dynamic CSPs.

A different approach to exploiting parallelism in the constraint propagation phase is reported in [GH00]. This approach involves an alternative iteration algorithm that applies the following steps until a fixed point of all reduction operators is reached.

1. Apply all pending reduction operators independently of each other to evaluate the amount of reduction they achieve on the current set of domains.

2. Compute a fixed point only of those operators that achieve the maximal reduction.

The first of these two steps can easily be parallelized because the operators are all applied independently of each other, on the same set of domains. The second step is computed sequentially, and constitutes a bottleneck for parallel performance. As a result, only modest speedup figures are obtained beyond running the alternative iteration algorithm on a single processor.

Finally, we would like to mention that ***concurrent constraint programming*** (CCP, see, e.g.,[SR90]), is a model of concurrent computation, where processes or agents interact by communicating with a shared constraint store. The communication with the store consists of ***Ask*** and ***Tell*** operations. Via the Ask operation, an agent inquires whether a constraint is entailed by the store or not. Via the Tell operator constraints are added to the store. Although distributed computing introduces concurrency, the algorithms discussed in this chapter are unrelated to CCP. We expect, however, that they can be used to realize a distributed implementation of a CCP constraint store, if needed. The multi-paradigm programming language Oz (see, e.g., [VRBD$^+$03]) incorporates the CCP model. Its main implementation, Mozart (see `http://www.mozart-oz.org/`), is an advanced platform for the development of distributed applications.

9.6 Summary

In this chapter we presented the DICE framework for distributed constraint solving, and discussed an optimization of it, which entails combining the functionality of several processes in order to limit communication overhead. Through a special coordination-layer plug-in, OpenSolver can be configured to implement this optimization. The resulting system is well suited for solving the DisCSP problem, and conformance to the IWIM model makes it a very flexible solution.

We also considered adapting OpenSolver/DICE for parallel search and competing search strategies. We argued that these are better dealt with according to the component-based approach of Chapter 8. In retrospect, we should not modify a software component to support functionality that can also be achieved by exogenous coordination of these components.

Chapter 10

Conclusions

In this thesis we have explored the possibilities for composing branch-and-propagate constraint solvers from a set of solver "building blocks." We have taken a practical approach, and implemented OpenSolver, a configurable constraint solving engine that supports a wide range of relevant solver configurations. Performance comparison with state-of-the-art solvers, some of them commercially available systems, demonstrates that the approach leads to realistic and efficient solvers.

In this chapter we review our approach. In Section 10.1 we return to forms of solver composition identified in Section 2.4, and evaluate what has been achieved. In Section 10.2 we compare our approach with other systems, this time taking a broader view than in Section 3.4.2, were we considered only object-oriented frameworks. In Section 10.3 we discuss the limitations of our approach, and identify directions for further research and development. In Section 10.4 we summarize the contributions of our work.

10.1 Composing Constraint Solvers

Propagation

All forms of composition related to constraint propagation that we identified in Sections 2.4.1 and 2.4.2 are supported by OpenSolver. In particular:

- OpenSolver allows that a separate reduction operator is used for each inversion of an (atomic) arithmetic constraint. We can then define a schedule for the operator-based scheduler that respects the hierarchical dependencies between these rules, following from the decomposition of general arithmetic constraints. This way we do not need heavy-weight operators like HC4. Instead, comparable functionality can be composed from the facilities for a decomposition-based approach, and a scheduler.

- Instead of implementing reduction operators for "stronger" forms of local consistency, such as box consistency and shaving, we compose these from the ingredients of the weaker consistency notions plus an operator for nested search. The operator-based scheduler allows for the implementation of priority schemes that postpone the application of computation-intensive operators until no more "cheap" reductions can be made. It does not support the dynamic schemes suggested in [SS04], but OpenSolver itself does not prevent their implementation.

- Because for every variable, a different domain-type plug-in can be used, hybrid solvers are naturally supported.

Search

OpenSolver has rich facilities for the composition of search strategies, as described in Section 2.4.3. We have seen the following examples.

- In Section 4.3 we composed a best-first search strategy according to Warnsdorf's heuristic for solving the knight's tour problem from a container, a selector, and an annotation scheme, all having wider applicability than this particular heuristic.

- In Section 6.4 we discussed how the composition of branching strategies, as supported by the SALSA language, can be implemented in OpenSolver.

In retrospect, the proposed mechanism for composing branching strategies could also have been applied to the memory-bounded LDS traversal strategy of Section 4.1.2. We did not exploit the full potential of the adapter mechanism here, and implemented a dedicated container plug-in. Instead, we could have composed it as suggested in Section 2.4.3.

OpenSolver also supports multiple search probes, to simulate parallel search, as suggested in that same section. Compared to schemes like interleaved depth-first search it has the additional benefit that it supports the possibility to abort constraint propagation before a fixed point has been reached. This would prevent that slow convergence in one node of the search tree blocks progress in other nodes. We have not performed experiments in this area, though. Another option would be to actually run a parallel solver on a single processor. In this case the multiplexing between the search probes is taken care of by the operating system, at the cost of unnecessary inter-process communication. In this case load-balancing is not an issue, and the experiments in Chapter 8 suggest that for the benchmark problems used in that chapter, the communication overhead would range from 4% to 16%, depending on the size of the solver configurations that are being communicated.

Solver Cooperation

OpenSolver is not intended as a framework for solver cooperation. It does not support generic solver cooperation schemes, as do BALI [AM98] and the system of [HSG01].

Other solvers can be embedded in OpenSolver, but this should be the case for all object-oriented frameworks that can be extended with user constraints. Conversely, OpenSolver can also be embedded in other solvers, and we have seen an example of this in Chapter 7. Because the coordination-layer mechanism gives fine-grained control over the solving process, we expect that OpenSolver is more versatile than other solvers in this respect.

For optimization purposes, the coordination layer mechanism makes it easy to incorporate new bounds that are calculated by external solvers. Also the configuration language makes OpenSolver particularly well suited for cooperation with symbolic solvers that are applied as a preprocessing step. In fact, this is how we implemented the decomposition of arithmetic constraints into simple and atomic constraints for our experiments in Chapter 5. We expect that the ability to publish the search frontier, which is also the basis for parallel search in Chapter 8, opens up new possibilities for cooperation with symbolic solvers ***during*** the solving process, instead of only as a preprocessing step. Despite the flexibility offered by the coordination layer mechanism, we think that for general cooperation schemes, a proper API would be a valuable addition.

Composition of Parallel and Distributed Solvers

The coordination layer mechanism was specifically designed to support distributed constraint propagation and parallel search, and OpenSolver is well suited for these forms of solver composition. While we did not experiment with distributed constraint propagation after we moved from DICE to OpenSolver (see Section 3.1 and Chapter 9), the design of OpenSolver supports the basic ingredients of distributed constraint propagation algorithms: distributed termination detection, and the communication of domain updates.

For parallel search, we use a time-out mechanism to achieve an implicit load balancing. This gives a very simple implementation of parallel search, but it leads to slightly less user-friendly systems because a tuning factor is introduced. Especially because our experiments indicated that performance is not overly sensitive to the actual time-out value that is used, we expect that it can be tuned automatically, based on heuristics.

For communicating sub-problems, our parallel solver uses the OpenSolver configuration language. We expect this to have wider applicability than parallel search, and that it increases the value of OpenSolver as a software component. We already mentioned the potential for pre-search and in-search transformations of CSPs and solver configurations. In addition, as we discussed in Section 8.6, it facilitates a checkpoint mechanism, where the search frontier is saved to disk so that a lengthy search process may survive a power failure or system crash. We use the textual format for steering the parallel search, where depending on the level annotation of a subproblem, a breadth-first or depth-first search is performed. We expect that in addition to this automated steering, there are also possibilities for interaction by users. For example, based on an inspection of the nodes, a user might decide to explore a particular subtree first, or to change the branching strategy for some areas of the search tree. The ability to publish the search frontier is essential for all these applications. The textual format of our configuration language increases the possibilities for external processing of the published nodes.

Returning to the subject of distributed constraint solving, OpenSolver is unsuitable for distributed search by asynchronous backtracking, or related algorithms proposed by Yokoo [Yok01]. These algorithms form an alternative approach, where the processes, or agents that are involved asynchronously propose value assignments to their peers with whom they share constraints. These algorithms rely less on termination detection than ours, and hence allow for greater autonomy of agents. It is unclear how the two approaches compare with respect to efficiency and flexibility.

10.2 Comparison with Other Systems

OpenSolver is an object-oriented ***framework***, aiming at reuse of the basic solver design. Through its coordination layer mechanism it can be deployed as a software component in many environments. In Section 3.4.2 we already compared OpenSolver with other component-based toolkits and frameworks, namely ILOG Solver, Koalog Constraint Solver, Elisa, Disolver, Figaro, EasyLocal++, and Localizer++.

In this section we compare it further with two other types of constraint solving environments: logic programming systems, and modeling languages. Both kinds of systems provide ***declarative*** languages, that shield the user from implementation details of the computational model that they are based on. In the category of logic programming systems we consider the ECL^{i}PS^e [WNS97] system. In the category of modeling languages we consider OPL [VH99] and COMET [MVH02a].

ECL^{i}PS^e is a logic programming system with extensive libraries for constraint solving on various domain types using many different techniques. It supports tree search and local search, and interfaces with CPLEX, an industrial solver for linear and mixed-integer programs. Furthermore it has interfaces for calling routines written in other programming languages, and can also be embedded in programs written in those languages.

OPL is a language for modeling combinatorial problems, and for specifying search strategies. The modeling facilities include high-level abstractions such as arrays, allowing for very compact and readable specifications that are close to the combinatorial problems that are being modeled. Compared to their implementation in a main-stream programming language the search abstractions are also at a very high level. OPL has dedicated facilities for specifying scheduling and resource allocation problems.

Comet is an object-oriented language for constraint-based local search. The modeling facilities are comparable to those of OPL. The facilities for specifying local search are comparable to those that are made available in a C++ environment through Localizer++ [MVH01].

From a certain perspective, the designs of ECL^{i}PS^e and OpenSolver are opposites. Both systems mean to provide a platform for composing constraint solvers by arbitrarily combining different solving techniques, but while OpenSolver is part of a toolbox of loosely connected building blocks, ECL^{i}PS^e provides an all-encompassing development environment. OpenSolver plug-ins are written in C++, and solver composition is a matter of black-box composition through a configuration language. A full constraint solver would then complete the OpenSolver core with an external user interface that offers modeling facilities. Conversely, in ECL^{i}PS^e, everything is done in the same language, which has a Prolog-like syntax. The basic solving algorithms such as branching strategies, the composite solver, and the problem specification are all coded in this language. A library mechanism is available to hide implementation details from users of facilities coded in ECL^{i}PS^e.

In our opinion, ECL^{i}PS^e and OpenSolver are both good approaches to composing constraint solvers. For users who are not acquainted with logic programming, the ECL^{i}PS^e user interface may have a steep learning curve. OpenSolver, on the other hand, is not intended as a full development environment, and can be coupled with a graphical "plug-and-play" user interface, or a modeling language, depending on the intended use. Being a configurable search engine rather than a development environment, OpenSolver allows for composition at a lower algorithmic level. For example, an algorithm like HC4 would be hard-wired in ECL^{i}PS^e. We could also introduce a dedicated OpenSolver plug-in for it, but a comparably efficient way to compute hull consistency can be composed from the facilities for decomposition-based hull consistency and a schedule for the operator-based scheduler. ECL^{i}PS^e does support parallel search, but being a closed system, we would not have been able to perform our experiments with the time-out mechanism with it.

As a dialect of Prolog, ECL^{i}PS^e is also a full programming language. In contrast, languages like OPL and COMET are suitable only for specifying combinatorial problems and solving strategies, and provide high-level abstractions for doing so. What is achieved in EasyLocal++ by providing C++ subclasses, can be accomplished directly by programming in COMET. OPL, which is targeted at tree search instead of local search, is at a comparably high level of abstraction. As we discussed in Section 6.4, at this level of abstraction the OpenSolver configuration language should be seen as an ***assembly language***. The rich facilities for specifying search in modeling languages can be implemented only if several assumptions about the variable domain types are made. For this reason, such systems depend on a fixed set of built-in data types. In OpenSolver, almost nothing is fixed. For a coherent set of plug-ins, though, the same facilities could probably be implemented as a compilation step, translating a higher level language to configuration specifications for OpenSolver, which is then used as a constraint solving engine.

10.3 Perspectives

In this section we identify some areas for further research and development related to OpenSolver, and composing constraint solvers in general.

In the first place, only a basic set of constraints has been implemented for OpenSolver. For example, global constraints such as the all_different constraint are missing, and mathematical modeling is currently limited to arithmetic constraints. Therefore the range of problems that can currently be solved is limited, especially compared to the commercial systems. This is not a fundamental limitation, though.

Also a modeling environment with a proper user interface is currently missing. For many experiments we used some peripheral programs to generate the configuration file for a problem instance of given dimensions. For other types of problems we wrote converters from standard file formats such as those used in the DIMACS test sets. Instead of these ad-hoc solutions, we would like to have a modeling layer that supports more abstract problem specifications than the OpenSolver configuration language. Our current program for rewriting arithmetic constraints could be used as a basis for this layer. It could well be coupled with a graphical user interface that offers menus from which the user can configure the solver for a given problem.

In Section 4.4 we concluded that the copying state restoration policy of OpenSolver impedes realizing efficient SAT solving schemes. The reason is that such schemes rely on non-chronological backtracking, which in turn is naturally combined with a trailing state restoration policy. It would be interesting to investigate to what extent this policy can be made configurable, like in Figaro [HMN99]. We expect that because this policy is closely related to the implementation of the variable domains, which is not fixed in OpenSolver, this is not straightforward. Another avenue would be to investigate the implementation of non-chronological backtracking on the basis of hierarchical information stored in node annotations.

From an implementation point of view, the following areas need attention:

- A problem with the current annotation mechanism is that two techniques may have conflicting uses for the annotation field. For example, best-first search, as discussed in Section 4.3 is now incompatible with the round-robin variable selection strategy, discussed in Section 4.1.2. This can be solved using the adapter mechanism, where a special annotation plug-in implements an array of annotations, and other adapters modify the plug-ins that access the annotations to use a specific element of this array. It would be better not to restrict the annotation mechanism to a single slot in the first place. Probably name/value pairs would be a good solution.

- The facility to break out of constraint propagation, and resume the fixed point computation has not been implemented in the operator-based scheduler. We have not experimented with multiple exploration "probes" as suggested at the end of Section 3.2.3.

Finally, we are convinced that we have not exploited the full potential of the combined features of being able to publish the search frontier, and using a textual format for the generated subproblems. As we already indicated in Section 10.1 this has wider applicability than facilitating parallel search, and should allow for forms of solver cooperation that would otherwise require that one solver is embedded in another.

10.4 Summary

Composing constraint solvers based on tree search and constraint propagation through generic iteration leads to efficient and flexible constraint solvers. This was demonstrated using OpenSolver, an abstract branch-and-propagate tree search engine that supports a wide range of relevant solver configurations. We gave an account of the design and implementation, and of many experiments that were performed to evaluate the approach.

The efficiency of OpenSolver-based constraint solvers compares well to that of existing solvers, some of which are successful commercial products. Yet, the following combination of features gives OpenSolver some unique advantages over each of the other systems that we considered.

- OpenSolver is a branch-and-propagate constraint solving ***engine***, and makes configurable those aspects that are hardwired in many other systems, such as the constraint propagation algorithm and implementation of the data types. Modeling languages can be implemented on top of it, by means of a compilation step.

- OpenSolver promotes the composition of complex strategies from atomic solver "building blocks," implemented as ***plug-ins***. This prevents duplicate solver code, and allows that techniques carry over to other data types and application domains.

- While it is a white-box framework from the perspective of writing new plug-ins for it, OpenSolver aims at black-box composition of solvers. This led to an inherently linguistic approach where solvers are composed through ***scripts*** in a simple configuration language.

- OpenSolver is designed as a stand-alone application. Compared to libraries for constraint solving, this makes it independent of a particular programming language. Composing constraint solvers around OpenSolver is primarily a matter of exogenous coordination and component-based software engineering. This is opposite to the approach taken by logic programming systems, which provide a full development environment.

- Because of the coordination layer mechanism, OpenSolver can be adapted to many different environments, and the solving process can be controlled externally. In particular, it is suited for distributed constraint solving.

- In combination with the configuration language, the coordination layer mechanism opens up new possibilities for implementing parallel search, in-search transformation of CSPs and solver configurations, and it allows for checkpoint mechanisms.

Thanks to the flexibility of the system, we could further achieve the following results related to the specific research questions addressed in some of the individual chapters.

- We demonstrated how a number of techniques that are normally hard-wired in solvers can be realized through composition.

- We performed a comparative study of several approaches to implementing arithmetic constraints on variables with integer interval domains. The best performance was observed for decomposition and hierarchical scheduling of the reduction operators. For this approach we characterized in part the effect of constraint propagation.

- We demonstrated the technique of constraining special purpose domain types.

- We demonstrated that several pruning techniques from various application domains can be expressed and implemented as applications of a generic operator for nested search.

- We evaluated a novel time-out mechanism for load balancing in parallel search, and demonstrated that it can lead to efficient and scalable parallel constraint solvers.

Appendix A

Proofs

We provide here the proofs of the Bounds consistency Theorems 5.8.2 and 5.8.3, and the *MULTIPLICATION* Theorem 5.9.1. This material is taken from unpublished joint work with Krzysztof Apt.

Proof of the Bounds consistency Theorem 5.8.2.
Let $\phi := \langle x \cdot y = z \; ; \; x \in D_x, y \in D_y, z \in D_z \rangle$. Call a variable u of ϕ ***bounds consistent*** if the bounds of its domain satisfy the condition of the bounds consistency (see Definition 5.8.1).

Given an integer interval $[l..h]$ denote by $\overline{[l..h]}$ the corresponding real interval $[l, h]$. Suppose that $D_x = [l_x..h_x], D_y = [l_y..h_y], D_z = [l_z..h_z]$. To show that ϕ is closed under the applications of the *MULTIPLICATION 1* rule it suffices to prove that

$$\{l_z, h_z\} \subseteq int(D_x \cdot D_y). \tag{A.1}$$

So take $c \in \{l_z, h_z\}$. By the bounds consistency of z we have $c = a \cdot b$ for some $a \in \overline{D_x}$ and $b \in \overline{D_y}$. Since D_x and D_y are integer intervals we have $\lfloor a \rfloor, \lceil a \rceil \in D_x$ and $\lfloor b \rfloor, \lceil b \rceil \in D_y$. To prove (A.1), by the definition of $D_x \cdot D_y$, we need to find $a_1, a_2 \in D_x$ and $b_1, b_2 \in D_y$ such that

$$a_1 \cdot b_1 \leq c \leq a_2 \cdot b_2.$$

The choice of a_1, a_2, b_1 and b_2 depends on the sign of a and of b and is provided in the following table:

condition	a_1	b_1	a_2	b_2
$a = 0$	a	$\lfloor b \rfloor$	a	$\lfloor b \rfloor$
$b = 0$	$\lfloor a \rfloor$	b	$\lfloor a \rfloor$	b
$a > 0, b > 0$	$\lfloor a \rfloor$	$\lfloor b \rfloor$	$\lceil a \rceil$	$\lceil b \rceil$
$a > 0, b < 0$	$\lceil a \rceil$	$\lfloor b \rfloor$	$\lfloor a \rfloor$	$\lceil b \rceil$
$a < 0, b > 0$	$\lfloor a \rfloor$	$\lceil b \rceil$	$\lceil a \rceil$	$\lfloor b \rfloor$
$a < 0, b < 0$	$\lceil a \rceil$	$\lceil b \rceil$	$\lfloor a \rfloor$	$\lfloor b \rfloor$

To prove that ϕ is closed under the applications of the *MULTIPLICATION 2* and *3* rules it suffices to prove

$$\{l_x, h_x\} \subseteq int(D_z/D_y) \text{ and } \{l_y, h_y\} \subseteq int(D_z/D_x). \tag{A.2}$$

We need to distinguish a number of cases. The case analysis depends on the position of 0 w.r.t. each of the intervals D_x and D_y. This leads to 9 cases, which by symmetry between x and y can be reduced to 6 cases. We present here the proofs for representative 3 cases.

Case 1. $l_x \geq 0,\ l_y \geq 0$.

By the bounds consistency of x for some $b \in [l_y, h_y]$ we have $l_x \cdot b \in [l_z, h_z]$. Then $b \leq h_y$ and $l_x \geq 0$, so $l_x \cdot b \leq l_x \cdot h_y$. Also $l_z \leq l_x \cdot b$, so

$$l_z \leq l_x \cdot h_y.$$

Next, by the bounds consistency of y for some $a \in [l_x, h_x]$ we have $a \cdot h_y \in [l_z, h_z]$. Then $l_x \leq a$ and $h_y \geq 0$, so $l_x \cdot h_y \leq a \cdot h_y$. Also $a \cdot h_y \leq h_z$, so

$$l_x \cdot h_y \leq h_z.$$

So $l_x \cdot h_y \in [l_z..h_z]$ and consequently by the definition of the integer intervals division

$$l_x \in D_z/D_y \text{ and } h_y \in D_z/D_x.$$

By a symmetric argument

$$h_x \in D_z/D_y \text{ and } l_y \in D_z/D_x.$$

Case 2. $l_x \geq 0,\ h_y \leq 0$.

By the bounds consistency of x for some $b \in [l_y, h_y]$ we have $h_x \cdot b \in [l_z, h_z]$. Then $b \leq h_y$ and $h_x \geq 0$, so $h_x \cdot b \leq h_x \cdot h_y$. Also $l_z \leq h_x \cdot b$, so

$$l_z \leq h_x \cdot h_y.$$

Next, by the bounds consistency of y for some $a \in [l_x, h_x]$ we have $a \cdot h_y \in [l_z, h_z]$. Then $a \leq h_x$ and $h_y \leq 0$, so $a \cdot h_y \geq h_x \cdot h_y$. Also $h_z \geq a \cdot h_y$, so

$$h_x \cdot h_y \leq h_z.$$

So $h_x \cdot h_y \in [l_z..h_z]$ and consequently by the definition of the integer intervals division

$$h_x \in D_z/D_y \text{ and } h_y \in D_z/D_x.$$

Further, by the bounds consistency of x for some $b \in [l_y, h_y]$ we have $l_x \cdot b \in [l_z, h_z]$. Then $l_y \leq b$ and $l_x \geq 0$, so $l_x \cdot l_y \leq l_x \cdot b$. Also $l_x \cdot b \leq h_z$, so

$$l_x \cdot l_y \leq h_z.$$

Next, by the bounds consistency of y for some $a \in [l_x, h_x]$ we have $a \cdot l_y \in [l_z, h_z]$. Then $l_x \leq a$ and $l_y < 0$, so $l_x \cdot l_y \geq a \cdot l_y$. Also $a \cdot l_y \geq l_z$, so

$$l_z \leq l_x \cdot l_y.$$

So $l_x \cdot l_y \in [l_z..h_z]$ and consequently by the definition of the integer intervals division

$$l_x \in D_z/D_y \text{ and } l_y \in D_z/D_x.$$

Case 3. $l_x < 0 < h_x,\ l_y \geq 0$.

The proof for this case is somewhat more elaborate. By the bounds consistency of x for some $b \in [l_y, h_y]$ we have $l_x \cdot b \in [l_z, h_z]$. Then $l_y \leq b$ and $l_x < 0$, so $l_x \cdot l_y \geq l_x \cdot b$. But also $l_x \cdot b \geq l_z$, so

$$l_z \leq l_x \cdot l_y.$$

Next, by the bounds consistency of y for some $a \in [l_x, h_x]$ we have $a \cdot l_y \in [l_z, h_z]$. Then $l_x \leq a$ and $l_y \geq 0$, so $l_x \cdot l_y \leq a \cdot l_y$. But also $a \cdot l_y \leq h_z$, so

$$l_x \cdot l_y \leq h_z.$$

So $l_x \cdot l_y \in [l_z..h_z]$ and consequently by the definition of the integer intervals division

$$l_x \in D_z/D_y \text{ and } l_y \in D_z/D_x.$$

Further, by the bounds consistency of x for some $b \in [l_y, h_y]$ we have $h_x \cdot b \in [l_z, h_z]$. Then $l_y \leq b$ and $h_x > 0$, so $h_x \cdot l_y \leq h_x \cdot b$. But also $h_x \cdot b \leq h_z$, so

$$h_x \cdot l_y \leq h_z.$$

Next, we already noted that by the bounds consistency of y for some $a \in [l_x, h_x]$ we have $a \cdot l_y \in [l_z, h_z]$. Then $a \leq h_x$ and $l_y \geq 0$, so $a \cdot l_y \leq h_x \cdot l_y$. But also $l_z \leq a \cdot l_y$, so

$$l_z \leq h_x \cdot l_y.$$

So $h_x \cdot l_y \in [l_z..h_z]$ and consequently by the definition of the integer intervals division

$$h_x \in D_z/D_y.$$

It remains to prove that $h_y \in D_z/D_x$. We showed already $l_x \cdot l_y \leq h_z$. Moreover, $l_x < 0$ and $l_y \leq h_y$, so $l_x \cdot h_y \leq l_x \cdot l_y$ and hence

$$l_x \cdot h_y \leq h_z.$$

Also we showed already $l_z \leq h_x \cdot l_y$. Moreover $h_x > 0$ and $l_y \leq h_y$, so $h_x \cdot l_y \leq h_x \cdot h_y$ and hence

$$l_z \leq h_x \cdot h_y.$$

So if either $l_z \leq l_x \cdot h_y$ or $h_x \cdot h_y \leq h_z$, then either $l_x \cdot h_y \in [l_z..h_z]$ or $h_x \cdot h_y \in [l_z..h_z]$ and consequently $h_y \in D_z/D_x$.

If both $l_x \cdot h_y < l_z$ and $h_z < h_x \cdot h_y$, then

$$[l_z..h_z] \subseteq [l_x..h_x] \cdot h_y.$$

In particular for some $a \in D_x$ we have $l_z = a \cdot h_y$, so $h_y \in D_z/D_x$, as well.

This concludes the proof for this case. □

Proof of the Bounds consistency Theorem 5.8.3.
We consider each variable in turn. We begin with x. Suppose that $D_x = [l_x..h_x]$. ϕ is closed under the applications of the *MULTIPLICATION 2* rule, so

$$\{l_x, h_x\} \subseteq int(D_z/D_y). \tag{A.3}$$

To show the bounds consistency of x amounts to showing

$$\{l_x, h_x\} \subseteq \overline{D_z} \oslash \overline{D_y}. \tag{A.4}$$

(Recall that given real intervals X and Y we denote by $X \oslash Y$ their division, defined in Section 5.3.)

Case 1. $int(D_z/D_y) = \mathbb{Z}$.

This implies that $0 \in D_z \cap D_y$, so by the definition of real intervals division $\overline{D_z} \oslash \overline{D_y} = \langle -\infty, \infty \rangle$. Hence (A.4) holds.

Case 2. $int(D_z/D_y) \neq \mathbb{Z}$.

So $int(D_z/D_y)$ is an integer interval, say $int(D_z/D_y) = [l_{zy}..h_{zy}]$. Two subcases arise.

Subcase 1. $\overline{D_z} \oslash \overline{D_y}$ is a, possibly open ended, real interval.

By (A.3) for some $b_1, b_2 \in D_y$ and $c_1, c_2 \in D_z$ we have

$$l_{zy} \cdot b_1 = c_1,$$

$$h_{zy} \cdot b_2 = c_2.$$

Let

$$\underline{b} := min(b_1, b_2), \overline{b} := max(b_1, b_2), \underline{c} := min(c_1, c_2), \overline{c} := max(c_1, c_2).$$

So $\{l_{zy}, h_{zy}\} \subseteq [\underline{c}, \overline{c}] \oslash [\underline{b}, \overline{b}]$. Also $[\underline{c}, \overline{c}] \oslash [\underline{b}, \overline{b}] \subseteq \overline{D_z} \oslash \overline{D_y}$. Hence $\{l_{zy}, h_{zy}\} \subseteq \overline{D_z} \oslash \overline{D_y}$ and consequently, by the assumption for this subcase, $[l_{zy}, h_{zy}] \subseteq \overline{D_z} \oslash \overline{D_y}$. This proves (A.4) since by (A.3) $\{l_x, h_x\} \subseteq [l_{zy}, h_{zy}]$.

Subcase 2. $\overline{D_z} \oslash \overline{D_y}$ is not a, possibly open ended, real interval.

In what follows for an integer interval $D := [l..h]$ we write $D > 0$ if $l > 0$, $D < 0$ if $h < 0$. Also recall that $\langle D \rangle := \{x \in \mathbb{Z} \mid l < x < h\}$.

This subcase can arise only when $D_z > 0$ and $0 \in \langle D_y \rangle$ or $D_z < 0$ and $0 \in \langle D_y \rangle$, see [Rat96] (reported as Theorem 4.8 in [HJvE01]), where the definition of the division of real intervals is considered.

Since ϕ is closed under the *MULTIPLICATION* rule 3

$$D_y \subseteq int(D_z/D_x).$$

So $int(D_z/D_x) \neq \emptyset$ since by assumption D_y is non-empty. Also, since $0 \notin D_z$, we have $int(D_z/D_x) \neq \mathbb{Z}$. So $int(D_z/D_x)$ is a non-empty integer interval such that $0 \in \langle int(D_z/D_x) \rangle$.

But $D_z > 0$ or $D_z < 0$, so if $D_x > 0$, then $int(D_z/D_x) > 0$ or $int(D_z/D_x) < 0$ and if $D_x < 0$, then $int(D_z/D_x) > 0$ or $int(D_z/D_x) < 0$, as well. So $0 \in \langle D_x \rangle$. Hence $0 \in \langle D_x \rangle \cap \langle D_y \rangle$ while $0 \notin D_z$. This contradicts (5.5). So this subcase cannot arise.

The proof for the variable y is symmetric to the one for the variable x.

Consider now the variable z. ϕ is closed under the applications of the *MULTIPLICATION 1* rule, so

$$D_z \subseteq int(D_x \cdot D_y).$$

Take now $c \in D_z$. Then there exist $a_1, a_2 \in D_x$ and $b_1, b_2 \in D_y$ such that $a_1 \cdot b_1 \leq c \leq a_2 \cdot b_2$. We can assume that both inequalities are strict, that is,

$$a_1 \cdot b_1 < c < a_2 \cdot b_2, \tag{A.5}$$

since otherwise the desired conclusion is established.

Let

$$\underline{a} := min(a_1, a_2), \overline{a} := max(a_1, a_2), \underline{b} := min(b_1, b_2), \overline{b} := max(b_1, b_2).$$

We now show that $a \in [\underline{a}..\overline{a}]$ and $b \in [\underline{b}..\overline{b}]$ exist such that $c = a \cdot b$. Since $[\underline{a}..\overline{a}] \subseteq \overline{D_x}$ and $[\underline{b}..\overline{b}] \subseteq \overline{D_y}$, this will establish the bounds consistency of z.

The choice of a and b depends on the signs of a_1 and b_2. When one of these values is zero, the choice is provided in the following table, where in each case on the account of (A.5) no division by zero takes place:

condition	a	b
$a_1 = 0$	c/b_2	b_2
$a_2 = 0$	c/b_1	b_1
$b_1 = 0$	a_2	c/a_2
$b_2 = 0$	a_1	c/a_1

It is straightforward to show that in each case the quotient belongs to the corresponding interval. For example, when $a_1 = 0$ we need to prove that $c/b_2 \in [\underline{a}..\overline{a}]$. By (A.5) $a_2 \neq 0$. If $a_2 > 0$, then again by (A.5), $b_2 > 0$, so $c/b_2 \in [0..a_2]$. In turn, if $a_2 < 0$, then also by (A.5) $b_2 < 0$, so, yet again by (A.5), $c/b_2 \in [a_2..0]$.

When neither a_1 nor b_2 is zero, the choice of a and b has to be argued case by case.

Case 1. $a_1 > 0,\ b_2 > 0$.
Then by (A.5) $b_1 < c/a_1$ and $c/b_2 < a_2$. Suppose that both $b_2 < c/a_1$ and $c/b_2 < a_1$. Then $a_1 \cdot b_2 < c < a_1 \cdot b_2$, which is a contradiction. So either $c/a_1 \leq b_2$ or $a_1 \leq c/b_2$, that is either $c/a_1 \in [b_1..b_2]$ or $c/b_2 \in [a_1..a_2]$.

Case 2. $a_1 > 0,\ b_2 < 0$.
Then by (A.5) $b_1 < c/a_1$ and $a_2 < c/b_2$. Suppose that both $b_2 < c/a_1$ and $a_1 < c/b_2$. Then $a_1 \cdot b_2 < c < a_1 \cdot b_2$, which is a contradiction. So either $c/a_1 \leq b_2$ or $c/b_2 \leq a_2$, that is either $c/a_1 \in [b_1..b_2]$ or $c/b_2 \in [a_2..a_1]$.

Case 3. $a_1 < 0,\ b_2 > 0$.
Then by (A.5) $c/a_1 < b_1$ and $c/b_2 < a_2$. Suppose that both $c/a_1 < b_2$ and $c/b_2 < a_1$. Then $a_1 \cdot b_2 < c < a_1 \cdot b_2$, which is a contradiction. So either $b_2 \leq c/a_1$ or $a_1 \leq c/b_2$, that is either $c/a_1 \in [b_2..b_1]$ or $c/b_2 \in [a_1..a_2]$.

Case 4. $a_1 < 0,\ b_2 < 0$.
Then by (A.5) $c/a_1 < b_1$ and $a_2 < c/b_2$. Suppose that both $c/a_1 < b_2$ and $a_1 < c/b_2$. Then $a_1 \cdot b_2 < c < a_1 \cdot b_2$, which is a contradiction. So either $b_2 \leq c/a_1$ or $c/b_2 \leq a_1$, that is either $c/a_1 \in [b_2..b_1]$ or $c/b_2 \in [a_2..a_1]$.

So in each of the four cases we can choose either $a := a_1$ and $b := c/a_1$ or $a := c/b_2$ and $b := b_2$. □

Proof of the *MULTIPLICATION* Theorem 5.9.1.
The weak interval division produces larger sets than the interval division. As a result the *MULTIPLICATION* rules 2w and 3w yield a weaker reduction than the original *MULTIPLICATION* rules *2* and *3*. So it suffices to prove that $\phi := \langle x \cdot y = z \ ;\ x \in D_x, y \in D_y, z \in D_z\rangle$ is closed under the applications of the *MULTIPLICATION 1*, *2* and *3* rules assuming that it is closed under the applications of the *MULTIPLICATION 1*, *2w* and *3w* rules. Suppose that $D_x = [l_x..h_x], D_y = [l_y..h_y], D_z = [l_z..h_z]$. The assumption implies

$$\{l_x, h_x\} \subseteq \text{int}(D_z : D_y) \tag{A.6}$$

and

$$\{l_y, h_y\} \subseteq \text{int}(D_z : D_x) \tag{A.7}$$

The proof is by contradiction. Assume that (A.6) and (A.7) hold, while ϕ is not closed under application of *MULTIPLICATION 2* and *3*. Without loss of generality, suppose that *MULTIPLICATION 2* is the rule that can make a further reduction. This is the case iff

$$\text{int}(D_z/D_y) \subset \text{int}(D_z : D_y).$$

By definition, the proper inclusion implies that $l_y \geq 0$ or $h_y \leq 0$. Assume $l_y \geq 0$, the case for $h_y \leq 0$ is similar. Let $l'_y := \max(1, l_y)$, and let $A := \{l_z/l'_y, l_z/h_y, h_z/l'_y, h_z/h_y\}$, and $B := \{l_z/l_x, l_z/h_x, h_z/l_x, h_z/h_x\}$. A further implication of the proper inclusion is that one or both of l'_y and h_y do not have a multiple in D_z: otherwise $\min(A)$ and $\max(A)$ would be elements of D_z/D_y, and we would have $\text{int}(D_z : D_y) = \text{int}(D_z/D_y)$. The cases for l'_y and h_y can be seen in isolation, and their proofs are similar, so here we only consider the case that l'_y does not have a multiple in D_z. In what follows we can assume $0 \notin D_z$, since otherwise l'_y and h_y do have a multiple in D_z.

Case 1. $l_z > 0$.
From (A.6) it follows that $h_x \leq \lfloor \max(A) \rfloor$, which for the case $l'_y, h_y, l_z, h_z > 0$ that we consider here implies $h_x \leq \lfloor h_z/l'_y \rfloor$. Because $[l_z..h_z]$ does not contain a multiple of l'_y, we have $\lfloor h_z/l'_y \rfloor = \lfloor l_z/l'_y \rfloor$, so

$$h_x \leq \lfloor l_z/l'_y \rfloor.$$

A further consequence of (A.6) is that $l_x, h_x > 0$. From (A.7) it follows that $l'_y \geq \lceil \min(B) \rceil$, which for $l_x, l_z > 0$ implies

$$l'_y \geq \lceil l_z/h_x \rceil \geq l_z/h_x \geq l_z/\lfloor h_z/l'_y \rfloor.$$

Because l'_y is no divisor of l_z, and both numbers are positive, we have $\lfloor l_z/l'_y \rfloor < l_z/l'_y$, and consequently $l'_y > l_z/(l_z/l'_y)$, leading to $l'_y > l'_y$, which is a contradiction.

Case 2. $h_z < 0$.
Similarly, because $l'_y, h_y > 0$ and $l_z, h_z < 0$, it follows from (A.6) that $l_x \geq \lceil \min(A) \rceil = \lceil l_z/l'_y \rceil$, and $l_x, h_x < 0$. Because $[l_z..l_h]$ does not contain a multiple of l'_y, we have $\lceil l_z/l'_y \rceil = \lceil h_z/l'_y \rceil$, so

$$l_x \geq \lceil h_z/l'_y \rceil.$$

We use this information in the following implication of (A.7):

$$l'_y \geq \lceil \min(B) \rceil = \lceil h_z/l'_x \rceil \geq h_z/l'_x$$

to get $l'_y \geq h_z/\lceil h_z/l'_y \rceil$. Because $|\lceil h_z/l'_y \rceil| < |h_z/l'_y|$, we have $l'_y > h_z/(h_z/l'_y)$, leading to $l'_y > l'_y$, which is a contradiction. □

Bibliography

[AB03] Krzysztof R. Apt and Sebastian Brand. Schedulers for rule-based constraint programming. In *Proceedings of the 2003 ACM Symposium on Applied Computing (SAC), March 9-12, 2003, Melbourne, FL, USA*, pages 14–22. ACM Press, 2003.

[ABC$^+$02] G. Audemard, P. Bertoli, A. Cimatti, A. Kornilowicz, and R. Sebastiani. A SAT based approach for solving formulas over boolean and linear mathematical propositions. In Andrei Voronkov, editor, *Proceedings of the 8th International Conference on Computer Aided Deduction (CADE 2002)*, volume 2392 of *LNCS*, pages 195–210. Springer-Verlag, 2002.

[ABRS04] Farhad Arbab, C. Baier, J.J.M.M. Rutten, and Marjan Sirjani. Modeling component connectors in reo by constraint automata (extended abstract). In *Proceedings of the 2nd International Workshop on Foundations of Coordination Languages and Software Architectures (FOCLASA 2003)*, volume 97 of *ENTCS*, pages 25–46, 2004.

[AC91] David Applegate and William Cook. A computational study of the job-shop scheduling problem. *ORSA Journal on Computing*, 3(2):149–156, 1991.

[AM98] Farhad Arbab and Eric Monfroy. Coordination of heterogeneous distributed cooperative constraint solving. *SIGAPP Applied Computing Review*, 6(2):4–17, 1998.

[AM00] Farhad Arbab and Eric Monfroy. Distributed Splitting of Constraint Satisfaction Problems. In Porto and Roman, editors, *Coordination Languages and Models*, volume 1906 of *LNCS*, pages 115–132. Springer-Verlag, 2000.

[Apt98] Krzysztof R. Apt. A proof theoretic view of constraint programming. *Fundamenta Informaticae*, 33(3):263–293, 1998.

[Apt99] Krzysztof R. Apt. The rough guide to constraint propagation. In Joxan Jaffar, editor, *Proceedings of the 5th International Conference on Principles and Practice of Constraint Programming (CP'99)*, volume 1713 of *LNCS*, pages 1–23. Springer-Verlag, 1999.

[Apt03] Krzysztof R. Apt. *Principles of Constraint Programming.* Cambridge University Press, 2003.

[AR02] Farhad Arbab and Jan J. M. M. Rutten. A coinductive calculus of component connectors. In Martin Wirsing, Dirk Pattinson, and Rolf Hennicker, editors, *Recent Trends in Algebraic Development Techniques, 16th International Workshop, WADT 2002, Frauenchiemsee, Germany, September 24-27, 2002, Revised Selected Papers*, volume 2755 of *LNCS*, pages 34–55. Springer-Verlag, 2002.

[Arb] F. Arbab. Manifold version 2: Language reference manual. Available from `http://www.cwi.nl/ftp/manifold/refman.ps.Z`.

[Arb96] Farhad Arbab. The IWIM model for coordination of concurrent activities. In Paolo Ciancarini and Chris Hankin, editors, *Coordination Languages and Models*, volume 1061 of *LNCS*, pages 34–56. Springer-Verlag, 1996.

[Arb98] Farhad Arbab. What do you mean, coordination? *Bulletin of the Dutch Association for Theoretical Computer Science, NVTI*, pages 11–22, 1998.

[Arb02] Farhad Arbab. Abstract behavior types: a foundation model for components and their composition. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects*, volume 2852 of *LNCS*, pages 33–70. Springer-Verlag, 2002.

[AZ04] Krzysztof R. Apt and Peter Zoeteweij. A comparative study of arithmetic constraints on integer intervals. In Krzysztof R. Apt, Francois Fages, Francesca Rossi, Peter Szeredi, and Jozsef Váncza, editors, *Proceedings of CSCLP 2003, Budapest, Hungary, June 30 - July 2, 2003*, volume 3010 of *LNAI*, pages 1–24. Springer-Verlag, 2004.

[Ben96] Frédéric Benhamou. Heterogeneous constraint solving. In Michael Hanus and Mario Rodríguez-Artalejo, editors, *Algebraic and Logic*

Programming, 5th International Conference, ALP'96, Aachen, Germany, September 25-27, 1996, Proceedings, volume 1139 of *LNCS*, pages 62–72. Springer-Verlag, 1996.

[BGGP99] Frédéric Benhamou, Frédéric Goualard, Laurent Granvilliers, and Jean-Francois Puget. Revising hull and box consistency. In *Proceedings of the 16th International Conference on Logic Programming (ICLP'99)*, pages 230–244. The MIT Press, 1999.

[BK98] J.A. Bergstra and P. Klint. The discrete time ToolBus – a software coordination architecture. *Science of Computer Programming*, 31(2-3):205–229, July 1998.

[BLPN01] Philippe Baptiste, Claude Le Pape, and Wim Nuijten. *Constraint-Based Scheduling: Applying Constraint Programming to Scheduling Problems*, volume 39 of *International Series in Operations Research and Management Science*. Kluwer Academic Publishers, 2001.

[BMVH94] F. Benhamou, D. McAllester, and P. Van Hentenryck. CLP(intervals) revisited. In *Proceedings of the 1994 International Symposium on Logic programming*, pages 124–138. MIT Press, 1994.

[BO97] Frédéric Benhamou and William J. Older. Applying interval arithmetic to real, integer, and Boolean constraints. *Journal of Logic Programming*, 32(1):1–24, 1997.

[CDR99] Hélène Collaviza, François Delobel, and Michel Rueher. Comparing partial consistencies. *Reliable Computing*, 5(3):213–228, 1999.

[CG89] Nicholas Carriero and David Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, 1989.

[CHN01] Chiu Wo Choi, Martin Henz, and Ka Boon Ng. Components for state restoration in tree search. In Toby Walsh, editor, *Principles and Practice of Constraint Programming - CP 2001, 7th International Conference, CP 2001, Paphos, Cyprus, November 26 - December 1, 2001, Proceedings*, volume 2239 of *LNCS*. Springer-Verlag, 2001.

[CHS$^+$03] Andrew M. Cheadle, Warwick Harvey, Andrew J. Sadler, Joachim Schimpf, Kish Shen, and Mark G. Wallace. ECLiPSe: an introduction. Technical Report IC-Parc-03-1, IC-PARC, Imperial College London, 2003.

[CP89] J. Carlier and E. Pinson. An algorithm for solving the job-shop problem. *Management Science*, 35(2):165–176, February 1989.

[DB97] Romuald Debruyne and Christian Bessière. Some practicable filtering techniques for the constraint satisfaction problem. In *Proceedings of the Fifteenth International Conference on Artificial Intelligence (IJCAI'97), August 23–29, Nagoya, Japan*. Morgan Kaufmann Publishers, Inc., 1997.

[Dec03] Rina Dechter. *Constraint Processing*. Morgan Kaufmann Publishers, 2003.

[Deu83] L. Peter Deutsch. Reusability in the Smalltalk-80 programming system. In *ITT Proceedings of the Workshop on Reusability in Programming*, 1983.

[DGS03] Luca Di Gaspero and Andrea Schaerf. EasyLocal++: an object-oriented framework for the flexible design of local-search algorithms. *Software — Practice and Experience*, 33(8):733–765, 2003.

[Dij87] Edsger W. Dijkstra. Shmuel Safra's version of termination detection. Note EWD998, circulated privately, January 1987.

[DLL62] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Journal of the ACM*, 5(7), 1962.

[DS95] V. Dubrovsky and A. Shvetsov. *Quantum* Cyberteaser. Available from `http://www.nsta.org/quantum/kyotoarc.asp`, May/June 1995.

[Eli04] Laboratoire d'Informatique de Nantes Atlantique (LINA). *Elisa 1.0.3 documentation*, 2004. Available from `http://sourceforge.net/projects/elisa/`.

[GC92] David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2):97–107, 1992.

[Gen02] Rosella Gennari. *Mapping Inferences*. PhD thesis, Institute for Logic, Language and Computation, Amsterdam, 2002.

[GH00] Laurent Granvilliers and Gaétan Hains. A conservative scheme for parallel interval narrowing. *Information Processing Letters*, 74(3–4):141–146, 2000.

[GM03] Laurent Granvilliers and Eric Monfroy. Implementing constraint propagation by composition of reductions. In Catuscia Palamidessi, editor, *Proceedings of ICLP 2003*, volume 2916 of *LNCS*, pages 300–314. Springer-Verlag, 2003.

[GMB01] Laurent Granvilliers, Eric Monfroy, and Frédéric Benhamou. Symbolic-interval cooperation in constraint programming. In *ISSAC '01: Proceedings of the 2001 international symposium on Symbolic and algebraic computation*, pages 150–166. ACM Press, 2001.

[Gou] Frédéric Goualard. *Gaol: NOT Just Another Interval Library.* Available from `http://gaol.sourceforge.net/`.

[Gra04a] Torbjörn Granlund. *GNU MP: The Gnu Multiple Precision Library.* Swox AB, September 2004. Edition 4.1.4, available from `http://www.swox.com/gmp`.

[Gra04b] Laurent Granvilliers. *RealPaver User's Manual: Solving Nonlinear Constraints by Interval Computations*, August 2004. Edition 0.4, for RealPaver Version 0.4, available from `http://sourceforge.net/projects/realpaver`.

[Ham05] Youssef Hamadi. *Disolver: the Distributed Constraint Solver, version 2.0.* Microsoft Research, 2005. Available via `http://research.microsoft.com/~youssefh/DisolverWeb/Disolver.html`.

[Hen01] Martin Henz. Scheduling a major college basketball conference—revisited. *Operations Research*, 49(1):163–168, 2001.

[HG95] W.D. Harvey and M.L. Ginsberg. Limited discrepancy search. In Mellish, editor, *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI-95)*, pages 607–613. Morgan Kaufmann Publishers, 1995.

[Hic01] Timothy J. Hickey. Metalevel interval arithmetic and verifiable constraint solving. *Journal of Functional and Logic Programming*, 2001(7), October 2001.

[HJvE01] Timothy J. Hickey, Qun Ju, and Maarten H. van Emden. Interval arithmetic: from principles to implementation. *Journal of the ACM*, 48(5):1038–1068, 2001.

[HKS01] Zineb Habbas, Michaël Krajecki, and Daniel Singer. Shared memory implementation of constraint satisfaction problem resolution. *Parallel Processing Letters*, 11(4):487–501, 2001.

[HMD97] Pascal Van Hentenryck, Laurent Michel, and Yves Deville. *Numerica: a Modeling Language for Global Optimization.* The MIT Press, 1997.

[HMN99] Martin Henz, Tobias Müller, and Ka Boon Ng. Figaro: Yet another constraint programming library. In *Proceedings of the Workshop on Parallelism and Implementation Technology for Constraint Logic Programming*, 1999. Held in conjunction with ICLP'99.

[HS03] Warwick Harvey and Peter J. Stuckey. Improving linear constraint propagation by changing constraint representation. *Constraints*, 8(2):173–207, 2003.

[HSG01] Petra Hofstedt, Dirk Seifert, and Eicke Godehardt. A framework for cooperating constraint solvers – a prototypic implementation. Presented at the Workshop on Cooperative Solvers in Constraint Programming – CoSolv, held in conjunction with CP, 2001.

[Ilo01] ILOG. *ILOG Solver 5.1 User's Manual*, 2001. See `http://www.ilog.com`.

[JF88] Ralph E. Johnson and Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, June/July 1988.

[KoaA] Koalog. *Koalog Constraint Solver (v2.4) tutorial.* Available from `http://www.koalog.com`.

[KoaB] Koalog. *An Overview of Koalog Constraint Solver.* Available from `http://www.koalog.com`.

[Lab00] François Laburthe. CHOCO: implementing a CP kernel. In *Proceedings of TRICS workshop, held in conjunction with CP 2000*, 2000.

[LC02] François Laburthe and Yves Caseau. Salsa: A language for search algorithms. *Constraints*, 7(3–4):255–288, 2002.

[Lho93] Olivier Lhomme. Consistency techniques for numeric CSPs. In *Proceedings of IJCAI93, Chambery, France*, pages 232–238, 1993.

[LMS03] Inês Lynce and João Marques-Silva. The effect of nogood recording in DPLL-CBJ SAT algorithms. In Barry O'Sullivan, editor, *Recent Advances in Constraints*, volume 2627 of *LNAI*, pages 144–158. Springer-Verlag, 2003.

[Mac77] A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.

[Mes97] Pedro Meseguer. Interleaved depth-first search. In Martha E. Pollack, editor, *Proceedings of IJCAI-97, Nagoya, Japan, August 23–29, 1997*, volume 2, pages 1382–1387. Morgan Kaufmann Publishers, Inc., 1997.

[MMZ+01] Mathew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 30th Design Automation Conference (DAC 2001), Las Vegas, June 2001*, pages 530–535. ACM, 2001.

[Mon00a] Eric Monfroy. A Coordination-based Chaotic Iteration Algorithm for Constraint Propagation. In Caroll, Damiani, Haddad, and Oppenheim, editors, *Proceedings of the 2000 ACM Symposium on Applied Computing*, pages 262–269. ACM Press, 2000.

[Mon00b] Eric Monfroy. The constraint solver collaboration language of BALI. In D.M. Gabbay and M. de Rijke, editors, *Frontiers of Combining Systems 2*, volume 7 of *Studies in Logic and Computation*, pages 211–230. Research Studies Press, 2000.

[Moo66] Ramon E. Moore. *Interval Analysis.* Prentice-Hall, Englewood Cliffs, NJ, 1966.

[MR99] Eric Monfroy and Jean-Hugues Réty. Chaotic iteration for distributed constraint propagation. In *Proceedings of the 14th ACM Symposium on Applied Computing*, pages 19–24, 1999.

[MS94] Shyam Mudambi and Joachim Schimpf. Parallel CLP on heterogeneous networks. In Pascal Van Hentenryck, editor, *Proceedings of ICLP 1994*, pages 124–141. MIT Press, 1994.

[MS96] P. Martin and D. B. Shmoys. A new approach to computing optimal schedules for the job-shop scheduling problem. In *Proceedings of the 5th Conference on Integer Programming and Combinatorial Optimization*, 1996.

[MSS96] João P. Marques-Silva and Karem A. Skallah. GRASP — a new search algorithm for satisfiability. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD 1996), San Jose, California, USA, November 1996*, pages 220–227, 1996.

[MVH00] Laurent Michel and Pascal Van Hentenryck. Localizer. *Constraints*, 5:43–84, 2000.

[MVH01] Laurent Michel and Pascal Van Hentenryck. Localizer++: An open library for local search. Technical Report CS-01-02, Brown University, Providence, Rhode Island, 2001.

[MVH02a] Laurent Michel and Pascal Van Hentenryck. A constraint-based architecture for local search. In *Proceedings of OOPSLA '02, Seattle, WA, USA, November 2002*, pages 83–100. ACM Press, 2002.

[MvH02b] Michela Milano and Willem Jan van Hoeve. Reduced cost-based ranking for generating promising subproblems. In Pascal Van Hentenryck, editor, *Principles and Practice of Constraint Programming, Proceedings of CP 2002, Ithaca, NY, USA, September 2002*, volume 2470 of *LNCS*, pages 1–16. Springer-Verlag, 2002.

[Per99] Laurent Perron. Search procedures and parallelism in constraint programming. In Joxan Jaffar, editor, *Proceedings of the 5th International Conference on Principles and Practice of Constraint Programming (CP'99)*, volume 1713 of *LNCS*, pages 346–360. Springer-Verlag, 1999.

[Rat96] Dietmar Ratz. Inclusion isotone extended interval arithmetic: a toolbox update. Technical Report D-76128, University of Karlsruhe, 1996.

[RH05] Georg Ringwelski and Youssef Hamadi. Boosting distributed constraint satisfaction. In Peter van Beek, editor, *Proceedings of CP 2005*, volume 3709 of *LNCS*, 2005.

[Rin05] Georg Ringwelski. An arc-consistency algorithm for dynamic and distributed constraint satisfaction problems. *Artificial Intelligence Review*, 2005.

[Sch99] Christian Schulte. Comparing trailing and copying for constraint programming. In De Schreye, editor, *Proceedings of the Sixteenth International Conference on Logic Programming*, pages 275–289. The MIT Press, 1999.

[Sch00] Christian Schulte. Parallel search made simple. In Nicolas Beldiceanu, Warwick Harvey, Martin Henz, François Laburthe, Eric Monfroy, Tobias Müller, Laurent Perron, and Christian Schulte, editors, *Proceedings of TRICS: Techniques foR Implementing Constraint programming Systems, a post-conference workshop of CP 2000*, Singapore, September 2000.

[Sil99] João P. Marques Silva. The impact of branching heuristics in propositional satisfiability algorithms. In Pedro Barahona and José Júlio Alferes, editors, *Progress in Artificial Intelligence, proceedings of EPIA '99, Évora, Portugal, September 21-24, 1999*, volume 1695 of *LNCS*, pages 62–74. Springer-Verlag, 1999.

[Smi95] Barbara M. Smith. A tutorial on constraint programming. Report 95.14, University of Leeds, School of Computer Studies, April 1995.

[SR90] Vijay A. Saraswat and Martin Rinard. Concurrent constraint programming. In *Proceedings of the 17th ACM Symposium on Principles of Programming Languages (POPL 90)*, pages 232–245, 1990.

[SS01] Christian Schulte and Peter J. Stuckey. When do bounds and domain propagation lead to the same search space. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 115–126. ACM Press, 2001.

[SS02] Christian Schulte and Gert Smolka. Finite domain constraint programming in Oz. A tutorial. Available from `http://www.mozart-oz.org/documentation/fdt/`, August 2002. Version 1.3.1 (20040616).

[SS04] Christian Schulte and Peter J. Stuckey. Speeding up constraint propagation. In Mark Wallace, editor, *Principles and Practice of Constraint Programming, Proceedings of CP 2004, Toronto, Canada, September/October 2004*, volume 3258 of *LNCS*, pages 619–633. Springer-Verlag, 2004.

[Sta02] Andries Stam. A framework for coordinating parallel branch and bound algorithms. In *Coordination Models and Languages*, volume 2315 of *LNCS*, pages 332–339. Springer-Verlag, 2002.

[TU96] Vitaly Telerman and Dmitry Ushakov. Data types in subdefinite models. In Jacques Calmet, John A. Campbell, and Jochen Pfalzgraf, editors, *Artificial Intelligence and Symbolic Mathematical Computation, Proceedings of AISMC-3*, volume 1138 of *LNCS*, pages 305–319. Springer-Verlag, 1996.

[vD02] M.R.C. van Dongen. AC-3_d an efficient arc-consistency algorithm with a low space-complexity. In P. Van Hentenryck, editor, *Proceedings of the Eighth International Conference on Principles and Practice of Constraint Programming (CP'2002)*, volume 2470 of *Lecture notes in Computer Science*, pages 755–760. Springer, 2002.

[VH99] Pascal Van Hentenryck. *The OPL Optimization Programming Language.* The MIT Press, 1999.

[VHPP00] Pascal Van Hentenryck, Laurent Perron, and Jean-Francois Puget. Search and strategies in OPL. *ACM Transactions on Computational Logic*, 1(2):285–320, October 2000.

[VHSD92] Pascal Van Hentenryck, Helmut Simonis, and Mehmet Dincbas. Constraint satisfaction using constraint logic progamming. *Artificial Intelligence*, 58(1–3):113–150, 1992.

[VHSD98] Pascal Van Hentenryck, Vijay A. Saraswat, and Yves Deville. Design, implementation, and evaluation of the constraint language cc(FD). *Journal of Logic Programming*, 37(1–3):139–164, 1998. Special Issue on Constraint Logic Programming (P. J. Stuckey and K. Marriot, Eds.).

[VRBD$^+$03] Peter Van Roy, Per Brand, Denys Duchier, Seif Haridi, Martin Henz, and Christian Schulte. Logic programming in the context of multiparadigm programming: the Oz experience. *Theory and Practice of Logic Programming*, 3(6):715–763, 2003.

[WNS97] Mark G. Wallace, Stefano Novello, and Joachim Schimpf. ECL^{i}PS^e: A Platform for Constraint Logic Programming. *ICL Systems Journal*, 12(1):159–200, May 1997.

[Yok01] Makoto Yokoo. *Distributed Constraint Satisfaction: Foundations of Cooperation in Multi-Agent Systems*. Springer-Verlag, 2001.

[ZA04] Peter Zoeteweij and Farhad Arbab. A component-based parallel constraint solver. In Rocco De Nicola, GianLuigi Ferrari, and Greg Meredith, editors, *Coordination Models and Languages, Proceedings of COORDINATION 2004, Pisa, Italy, February 2004*, volume 2949 of *LNCS*, pages 307–322. Springer-Verlag, 2004.

[Zho97] Jianyang Zhou. A permutation-based approach for solver the jobshop problem. *Constraints*, 2:185–213, 1997.

[ZM02] Lintao Zhang and Sharad Malik. The quest for efficient boolean satisfiability solvers. In Andrei Voronkov, editor, *Proceedings of the 8th International Conference on Computer Aided Deduction (CADE 2002)*, volume 2392 of *LNCS*, pages 295–313. Springer-Verlag, 2002.

[Zoe03a] Peter Zoeteweij. Coordination-based distributed constraint solving in DICE. In *Proceedings of the 2003 ACM Symposium on Applied Computing*, pages 360–366, 2003.

[Zoe03b] Peter Zoeteweij. A coordination-based framework for parallel constraint solving. In Barry O'Sullivan, editor, *Recent Advances in Constraints*, volume 2627 of *LNAI*, pages 171–184. Springer-Verlag, 2003.

[Zoe03c] Peter Zoeteweij. Opensolver: A coordination-enabled abstract branch-and-prune tree search engine (abstract). In Francesca Rossi, editor, *Proceedings of CP 2003*, volume 2833 of *LNCS*, page 1002. Springer-Verlag, 2003.

[Zoe04a] Peter Zoeteweij. Applications of nested search. In Boi Faltings, Francois Fages, Francesca Rossi, and Adrian Petcu, editors, *Proceedings of CSCLP'04, Lausanne, Switzerland, June 23 – 25, 2003*, 2004. Available from `http://liawww.epfl.ch/Events/ercim04/`.

[Zoe04b] Peter Zoeteweij. Constraining special-purpose domain types (abstract). In Mark Wallace, editor, *Proceedings of CP 2004*, volume 3258 of *LNCS*, page 809. Springer-Verlag, 2004.

Index

Samenvatting

Veel vraagstukken uit het dagelijks leven, de wetenschap en de industrie zijn ***combinatorische problemen***, of combinatorische ***optimalisatieproblemen***. Een mogelijke oplossing voor dit soort vraagstukken bestaat uit de combinatie van verschillende keuzes, die elkaar allemaal beïnvloeden, waardoor een menselijke probleemoplosser het overzicht verliest. Het systematisch nagaan van alle mogelijke combinaties van keuzes door een computer kan hier uitkomst bieden, maar de zoekruimte die zo ontstaat is in het algemeen te groot (men spreekt wel van een combinatorische explosie) om binnen redelijke tijd een antwoord te kunnen verwachten.

Voor sommige klassen van problemen zijn echter efficiënte oplossingsmethoden gevonden, die in de praktijk goed blijken te werken. In andere gevallen kan worden volstaan met niet-systematische zoekmethoden, die weliswaar snel tot resultaat kunnen leiden, maar geen uitsluitsel kunnen geven over het bestaan van een oplossing, of over optimaliteit hiervan. In dit proefschrift richten we ons echter op systematische methoden voor problemen waarvoor geen specifieke efficiënte methode voorhanden is, geformuleerd als een verzameling voorwaarden, of ***constraints***, waaraan de waarden die kunnen worden toegekend aan verschillende variabelen moeten voldoen. Het opstellen en oplossen van dit soort problemen staat bekend onder de naam ***constraint programming***, en omvat een groot aantal technieken voor systematisch zoeken, en voor het verkleinen van de zoekruimte. Doordat op voorhand vaak niet duidelijk is met welke technieken een dergelijk probleem kan worden opgelost, is het van groot belang dat we kunnen experimenteren met het samenstellen, of componeren, van constraint solvers (oplosprocedures) uit het palet van beschikbare technieken.

Allereerst definiëren we wat we precies verstaan onder het oplossen van een constraint probleem. In deze definitie staat het concept van een zgn. opgeloste vorm van een constraint probleem centraal. Een opgeloste vorm van een probleem is niet noodzakelijk een oplossing, maar alle opgeloste vormen bij elkaar omvatten wel degelijk alle oplossingen. Op deze manier abstraheren we van beperkingen die inherent zijn aan het representeren van reëele getallen in een computer, en kunnen we sommige variabelen typeren als hulpvariabelen, waarmee we aangeven dat hun precieze waarde er niet toe doet. Vervolgens illustreren we het belang van ons werk door een aantal technieken die in de praktijk worden gebruikt te beschrijven als constraint solver composities.

Om te kunnen experimenteren met constraint solver composities hebben we een computerprogramma geschreven. Dit programma, OpenSolver, is in de eerste plaats een uitbreidbare verzameling constraint solver bouwstenen, waarmee op laag niveau verschillende technieken kunnen worden gecombineerd. In de tweede plaats is het programma ook een autonome applicatie, die er helemaal op is ingericht om op verschillende manieren van buitenaf te kunnen worden gecoördineerd, en zo als component aan een groter samenwerkingsverband van softwarecomponenten te kunnen deelnemen.

Na een beschrijving van deze software vervolgen we het betoog met een demonstratie van de wijze waarop een aantal standaardtechnieken voor het oplossen van constraint problemen in OpenSolver zijn gerealiseerd. Typerend hiervoor is dat waar deze standaardtechnieken gewoonlijk zijn vastgelegd in de broncode van constraint solvers, ze in OpenSolver worden opgebouwd uit bouwstenen voor kleinere deeltechnieken. Dit biedt mogelijkheden voor het combineren, en voor domeinoverschreidend gebruik van bestaande technieken.

Het open karakter van onze software stelt ons vervolgens in staat te onderzoeken wat de efficiëntste manier is om rekenkundige constraints op geheeltallige variabelen op te lossen. Hierbij richten we ons specifiek op een intervalrepresentatie van de verzamelingen toegestane waarden voor deze variabelen. Veel bestaande constraint solvers bieden mogelijkheden hiervoor, maar een systematische analyse ontbrak tot nu toe, voor zover wij hebben kunnen nagaan.

Ook laten we zien hoe OpenSolver kan worden aangevuld met bouwstenen die voor een specifieke toepassing zijn ontwikkeld, in dit geval het oplossen van het zgn. job-shop scheduling probleem. Op die manier ontstaat een solver voor een specifiek probleem, maar doordat we voor een groot deel van de functionaliteit gebruik kunnen maken van reeds bestaande bouwstenen, is de ontwikkeltijd van zo'n solver gering vergeleken met de ontwikkeling van een volledig nieuwe solver. Bovendien kunnen nu ook voor het specifieke probleem eenvoudig variaties in de oplosmethode worden aangebracht.

Na deze voorbeelden van het componeren van constraint solvers uit bouwstenen te hebben besproken, verleggen we onze aandacht naar constructies waarin OpenSolver zelf als software component deel uit maakt van een groter systeem. Allereerst introduceren we een operator voor genest zoeken, en laten we zien hoe een drietal veelgebruikte constraint solving technieken kunnen worden beschreven als toepassingen van deze operator. We implementeren de operator met een vrijwel op zichzelf staand OpenSolver exemplaar, en we beschrijven een aantal experimenten die aantonen dat dit tot bruikbare en flexibele solvers leidt.

Een tweede voorbeeld van het gebruik van OpenSolver als software component is het parallel zoeken naar oplossingen van een constraint probleem. Het doel van parallellisatie in het algemeen is het reduceren van de rekentijd, door het rekenwerk te verdelen over verschillende processoren. Onze parallelle solver bestaat uit een aantal OpenSolver exemplaren, die regelmatig al hun nog te doorzoeken deelproblemen aan een centraal distributiepunt teruggeven, van waaruit ze weer worden herverdeeld. Dit gebeurt telkens als een vooraf ingestelde periode verstrijkt. Voor zover we hebben kunnen nagaan is deze aanpak van parallel zoeken nieuw, en onze experimenten tonen aan dat dit tot efficiënte en schaalbare parallelle solvers leidt.

Als derde voorbeeld beschrijven we hoe verschillende OpenSolver exemplaren kunnen samenwerken aan het verkleinen van de zoekruimte, als onderdeel van een groter zoekproces. Hoewel dit niet efficiënt is, kan het nodig zijn in het geval dat het constraint probleem zelf fysiek gedistribueerd is. We gebruiken hiervoor bestaande algoritmen die in OpenSolver met alle andere beschikbare technieken gecombineerd kunnen worden.

Tot slot vatten we de behaalde resultaten samen, vergelijken we onze aanpak met een aantal alternatieven, en bespreken we mogelijkheden voor verder onderzoek en ontwikkeling.