Composing constraint solvers
Zoeteweij, P.

Citation for published version (APA):

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: http://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
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 \((e_1, \ldots, e_n)\). 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 \( s := i_1, \ldots, i_t \), let \( t[s] \) denote the sequence \( e_{i_1}, \ldots, e_{i_t} \), 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 \( t[i] = (e_i) = e_i \).

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 \( D := D_1, \ldots, D_n \), together with a finite set \( C \) 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.

\[
(C : x_1 \in D_1, \ldots, x_n \in D_n).
\]  

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. Definitions

2.2.1. Example. Consider the following CSP.

\[ x < y, \ y \neq z ; \ x, y, z \in \{1, 2, 3\} \]

The constraint \( x < y \) denotes the subset \( \{1, 2\}, \{1, 3\}, \{2, 3\} \) 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 \( s := i_1, \ldots, i_t \), let \( C[\mathcal{D}'[s]] \) denote \( C \cap D'_{i_1} \times \ldots \times D'_{i_t} \), the projection of \( \mathcal{D}'[s] \) on \( C \). \( C[\mathcal{D}] \) 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

\[(x \neq y, y \neq z ; x, y, z \in \{0, 1\})\]

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 \((x, y) \in \{(1, 2), (1, 3), (2, 3)\}\) 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

\[\langle x < y, y \neq z ; x \in \{1, 2\}, y \in \{2, 3\}, z \in \{1, 2, 3\}\]\n
which is equivalent to the original CSP. \(\square\)

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 \(P(D_i)\), 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 \(P(D_i)\) may not be the most efficient choice.

Moreover, if \(D_i\) is a set of real numbers, using \(P(D_i)\) 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 \(D_i \subseteq \mathbb{R}\) 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 Telemann 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 $T$ 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 $T^\uparrow$ 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 $T$.

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 $T$ is a partial order relation $R$ such that every non-empty subset of $T$ 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 $T$ and a set $D \subseteq T^\uparrow$, let $T(D)$ denote the smallest element of $T$ that is a superset of $D$. $T(D)$ is called the representation of $D$ in $T$.

2.2.6. Example. Let $T$ denote the domain type containing the following 11 domains.

\[
\begin{align*}
\{1, 2, 3, 4, 5, 6, 7\} & \\
\{1, 2, 3, 4, 5\} & \{3, 4, 5, 6, 7\} \\
\{1\} & \{1, 2, 3\} \quad \{3\} \quad \{3, 4, 5\} \quad \{5\} \quad \{5, 6, 7\} \quad \{7\} \\
\emptyset
\end{align*}
\]
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 \(\mathcal{T}^+ = \{1,2,3,4,5,6,7\}\) 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

\[
[\mathcal{T}] := \{\mathcal{T}(\{x\}) \mid x \in \mathcal{T}^+\}.
\]

The elements of \([\mathcal{T}]\) 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 \(\mathbb{R}^\infty\) denote \(\mathbb{R} \cup (-\infty, \infty)\), 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 \(x \in \mathbb{R}\). \(\mathbb{F}\) denotes a finite subset of \(\mathbb{R}^\infty\) that contains \(-\infty\) and \(\infty\), and is used to model a set of floating-point numbers of unspecified, but fixed precision. For \(a, b \in \mathbb{R}^\infty\), let \([a, b]\) denote the set \(\{x \in \mathbb{R} \mid a < x < b\}\), and for two integers \(a\) and \(b\), let \([a..b]\) denote the set \(\{i \in \mathbb{Z} \mid a < i < b\}\).

- \(\mathcal{B}\) denotes the domain type \(\{\text{true, false}\}, \{\text{true}\}, \{\text{false}\}, \emptyset\)\), 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 \(a \leq b\). Elements of \(\mathcal{I}\) are called **integer intervals**.
- \(\mathcal{F}\) denotes the domain type that consists of \(\emptyset\) and all intervals \([a, b]\), where \(a, b \in \mathbb{F}\) and \(a \leq b\). Elements of \(\mathcal{F}\) are called **floating-point intervals**.

Variables with domain type \(\mathcal{Z}\) are usually called **finite domains** variables. Note that \(\mathbb{R} \in \mathcal{F}\), and that \(\mathcal{I} \subset \mathcal{Z}\,\text{and}\, |\mathcal{I}| = |\mathcal{Z}| = \{\{x\} \mid x \in \mathbb{Z}\}\).

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** \([\mathcal{F}]\) contains a singleton set for each of the elements of \(\mathbb{F} - \{-\infty, \infty\}\), and an interval representation for all
other real numbers. The representation of a set of real numbers is usually called
the \textbf{hull} of this set, and to conform our notation, for $D \subseteq \mathbb{R}$ we define
\[ \text{hull}(D) := \mathcal{F}(D). \]

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

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 \textbf{hull consistent} if for all $1 \leq i \leq n,$
\[ D_i = \text{hull}(\{x_i \in \mathbb{R} \mid \exists x_1 \in D_1, \ldots, x_{i-1} \in D_{i-1}, x_{i+1} \in D_{i+1}, \ldots, x_n \in D_n \}
\text{ for which } (x_1, \ldots, x_n) \in C). \]
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.

\textbf{2.2.10. Example.} Let $a = \max(\{x \in \mathbb{F} \mid x < -\frac{1}{10}\})$ and $b = \min(\{x \in \mathbb{F} \mid x > \frac{1}{10}\})$. The CSP
\[ \langle 100x^2 - 1 ; \ x \in \text{hull}([-\frac{1}{10}, \frac{1}{10}]) \rangle \]
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. \( \square \)

\textbf{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 $A \subseteq T$ qualifies as a set of final domains of domain type $T$ if it has the following properties:

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

$$\forall D \in A, (\mathcal{P}(D) \cap T) - \{\emptyset\} \subseteq A.$$

We will be using three specific sets of final domains:

- In general we will use $A = [T]$ 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 $\epsilon$ only. In this case we use the set

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

- 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 $A = T - \{\emptyset\}$ to indicate that we will accept all non-empty domains of the domain type. The variables for which $A \subseteq T - \{\emptyset\}$ 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. Definitions

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

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

(2.2)

where $$( \mathcal{C} ; x_1 \in D_1, \ldots, x_n \in D_n )$$ 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 ECSPP 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 $D_x, D_y, D_z \in \mathbb{Z}$ and $\mathcal{A}_x, \mathcal{A}_y, \mathcal{A}_z = [\mathbb{Z}]$.

2.2.13. Example. In the ECSPP

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

having $\mathcal{A}_x = [\mathbb{Z}], \mathcal{A}_y = \mathbb{Z} - \{0\}$, and $\mathcal{A}_z = [\mathbb{Z}]$, $x$ and $z$ are decision variables, and $y$ is an auxiliary variable. Their domains are represented by elements of the domain type $\mathbb{Z}$.

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

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

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

Further, for two ECSPPs

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

and

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

having $D'_1 \subseteq D_1, \ldots, D'_n \subseteq D_n$, we say that
Chapter 2. Constraint Solving

- $P'$ is a **γ 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 $C$ substituted by the projection $C[D'_1, \ldots, D'_n]$ 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
  - $D'_i \subseteq D_i$, 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 $C'$ are needed because constraints are defined as subsets of Cartesian products of domains.

2.2.15. Example. The ECSP

$$\langle 100x^2 = 1 : x \in \text{hull}\{-\frac{1}{10}, \frac{1}{10}\} ; \ D_x \in \mathcal{F} ; \ A_x = [\mathcal{F}] \rangle$$

is hull consistent, but is not in hull solved form because $D_x \notin A_x = [\mathcal{F}]$.

The ECSP

$$\langle x < y, y \neq z : x = 0, y \in \{1, 2\}, z = 0 ; \ D_x, D_y, D_z \in \mathcal{Z} ; \ A_x, A_y, A_z \rangle$$

with $A_x, A_z = [\mathcal{Z}]$ and $A_y = \mathcal{Z} - \{0\}$ 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 \( P = (C ; x_1 \in D_1, \ldots, x_n \in D_n) \),
- a sequence of domain types \( T_1, \ldots, T_n \) such that for all \( 1 \leq i \leq n \), \( T_i(D_i) \) exists, and
- a sequence of sets of respective final domains \( A_1, \ldots, A_n \),

let \( P_E \) denote the ECSP

\[
(C ; x_1 \in T_1(D_1), \ldots, x_n \in T_n(D_n) ; T_1, \ldots, T_n ; A_1, \ldots, A_n).
\]

Now by solving \( P \) we mean constructing an ECSP that is a \( \gamma \) solved form of \( P_E \), for some notion of local consistency \( \gamma \). 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
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

\[
\langle x < y, y \neq z \mid x, y, z \in \{0, 1, 2\} \rangle
\]

We deal with integer domains, and all variables are decision variables, so no explicit reference to the ECS 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

\[
f : \mathcal{T}_{i_1} \times \cdots \times \mathcal{T}_{i_l} \to \mathcal{T}_{j_1} \times \cdots \times \mathcal{T}_{j_m}
\]

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

\[
\langle D'_1, \ldots, D'_n \rangle = f(D_1, \ldots, D_n)
\]

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

\[
f^+ : \mathcal{D}_1 \times \cdots \times \mathcal{D}_n \to \mathcal{D}_1 \times \cdots \times \mathcal{D}_n
\]

is called the domains extension of \( f \). Applying it transforms an ECSP

\[
P := \langle \mathcal{C} ; x_1 \in D_1, \ldots, x_n \in D_n ; \mathcal{D}_1, \ldots, \mathcal{D}_n ; A_1, \ldots, A_n \rangle,
\]

into

\[
P' := \langle \mathcal{C}' ; x_1 \in D'_1, \ldots, x_n \in D'_n ; \mathcal{D}_1, \ldots, \mathcal{D}_n ; A_1, \ldots, A_n \rangle,
\]

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

\[
D'_y = \begin{cases} D_y - D_z & \text{if } D_z = \{z\} \\ D_y & \text{otherwise} \end{cases} \quad D'_z = \begin{cases} D_z - D_y & \text{if } D_y = \{y\} \\ D_z & \text{otherwise} \end{cases}
\]
enforces the constraint \( y \neq z \). Its domains extension is \( f_{NE}^+ : T_x \times T_y \times T_z \to T_x \times T_y \times T_z \), having \( f_{NE}(D_x, D_y, D_z) = (D_x', D_y', D_z') \) 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 \( f_{LT1} : T_x \times T_y \to T_x \), and \( f_{LT2} : T_x \times T_y \to T_y \), having

\[
\begin{align*}
  f_{LT1}(D_x, D_y) &= \{ x \in D_x \mid \exists y \in D_y \; x < y \} \\
  f_{LT2}(D_x, D_y) &= \{ y \in D_y \mid \exists x \in D_x \; x < y \}
\end{align*}
\]

The domains extensions of these two functions have the same signature as \( f_{NE}^+ \). We have \( f_{LT1}^+(D_x, D_y, D_z) = (D_x', D_y', D_z') \), 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

\[
(x \leq y, y \neq z ; x \in D_x, y \in D_y, z \in D_z ; D_x, D_y, D_z \in \mathcal{Z} ; A_x, A_y, A_z)
\]

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 \( f_{LE2}^+ \). \( \square \)

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 \( T \), 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, absence 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

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

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 $f$
    $G := G - \{f\}$
    $D'[t] := D[t] \cap f(D[s])$
    $G := G \cup \text{update}(F, D, D')$
    $D[t] := D'[t]$
    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

\[ f : T_1 \times \ldots \times T_n \rightarrow \mathcal{P}(T_1 \times \ldots \times T_n) \]

such that if \( \langle D_1, \ldots, D_n \rangle \in T_1 \times \ldots \times T_n \) 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

\[ \{ D_1' \times \ldots \times D_n' | \langle D_1', \ldots, D_n' \rangle \in f(D_1, \ldots, D_n) \} \]

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 \( f : \mathbb{Z}^n \rightarrow \mathcal{P}(\mathbb{Z}^n) \) having

\[ f(D_1, \ldots, D_n) = \{ \langle D_1, \ldots, D_{j-1}, \{ x \}, D_{j+1}, \ldots, D_n \} | x \in D_j \} \]

with \( j = \min(\{i | 1 \leq i \leq n, |D_i| > 1\}) \) is a domain branching function for an ECSP of the form (2.2) with \( T_i = \mathbb{Z} \) and \( A_i = [\mathbb{Z}] \), 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.
Chapter 2. Constraint Solving

<table>
<thead>
<tr>
<th>variable selection strategy</th>
<th>value selection strategy</th>
</tr>
</thead>
<tbody>
<tr>
<td>chronological</td>
<td>the variables are in some explicit order, and the first variable $x_i$ in this order whose domain $D_i$ is not yet in $A_i$ is selected.</td>
</tr>
<tr>
<td>fail-first</td>
<td>selects a variable with the largest domain size. Used primarily with domain type $Z$.</td>
</tr>
<tr>
<td>fail-last</td>
<td>the opposite of the previous strategy.</td>
</tr>
<tr>
<td>round robin</td>
<td>selects the variable $x_i$ whose domain $D_i$ is not in $A_i$ that has least recently been selected</td>
</tr>
</tbody>
</table>

| Table 2.1: Variable selection strategies |

| enumeration                  | used primarily with domain types $Z$ and $B$: a subdomain of size 1 is created for each of the values in the original domain. |
| L/R-enumeration              | used with domain types $Z$ and $I$: 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 $I$ and $F$. 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:

\[
\text{failed}((D_1, \ldots, D_n)) \text{ is false iff } D_1 \neq \emptyset, \ldots, D_n \neq \emptyset.
\]

\[
\text{final}((D_1, \ldots, D_n)) \text{ is true iff } D_1 \in A_1, \ldots, D_n \in A_n.
\]
parameters: function \textit{select},
function \textit{propagate}.

input: an ECSP \( P := (C ; x_1 \in D_1, \ldots, x_n \in D_n ; \tau_1, \ldots, \tau_n ; A_1, \ldots, A_n) \),
a domain branching function \( f \),
a set \( R \) of domain reduction functions,

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

\[ \langle C[D'_1, \ldots, D'_n] ; x_1 \in D'_1, \ldots, x_n \in D'_n ; \tau_1, \ldots, \tau_n ; A_1, \ldots, A_n \rangle \]
is a \( \gamma \) solved form of \( P \), where \( \gamma \) is the notion of consistency enforced
for the constraints in \( C \) by \textit{propagate} and \( R \).

\[
F := \{(D_1, \ldots, D_n)\}
\]
\[
S := \emptyset
\]
while \( F \neq \emptyset \) do
\[
select \ D_w \in F
\]
\[
F := F \setminus \{D_w\}
\]
\[
D'_w := \text{propagate}(D_w, R)
\]
if \( \neg \text{failed}(D'_w) \)
then
if \( \text{final}(D'_w) \)
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 \textit{solution}, \textit{failure}, or \textit{internal} node of the search tree. If after constraint propagation all domains are in their respective set of final domains, the \textit{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 \textit{failed} predicate holds, which characterizes the node as a failure. Note that \textit{failed} and \textit{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 \textit{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 \textit{stack}, which implies that we always select one of the most recent additions, the algorithm essentially performs a \textit{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 \textit{queue}, selecting always one of the oldest additions, results in a \textit{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 not 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 Bessièrè. 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
2 8 8

Chapter 2. Constraint Solving

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 $F$ 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 \( f : \mathcal{F} \times \mathcal{I} \rightarrow \mathcal{F} \times \mathcal{I} \), having \( f((D_r, D_i)) = (D'_r, D'_i) \), where \( D'_r := \text{hull}(D_r \cap D_i) \) and \( D'_i := [\text{min}(D_i \cap D_r)] .. [\text{max}(D_i \cap D_r)] \), 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 breadth-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 pre-processing 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 \textit{choose} and \textit{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.