Chapter 14

Finite Domain Constraint Programming Systems

Christian Schulte and Mats Carlsson

One of the main reasons why constraint programming quickly found its way into applications has been the early availability of usable constraint programming systems. Given the wide range of applications using constraint programming it is obvious that one of the key properties of constraint programming systems is their provision of widely reusable services for constructing constraint-based applications.

A constraint programming system can be thought of as providing a set of reusable services. Common services include constraint propagation, search, and services for interfacing to the system. This chapter looks in more detail at which services are provided by a constraint programming system and in particular what are the key principles and techniques in constructing and coordinating these services.

To give the chapter a clear focus and a reasonably uniform presentation, we mostly restrict our attention to propagation-based finite domain constraint programming systems. That is, systems that solve problems using constraint propagation involving variables ranging over some finite set of integers. The focus on finite domain constraint programming systems coincides with both practical relevance and known principles and techniques: systems at least offer services for finite domains; much of the known principles and techniques have been conceived and documented for finite domains.

Essential for a system in providing the services mentioned above are some important abstractions (or objects) to be implemented by a system: variables, implementations for constraints, and so on. Important abstractions for propagation, search, and interfacing are as follows.

Constraint propagation. To perform constraint propagation a system needs to implement *variables* ranging over finite domains. Constraints expressing a relation among variables are implemented by *propagators*: software abstractions which by execution perform constraint propagation. Finally, a *propagation engine* coordinates the execution of propagators in order to deliver constraint propagation for a collection of constraints.

Search. Search in a finite domain constraint programming system has two principal dimensions. The first dimension is concerned with how to describe the search tree, typically achieved by a *branching* or *labeling*. The second dimension is concerned with how to explore a search tree, this is typically achieved by an *exploration strategy* or *search strategy*. Any system implementing search must provide a *state restoration* service which maintains computation states for the nodes of the search tree.

Interfacing. A system must provide access to the services mentioned above so that applications can use them. Depending on the underlying constraint programming system, the services can be tightly integrated into some host language (such as Prolog) or being provided by some library (pioneered by ILOG Solver as a C++-based library).

Different levels of interfaces can be observed with different systems. Clearly, all systems offer at least interfaces which allow to use the system-provided services in applications. Even though the constraints, search engines, and so on provided by a system are sufficient for many applications, some applications might require more. For these applications, a system must be extensible by new propagators for possibly new constraints, new branching strategies, and new exploration strategies.

Chapter structure. The structure of this chapter is as follows. The next section gives a simple architecture for finite domain constraint programming systems. It describes *what* a system computes and *how* computation is organized in principle. The following two Sections 14.2 and 14.3 describe how systems implement this architecture. Section 14.2 describes how propagation is implemented while the following section describes how search is implemented. An overview over existing finite domain constraint programming systems is provided by Section 14.4. The last section of this chapter summarizes the key aspects of a finite domain constraint programming system and presents current and future challenges.

14.1 Architecture for Constraint Programming Systems

This section defines a simple architecture of a finite domain constraint programming system. The section describes *what* results a system computes and *how* it computes them. The focus is on the basic entities and principles that are used in systems; the actual implementation techniques used in systems are discussed in the following sections.

Much of the content follows the presentation in [58]. Essential parts of the architecture described here have been first identified and discussed by Benhamou in [13].

14.1.1 Propagation-based Constraint Solving

This section defines terminology and *what* a system actually computes.

Domains. A *domain* D is a complete mapping from a fixed (countable) set of variables $\mathcal{V}$ to finite sets of integers. A domain D is *failed*, if $D(x) = \emptyset$ for some $x \in \mathcal{V}$. A variable $x \in \mathcal{V}$ is *fixed* by a domain D, if $|D(x)| = 1$. The *intersection* of domains D_1 and D_2, denoted $D_1 \sqcap D_2$, is defined by the domain $D(x) = D_1(x) \cap D_2(x)$ for all $x \in \mathcal{V}$.

A domain D_1 is *stronger* than a domain D_2, written $D_1 \sqsubseteq D_2$, if $D_1(x) \subseteq D_2(x)$ for all $x \in \mathcal{V}$.

We use range notation $[l, u]$ for the set of integers $\{n \in \mathbb{Z} \mid l \leq n \leq u\}$.

Assignments and constraints. An *integer assignment* a is a mapping of variables to integer values, written $\{x_1 \mapsto n_1, \ldots, x_k \mapsto n_k\}$. We extend the assignment a to map expressions and constraints involving the variables in the natural way.

Let vars be the function that returns the set of variables appearing in an assignment. In an abuse of notation, we define an assignment a to be an element of a domain D, written $a \in D$, if $a(x_i) \in D(x_i)$ for all $x_i \in \mathrm{vars}(a)$.

The *minimum* and *maximum* of an expression e with respect to a domain D are defined as $\min_D e = \min\{a(e) \mid a \in D\}$ and $\max_D e = \max\{a(e) \mid a \in D\}$.

A *constraint* c over variables $x_1, \ldots, x_n$ is a set of assignments a such that $\mathrm{vars}(a) = \{x_1, \ldots, x_n\}$. We also define $\mathrm{vars}(c) = \{x_1, \ldots, x_n\}$.

Propagators. A constraint is defined extensionally by a collection of assignments for its variables. Typical systems do not compute with these extensional representations directly for two reasons:

1. Representing all possible assignments of a constraint might take too much space to be feasible (exponential space in the number of variables). In particular, space becomes an issue if $\mathrm{vars}(c)$ contains more than two variables.

2. Common constraints have a certain *structure* (such as representing a linear equation constraint or an alldifferent constraint). Representing a constraint extensionally will make it difficult or even impossible to take advantage of this underlying structure.

Constraint propagation systems *implement* a constraint c by a collection of *propagators*. Propagators are also known as filters (implemented by some *filtering algorithm*) and *narrowing operators* [13]. A propagator p is a function that maps domains to domains. In order to make constraint propagation well-behaved (to be discussed in Section 14.1.2), propagators are *decreasing* and *monotonic*.

- A propagator p must be a *decreasing* function: $p(D) \sqsubseteq D$ for all domains D. This property is obvious and guarantees that constraint propagation only removes values.

- A propagator p must be a *monotonic* function: $p(D_1) \sqsubseteq p(D_2)$ whenever $D_1 \sqsubseteq D_2$. That is, application of p to stronger domains also yields stronger domains.

Propagators must faithfully implement constraints. A propagator p is *correct* for a constraint c iff it does not remove any assignment for c. That is, for all domains D

$$\{a \in D\} \cap c = \{a \in p(D)\} \cap c$$

This is a very weak restriction, for example the identity propagator i with $i(D) = D$ for all domains D is correct for all constraints c.

A propagator must also provide sufficient propagation to distinguish solutions from non-solutions. Hence, a set of propagators P is *checking* for a constraint c, if for domains D where all variables $\mathrm{vars}(c)$ are fixed the following holds: $p(D) = D$ for all $p \in P$, iff the unique assignment $a \in D$ where $\mathrm{vars}(a) = \mathrm{vars}(c)$ is a solution of c ($a \in c$). In

other words, all domains D corresponding to a solution of a constraint c on its variables are required to be a fixpoint of the propagator p.

A set of propagators P *implements* a constraint c, if all $p \in P$ are correct for c and P is checking for c. We denote this fact by $P = \mathrm{prop}(c)$.

Systems consider sets of propagators rather than a single propagator as implementations of constraints to have more freedom in implementing a constraint. For example, a common way to implement simple constraints is by indexicals (to be discussed in Section 14.2.5): a collection of indexical propagators is used to implement a single constraint. On the other hand, systems provide global constraints with the idea that a single propagator implements a constraint involving many variables.

Note that only very little propagation is required for a set of propagators to be checking (as the term suggests, checking is sufficient; no actual propagation is required). The level of consistency provided by a propagator set is irrelevant to this model. Consistency levels only provide a convenient way to refer to the strength of propagators. As far as achieving good propagation is concerned, it does not matter whether a propagator set corresponds to a predefined consistency level. What matters is that the propagator set offers a good compromise between strength and cost of propagation.

To simplify our presentation we assume that propagators are defined for all variables $\mathcal{V}$. In a system, a propagator p will be only interested in some variables: the variables $\mathrm{vars}(c)$ of the constraint c that is implemented by p. Two sets of variables which are important are the *input* and *output* variables.

The *output* variables $\mathrm{output}(p) \subseteq \mathcal{V}$ of a propagator p are the variables changed by the propagator: $x \in \mathrm{output}(p)$ if there exists a domain D such that $p(D)(x) \neq D(x)$.

The *input* variables $\mathrm{input}(p) \subseteq \mathcal{V}$ of a propagator p is the smallest subset $V \subseteq \mathcal{V}$ such that for all domains D_1 and D_2: $D_1(x) = D_2(x)$ for all $x \in V$ implies that $D_1'(x) = D_2'(x)$ for all $x \in \mathrm{output}(p)$ where $D_1' = D_2 \sqcap p(D_1)$ and $D_2' = D_1 \sqcap p(D_2)$. Only the input variables are useful in computing the application of the propagator to the domain. We say that a propagator p *depends* on a variable x, if $x \in \mathrm{input}(p)$.

Example 1 (Propagators)**.** For the constraint $c \equiv x_1 \leq x_2 + 1$ the function p_1 defined by $p_1(D)(x_1) = \{n \in D(x_1) \mid n \leq \max_D x_2 + 1\}$ and $p_1(D)(x) = D(x), x \neq x_1$ is a correct propagator for c. Its output variables are $\{x_1\}$ and its input variables are $\{x_2\}$. Let $D_1(x_1) = \{1, 5, 8\}$ and $D_1(x_2) = \{1, 5\}$, then $p_1(D_1) = D_2$ where $D_2(x_1) = D_2(x_2) = \{1, 5\}$.

The propagator p_2 defined as $p_2(D)(x_2) = \{n \in D(x_2) \mid n \geq \min_D x_1 - 1\}$ is another correct propagator for c. Here and in the following we assume that a propagator is defined as identity for variables not mentioned in the definition. Its output variables are $\{x_2\}$ and input variables $\{x_1\}$.

The set $\{p_1, p_2\}$ is checking for c. For example, the domain $D(x_1) = D(x_2) = \{2\}$ corresponding to a solution of c is a fixpoint of both propagators. The non-solution domain $D(x_1) = \{2\}$, $D(x_2) = \{0\}$ is not a fixpoint (of either propagator).

Now we are in the position to describe what a constraint programming system computes. A *propagation solver* for a set of propagators P and some initial domain D, $\mathrm{solv}(P, D)$, finds the greatest mutual fixpoint of all the propagators $p \in P$. In other words, $\mathrm{solv}(P, D)$ returns a new domain defined by

$$\mathrm{solv}(P, D) = \mathrm{gfp}(\lambda d.\, \mathrm{iter}(P, d))(D) \qquad \mathrm{iter}(P, D) = \bigsqcap_{p \in P} p(D)$$

```
    propagate(P_f,P_n,D)
 1: N ← P_n
 2: P ← P_f ∪ P_n
 3: while N ≠ ∅ do
 4:    p ← select(N)
 5:    N ← N − {p}
 6:    D' ← p(D)
 7:    M ← {x ∈ V | D(x) ≠ D'(x)}
 8:    N ← N ∪ {p' ∈ P | input(p') ∩ M ≠ ∅}
11:    D ← D'
12: return D
```

Figure 14.1: Basic propagation engine propagate.

where gfp denotes the greatest fixpoint w.r.t $\sqsubseteq$ lifted to functions.

14.1.2 Performing Propagation

A constraint programming system is concerned with performing propagation and search. In this section, we consider the propagation engine and postpone the discussion of search to Section 14.1.5.

The *propagation engine* propagate shown in Figure 14.1 computes $\mathrm{solv}(P, D)$ for a given set of propagators P and a domain D. Note that lines 9 and 10 are left out for an extension to be discussed later. The engine propagate takes two sets of propagators as input where P_f contains propagators already known to be at fixpoint for D. This is an important feature to obtain incremental propagation during search. If no fixpoint knowledge on propagators is available, it is safe to execute propagate$(\emptyset, P, D)$.

The algorithm uses a set N of propagators to apply (N stands for *not* known to be at fixpoint). Initially, N contains all propagators from P_n. Each time the while loop is executed, a propagator p is deleted from N, p is applied, and the set of *modified variables* M is computed. All propagators that share input variables with M are added to the set of propagators not known to be at fixpoint. Adding a propagator p to the set N is called *scheduling* p.

An invariant of the engine is that at the while statement $p(D) = D$ for all $p \in P - N$. The loop terminates, since in each iteration either a propagator is removed from N or a strictly smaller domain D' is computed (as a propagator is a decreasing function and there are only finitely many domains). After termination, the invariant yields that $D' =$ propagate(P_f, P_n, D) is a fixpoint for all $p \in P_f \cup P_n$, that is propagate$(P_f, P_n, D) = \mathrm{solv}(P_f \cup P_n, D)$.

As mentioned, the fact that a propagator is a decreasing function is essential for termination. The fact that a propagator is monotonic guarantees that propagate(P_f, P_n, D) actually computes $\mathrm{solv}(P_f \cup P_n, D)$ and that the order in which propagators are executed does not change the result of propagate(P_f, P_n, D). The propagation engine (assuming $P_f = \emptyset$) is more or less equivalent to the propagation algorithm of Apt [7, Section 7.1.3].

The engine is geared at simplicity, one particular simplification being that it does not pay particular attention to failed domains. That is, even though the domain D becomes failed, propagation continues until a domain is computed that is both failed and a fixpoint for all propagators in P. A concrete system might optimize this, as is discussed in Section 14.1.6.

Note that propagate leaves undefined how a propagator p is selected from N. Strategies for selecting propagators are discussed in Section 14.2.1.

Example 2 (Propagation)**.** Consider the propagator p_1 for the constraint $x_1 \leq x_2$ defined by

$$\begin{aligned} p_1(D)(x_1) &= \{n \in D(x_1) \mid n \leq \max_D x_2\} \\ p_1(D)(x_2) &= \{n \in D(x_2) \mid n \geq \min_D x_1\} \end{aligned}$$

Also consider the propagator p_2 for the constraint $x_1 \geq x_2$ defined analogously.

Let us start propagation for the domain D_0 with $D(x_1) = \{0, 2, 6\}$ and $D(x_2) = \{-1, 2, 4\}$ by executing propagate$(\emptyset, \{p_1, p_2\}, D_0)$. This initializes both P and N to $\{p_1, p_2\}$.

Let us assume that p_1 is selected for propagation. Then, p_1 is removed from N and yields $D'(x_1) = \{0, 2\}$ and $D'(x_2) = \{2, 4\}$. The set of modified variables M is $\{x_1, x_2\}$ and hence after this iteration N is $\{p_1, p_2\}$.

In the second iteration, let us assume that p_2 is selected for propagation. This yields $D'(x_1) = D'(x_2) = \{2\}$. Again, the set of modified variables M is $\{x_1, x_2\}$ and hence N is $\{p_1, p_2\}$.

Assume that in the next iteration p_1 is selected. Now D is already a fixpoint of p_1 and the set of modified variables M is empty. This in turn means that N is just $\{p_2\}$ after this iteration.

The last iteration selects p_2 for execution, does not modify the domain, and N becomes empty. Hence, propagate$(\emptyset, \{p_1, p_2\}, D_0)$ returns a domain D with $D(x_1) = D(x_2) = \{2\}$.

14.1.3 Improving Propagation

The propagation engine shown in Figure 14.1 is naive in that it does not exploit additional information about propagators. Improved engines being the base for existing systems try to avoid propagator execution based on the knowledge whether a domain is a fixpoint for a propagator.

In the following we discuss common properties of propagators, which help to avoid useless execution. How a system implements these properties (or detects these properties) is discussed in Section 14.2.

Idempotent propagators. Assume that a propagator p has actually made the domain stronger, that is, $D' \neq D$. This means that there exists a variable $x \in \mathcal{V}$ for which $D'(x) \subset D(x)$. Assume further that $x \in \mathrm{input}(p)$. Hence p will be included in N.

Quite often, however, propagators happen to be idempotent: a propagator p is idempotent, if the result of propagation is a fixpoint of p. That is, $p(p(D)) = p(D)$ for all domains D.

Hence, to avoid inclusion of an idempotent propagator p, the following lines can be added to the propagation engine after line 8:

```
 9: if (p idempotent) then
10:    N ← N − {p}
```

Example 3 (Idempotent propagators)**.** The propagators p_1 and p_2 from Example 2 are both idempotent as can be seen easily.

Taking this into account, propagation takes only three iterations. If the same selection of propagators is done as above, then N is $\{p_2\}$ after the first iteration and $\{p_1\}$ after the second iteration.

Note that in particular all domain consistent propagators are idempotent.

Entailment. An idempotent propagator can be exempted from being included in N directly after p has been applied. A much stronger property for a propagator p is *entailment*. A propagator p is *entailed* by a domain D, if all domains D' with $D' \sqsubseteq D$ are fixpoints of p, that is $p(D') = D'$. This means that as soon as a propagator p becomes entailed, it can be safely deleted from the set of propagators P.

Example 4 (Entailed propagator)**.** Consider the propagator p_1 for the constraint $x_1 \leq x_2$ from Example 2. Any domain D with $\max_D x_1 \leq \min_D x_2$ entails p_1.

Propagator rewriting. During propagation the domain D might fix some variables in $\mathrm{input}(p) \cup \mathrm{output}(p)$ of a propagator p. Many propagators can be replaced by simpler propagators after some variables have become fixed.

Example 5 (Propagator rewriting for fixed variables)**.** Consider the propagator p with $p \in \mathrm{prop}(c)$ where $c \equiv x_1 + x_2 + x_3 \leq 4$:

$$p(D)(x_1) = \{n \in D(x_1) \mid n \leq \max_D(4 - x_2 - x_3)\}$$

Assume that propagation has computed a domain D which fixes x_2 to 3 (that is, $D(x_2) = \{3\}$). Then p can be replaced by the simpler (and most likely more efficient) propagator p' defined by:

$$p'(D)(x_1) = \{n \in D(x_1) \mid n \leq \max_D(1 - x_3)\}$$

A propagator p can always be rewritten to a propagator p' for a domain D, if $p(D') = p'(D')$ for all domains D' with $D' \sqsubseteq D$. This means that propagator rewriting is not only applicable to domains that fix variables.

This is for example exploited for the "type reduction" of [52] where propagators are rewritten as more knowledge on domains (there called types) becomes available. For example, the implementation of $x_0 = x_1 \times x_2$ will be replaced by a more efficient one, when all elements in $D(x_1)$ and $D(x_2)$ are non-negative.

14.1.4 Propagation Events

For many propagators it is simple to decide whether they are still at a fixpoint for a changed domain based on *how* the domain has changed. How a domain changes is described by *propagation events* (or just *events*).

Example 6 (Disequality propagators)**.** Consider the propagator p with $\{p\} = \mathrm{prop}(c)$ for the constraint $c \equiv x_1 \neq x_2$:

$$\begin{array}{rcl} p(D)(x_1) & = & D(x_1) - \mathrm{single}(D(x_2)) \\ p(D)(x_2) & = & D(x_2) - \mathrm{single}(D(x_1)) \end{array}$$

where $\mathrm{single}(N)$ for a set N is defined as N if $|N| = 1$ and $\emptyset$ otherwise.

Clearly, any domain D with $|D(x_1)| > 1$ and $|D(x_2)| > 1$ is a fixpoint of D. That is, p only needs to be applied if x_1 or x_2 are fixed.

Similarly, the propagator p_1 from Example 1 only needs to be applied to a domain D if $\max_D x_2$ changes and p_2 from the same example needs to be applied to a domain D if $\min_D x_1$ changes.

Assume that the domain D changes to the domain $D' \sqsubseteq D$. The usual events defined in a constraint propagation system are:

- $\mathrm{fix}(x)$: the variable x becomes fixed.
- $\mathrm{minc}(x)$: the minimum of variable x changes.
- $\mathrm{maxc}(x)$: the maximum of variable x changes.
- $\mathrm{any}(x)$: the domain of variable x changes.

Clearly the events overlap. Whenever a $\mathrm{fix}(x)$ event occurs then a $\mathrm{minc}(x)$ event, a $\mathrm{maxc}(x)$ event, or both events must also occur. If any of the first three events occur then an $\mathrm{any}(x)$ event occurs. This is captured by the following definition of $\mathrm{events}(D, D')$ for domains $D' \sqsubseteq D$:

$$\begin{array}{rcl} \mathrm{events}(D, D') & = & \{\mathrm{any}(x) \mid D'(x) \subset D(X)\} \\ & \cup & \{\mathrm{minc}(x) \mid \min_{D'} x > \min_D x\} \\ & \cup & \{\mathrm{maxc}(x) \mid \max_{D'} x < \max_D x\} \\ & \cup & \{\mathrm{fix}(x) \mid |D'(x)| = 1 \text{ and } |D(x)| > 1\} \end{array}$$

Events satisfy an important monotonicity condition: suppose domains $D'' \sqsubseteq D' \sqsubseteq D$, then

$$\mathrm{events}(D, D'') = \mathrm{events}(D, D') \cup \mathrm{events}(D', D'').$$

So an event occurs on a change from D to D'' iff it occurs in the change from D to D' or from D' to D''.

Example 7 (Events)**.** Let $D(x_1) = \{1, 2, 3\}$, $D(x_2) = \{3, 4, 5, 6\}$, $D(x_3) = \{0, 1\}$, and $D(x_4) = \{7, 8, 10\}$ while $D'(x_1) = \{1, 2\}$, $D'(x_2) = \{3, 5, 6\}$ $D'(x_3) = \{1\}$ and $D'(x_4) = \{7, 8, 10\}$. Then $\mathrm{events}(D, D')$ is

$$\{\mathrm{maxc}(x_1), \mathrm{any}(x_1), \mathrm{any}(x_2), \mathrm{fix}(x_3), \mathrm{minc}(x_3), \mathrm{any}(x_3)\}$$

For a propagator p, the set $\mathrm{es}(p) \subseteq \{\mathrm{fix}(x), \mathrm{minc}(x), \mathrm{maxc}(x), \mathrm{any}(x) \mid x \in \mathcal{V}\}$ of events is an *event set* for p if the following two properties hold:

1. For all domains D' and D with $D' \sqsubseteq D$ and $D(x) = D'(x)$ for all $x \in \mathcal{V} - \mathrm{input}(p)$: if $p(D) = D$ and $p(D') \neq D'$, then $\mathrm{es}(p) \cap \mathrm{events}(D, D') \neq \emptyset$.

2. For all domains D with $p(D) \neq p(p(D))$: $\mathrm{es}(p) \cap \mathrm{events}(D, p(D)) \neq \emptyset$.

The first clause of the definition captures the following. If the domain D is a fixpoint and the stronger domain D' (stronger only on the input variables) is not a fixpoint for a propagator p, then an event occurring from changing the domain D to D' must be included in the event set $\mathrm{es}(p)$. The second clause refers to the case when a propagator p does not compute a fixpoint (that is, $p(d) \neq p(p(D))$). In this case, an event must occur when the domain changes from D to $p(D)$. Note that the second clause never applies to an idempotent propagator.

An event set plays an analogous role to the set of input variables: if an event from the event set occurs when going from a domain D to a domain D', the propagator is no longer guaranteed to be at a fixpoint and must be re-applied.

Note that the definition of an event set is rather liberal as the definition does not require the event set to be the smallest set: any set that guarantees re-application is allowed. In particular, for any propagator p the set $\{\mathrm{any}(x) \mid x \in \mathrm{input}(p)\}$ is an event set: this event set makes propagation behave as if no events at all are considered. However, an implementation will try to use event sets that are as small as possible.

Example 8 (Event sets)**.** The propagator p_1 from Example 2 depends on the event set $\{\mathrm{minc}(x_1), \mathrm{maxc}(x_2)\}$. The propagator p from Example 6 depends on the event set $\{\mathrm{fix}(x_1), \mathrm{fix}(x_2)\}$.

Now it is obvious how the propagation engine from Figure 14.1 can take advantage of events: instead of considering the set of modified variables and the input variables of a propagator for deciding which propagators are to be included into N, consider the events and an event set for a propagator. In other words, replace line 8 by:

8: $N \leftarrow N \cup \{p' \in P \mid \mathrm{es}(p') \cap \mathrm{events}(D, D') \neq \emptyset\}$

14.1.5 Performing Search

A constraint programming system evaluates $\mathrm{solv}(P, D)$ during search. We assume an execution model for solving a constraint problem with a set of constraints C and an initial domain D_0 as follows. We execute the procedure search$(\emptyset, P, D_0)$ for an initial set of propagators $P = \bigcup_{c \in C} \mathrm{prop}(c)$. This procedure (shown in Figure 14.2) serves as an architecture of a constraint programming system.

The procedure requires that D be a fixpoint for all propagators in P_f (f for fixpoint). The propagators included in P_n do not have this requirement (n for not at fixpoint). This partitioning of propagators is used for incremental propagation with respect to recursive calls to search (as discussed below).

The somewhat unusual definition of search is quite general. The default *branching* strategy (also known as *labeling* strategy) for many problems is to choose a variable x such that $|D(x)| > 1$ and explore $x = \min_D x$ or $x \geq \min_D x + 1$. This is commonly thought of as changing the domain D for x to either $\{\min_D x\}$ or $\{n \in D(x) \mid n > \min_D x\}$. Branching based on propagator sets for constraints allows for more general strategies, for example $x_1 \leq x_2$ or $x_1 > x_2$.

```
    search(P_f, P_n, D)
 1: D ← propagate(P_f, P_n, D)
 2: if D is failed domain then
 3:    return false
 4: if ∃x ∈ V.|D(x)| > 1 then
 5:    choose {c_1, ..., c_m} where C ∧ D ⊨ c_1 ∨ ··· ∨ c_m
 6:    for all i ∈ [1, m] do
 7:       if search(P_f ∪ P_o, prop(c_i), D) then
 8:          return true
 9:    return false
10: return true
```

Figure 14.2: Architecture of constraint programming system.

Note that search has two dimensions: one describes how the search tree looks and the other describes how the search tree is explored. In the above architecture the selection of the c_i together with the selection of propagators $\mathrm{prop}(c_i)$ for the c_i describes the shape of the search tree. The sets $\mathrm{prop}(c_i)$ we refer to as *alternatives* and the collection of all alternatives is called *choice point*. Completely orthogonal is how the search tree is explored. Here, the architecture fixes exploration to be depth-first. Exploration is discussed in more detail in 14.3.

Note that search performs incremental propagation in the following sense: when calling propagate only the propagators $\mathrm{prop}(c_i)$ for the alternatives are not known to be at a fixpoint.

14.1.6 Implementing the Architecture

This section discusses general approaches to implementing the architecture for a constraint programming system introduced above. The focus is on what needs to be implemented and how the architecture (as an abstraction of a system) relates to a real system.

Detecting failure and entailment. In our architecture, failure is only detected inside search after returning from propagate by testing whether the domain obtained by propagation is failed. It is clear that a system should optimize detecting failure such that no propagation is wasted if a domain becomes failed and that no inspection of a domain is required to detect failure.

A typical way to make the detection of failure or entailment of a propagator more efficient is to let the propagator not only return a domain but also some status information describing whether propagation has resulted in a failed domain or whether the propagator has become entailed.

Implementing domains. The architecture describes that a propagator takes a domain as input and returns a new domain. This is too memory consuming for a real system. Instead, a system maintains a single data structure implementing one domain and propagators update this single domain when being applied.

Inspecting propagate in Figure 14.1 it becomes clear that maintaining a single domain is straightforward to achieve. The only reason for having a domain D and D' is in order to be able to identify the modified variables M.

State restoration. For propagation, systems maintain a single domain as has been argued above. However, a single domain becomes an issues for search: when calling search recursively as in Figure 14.2 and a domain is not transferred as a copy, backtracking (that is, returning from the recursive call) needs to restore the domain.

State restoration is not limited to domains but also includes private states of propagators and also whether propagators have become entailed. State restoration is a key service required in a constraint programming system and is discussed in Section 14.3.2 in detail.

Finding dependent propagators. After applying a propagator, propagate must compute the events (similar to the set of modified variables) in order to find all propagators that depend on these events. Clearly, this requires that a system be able to compute the events and find the dependent propagators efficiently.

Variables for propagators. In addition to the fact that propagators update a single domains rather than returning domains, implementations need to be careful in how many variables are referenced by a propagator. In our architecture, propagators are defined for all variables in $\mathcal{V}$. However, from the above discussion it is clear that a propagator p is only concerned with variables $\mathrm{input}(p) \cup \mathrm{output}(p)$. Quite often, the variables in $\mathrm{input}(p) \cup \mathrm{output}(p)$ are called the *parameters* of p.

A system will implement a propagator p such that it maintains its $\mathrm{input}(p) \cup \mathrm{output}(p)$ in some datastructure, typically as an array or list of variables. While most of the properties discussed for our architecture readily carry over to this extended setup, the case of multiple occurrences of the same variable in the datastructure maintained by a propagator needs special attention.

Multiple variable occurrences and unification. Depending on the actual system, multiple occurrences may both be common and appear dynamically. Here, dynamic means that variable occurrences for a propagator p might become the same during some computation not performed by p itself. This is typically the case when the constraint programming system is embedded in a constraint logic programming host language featuring *unification* for logical variables. Unification makes two variables x and y equal without the requirement to assign the variables a particular value. In this case the variables x and y are also said to be *aliased*.

The main issues with multiple occurrences of the same variable are that they make (a) detection of idempotence and (b) achieving good propagation more difficult, as is discussed in Section 14.2.3.

Private state. In our architecture, propagators are functions. In systems, propagators often need to maintain some *private state*. Private state is for example used to achieve incrementality, more information is given in Section 14.2.3.

14.2 Implementing Constraint Propagation

In this section, we detail the software architecture introduced on an abstract level in Section 14.1. This architecture can be roughly divided into *domain variables*, data structures implementing the problem variables; *propagators*, coroutines implementing the problem constraints by executing operations on the variables that it constrains; and *propagation services*, callable by the propagators in order to achieve the overall fixpoint computation.

Domain variables and propagators form a bipartite graph: every propagator is linked to the domain variables that it constrains, and every domain variable is linked to the propagators that constrain it.

In the following, we will not discuss issues related to the state restoration policy used, this is discussed in Section 14.3.2.

14.2.1 Propagation Services

From an operational point of view, a constraint programming system can be described in terms of coroutines (propagators) and events (domain changes). Propagators raise events, which leads to other propagators being resumed, until a fixpoint is reached. The management of events and selection (scheduling) of propagators are the main tasks of the propagation services.

Events. Most integer propagation solvers use the events defined in Section 14.1.4, although some systems collapse $\mathrm{minc}(x)$ and $\mathrm{maxc}(x)$ into a single event (for example, ILOG Solver [32]). Choco [36] maintains an event queue and interleaves propagator execution with events causing more propagators to be added to the queue.

Other events than those discussed in Section 14.1.4 are also possible. For example, *neq*(x, n): the variable x can no longer take the value n, that is, $n \in D(x)$ and $n \notin D'(x)$ for domains D and D'. These events have been used in e.g. B-Prolog [75].

Selecting the next propagator. It is clear that the number of iterations performed by the propagation engine shown in Figure 14.1 depends also on which propagator is selected to be applied next. The selection policy is system-specific, but the following guiding principles can be observed:

- Events providing much information, for example fix events, yield quicker reaction than events providing less information. This captures selecting propagators according to expected *impact*.
- Propagators with low complexity, e.g. small arithmetic propagators, are given higher priority than higher complexity propagators. This captures selecting propagators according to *cost*.
- Starvation is avoided: no event or propagator should be left unprocessed for an unbounded amount of time (unless there is a propagator of higher priority or higher impact to run). This is typically achieved by selecting propagators for execution in a last-in last-out fashion (that is, maintaining the set N in Figure 14.2 as a queue).

Most systems have some form of static priorities, typically using two priority levels (for example, SICStus Prolog [35], Mozart [43]). The two levels are often not entirely based on cost: in SICStus Prolog all indexicals (see Section 14.2.5) have high priority and global constraints lower priority. While ECL^{i}PS^e [14, 28] supports 12 priority levels, its finite domain solver also uses only two priority levels where another level is used to support constraint debugging. A similar, but more powerful approach is used by Choco [36] using seven priority levels allowing both LIFO and FIFO traversal.

Schulte and Stuckey describe a model for dynamic priorities based on the complexity of a propagator in [58]. They describe how priorities can be used to achieve staged propagation: propagators dynamically change priority to first perform weak and cheap propagation and only later perform stronger and more complex propagation. Another model for priorities in constraint propagation based on composition operators is [25]. This model runs all propagators of lower priority before switching propagation back to propagators of higher priority.

Prioritizing particular operations during constraint propagation is important in general. For (binary) arc consistency algorithms, ordering heuristics for the operations performed during propagation can reduce the total number of operations required [72]. For interval narrowing, prioritizing constraints can avoid slow convergence, see for example [38].

14.2.2 Variable Domains

In a reasonable software architecture, propagators do not manipulate variable domains directly, but use the relevant propagation services. These services return information about the domain or update the domain. In addition, they handle failure (the domain becomes empty) and control propagation.

Value operations. A *value operation* on a variable involves a single integer as result or argument. We assume that a variable x with $D = \mathrm{dom}(x)$ provides the following value operations: $x.\mathrm{getmin}()$ returns $\min(D)$; $x.\mathrm{getmax}()$ returns $\max(D)$; $x.\mathrm{hasval}(n)$ returns $n \in D$; $x.\mathrm{adjmin}(n)$ updates $\mathrm{dom}(x)$ to $\{m \in D \mid m \geq n\}$; $x.\mathrm{adjmax}(n)$ updates $\mathrm{dom}(x)$ to $\{m \in D \mid m \leq n\}$; and $x.\mathrm{excval}(n)$ updates $\mathrm{dom}(x)$ to $\{m \in D \mid m \neq n\}$. These operations are typical for finite domain constraint programming systems like Choco, ILOG Solver, ECL^{i}PS^e, Mozart, and SICStus Prolog. Some systems provide additional operators such as for fixing values.

Iterators. It is quite common for a propagator to iterate over all values of a given variable. Suppose that i is a value iterator for some variable providing the following operations: $i.\mathrm{done}()$ tests whether all values have been iterated; $i.\mathrm{value}()$ returns the current value; and $i.\mathrm{next}()$ moves to the next value.

Domain operations. A *domain operation* supports simultaneous access or update of multiple values of a variable domain. If the multiple values form a consecutive interval $[n, m]$, such operations need only take n and m as arguments. Many systems provide general sets of values by supporting an abstract set type, e.g. Choco, ECL^{i}PS^e, Mozart and SICStus Prolog. Schulte and Tack describe in [60] domain operations based on generic

range and value iterators. Other systems like ILOG Solver only allow access by iteration over the values of a variable domain.

Subscription. When a propagator p is created, it *subscribes* to its input variables. Subscription guarantees that p is executed whenever the domain of one of its variables changes according to an event. Options for representing the subscriber set of a given variable x include the following:

1. A single suspension list of pairs $E_i.p_i$ where E_i denotes the event set for which propagator p_i requires execution. When an event on x occurs, the list is traversed and the relevant propagators are selected for execution. Obviously, a lot of pairs that do not match the event could be scanned.

2. Multiple suspension lists of propagators for different events. On subscription, the propagator is placed in one of the lists. When an event on x occurs, all propagators on the relevant lists are selected for execution. Typically, there is one list for each event type $e \in \{\mathrm{fix}(x), \mathrm{minc}(x), \mathrm{maxc}(x), \mathrm{any}(x)\}$ plus one list for propagators whose event set contains both $\mathrm{minc}(x)$ and $\mathrm{maxc}(x)$. This is the design used in Choco, ECL^{i}PS^e, Mozart, and SICStus Prolog. Other systems collapse $\mathrm{minc}(x)$ and $\mathrm{maxc}(x)$ into a single event $\mathrm{minmaxc}(x)$ (for example, ILOG Solver [32] and Gecode [24]).

3. An array of propagators, partitioned according to the various events. When an event on x occurs, all propagators in the relevant partitions are selected for execution. This representation is particularly attractive if the possible events are $e \in \{\mathrm{fix}(x), \mathrm{minmaxc}(x), \mathrm{any}(x)\}$, in which case the relevant partitions form a single interval.

Domain representation. Popular representations of $D = \mathrm{dom}(X)$ include range sequences and bit vectors. A *range sequence* for a finite set of integers I is the shortest sequence $s = \{[n_1, m_1], \ldots, [n_k, m_k]\}$ such that I is covered $\left(I = \cup_{i=1}^{k} [n_i, m_i]\right)$ and the ranges are ordered by their smallest elements ($n_i \leq n_{i+1}$ for $i \leq i < k$). Clearly, a range sequence is unique, none of its ranges is empty, and $m_i + 1 < n_{i+1}$ for $1 \leq i < k$. A *bit vector* for a finite set of integers I is a string of bits such that the i^{th} bit is 1 iff $i \in I$.

Table 14.1 compares the worst-case complexity of the basic operations for these representations. Range sequences are usually represented as singly or doubly linked lists. Bit vectors are typically represented as a number of consecutive memory words, with an implementation defined size limit, usually augmented with direct access to $\min(D)$ and $\max(D)$. Range sequence thus seem to be more scalable to problems with large domains.

14.2.3 Propagators

A propagator p is a software entity with possibly private state (we allow ourselves to refer to the function as well as its implementation as propagator). It (partially) implements a constraint c over some variables or *parameters*. The task of a propagator is to observe its parameters and, as soon as a value is removed from the domain of a parameter, try to remove further values from the domains of its parameters. The algorithm employed in the

Table 14.1: Complexity of basic operations for range sequences of length r and bit vectors of size v augmented with explicit bounds.

Operations	**Range sequence**	**Bitvector**
$x.\mathrm{getmin}()$	$O(1)$	$O(1)$
$x.\mathrm{getmax}()$	$O(1)$	$O(1)$
$x.\mathrm{hasval}(n)$	$O(r)$	$O(1)$
$x.\mathrm{adjmin}(n)$	$O(r)$	$O(1)$
$x.\mathrm{adjmax}(n)$	$O(r)$	$O(1)$
$x.\mathrm{excval}(n)$	$O(r)$	$O(v)$
$i.\mathrm{done}()$	$O(1)$	$O(v)$
$i.\mathrm{value}()$	$O(1)$	$O(1)$
$i.\mathrm{next}()$	$O(1)$	$O(v)$

process is called a *filtering algorithm*. Thus, the filtering algorithm is repeatedly executed in a coroutining fashion.

The main work of a filtering algorithm consists in computing values to remove and to perform these value removals via the value and domain operations described above. The events raised by these operations cause other propagators to be scheduled for execution.

Life cycle. The life cycle of a propagator p is depicted in Figure 14.3. When a constraint c is posted, its parameters are checked and subscribed to, p is created, its private state is allocated, and it is scheduled for execution. If the constraint c is implemented by more than one propagator, all propagators implementing c are created likewise.

One run of p has one of three possible outcomes:

- p may realize that the constraint has no solution, e.g. by a domain becoming empty. The parameters are unsubscribed to, the private state is deallocated, and the current search node fails.
- p may discover that the constraint holds no matter what of the remaining values are taken by the parameters. The parameters are unsubscribed to and the private state is deallocated,
- None of the above. p is moved to the set of suspended propagators, and will remain there until the relevant events are raised.

Idempotent propagators. Suppose a propagator p runs and removes some values. This raises some events, which would normally reschedule p for execution, as p subscribes to the very variables whose domains it just pruned. But suppose now that p is idempotent. Then by definition running p again would be useless. Thus, idempotence is a desirable property of propagators: if p is known to be idempotent, then p itself can be excluded from the set of propagators scheduled for execution by events raised by p.

However, guaranteeing idempotence may be a serious difficulty in the design of a filtering algorithm—it is certainly more convenient to not guarantee anything and instead

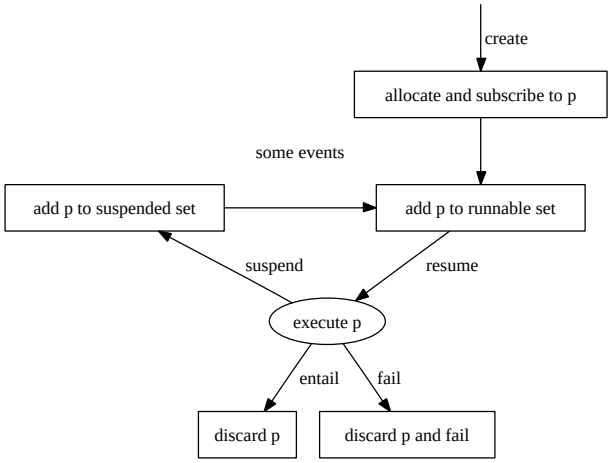

Figure 14.3: Life cycle of propagator p

leave the fixpoint computation to the propagation loop, at the cost of some redundant runs of p. Also, if the same variable occurs multiple times in the parameters, there is usually no straightforward way to guarantee idempotence.

Most systems do not require of propagators to be idempotent; some optimize the scheduling of propagators that are known to be idempotent. Mozart, as an exception, only supports idempotent propagators [45].

Schulte and Stuckey describe *dynamic idempotence* in [58] as a generalization: a propagator p signals after application whether the new domain is a fixpoint of p (similar to signaling failure or entailment as described above).

Multiple value removals. Suppose a propagator p runs and multiple values are removed by multiple operations from the same variable x. It would be very wasteful to traverse the suspension list(s) of x and schedule subscribing propagators for each removal. A much more reasonable design is to perform such traversal once per modified parameter, at the end of executing p. Therefore, the value and domain operations described above usually do not perform such scheduling. Instead, propagators call the relevant propagation services near the end of the filtering algorithm.

This is already manifest in the propagate function described in Section 14.1.2: it records the modified variables (or the events that occurred) and schedules the propagators only after the propagator has been applied.

Amount of information available. When a propagator p is resumed, it is usually interested in knowing which values have been deleted from which parameters since last time. The propagation services may provide part of this information, or even all of it. Of course, there is a trade-off between the efforts spent by the propagation services maintaining this

information and the efforts spent by the propagators discovering it. One can distinguish three levels of granularity of the information provided to p:

coarse p is told that something has changed, but not what. This is the information provided in SICStus Prolog and Mozart.

medium p is told which parameters have been changed. This is the information provided in CHIP.

fine p is told which parameters have been changed, as well as the set of removed values. This is the information provided in ILOG Solver.

Use of private state. The private state that each propagator maintains can be used for a number of things:

auxiliary data structures Many filtering algorithms contain as components algorithms operating on specific data structures (digraphs, bipartite graphs, heaps etc.). The private state stores such data structures.

incrementality When a propagator is resumed, it is often the case that a single parameter has been changed, and that a single change has been made to it. Many filtering algorithms can exploit this fact and be incremental, i.e. perform its computation based on a previous state and changes made since that state was saved. An incremental computation is typically an order of magnitude cheaper than computing from scratch, but if many changes have been made, it is not necessarily the case.

domain information Suppose that the propagation services do not provide fine-grained information when a propagator is resumed. Nevertheless, by maintaining in its private state a copy of the parameter domains, or some suitable abstraction, the propagator can compute the required information.

fixed parameters It is often useful for a propagator of arity m to maintain a partitioning of its parameters X into two sets X_f, whose values have been fixed, and X_v, whose values have not yet been fixed. Most filtering algorithms focus their attention on the set X_v. This partitioning is easily achieved by a simple array A of pointers or indices, such that X_f occupies array elements $A[1,\ldots,k]$ and X_v array elements $A[k+1,\ldots,m]$, where $k = |X_f|$. As a parameter is fixed, the relevant array elements are simply swapped and k is incremented.

Multiple variable occurrences. The same variable may occur multiple times among the parameters of a propagator p, initially as well as dynamically due to unification. This by itself does not cause problems, except, as noted above, any guarantees of idempotence are usually given under the assumption that no variable aliasing occurs.

Some propagators may also use variable aliasing in order to propagate more. For example, suppose that in a propagator for the constraint $x - y = z$, x and y are aliased. The propagator can then conclude $z = 0$, no matter what value is taken by x and y. Harvey and Stuckey discuss multiple occurrences of the same variable for linear integer constraints introduced by substitution in [29] and show how the amount of propagation changes with allowing substitution.

14.2.4 Daemons

So far, we have been assuming that an element of a suspension list is just a passive data structure pointing at a propagator p to schedule for execution. One can extend this design by associating with such elements a procedure called a *daemon*, which has access to p and its private state. If during traversal of the suspension list a daemon is encountered, instead of scheduling the propagator for execution, the daemon is simply run. This design is motivated by the following reasons:

- If the propagation services do not tell propagators which parameters have been changed, the daemon can maintain that information, since a daemon is always associated with a given parameter.
- Scheduling and resuming a propagator often involves larger overhead than running a daemon. If there is some simple test to determine whether the propagator can propagate anything, then the daemon can run that test, and if successful, schedule the propagator for execution.
- If there is some information in the private state that needs to be updated incrementally as the parameters are modified, daemons are a convenient mechanism for doing so.

Systems using daemons include CHIP, SICStus Prolog, and ILOG Solver.

14.2.5 Indexicals

Indexicals [66, 19, 15], also known as projection constraints [62], are a popular approach to implement simple propagators using a high-level specification language.

An indexical is a propagator with a single output variable and is defined in terms of a range expression. A constraint $c(x_1, \ldots, x_n)$ is then implemented by n indexicals p_i. Each indexical p_i is defined by $x_i \text{ in } r_i$ where r_i is a range expression (to be explained later). Each of the indexicals p_i has the input variables $\{x_1, \ldots, x_n\}$.

Executing an indexical p of the form $x_i \text{ in } r_i$ with a current domain D computes the projection $\hat{c}_i$ of c onto x_i from $D(x_1), \ldots, D(x_{i-1}), D(x_{i+1}), \ldots, D(x_n)$. The domain returned by p is $p(D)(x_i) = D(x_i) \cap \hat{c}_i$ and $p(D)(x_j) = D(x_j)$ for all $1 \leq i \neq j \leq n$.

Indexicals can be seen as a programming interface for fine-grained control over propagation. They do not provide for the integration of sophisticated filtering algorithms for global constraints. Figure 14.4 shows a subset of the range expressions used in SICStus Prolog.

Example 9. To illustrate the use of indexicals for controlling the amount of propagation, consider the constraint $x = y + c$ where c is assumed to be a constant. This may be expressed with indexicals maintaining arc consistency:

$$(x \text{ in } \mathrm{dom}(y) + c,\ y \text{ in } \mathrm{dom}(x) - c)$$

or with indexicals maintaining bounds consistency:

$$(x \text{ in } \min(y) + c \,..\, \max(y) + c,\ y \text{ in } \min(x) - c \,..\, \max(x) - c)$$

$$
\begin{array}{rcl}
R & ::= & T\,..\,T \mid R \cap R \mid R \cup R \mid R \,?\, R \mid \backslash R \\
 & \mid & R + T \mid R - T \mid R \bmod T \mid \mathrm{dom}(x) \\
 & \mid & \text{a finite subset of } \mathbb{Z} \\
T & ::= & N \mid T + T \mid T - T \mid T * T \mid \lceil T/T \rceil \mid \lfloor T/T \rfloor \mid T \bmod T \\
 & \mid & \min(x) \mid \max(x) \mid \mathrm{card}(x) \\
N & ::= & x \mid i, \text{ where } i \in \mathbb{Z} \mid \infty \mid -\infty
\end{array}
$$

Figure 14.4: Range expressions in SICStus Prolog indexicals

As discussed in [66, 19, 15], range expressions for indexicals must comply with certain monotonicity rules to make sense logically and operationally (corresponding to the properties that hold true for propagators and for propagators being the implementation of a constraint).

14.2.6 Reification

A *reified constraint* (also known as *meta constraint*) $c \leftrightarrow b$ reflects the truth value of the constraint c onto a 0/1-variable b. So if c is entailed (disentailed) by the constraint store, b is constrained to 1 (0), and if $b = 1$ ($b = 0$), c ($\neg c$) is posted.

One way of providing reification of a class of constraints is by extending the indexical mechanism, as proposed in [66] and implemented in SICStus Prolog [15]. Indexicals as described above are used for posting a constraint c. For reification, however, we also need to be able to post $\neg c$, and to check whether c is entailed or disentailed. This can be done by introducing *checking indexicals*. A checking indexical has the same format as a regular one, x_i in r_i, but instead of updating the domain for x_i, it checks whether $D(x_i) \subseteq \hat{c}_i$ holds for a given domain D.

14.3 Implementing Search

This section describes how systems implement search. As introduced in Section 14.1, a system needs to implement branching, state restoration, and exploration. The section discusses each of these issues in turn.

14.3.1 Branching

How a system implements branching depends in particular on whether the system is based on a programming language that has search built-in (such as Prolog or Oz). In this case, a branching strategy is expressed easily from the primitive of the language that controls search. In Prolog-based systems, for example, several clauses of a predicate then define the branching strategy. With relation to Figure 14.2, each clause corresponds to one of the alternatives $\mathrm{prop}(c_i)$).

Other languages provide special constructs that allow to express several alternatives. OPL, for example, offers a `try`-statement with several clauses corresponding to the alternatives [69]. A similar approach is taken in Oz, where a `choice`-statement serves the same purpose [63, 54]. For Oz, Schulte gives in [54] a reduction to a primitive that only allows one to assign an integer to a variable.

If the underlying language does not have search built-in (such as for libraries built on top of C++) systems provide some other means to describe a choice point. ILOG Solver [32], for example, provides the concept of a choice point (some data structure), which consists of several alternatives called *goals*. Goals themselves can be composed from several subgoals.

A common pattern for branching strategies is to select a particular not-yet fixed variable x according to some criteria. A common example is first-failure branching, which selects a variable with smallest domain. Here it is important to understand what the strategy does in case of ties: which one of the possibly many variables with a smallest domain is selected. For example, Wallace, Schimpf, et al. note in [71] that comparing search in systems can be particularly difficult due to a different treatment of ties in different systems.

14.3.2 State Restoration

As described in Section 14.1.6 search requires that a previous state of the system can be restored. The state includes the domain of the variables, propagators (for example, propagators that became entailed need to be restored), and private state of propagators.

Exploration (to be discussed in Section 14.3.3) creates a search tree where the nodes of the search tree correspond to the state of the system. In relation to `search` as shown in Figure 14.2, a new node is defined by each recursive invocation of `search`.

Systems use basically three different approaches to state restoration (the term state restoration has been coined by Choi, Henz, et al. in [18]):

copying A copy of a node is created before the node is changed.

trailing Changes to nodes are recorded such that they can be undone later.

recomputation If needed, a node is recomputed from scratch.

Expressiveness. The main difference as it comes to expressiveness is the number of nodes that are simultaneously available for further exploration. With copying, all nodes that are created as copies are directly ready for further exploration. With trailing, exploration can only continue at a single node at a time.

In principle, trailing does not exclude exploration of multiple nodes. However, they can be explored in an interleaved fashion only and switching between nodes is a costly operation.

Having more than a single node available for exploration is essential to search strategies like concurrent, parallel, or breadth-first.

Trailing. A trailing-based system uses a trail to store undo information. Prior to performing a state-changing operation, information to reconstruct the state is stored on the trail. In a concrete implementation, the state changing operations considered are updates of memory locations. If a memory update is performed, the location's address and its old content is stored on the trail. To this kind of trail we refer as single-value trail. Starting

exploration from a node puts a mark on the trail. Undoing the trail restores all memory locations up to the previous mark. This is essentially the technology used in Warren's Abstract Machine [74, 8].

In the context of trailing-based constraint programming systems two further techniques come into play:

time-stamping With finite domains, the domain of a variable can be narrowed multiple times. However it is sufficient to trail only the original value, intermediate values need no restoration: each location needs to appear at most once on the trail. Otherwise memory consumption is no longer bounded by the number of changed locations but by the number of state-changing operations performed. To ensure this property, time-stamping is used: as soon as an entity is trailed, the entity is stamped to prevent it from further trailing until the stamp changes again.

The time-stamp changes every time a new mark is put on the trail. Note that time-stamping concerns both the operations and the data structures that must contain the time-stamp.

multiple-value trail A single-value trail needs $2n$ entries for n changed locations. A multiple value trail uses the optimization that if the contents of $n > 1$ successive locations are changed, $n + 2$ entries are added to the trail: one for the first location's address, a second entry for n, and n entries for the locations' values.

For a discussion of time-stamps and a multiple value trail in the context of the CHIP system, see [1, 3].

Copying. Copying needs for each data structure a routine that creates a copy and also recursively copies contained data structures. A system that features a copying garbage collector already provides almost everything needed to implement copying. For example in the Mozart implementation of Oz [43], copying and garbage collection share the same routines parameterized by a flag that signals whether garbage collection is performed or whether a node is being copied.

By this all operations on data structures are independent of search with respect to both design and implementation. This makes search in a system an orthogonal issue.

Discussion. Trailing is the predominating approach used for state restoration in finite domain constraint programming systems. Clearly, all Prolog-based systems use trailing but also most other systems with the exception of Oz/Mozart [43], Figaro [31], and Gecode [24].

Trailing requires that all operations be search-aware: search is not an orthogonal issue to the rest of the system. Complexity in design and implementation is increased: it is a matter of fact that a larger part of a system is concerned with operations rather than with basic data structure management. A good design that encapsulates update operations will avoid most of the complexity. To take advantage of multiple value trail entries, however, operations require special effort in design and implementation.

Semantic backtracking as an approach to state restoration that exploits the semantics of an underlying solver for linear constraints over the reals is used in CLP(R) [33] and also in [65]. Semantic backtracking stores constraints that are used to reestablish an equivalent

state of the system rather than trailing all changes to the underlying constraints. By this the approach can be seen as a hybrid between trailing and recomputation. A similar technique is used by Régin in [51], where the author describes how to maintain arc consistency by restoring equivalent states rather than identical states.

Recomputation. Recomputation trades space for time, a node is reconstructed on demand by redoing constraint propagation. The space requirements are obviously low: only the path in the search tree leading to the node must be stored. Basing exploration on recomputation alone is infeasible. Suppose a complete binary search tree of height n, which has 2^n leafs. To recompute a single leaf, n exploration steps are needed. This gives a total of $n2^n$ exploration steps compared to $2^{n+1} - 2$ exploration steps without recomputation (that is, the number of arcs).

The basic idea of combining recomputation with copying or trailing is as follows: copy (or start trailing) a node from time to time during exploration. Recomputation then can start from the last copied (or trailed) node on the path to the root. The implementation of recomputation is straightforward, see [56, 54] for example.

If exploration exhibits a failed node it is quite likely that not only a single node is failed but that an entire subtree is failed. It is unlikely that only the last decision made in exploration was wrong. This suggests that as soon as a failed node occurs during exploration, the attitude for further exploration should become more pessimistic. *Adaptive recomputation* [54] takes a pessimistic attitude by creating intermediate copies as follows: during recomputation an additional copy is created at the middle of the path for recomputation.

Performance of recomputation depends critically on the amount of information stored for the path. In naive recomputation, the path is stored as a list of integers identifying which alternative (that is, the i in $\mathrm{prop}(c_i)$) needs to be recomputed. While this makes the space requirements for recomputation problem independent, n fixpoints need to be computed for a path of length n.

In *batch recomputation* [18], the alternatives $\mathrm{prop}(c_i)$ are stored. To recompute a node it is sufficient to compute a single fixpoint. Batch recomputation is shown to be considerably more efficient than naive recomputation in [18]. *Decomposition-based search* as a similar idea to batch recomputation is reported by Michel and Van Hentenryck in [41]. Here also the alternatives rather than just integers are stored for recomputation.

14.3.3 Exploration

The architecture for search in a finite domain constraint programming system described in Section 14.1.5 only considers left-most depth-first exploration of the search tree. Clearly, systems offer more exploration strategies to allow for example search for a best solution. A few systems also provide abstractions from which new exploration strategies can be programmed.

Predefined exploration strategies. All Prolog-based languages systems support single- and all-solution search following depth-first exploration as sketched in Section 14.1.5. Best-solution search is controlled by a single cost variable and amounts to search for a solution with smallest or largest cost. CLP-based systems offer an interactive toplevel for controlling exploration that allows the user to prompt for multiple solutions. The interactive toplevel cannot be used within programs. ECL^{i}PS^e provides visual search through

the Grace tool [39] and other strategies such as LDS [30] and time and resource bounded search.

ILOG Solver [32] and OPL [64] offer LDS [30], DDS [73], and IDFS [40]. Best-solution search in ILOG Solver also uses a cost variable. To avoid recomputation of the best solution, the program must be modified to explicitly store solutions. Search in ILOG Solver is incremental in that solutions can be computed on request.

Programming exploration. The first system to offer support for programming exploration has been Oz/Mozart. Schulte and Smolka introduce the solve combinator in [57], which allows to program exploration based on the idea of having a first-class representation of nodes in the search tree. Schulte describes computation spaces as a refinement [56, 54] of the solve combinator, which also allows to program strategies supporting recomputation and parallel execution. Computation spaces have been used to realize user-controlled interactive search [55] and parallel search on networked computers [53]. Curry [27] offers the same programming model as the solve combinator.

Another system providing support for programming exploration is ILOG Solver [32] (OPL [64] offers an equivalent model for programming exploration). Programming exploration in ILOG Solver is based on limits and node evaluators [47, 69]. Programmable limits allow to stop exploration (time limit, for example). Node evaluators map search tree nodes to priorities. Node priorities determine the exploration order of nodes. Additionally, a special priority discards nodes.

ILOG Solver supports switching between arbitrary nodes in the search tree by full recomputation. For example, best-first search needs to switch between arbitrary nodes. To limit the amount of switching, Solver uses an additional threshold value. Only if the cost improvement exceeds the threshold, nodes are switched. This results in an approximation of best-first search. Fully interactive exploration is not feasible with full recomputation.

SALSA [37] is a language for the specification of search algorithms that cover exploration strategies for tree search as well as neighborhood-based search (local search). SALSA requires a host language that supports search (for example, Claire [16]) as compilation target.

14.4 Systems Overview

This section discusses different approaches used for finite domain programming systems and a brief overview of existing systems.

14.4.1 Approaches

Several approaches and systems have been suggested to solve combinatorial problems with finite domain constraints. Historically, many systems have been implemented by embedding into an existing Prolog host system. There are many reasons for such an approach:

1. Much of the required infrastructure of the constraint solver is provided by the host language: data structures, memory management, support for search and backtrackable updates.

2. The high level and declarative nature of Prolog makes it a reasonable choice of language for expressing combinatorial problems.

From the point of view of writing applications in mainstream object-oriented programming languages such as C++ and Java, although they can readily interface to modules written in other languages, providing a constraint solver as a class library is arguably a more attractive approach. This requires a larger implementation effort to provide the necessary infrastructure, but also gives more opportunities for optimization, as there are no design constraints imposed by a host system.

14.4.2 Prominent Systems

This section gives a brief overview of some finite domain constraint programming systems. As it is impossible to cover all systems that exist or have existed, we have selected systems that introduced some new ground-breaking ideas or that are prominent in other ways. The systems are partitioned into autonomous systems and library systems.

Autonomous Systems

B-Prolog [75]. Extends a Prolog virtual machine with instructions for constraint propagation. Introduces action rules, a generalization of indexicals.

`cc(FD)` [66, 67, 68]. A representative of the concurrent and glass-box constraint programming research directions. Significant contributions include indexicals and constraint combinators.

`clp(FD)` [22, 19]. A representative of the approach of extending a Prolog virtual machine [74] with instructions for constraint propagation. Uses indexicals. Precursor of GNU Prolog.

CHIP [1, 2, 3]. A Prolog system with a constraint solver written in C. A pioneer in the global constraints research area [4, 11]. Provides a rich set of global constraints. Also available as C/C++ libraries.

ECL^{i}PS^e [70, 5, 14]. A Prolog system with constraint solving based on a general coroutining mechanism and attributed variables. A pioneer in the areas of integration with MIP solvers such as CPLEX and XPRESS-MP and using hybrid methods for constraint solving [61].

GNU Prolog [20, 21]. The successor of `clp(FD)`, compiles Prolog programs with constraints to native binaries, extending a Prolog virtual machine [74] with instructions for constraint propagation. Uses indexicals.

Mozart [63, 43]. A development platform based on the Oz language, mixing logic, constraint, object-oriented, concurrent, distributed, and multi-paradigm programming. Search in Mozart is based on copying and recomputation.

Nicolog [62]. Extends a Prolog virtual machine with instructions for constraint propagation. Introduces *projection constraints*, extending indexicals with conditional expressions and tests.

PaLM [34]. PaLM (Propagation and Learning with Move) is a constraint programming system, based on the Choco constraints library. Its most important contributions are its explanation-based features, which can be used to control the search as well as provide answers to questions such as:

- Why does my problem not have any solution?
- Why can variable x not take value a in a solution?
- Why is variable x currently assigned to a?

SICStus Prolog [35, 15]. A Prolog system with constraint solving based on a general coroutining mechanism and attributed variables. Constraint solver written in C using global constraints as well as indexicals.

Library Systems

CHIP [1, 2, 3]. C/C++ library version of the CHIP constraint solver as described above.

Choco [36]. A constraint solver kernel, originally written in the Claire programming language. A more recent Java version is available. Designed to be a platform for CP research, allowing for easy extensions and experimentation with event handling and scheduling policies. A library of global constraints, Iceberg, is available.

FaCiLe [9]. A constraint programming library written in OCaml, featuring constraints over integer as well as integer set finite domains.

Gecode [24, 58, 60]. A constraint solver library implemented in C++. Designed to be not used directly for modeling but for interfacing to systems offering modeling support (for example, Alice [6] an extension to Standard ML, interfaces to Gecode). Gecode is based on copying and recomputation rather than trailing.

ILOG Solver and JSolver [32]. A constraint solver library tightly integrated into the C++ and Java languages. Features constraints over integer as well as integer set finite domains. A pioneer in constraint solver libraries and in integrating constraint and object-oriented programming.

14.5 Outlook

Finite-domain constraint programming systems have proven useful tools for solving many problems in a wide range of application areas. As witnessed by this chapter many useful techniques for the implementation of constraint systems are available, both for constraint propagation as well as for search.

However, due to the change of available hardware platforms, the advent of new methods for problem solving and new constraints and propagators, and new requirements for systems, it is quite clear that the development of constraint programming systems will be faced with many new and difficult challenges. Some of the challenges are as follows.

Parallelism. Search for constraint programming offers great potential for parallelism: rather than exploring a single node at a time, explore several nodes of the search tree in parallel. There has been considerable work in the area of parallel search in general and parallel search for logic programming in particular [17, 26], however only little attention has been given to parallel search for constraint programming: only few systems support parallel search (ECL^{i}PS^e [44, 50], ILOG Solver [47], and Mozart [53, 54]) and only little experience in using parallel search for solving real-life problems is available [46, 48].

This is in sharp contrast to the fact that solving constraint problems is difficult and parallel computers are commodities. Networked computers are available everywhere and are mostly being idle. Pretty much all desktop machines sold in the next few years will feature processors providing parallel execution by multiple processing cores. The challenge for systems is to exploit the resources provided by parallel computers and making their useful exploitation simple.

Hybrid architectures. Propagation-based constraint programming is clearly not the only approach for solving combinatorial optimization problems. Other approaches such as integer programming and local search have shown their potential and even hybrid approaches are emerging. The questions for a system is how to best combine and provide services based on different approaches. One of the key questions is of course how tight the integration can and should be. Shen and Schimpf discuss the integration of linear integer solvers in ECL^{i}PS^e in [61].

Correctness. The last decade has seen the advent of an ever increasing number of powerful filtering algorithms used in propagators for the implementation of global constraints. However, implementing these propagators is typically complex and it is far from obvious that an implementation is actually correct for a given constraint. Additionally, taking advantage of properties such as idempotence and entailment add additional complexity to the implementation.

Ideally, a system should only offer propagators that are known to be correct. So far, a systematic methodology for proving correctness of these algorithms is missing. Worse still, even approaches for the systematic testing with sufficient coverage for propagators are not available. Correctness is important as the usefulness of constraint programming relies on the very fact that what is computed by a system is actually a solution to the problem solved.

Open interfaces. Today's development and deployment of constraint-based applications is often system specific: a programmer develops a constraint-based solution to a problem and integrates it into some larger software system. Development is system-specific as the model used can not easily be ported or adapted to a different system. Deployment is system-specific as many systems (notably language-based systems) require quite some effort for integrating constraint-based components into larger software systems.

The challenge is to devise open interfaces such that the same model can be used with many different systems without any porting effort and that the integration into software systems is easy. The former issue is partly addressed by using *modeling languages* such as OPL [64] or ESRA [23], for example. Modeling languages, however, only address part of the challenge as different systems offer vastly different services (think of what collection of global constraints systems support).

Richer coordination. One of the beauties of constraint programming is the simplicity of how constraint propagation can be coordinated: propagators are connected by variables acting as simple communication channels enjoying strong properties such as being decreasing. The beauty comes at the price of making communication low-level: only value removal is communicated.

The challenge is to provide richer communication to achieve stronger propagation. A potential candidate for communication are graph properties expressing information on a collection of constraints. Another approach, which chooses propagators for constraints to minimize propagation effort while retaining search effort, is based on properties that characterize the interaction among several constraints sharing variables [59].

Acknowledgments

The authors are grateful to Peter Stuckey for much of the material in Section 14.1, which is based on joint work by Peter Stuckey and Christian Schulte. Martin Henz, Mikael Lagerkvist, and Peter Stuckey provided helpful comments, which considerably improved this chapter. The authors thank Pascal Van Hentenryck for convincing them to give an invited tutorial at CP 2002 on finite domain constraint programming systems, which has served as a starting point for this chapter. Christian Schulte is partially funded by the Swedish Research Council (VR) under grant 621-2004-4953.

Bibliography

[1] Abderrahmane Aggoun and Nicolas Beldiceanu. Time Stamps Techniques for the Trailed Data in Constraint Logic Programming Systems. In S. Bourgault and M. Dincbas, editors, *Actes du Séminaire 1990 de programmation en Logique*, pages 487–509, Trégastel, France, May 1990. CNET, Lannion, France.

[2] Abderrahmane Aggoun and Nicolas Beldiceanu. Overview of the CHIP Compiler System. In Koichi Furukawa, editor, *Proceedings of the Eight International Conference on Logic Programming*, pages 775–788, Paris, France, June 1991. The MIT Press.

[3] Abderrahmane Aggoun and Nicolas Beldiceanu. Overview of the CHIP compiler system. In Frédéric Benhamou and Alain Colmerauer, editors, *Constraint Logic Programming: Selected Research*, pages 421–437. The MIT Press, Cambridge, MA, USA, 1993.

[4] Abderrahmane Aggoun and Nicolas Beldiceanu. Extending CHIP in order to solve complex scheduling and placement problems. *Journal of Mathematical and Computer Modelling*, 17(7):57–73, 1993.

[5] Abderrahmane Aggoun, David Chan, Pierre Dufresne, Eamon Falvey, Hugh Grant, Warwick Harvey, Alexander Herold, Geoffrey Macartney, Micha Meier, David Miller, Shyam Mudambi, Stefano Novello, Bruno Perez, Emmanuel Van Rossum, Joachim Schimpf, Kish Shen, Periklis Andreas Tsahageas, and Dominique Henry de Villeneuve. ECLiPSe 5.0. User manual, IC Parc, London, UK, November 2000.

[6] Alice Team. The Alice system, 2003. Programming Systems Lab, Universität des Saarlandes. Available from `www.ps.uni-sb.de/alice/`.

[7] Krzysztof R. Apt. *Principles of Constraint Programming*. Cambridge University Press, 2003.

[8] Hassan Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction*. Logic Programming Series. The MIT Press, Cambridge, MA, USA, 1991.

[9] Nicolas Barnier and Pascal Brisset. FaCiLe: a functional constraint library. *ALP Newsletter*, 14(2), May 2001. Available from `www.recherche.enac.fr/opti/facile/`.

[10] Peter Van Beek, editor. *Eleventh International Conference on Principles and Practice of Constraint Programming*. Lecture Notes in Computer Science. Springer-Verlag, Sitges, Spain, October 2005.

[11] Nicolas Beldiceanu and Evelyne Contejean. Introducing global constraints in CHIP. *Journal of Mathematical and Computer Modelling*, 20(12):97–123, 1994.

[12] Nicolas Beldiceanu, Warwick Harvey, Martin Henz, François Laburthe, Eric Monfroy, Tobias Müller, Laurent Perron, and Christian Schulte. Proceedings of TRICS: Techniques foR Implementing Constraint programming Systems, a post-conference workshop of CP 2000. Technical Report TRA9/00, School of Computing, National University of Singapore, 55 Science Drive 2, Singapore 117599, September 2000.

[13] Frederic Benhamou. Heterogeneous Constraint Solving. In *Proceedings of the Fifth International Conference on Algebraic and Logic Programming (ALP'96), LNCS 1139*, pages 62–76, Aachen, Germany, 1996. Springer-Verlag.

[14] Pascal Brisset, Hani El Sakkout, Thom Frühwirth, Warwick Harvey, Micha Meier, Stefano Novello, Thierry Le Provost, Joachim Schimpf, and Mark Wallace. ECLiPSe Constraint Library Manual 5.8. User manual, IC Parc, London, UK, February 2005.

[15] Mats Carlsson, Greger Ottosson, and Björn Carlson. An open-ended finite domain constraint solver. In Hugh Glaser, Pieter H. Hartel, and Herbert Kuchen, editors, *Programming Languages: Implementations, Logics, and Programs, 9th International Symposium, PLILP'97*, volume 1292 of *Lecture Notes in Computer Science*, pages 191–206, Southampton, UK, September 1997. Springer-Verlag.

[16] Yves Caseau, François-Xavier Josset, and François Laburthe. CLAIRE: Combining sets, search and rules to better express algorithms. In Danny De Schreye, editor, *Proceedings of the 1999 International Conference on Logic Programming*, pages 245–259, Las Cruces, NM, USA, November 1999. The MIT Press.

[17] Jacques Chassin de Kergommeaux and Philippe Codognet. Parallel logic programming systems. *ACM Computing Surveys*, 26(3):295–336, September 1994.

[18] Chiu Wo Choi, Martin Henz, and Ka Boon Ng. Components for state restoration in tree search. In Toby Walsh, editor, *Proceedings of the Seventh International Conference on Principles and Practice of Constraint Programming*, Lecture Notes in Computer Science, vol. 2239. Springer Verlag, 2001.

[19] Philippe Codognet and Daniel Diaz. Compiling constraints in `clp(FD)`. *The Journal of Logic Programming*, 27(3):185–226, June 1996.

[20] Daniel Diaz and Philippe Codognet. GNU prolog: Beyond compiling Prolog to C. In Enrico Pontelli and Vítor Santos Costa, editors, *Practical Aspects of Declarative Languages, Second International Workshop, PADL 2000*, volume 1753 of *Lecture Notes in Computer Science*, pages 81–92, Boston, MA, USA, January 2000. Springer-Verlag.

[21] Daniel Diaz and Philippe Codognet. Design and implementation of the GNU prolog system. *Journal of Functional and Logic Programming*, 2001(6), 2001.

[22] Daniel Diaz and Philippe Codognet. A minimal extension of the WAM for clp(FD). In David S. Warren, editor, *Proceedings of the Tenth International Conference on Logic Programming*, pages 774–790, Budapest, Hungary, June 1993. The MIT Press.

[23] Pierre Flener, Justin Pearson, and Magnus Ågren. Introducing ESRA, a relational language modelling combinatorial problems. In Maurice Bruynooghe, editor, *Logic Based Program Synthesis and Transformation: 13th International Symposium*, volume 3108 of *Lecture Notes in Computer Science*, pages 214–229, Uppsala, Sweden, August 2004. Springer-Verlag.

[24] Gecode. Gecode: Generic constraint development environment, 2005. Available from `www.gecode.org`.

[25] Laurent Granvilliers and Eric Monfroy. Implementing constraint propagation by composition of reductions. In *ICLP'03*, volume 2916 of *Lecture Notes in Computer Science*, pages 300–314. Springer-Verlag, 2003.

[26] Gopal Gupta, Enrico Pontelli, Khayri Ali, Mats Carlsson, and Manuel Hermenegildo. Parallel execution of Prolog programs. *ACM Transactions on Programming Languages and Systems*, 23(4):472–602, July 2001.

[27] Michael Hanus. A unified computation model for functional and logic programming. In Neil D. Jones, editor, *The 24th Symposium on Principles of Programming Languages*, pages 80–93, Paris, France, January 1997. ACM Press.

[28] Warwick Harvey. Personal communication, April 2004.

[29] Warwick Harvey and Peter J. Stuckey. Improving linear constraint propagation by changing constraint representation. *Constraints*, 7:173–207, 2003.

[30] William D. Harvey and Matthew L. Ginsberg. Limited discrepancy search. In Chris S. Mellish, editor, *Fourteenth International Joint Conference on Artificial Intelligence*, pages 607–615, Montréal, Québec, Canada, August 1995. Morgan Kaufmann Publishers.

[31] Martin Henz, Tobias Müller, and Ka Boon Ng. Figaro: Yet another constraint programming library. In Inês de Castro Dutra, Vítor Santos Costa, Gopal Gupta, Enrico Pontelli, Manuel Carro, and Peter Kacsuk, editors, *Parallelism and Implementation Technology for (Constraint) Logic Programming*, pages 86–96, Las Cruces, NM, USA, December 1999. New Mexico State University.

[32] ILOG S.A. *ILOG Solver 6.0: Reference Manual*. Gentilly, France, October 2003.

[33] Joxan Jaffar, Spiro Michaylov, Peter J. Stuckey, and Roland H. C. Yap. The CLP(R) language and system. *Transactions on Programming Languages and Systems*, 14(3): 339–395, 1992.

[34] Narendra Jussien and Vincent Barichard. The PaLM system: explanation-based constraint programming. In Beldiceanu et al. [12], pages 118–133.

[35] Intelligent Systems Laboratory. SICStus Prolog user's manual, 3.12.1. Technical report, Swedish Institute of Computer Science, Box 1263, 164 29 Kista, Sweden, April 2005.

[36] François Laburthe. CHOCO: implementing a CP kernel. In Beldiceanu et al. [12], pages 71–85.

[37] François Laburthe and Yves Caseau. SALSA: A language for search algorithms. In Michael Maher and Jean-François Puget, editors, *Proceedings of the Fourth International Conference on Principles and Practice of Constraint Programming*, volume 1520 of *Lecture Notes in Computer Science*, pages 310–324, Pisa, Italy, October 1998. Springer-Verlag.

[38] Olivier Lhomme, Arnaud Gotlieb, and Michel Rueher. Dynamic optimization of interval narrowing algorithms. *The Journal of Logic Programming*, 37(1–3):165–183, 1998.

[39] Micha Meier. Debugging constraint programs. In Montanari and Rossi [42], pages 204–221.

[40] Pedro Meseguer. Interleaved depth-first search. In Pollack [49], pages 1382–1387.

[41] Laurent Michel and Pascal Van Hentenryck. A decomposition-based implementation of search strategies. *ACM Transactions on Computational Logic*, 5(2):351–383, 2004.

[42] Ugo Montanari and Francesca Rossi, editors. *Proceedings of the First International Conference on Principles and Practice of Constraint Programming*, volume 976 of *Lecture Notes in Computer Science*. Springer-Verlag, Cassis, France, September 1995.

[43] Mozart Consortium. The Mozart programming system, 1999. Available from `www.mozart-oz.org`.

[44] Shyam Mudambi and Joachim Schimpf. Parallel CLP on heterogeneous networks. In Pascal Van Hentenryck, editor, *Proceedings of the Eleventh International Conference on Logic Programming*, pages 124–141. The MIT Press, Santa Margherita Ligure, Italy, 1994.

[45] Tobias Müller. *Constraint Propagation in Mozart*. Doctoral dissertation, Universität des Saarlandes, Fakultät für Mathematik und Informatik, Fachrichtung Informatik, Im Stadtwald, 66041 Saarbrücken, Germany, 2001.

[46] Claude Le Pape, Laurent Perron, Jean-Charles Régin, and Paul Shaw. Robust and parallel solving of a network design problem. In *Eigth International Conference on Principles and Practice of Constraint Programming*, volume 2470 of *Lecture Notes in Computer Science*, pages 633–648, Ithaca, NY, USA, September 2002. Springer-Verlag.

[47] Laurent Perron. Search procedures and parallelism in constraint programming. In Joxan Jaffar, editor, *Proceedings of the Fifth International Conference on Principles and Practice of Constraint Programming*, volume 1713 of *Lecture Notes in Computer Science*, pages 346–360, Alexandra, VA, USA, October 1999. Springer-Verlag.

[48] Laurent Perron. Practical parallelism in constraint programming. In Narendra Jussien and François Laburthe, editors, *Proceedings of the Fourth International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimisation Problems (CP-AI-OR'02)*, pages 261–275, Le Croisic, France, March, 25–27 2002.

[49] Martha E. Pollack, editor. *Fifteenth International Joint Conference on Artificial Intelligence*. Morgan Kaufmann Publishers, Nagoya, Japan, August 1997.

[50] Steven Prestwich and Shyam Mudambi. Improved branch and bound in constraint logic programming. In Montanari and Rossi [42], pages 533–548.

[51] Jean-Charles Régin. Maintaining arc consistency algorithms during the search without additional cost. In Beek [10], pages 520–533.

[52] Pierre Savéant. Constraint reduction at the type level. In Beldiceanu et al. [12], pages 16–29.

[53] Christian Schulte. Parallel search made simple. In Beldiceanu et al. [12], pages 41–57.

[54] Christian Schulte. *Programming Constraint Services*, volume 2302 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, Berlin, Germany, 2002.

[55] Christian Schulte. Oz Explorer: A visual constraint programming tool. In Lee Naish, editor, *Proceedings of the Fourteenth International Conference on Logic Programming*, pages 286–300, Leuven, Belgium, July 1997. The MIT Press.

[56] Christian Schulte. Programming constraint inference engines. In Gert Smolka, editor, *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming*, volume 1330 of *Lecture Notes in Computer Science*, pages 519–533, Schloß Hagenberg, Linz, Austria, October 1997. Springer-Verlag.

[57] Christian Schulte and Gert Smolka. Encapsulated search in higher-order concurrent constraint programming. In Maurice Bruynooghe, editor, *Logic Programming: Proceedings of the 1994 International Symposium*, pages 505–520, Ithaca, NY, USA, November 1994. The MIT Press.

[58] Christian Schulte and Peter J. Stuckey. Speeding up constraint propagation. In Mark Wallace, editor, *Tenth International Conference on Principles and Practice of Constraint Programming*, volume 3258 of *Lecture Notes in Computer Science*, pages 619–633, Toronto, Canada, September 2004. Springer-Verlag.

[59] Christian Schulte and Peter J. Stuckey. When do bounds and domain propagation lead to the same search space? *Transactions on Programming Languages and Systems*, 27(3):388–425, May 2005.

[60] Christian Schulte and Guido Tack. Views and iterators for generic constraint implementations. In Beek [10], pages 817–821.

[61] Kish Shen and Joachim Schimpf. Eplex: An interface to mathematical programming solvers for constraint logic programming languages. In Beek [10], pages 622–636.

[62] Gregory Sidebottom. *A Language for Optimizing Constraint Propagation*. PhD thesis, Simon Fraser University, 1993.

[63] Gert Smolka. The Oz programming model. In Jan van Leeuwen, editor, *Computer Science Today*, volume 1000 of *Lecture Notes in Computer Science*, pages 324–343. Springer-Verlag, Berlin, 1995.

[64] Pascal Van Hentenryck. *The OPL Optimization Programming Language*. The MIT Press, Cambridge, MA, USA, 1999.

[65] Pascal Van Hentenryck and Viswanath Ramachandran. Backtracking without trailing in clp(r-lin). *ACM Trans. Program. Lang. Syst.*, 17(4):635–671, 1995.

[66] Pascal Van Hentenryck, Vijay Saraswat, and Yves Deville. Constraint processing in cc(FD). Manuscript, 1991.

[67] Pascal Van Hentenryck, Vijay Saraswat, and Yves Deville. Design, implementation and evaluation of the constraint language cc(FD). In Andreas Podelski, editor, *Constraint Programming: Basics and Trends*, volume 910 of *Lecture Notes in Computer Science*, pages 293–316. Springer-Verlag, 1995.

[68] Pascal Van Hentenryck, Vijay Saraswat, and Yves Deville. Design, implementation, and evaluation of the constraint language cc(FD). *The Journal of Logic Programming*,

37(1–3):139–164, October 1998.

[69] Pascal Van Hentenryck, Laurent Perron, and Jean-François Puget. Search and strategies in OPL. *ACM Transactions on Computational Logic*, 1(2):285–320, October 2000.

[70] Mark Wallace, Stefano Novello, and Joachim Schimpf. Eclipse: A platform for constraint logic programming. Technical report, IC-Parc, Imperial College, London, GB, August 1997.

[71] Mark Wallace, Joachim Schimpf, Kish Shen, and Warwick Harvey. On benchmarking constraint logic programming platforms. *Constraints*, 9(1):5–34, 2004.

[72] Richard J. Wallace and Eugene C. Freuder. Ordering heuristics for arc consistency algorithms. In *Ninth Canadian Conference on Artificial Intelligence*, pages 163–169, Vancouver, Canada, 1992.

[73] Toby Walsh. Depth-bounded discrepancy search. In Pollack [49], pages 1388–1393.

[74] David H. D. Warren. An abstract Prolog instruction set. Technical Note 309, SRI International, Artificial Intelligence Center, Menlo Park, CA, USA, October 1983.

[75] Neng-Fa Zhou. Programming finite-domain constraint propagators in action rules. *Theory and Practice of Logic Programming*, 6(1):1–26, 2006. To appear.