Constraint programming

Patrick Esquirol, Pierre Lopez
Laboratoire d'Analyse et d'Architecture des Systèmes du CNRS
7, avenue du Colonel Roche
31077 Toulouse Cedex (France)
e-mail: {esquirol,lopez}@laas.fr

Hélène Fargier
Université Paul Sabatier (IRIT)
118, route de Narbonne
31062 Toulouse Cedex (France)
e-mail: fargier@irit.fr

Thomas Schiex
Institut National de la Recherche Agronomique
Chemin de Borde Rouge, BP 27
31326 Castanet-Tolosan Cedex (France)
e-mail: tschiex@toulouse.inra.fr

Abstract

There has been a lot of interest lately from people solving constrained optimization problems for *Constraint Programming* (CP). Constraint programming cannot be described as a technique by itself but perhaps better as a class of computer languages tailored to the expression and resolution of problems which are non-deterministic in nature, with a fast program development and efficient runtime performances.

Constraint programming derives from logic programming, operational research and artificial intelligence. Logic programming offers the general non-deterministic host language which accommodates dedicated constraint solvers from OR and AI such as linear programming or constraint satisfaction techniques.

In this paper, we first review how pure logic programming languages evolved into *Constraint Logic Programming* (CLP) languages, bringing to light the interface between logic programming and constraint solvers. Some extra attention is given to a specific class of constraint solving techniques which have been developed in the *Constraint Satisfaction Problem* framework and which are currently used in most CLP languages to solve constrained problems in finite domains. We finally conclude by an overview of some existing CP languages, with some examples taken from scheduling.

Keywords: Constraint Logic Programming, Constraint Satisfaction, Scheduling.

1 Introduction

A large number of problems which are considered by operational research and which have also been tackled by artificial intelligence techniques, such as job-shop scheduling, resource allocation problems or digital circuit validation are combinatorial problems in nature. More formally, such problems are often NP-hard [GJ79] and it seems unlikely that algorithms with a reasonable worst-case complexity exist to solve them.

A traditional approach to solve such problems is to design a specific computer program that uses enumerative techniques such as Branch and Bound or, loosing some guarantees, local search techniques (Taboo search, simulated annealing). This may lead to the utmost efficiency but usually necessitates a lot of work, which may rapidly become useless if the model evolves in an unexpected direction and whose reuse is highly improbable.

Another possible approach is to cast the problem in a general framework such as integer linear programming and to use a dedicated solver to solve the linear model which has been designed. Usually, the rewriting tends to enlarge the problem size, to distort (to some extent) the original problem, to ignore nice properties such as possible heuristics or symmetries... In the worst cases, the final efficiency may be poor and solutions almost meaningless.

Ideally, one would like a general framework for stating large classes of combinatorial problems, with a language which is general enough to limit distortion and open enough to make it possible to use the specific knowledge about the problem, in order to enhance efficiency.

We now try to show the assets of constraint programming languages as general tools for representing and solving various classes of constrained problems, with illustrations in job-shop scheduling. These assets are inherited from logic programming but also from the underlying constraint solvers. In this paper, we will more specifically focus on AI techniques for solving constraint satisfaction problems on finite domains [Tsa93], even if other solvers, such as Simplex for sets of linear inequations are often used in CP languages.

2 From Logic Programming to Constraint Programming

Logic programming languages are general programming languages based on mathematical logic, and more specifically first order logic. The language Prolog, with its numerous dialects, is the main representative of the logic programming languages community. The original idea of Prolog is to let the user specify the properties he wants to satisfy using a subset of the first-order logic language and to use a general logical inference mechanism, the *resolution principle* [Rob65], to prove either that these properties cannot be met or to exhibit solutions.

2.1 Logic programming

2.1.1 Description and good properties of Prolog programming

Logic Programming will be presented here through its most representative implementation, the Prolog language. This language is a restriction of the first-order logic language to Horn clauses. It acts as a theorem prover, and applies a general inference mechanism based on the resolution principle. To get started with Prolog and the use of logic for problem solving, we recommend the reading of [CM81, CKvC83, Kow79, ACM92, AS93].

Prolog manipulates objects, called *terms*, that form the so-called "Herbrand universe". A term may be a constant such as "`1`", a variable such as "`X`", an atom such as "`a`", or a compound term such as functional terms ("`f(X,1)`") and lists. A Prolog program can be viewed as a sequence of declarative statements, the *clauses*, each of them stating how a given relation between some terms (the *head* of the clause) may be derived logically from a conjunctive set of relations (the *tail* of the clause), defined somewhere else in the program. n-ary relations (possibly $n = 0$) are all built from a predicate symbol and a set of terms, the arguments of the relations.

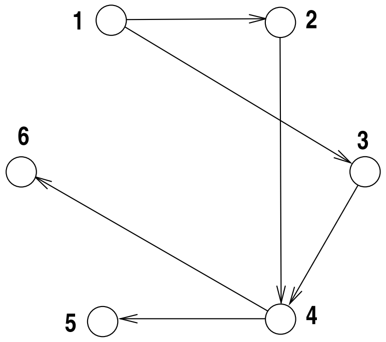
In the sequel, the chosen syntax to write the clauses will be the *Edinburgh syntax*. The head is separated from the tail by "`:-`"; if not empty, the tail consists of a set of terms separated by "`,`"; and the clause ends with "`.`". The character "`%`" introduces a comment.

```
% A comment: next line is an example of a clause
head :- term1, term2.
```

Moreover a variable is denoted by an upper case letter (`A-Z`); a list is represented by `[H|T]` where `H` is the head of the list (*i.e.,* the first element), `T` its tail (the other elements). Programs are clustered into packets of clauses, each packet grouping clauses which head refers to the same relation predicate. In each packet, one clause represents one alternative to prove the relation. In the following example (a path-finding problem), a first packet of 6 clauses describes the set of pairs of nodes that satisfy the arc-relation of some graph. These clauses have an empty tail (no conditions) since the graph is given. A second packet of clauses states the path relation, recursively defined from the arc-relation; a path exists between node `X` and node `Y`, either if the nodes are connected directly by an arc (first clause) or if a path exists between an intermediate node `Z` and node `Y`, such that node `X` is connected to node `Z` by an arc-relation (second clause).

```
% Directed non-reflexive graph
% described by its arcs
arc(1,2).
arc(1,3).
arc(2,4).
arc(3,4).
arc(4,5).
arc(4,6).

% Recursive definition of the path
% relation between any 2 nodes
path(X,Y,[X,Y]) :- arc(X,Y).
path(X,Y,[X|L]) :- arc(X,Z), path(Z,Y,L).
```



Prolog programs are launched by writing queries about one or more relations that must simultaneously be proved. The standard strategy of Prolog is based on an embedded chronological *backtrack search algorithm*, that enables to collect all the possible solutions for a given query.

If some variables appear in a query, Prolog attempts to give them values, or at least returns a minimal set of variables substitutions (equalities constraints), that satisfies the query. Values returned in the answer form one solution to the problem stated by the query. The query: "`?-path(1,Y,L).`" ask for all the paths `L` and their extremity `Y`, such that path `L` starts from node `1`. When the query has no variables, answer is '`Yes`' if the query is proved to be true, as for instance for the query "`?-path(1,4,[1,2,4]).`" that checks the path `<1,2,4>` between nodes `1` and `4`, or '`No`' when the query is not satisfiable, as for instance for the query "`?-path(X,X,L).`", which proves that no circuit can be found in this graph.

Let us note that the same program can also answer to queries like "`?-path(X,5,L).`" which returns all the paths terminating on node `5`, or like "`?-path(X,Y,L).`" which returns all the paths of the graph. This program is a *reversible* one, since arguments of the path relation are not forced to have a fixed input or output role, as opposed to classical programs in procedural or functional programming. The relational semantics of logic programming gives its declarativeness and its genericity. The embedded backtrack search algorithm and the high-level facilities for lists processing play also a major role in the conciseness of Prolog programs.

2.1.2 Limitations of logic programming for numerical problem solving

Pure logic programming needs and only allows the statement of relations between terms, without any assumption on numerical properties. When tackling numerical problems or reasoning in domains more structured than the domain of syntactic terms (Herbrand universe), the constraint solving algorithm on terms (the so-called *unification algorithm*) appears to be too weak for numerical computations, in particular when the search space developed by the standard backtracking can be pruned by an active interpretation of the numerical constraints, as shows the next example.

Suppose one wants to determine the pairs `(X,Y)` which satisfy the property `X < Y`, numbers `X` and `Y` belonging to a given enumerated subset of integers.

```
% Definition of a given subset of integers
is_number(1).
is_number(2).
is_number(3).

% Definition of the ordered-pair relation
ordered_pair(X,Y) :- is_number(X), is_number(Y), X < Y.
```

Numerical operations and checks (such as `X < Y`) are extra-logical relations and have been added under the form of predefined predicates just to make Prolog able to produce elementary numerical computations. The search tree developed in order to answer the query "`?-ordered-pair(X,Y).`" which lists all the ordered pairs of numbers amongst {1,2,3} is figured below.

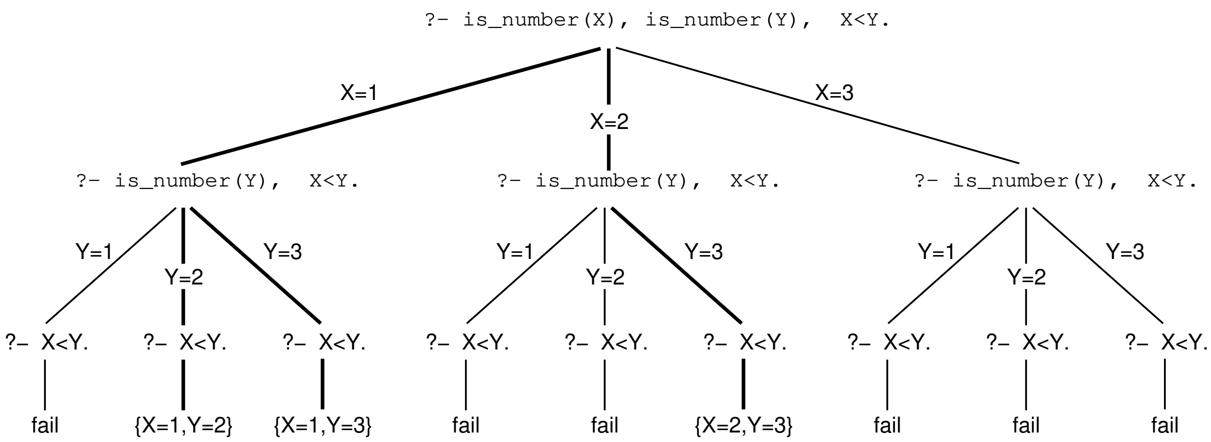


Figure 1: Search tree

This program first generates values for numbers `X` and `Y` and then checks the ordering relation. This approach (*generate-and-test*) is very inefficient considering the failing branches of the search tree. For example, `X=3` is tried even if it is obvious that no value can be associated with `Y` because `X` has its maximum value and the order is a strict one. For symmetrical reasons, failures due to `Y=1` are unnecessarily explored. Preferably, one could first restrict the domains of `X` and `Y` before enumerating their possible values. This necessitates that domains be associated to variables and that the relation `X < Y` be actively used to restrict the domains before values have been proposed for `X` and `Y` (*constrain-and-generate*), rather than simply passively checking it once the variables it involves are assigned.

2.1.3 Conclusion

Logic programming and its well-known implementation through the Prolog language offers several attractive properties for problem solving (declarativity, genericity, conciseness, exhaustive search). Facing problems where constraints must actively be exploited to improve efficiency, the unification constraint alone appears too weak. It has to be enlarged by other inference mechanisms based on more structured domains of interpretation, such as finite domain variables, integers, rationals, booleans, for which efficient constraint solving strategies can be implemented. The next section lists some of the domains of variables for which constraints can actively be interpreted in order to improve the standard strategy of Prolog.

2.2 Constraint Logic Programming

2.2.1 General principle

Constraint Logic programming (CLP) is an attempt to overcome the limitations of Logic Programming by enhancing it with constraint solving mechanisms.

The first objective was to replace an explicit coding of semantic properties between objects submitted to constraints by an implicit description with primitive constraints. This led to distinguish usual predicates and constraint predicates, usual variables and typed variables. For example, besides the original predicate "`is(X,Expr)`" that binds a variable to the evaluation of a closed arithmetic expression, new equality constraints have been introduced, which state equations in typed domains. This work was concerned with the integration of new computation domains and their associated constraint solvers. In the remainder of this paper we will denote by CLP($\mathcal{X}$) a CLP system acting on some constraint domain $\mathcal{X}$.

The second objective was to replace the depth-first search strategy of Prolog and its resulting generate-and-test behavior. Constraint propagation has been investigated in the Artificial Intelligence community since late 70's [Mon74, Mac77, Ste80]. Techniques like local value propagation, data-driven computation, forward checking and look ahead have been developed. Some of these techniques have been integrated as new inference rules in CLP systems.

It is often admitted that the embryonic attempts to introduce constraints in Prolog are due to Alain Colmerauer with its design of PROLOG II [Col82b], able to solve equations in the domain of infinite trees [Col82a]. PROLOG II also initiated the use of the "`dif(X,Y)`" primitive with a *co-routining* mechanism that postpones the disequality check until both arguments become instantiated and produces a backjumping to the clause containing the `dif` in case of failure.

The core idea of CLP was to replace the computational heart of a logic programming system, namely the unification algorithm, by a constraint handling mechanism in a constraint domain.

2.3 The CLP scheme

A good abstract of the theoretical foundations of CLP languages can be found in [Coh90]. For a deeper analysis, the survey of Jaffar & Maher [JM94] is worth reading. For the sake of simplicity and understandability we propose to describe what a CLP scheme may be with a model very close to the one of the abstract PROLOG III machine. This model supposes that constraints and predicates are syntactically separated in a clause, as opposed to the operational model given in [JM94]. Thus, a clause of a logic program will consist of a triplet $\langle h, B, C\rangle$ where h is the head, B a sequence of predicates (possibly empty) and C a set of constraints (possibly empty). A program clause $\langle h, B, C\rangle$ means that h can be rewritten as B under the constraints C. Suppose that the state of the abstract machine at a given time is $\langle G, R\rangle$, where:

- G is the (ordered) set of goals which remain to be executed;
- R forms the current *constraint store*.

Initially, goals and constraints are those of the query: $S_0 = \langle Q_g, Q_r\rangle$. One final state S_f is defined by an empty set of goals: $S_f = \langle \varnothing, R_f\rangle$. The final constraint store R_f may be non empty. In that case, the final answer may differ from a CLP system to another, depending on the constraint domains and some properties presented below.

The computation can be represented by a tree, similar to the search tree developed by a pure logic program: nodes are labeled by states, edges are labeled by rules (primitive or program-defined), and leaves are of two types: *fail* or *answer*. Fails occur when no rule can be applied to a state with a non empty set of goals. Suppose g is the next goal to be executed (the left-most term of G) in the current state $\langle G, R\rangle$. If a program clause $\langle h, B, C\rangle$ can be applied to g, the following state is $\langle G', C'\rangle$ where:

- $G' = G\backslash\{g\} \bigcup B$
- $R' = \mathsf{solve}(R \bigcup \{g = h\} \bigcup C)$.

The equation $\{g = h\}$ means that g and h have the same predicate symbol and satisfy the unification constraints between their respective arguments. When considering numerical domains, unification means equation, as for example in the unification of $g = \mathsf{foo}(X, 3)$ and $h = \mathsf{foo}(Y, Y+3)$: the result is $X = Y = 0$ in a CLP system on numerical domains whereas it fails in pure Prolog (as 3 cannot be made *syntactically* equivalent to $Y+3$). The solve function performs a satisfiability check, that, if failed, entails a backtrack and the selection of another clause of the program. This satisfiability check may be complete or not, depending on the constraint domains. In the following, we list some important properties of the CLP systems relating to the constraint domains they propose.

2.3.1 Properties

The major CLP systems offer, in a common Prolog environment, several constraint domains and their attached set of primitives. Programmers are generally interested in the expressiveness of the constraint domains and in the efficiency of the solvers. It is thus important to evaluate the various systems and their computation domains through relevant properties such as:

- Satisfaction-completeness, solution-compactness of the constraint domain.
- Incrementality of the constraint satisfaction algorithm.
- Existence of canonical representations of the constraints set.
- Simplification ability for the constraint-handling system.

A theory on a given domain is *satisfaction-complete* if it is always provable that every constraint is either satisfiable or not. It is the case for example for reals with the set $\{=, \neq, >, \geq, <, \leq\}$ of constraints and the set $\{+, \times\}$ of functions. But it is not the case for integers with $\{=\}$ constraint and $\{+, \times\}$ functions (the satisfiability of Diophantine equations being undecidable). CLP systems answer '`Maybe`', or '`N constraints delayed`' when this property is not filled, whereas when it is, the answer is '`Yes`' or '`No`'.

A theory on a constraint domain is *solution-compact* when any value of the domain can be represented by a (possibly infinite) set of constraints. True for the domain of real numbers with $\{=, \neq, >, \geq, <, \leq\}$ constraints and $\{+, \times\}$ functions, this property is lost for real numbers with only $\{=\}$ constraint and the same functions.

Furthermore, the set of constraints that is handled at each node is obtained by adding some constraints to a constraint set which was previously proved satisfiable[1]. Thus, the efficiency of the overall language would benefit from some "incrementality" in the constraint solver: once a set R of constraints has been shown satisfiable, the proof of satisfiability of $\mathsf{R} \cup \mathsf{R}'$ should, whenever possible, be made more efficient than a proof from scratch, by using some results from the previous satisfiability proof of R.

In practice, solvers often give a "solved form" of the initial set of constraints whose satisfiability is "obvious" and which is equivalent to the original set. The "solved form" given by the solvers can usually be directly used instead of the original constraint set when a constraint is added, thus automatically giving some "incrementality". The "solved form" may even be used for projection if it is an explicit representation of the solution space.

Simplification is sometimes called a *semantic garbage collection*. For example the two constraints $\mathsf{X} \geq 1$ and $\mathsf{X} \geq 2$ can be rewritten as only one: $\mathsf{X} \geq 2$. Another obvious type of simplification is to rewrite the subset of constraints on a given variable as soon as this variable receives a single value. Finally redundancies may also be eliminated, but this feature raises the more general problem of finding a minimal representative set of constraints, for which one cannot assert that a unique optimal strategy always exists.

To that aim, the existence of canonical forms of sets of constraints may help. A canonical form can be used as a "solved form" which facilitates both the satisfiability tests and simplifications, for example because constraints are ordered according to their arity, and/or variables are lexicographically ordered.

Finally, for satisfaction-complete systems, queries can receive non ground answers (*i.e.,* each variable of the query does not receive a single value): the answer is a projection of the system of constraints on these variables. The understandability of the answer obviously depends on simplification and canonical representation facilities.

2.4 Constraint Programming

Although constraint programming derives from logic programming and the major CLP systems have been designed as compatible extensions of Prolog (which remains the kernel), some other important systems have been developed independently. As they keep the non-deterministic computation principle, and their solvers are based on the same constraints domains, they deserve to be also mentioned in this paper (see Section 4.4). Constraint programming is the general framework that covers researches and applications of such constraint-based languages and systems.

[1] Or not yet disproved to be satisfiable in case of satisfaction-incomplete theories.

3 Constraint Solvers

As we have seen, CLP needs several services from the underlying constraint solvers: a satisfiability test to avoid useless exploration of space and the ability to project the solution space of the current constraint set on a subset of the variables involved (to be able to answer the user query using only the variables that appeared in the query).

We now consider several domains, with their available constraints solvers to see how these demands are actually met or relaxed.

3.1 Equalities in finite and rational trees

The domain of finite trees (FT) is the original domain of pure logic programming languages: Prolog terms that appear in Prolog predicates, such as $\mathsf{f(X)}$ or $\mathsf{f(g(X), 1, Z)}$, are simply the syntactic expression of labeled trees[2]. We present this domain for historic reasons, even if it may look somewhat artificial to the operation research community. The problem of satisfiability of a set of equalities between terms with variables is known as the *unifiability* problem. A solution is given by an *unifier*: a set of variable/term substitutions which, when applied to all the terms, make them identical. The problem is easy, solvable in linear time [PW78]. Furthermore, the algorithms build a most general solution to the constraint set (called a *most general unifier* or *mgu*) which can be used as a "solved form": it may easily be projected on a variable subset and also used as the basis for the next satisfiability test when new constraints are added.

In the domain of FT, the terms $\mathsf{f(X)}$ and $\mathsf{f(g(1, X))}$ are not unifiable: no single FT can simultaneously match both terms. However, unification algorithms naturally build the substitution $\mathsf{X/g(1, X)}$ as a possible solution whereas substitutions of the form $\mathsf{X/t}$, where t is a term that contains the variable X cannot define a solution since they implicitly represent *infinite* trees. Therefore, a specific test, called the *occur-check* should be inserted in the algorithm to avoid infinite cycles in the solution as above. This test yields a best-case complexity which is always equal to the worst-case complexity and was therefore omitted, for efficiency reasons, in most old Prolog dialects.

Nowadays, in order to sanely avoid the occur-check, Prolog languages prefer to solve the unifiability problem in the domain of *rational trees* (RT), which may be infinite, but which have a finite representation (the finite representation of an infinite tree contains cycles). In RT, $\mathsf{f(X)}$ and $\mathsf{f(g(1, X))}$ are unified by $\mathsf{X} = \mathsf{g(1, X)}$, which implicitly gives the infinite tree $\mathsf{f(g(1, g(1, g(1, \ldots))))}$ as the solution. Almost linear algorithms are available. The language PROLOG II of Marseille [Col82b] has been the first Prolog which chose to solve the unifiability problem in the domain of RT.

3.2 Linear (in)equations in $\mathbb{R}$ or $\mathbb{Q}$

Various CLP languages enable the user to express linear equalities and inequalities in the domain of $\mathbb{R}$ (approximated using floating point numbers) or $\mathbb{Q}$, using infinite precision rational numbers.

For linear equations, the usual technique of Gaussian elimination may be used, with a quadratic worst-case time complexity. When inequalities are introduced, polynomial time algorithms are still available [Kha79, Kar84], but since these algorithms are either less efficient practically or difficultly made incremental, all existing languages have, to our knowledge, decided to rely on variations of the Simplex algorithm, despite its exponential time worst case

[2] $\mathsf{f(X)}$ represents all trees with a root labelled f and a single son, which may be any tree.

complexity. The main variations consist in extending the Simplex to deal with negative numbers and strict inequalities [LM92]. As it is well known in the integer linear programming community, the Simplex can be extended to efficiently cope with a growing set of constraints and some incrementality is possible. The projection of a polyhedral set on a variable subset can be performed using a simple algorithm from Fourier, which has been tuned for redundancy elimination in [Imb93, JMSY93].

Even if satisfiability alone is actually needed in the CLP framework, the possible optimization performed by the Simplex is usually made available at the user level through a specific predicate.

3.3 Finite domains

The introduction of finite domains (FD) into CLP languages is certainly connected with the important growth of the field of "Constraint Satisfaction Problems" (CSP) in the artificial intelligence community [Tsa93, AS93]. The CSP framework is devoted to the problem of satisfying a set of constraints, without any limitation on the constraint types, in any FD.

Definition 1 *A CSP* $= (X, D, C)$ *is defined by:*

- *a set* $X = \{x_1, \dots, x_n\}$ *of* n *variables;*
- *a set* $D = \{d_1, \dots, d_n\}$ *of domains. Domain* d_i *contains the set of values that may be considered for variable* x_i*. We note* d *the size of the largest domain;*
- *a set of constraints* $C = \{c_1, \dots, c_e\}$ *of* e *constraints. Each constraint* c_i *is defined by:*
  - *the set of variables* $X(c_i) \subset X$ *involved in constraint* c_i*;*
  - *a relation* $R(c_i)$ *on the variables of* $X(c_i)$*, i.e., a subset of the Cartesian product of the domains of the variables of* $X(c_i)$*. This relation defines the tuples of values which may simultaneously be assigned to the variables involved in the constraint.*

Example: *Consider a CSP with 3 variables (*$X = \{x_1, x_2, x_3\}$*), all variables having the same domain* $\{$*white, black*$\}$*. Three constraints,* c_1*,* c_2*, and* c_3 *involve respectively* $\{x_1, x_2\}$*,* $\{x_2, x_3\}$ *and* $\{x_1, x_3\}$ *and are defined by the same relation* $R = \{(\textit{white}, \textit{black}), (\textit{black}, \textit{white})\}$*.*

An assignment of values to a subset $Y \subset X$ of the variables is said to be *consistent* (or *locally consistent*) iff all the constraints $c_i \in C$ such that $X(c_i) \subset Y$ are satisfied by the assignment, *i.e.*, only authorized combinations of values, as specified in the relations are used in the assignment. In our example, the assignment $\{x_1 \leftarrow \textit{white}, x_3 \leftarrow \textit{black}\}$ is locally consistent while $\{x_1 \leftarrow \textit{black}, x_3 \leftarrow \textit{black}\}$ is not.

A solution of a CSP is simply a consistent assignment of the whole set X of variables. A satisfiable CSP (a CSP with at least one solution) is also said to be consistent[3]. Our example CSP has no solution, the CSP is inconsistent.

In the "pure" CSP framework, the domains d_i are supposed to be finite domains and, when their size remains reasonable, the relation associated to each constraint can effectively be described by the set of tuples of values that correspond to authorized combinations. The

[3] One should not confuse the property of consistency of an assignment, which is decidable in polynomial time, and the property of consistency of a CSP, which defines an NP-complete decision problem.

interesting point is that any type of constraint, either linear or not, can be expressed. Numbers and symbols may also be simply mixed together. A lot of CSP techniques may be extended to the case of relations expressed in intention (linear or non linear constraints over $\mathbb{N}$ for example) [Dav87]. Some of these techniques have also been extended to the case of infinite domains (subsets of $\mathbb{R}$ usually) and are available in some CLP languages [SH91, Hyv92, Lho93]. In the following, we only consider FD.

An usual restriction in the "pure" CSP field is also to restrict oneself to binary constraints, involving only two variables. In that case, two graphs, called respectively the constraint graph and the consistency graph of the CSP, may be defined:

- The *constraint graph* has one vertex for each variable and one edge for each constraint. For non binary CSP, a similar hyper-graph can be defined. This graph only describes the problem structure.
- The *consistency graph* has one vertex for each value of each variable and one edge for each compatible pair that appear in a constraint. This graph is n-partite since no edge can appear between two values of a given variable. It completely defines the problem.

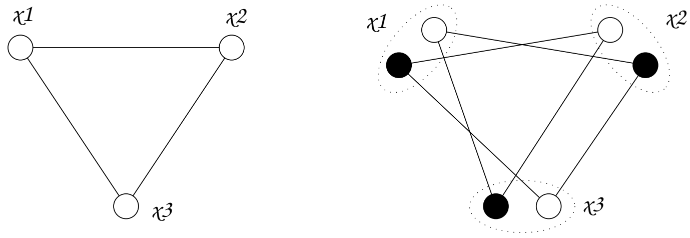


Figure 2: The constraint and consistency graphs of a CSP

These two representations are used in Fig. 2 to describe our example CSP. In the FD case, an important difference with the previous domains of trees and linear programming in $\mathbb{R}$ is that the satisfiability problem becomes NP-complete. This is shown simply by the following observation:[4]:

> **Example:** *the* GRAPH k-COLORABILITY *problem can simply be expressed as a CSP. If you consider a graph* (V, E)*, one variable* x_v *is associated to each vertex* v*, all the domains are identical and contain* k *values. Finally, one difference constraint between* x_v *and* x_w *is associated to each edge* $(v, w) \in E$*. Note that our example is simply the* GRAPH 2-COLORABILITY *problem on a fully connected graph of 3 vertices (a 3-clique).*

The most naive technique that may be used to solve the satisfiability problem is the backtrack algorithm: given an initial consistent assignment A_k of size k, it tries to assign a new variable x_{k+1}. If no value in d_{k+1} yields a consistent assignment, a backtrack occurs on the previous variable assigned. The algorithm starts with an empty (and therefore consistent) assignment and has a worst-case complexity in $O(ed^n)$.

Since this algorithm is highly inefficient, it would be unreasonable to use it to prove satisfiability in the frame of a CLP language (one satisfiability test being performed after each step of the computation). Furthermore, it seems difficult to give this algorithm some incrementality. Therefore, all existing implementations have chosen to weaken the "satisfiability" property by using the polynomial worst-case time *local consistency* properties defined in [Mon74, Mac77].

[4] This result shows that the satisfiability problem in the binary CSP restriction is still NP-complete and this restriction is made, in theory, "without loss of generality".

3.3.1 Local consistency

The notion of "local consistency enforcing" defines a whole family of techniques which, in practice, transform an initial CSP in a (hopefully) simpler problem, with the same solution set. This process is also known as a "filtering" process, or as a "constraint propagation" mechanism[5].

To each type of filtering process is associated a so-called "local consistency" property that is actually enforced by the filtering. The main class of local consistencies has been defined in [Fre78] and is called k-*consistency*.

Definition 2 *A CSP* (X, D, C) *is said to be* k*-consistent iff any consistent assignment of* $(k-1)$ *variables can be extended to a consistent assignment of* k *variables on any unassigned variable.*

A CSP (X, D, C) *is said to be strongly* k*-consistent iff it is* j*-consistent for all* $j = 1, \ldots, k$.

It is quite easy to prove that a CSP of n variables which is strongly n-consistent is satisfiable (and furthermore, any consistent assignment can be extended to a solution). However, a satisfiable CSP is not necessarily n-consistent[6].

A polynomial time algorithm (in $O(n^k.d^k)$) that enforces strong k-consistency has been proposed in [Coo89]. The CSP built is called the k-consistent closure of the CSP. Since the CSP obtained is equivalent to the original CSP, we get the following property:

Property 1 *If a CSP has an empty* k*-consistent closure (a domain is empty), then it is unsatisfiable. The converse is naturally false.*

Therefore, strong k-consistency enforcing offers a polynomial time relaxation of satisfiability. The approach followed by most CLP languages is to only enforce some local consistency property instead of satisfiability. Backtracking occurs when an empty closure is obtained, since this actually proves unsatisfiability. Usually, only strong 2-consistency, also called *arc-consistency* [Ull66, Wal72, Mon74] in the framework of binary CSP, is enforced.

Definition 3 *A binary CSP is said to be arc-consistent iff none of its domains is empty (this is 1-consistency) and every assignment of one variable can be extended to a consistent assignment of size 2 on any unassigned variable.*

> **Example:** *Our example problem of Fig. 2 is unsatisfiable, but it has an non empty arc-consistent closure, and is indeed already arc-consistent (enforcing arc-consistency on this problem is completely useless).*

Basically, arc-consistency enforcing works as follows: if a value of a given domain d_i does not appear in any of the authorized combinations of one of the constraints that involve x_i, then this value cannot belong to a solution and can simply be deleted. A single pass on all variables and constraints is usually not sufficient and the process is performed iteratively until quiescence.

---

[5]This process can be related to cutting plane generation in integer linear programming considering the combined results of [Hoo88] and [dK89].

[6]It can be the case that some consistent assignment of $n-1$ variables can not be consistently extended to the n^{th} variable. Satisfibility simply implies that some consistent assignment of $k-1$ variables do extend to a consistent assignment of all variables.

Example: *The CSP whose consistency graph is illustrated in Fig. 3, simply obtained by removing the edge (white, black) between x_1 and x_2, is a good example of non consistent CSP which is also non arc-consistent and whose arc-consistent closure is empty. On a first pass, the values numbered 1 will be deleted because they are not connected to any value on an adjacent variable, then the values numbered 2 will be deleted, for the same reason (thanks to the previous deletion)... until quiescence: here, all the values are deleted. Actually, one could stop on step 2, when the domain of x_3 becomes empty, since it suffices to prove inconsistency.*

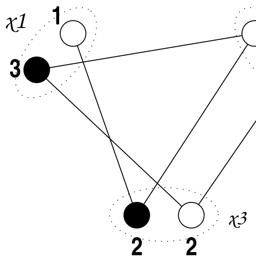


Figure 3: A CSP with an empty arc-consistent closure

This simple arc-consistency enforcing algorithm has undergone several optimizations, from AC-1 (described in [Mon74, Mac77]) to AC-7 (described in [BFR95]). With AC-4, we get optimal time complexity in $O(ed^2)$ and AC-6 gives further efficiency and a space complexity in $O(ed)$. However, most CLP languages rely on modified versions of AC-5 [vHDT92], because it is easily tuned to handle specific classes of constraints more efficiently.

Actually, no CLP language, to our knowledge, does enforce local consistency at a level higher than strong 2-consistency. Pragmatically, a stronger level would mean earlier unsatisfiability detection, and therefore less nodes explored in the Prolog search tree, but it would also mean more work at each node. Experimentally, arc-consistency appears to be a reasonable tradeoff.

A last interesting property of all the local consistency enforcing algorithms is their natural incrementality with respect to the addition of constraints in the problem. Indeed, the CSP obtained after arc-consistency enforcing, if non empty, has smaller domains than the initial CSP and is nevertheless equivalent to the original CSP: it may simply be used instead of the original CSP.

3.3.2 From local consistency to satisfaction

Since CLP languages only enforce arc-consistency, the information available at each node of the Prolog tree is not a "solved form" of the CSP which could be projected on the query variables, but simply reduced domains with no guarantee of satisfiability[7].

Since satisfaction is a practically important problem, an additional mechanism must be used to solve the satisfaction problem. The idea, which originates in the CSP field, is to use a hybrid of backtrack tree search and arc-consistency enforcing [Nad89, vH89].

The algorithm, known as "Really Full Look Ahead"[8], is simply a backtrack tree search where the assignment consistency test is replaced by arc-consistency enforcing. Given an initial

[7]The strong n-consistent closure of a CSP would give a "solved form", but at the cost of an exponential amount of memory in the worst case.

[8]Some CLP languages actually use a weaker level of local consistency enforcing than full arc-consistency, defining algorithms known as "Forward-checking" or "Partial Look-Ahead". See [Nad89, vH89].

arc-consistent CSP with variables $x_1, \ldots, x_k$ assigned, it tries to assign a new variable x_{k+1}. If no value in d_{k+1} yields a CSP with a non-empty arc-consistent closure, a backtrack occurs on the previous variable assigned and domains should be restored to the state they had before this assignment. The algorithm starts with an empty (and therefore consistent) assignment.

The algorithm always explores less nodes than the simple backtrack algorithm sketched in Section 3.3 since a locally inconsistent variable assignment of a subset of X will induce an empty arc-consistent closure. In practice, the algorithm is often several order of magnitude more efficient than the backtrack algorithm. It benefits from two main types of heuristics:

- *variable ordering* heuristics decide dynamically, during search, which will be the next variable x_k that will be assigned. The general principle underlying all these heuristics is the so-called *first-fail principle*: choose a variable which is highly constrained so that if a failure must occur, it is rapidly detected. A good choice is to favor variables with a small actual domain and with a high degree in the constraint graph.

- *value ordering* heuristics decide dynamically which value should be assigned successively to the variable x_k that has been chosen. Even if some "general purpose" value ordering heuristics have been proposed (see [MJPL90]), the best results are usually obtained with heuristics that take into account the precise problem class at hand.

  See [Tsa93], chapter 6 for further information on variable and value ordering heuristics.

In CLP languages, the actual tree search is done using the underlying non-determinism of the Prolog language. A specific predicate, usually called `indomain(X)`, enumerates and successively assigns each of the values of the domain of the CSP variable `X` to this variable. Since this mechanism is programmed in Prolog, it can actually be *completely* modified and tuned to exploit heuristics and specific properties of the problem at hand.

Example: *the following* CHIP *code*[9] *solves the* GRAPH 2-COLORABILITY *instance of Fig. 2 using the finite domain solver. The predicate* `2-color` *defines the three domain variables and the three constraints, while the* `labeling` *predicate is in charge of the enumeration process. The predefined predicate* `delete(X,L,R,first_fail)` *chooses a CSP variable* `X` *in the list* `L` *with a smallest domain and leaves the remaining variables in* `R`.

```
2-color(X1,X2,X3) :-
   % Domain declaration
   X1 :: 1..2,
   X2 :: 1..2,
   X3 :: 1..2,
   % Three ≠ constraints
   X1 #\= X2,
   X2 #\= X3,
   X3 #\= X1,
   % Look for a solution
   labeling([X1,X2,X3]).
```

```
labeling([]).
labeling([X|Y]) :-
   % Choice of a variable
   delete(Var,[X|Y],Rest,first_fail),
   % Choice of a value
   indomain(Var),
   labeling(Rest).
```

[9]See the beginning of Section 5 for more specific syntax explanations.

If the query `2-color(A,B,C)` *is given to the CLP language, the tree search will be very small, since as soon as any of the three variables is assigned, and whatever its value, an empty arc-consistent closure is obtained and backtrack occurs. The CSP being unsatisfiable, the answer will simply be* '`No`'.

Beyond satisfaction, optimization is also possible through specific predicates using variants of the Branch and Bound algorithm, often related to the "Depth First Iterative Deepening" [Kor85].

3.3.3 Integrating CSP in CLP

As we said, in "pure" CSP, constraints are often supposed to be expressed by the set of authorized tuples. If this language is extremely powerful, since it allows the expression of any constraint on a finite domain, it is also very cumbersome. Therefore, all CLP languages have limited the set of constraints that the user may express to a specific language which usually contains basic arithmetic constraints, often linear (in)equations with unlimited arity.

The traditional arc-consistency enforcing algorithms of general CSP can be finely tuned to better take into account the nature of these constraints. Most languages, to our knowledge, essentially perform a so-called "bound propagation", related to interval calculus, which can be easily extended to real (floating point) domains [Dav87, vH89, Lho93].

Example: *for a linear equality* $\sum \mathsf{a}_i.\mathsf{x}_i = 0$*, all variables* x_i *having a finite integer domain, and considering, for the sake of simplicity, that all the* a_i *are positive, it is possible to update the domain of* x_j *as follows. Since*

$$\mathsf{x}_j \leq \sum_{i \neq j} - \min(\mathsf{a}_i.\mathsf{x}_i)/\mathsf{a}_j$$

then all the values in the domain of x_j *which are larger than the right member can be removed, without losing any solution. By symmetry, an analogous lower-bound may be built.*

As in traditional arc-consistency enforcing, if the domain of x_j *is modified, then the domains of all the variables connected to* x_j *through a constraint should also be updated (and so on, iteratively, until quiescence).*

For some classes of constraints, such simple bound propagations may suffice to enforce arc-consistency [vHDT92].

However, the restriction to some arithmetic language is often too strong to easily express some natural and useful constraints, especially constraints on symbolic domains. Therefore, these constraints are usually introduced in the constraint language using specific predicates which are naturally called "symbolic constraints". An example of such a constraint is the "`element(I,L,X)`" constraint which involves an integer or a domain integer variable `I`, any finite domain variable `X` and a list of values `L`. The constraint is satisfied iff the $\mathtt{I}^{\text{th}}$ element of the list `L` is equal to the value of `X`. Such constraints are usually handled using *ad-hoc* propagation mechanisms, which may or not, enforce arc-consistency. The important issue here is to get the right tradeoff between the power of the local consistency enforcing (in terms of number of values deleted) and the actual efficiency of the algorithm.

Since one cannot hope to extensively introduce all "useful" constraint types in a fixed language, some of the expressive power of the CSP framework is lost. For example, the disjunctive constraints that naturally appear in job-shop scheduling (see Section 5) cannot be naturally expressed as linear inequalities whereas they can be expressed as a single CSP constraint, which could naturally be propagated using any arc-consistency algorithm.

3.4 Other domains

The three previous domains are certainly the most frequent ones in CLP languages. We rapidly review here solvers from other domains that have been implemented in a CLP language:

- *boolean algebra*: this domain, which only contains the 2 truth values (true and false) is a special case of finite domain and the previous techniques can be used [CD93], with all their limitations. The usual constraint language is the language of mathematical propositional logic and includes conjunction, disjunction, implication, equivalence... The satisfiability problem is the GENERAL SAT problem [GJ79], which is again NP-complete.

  A large number of techniques have been proposed to solve the satisfiability problem. Enumerative techniques such as Davis and Putnam's procedure [DP60], which are closely related to CSP tree search algorithms, are not easily made incremental. PROLOG III uses a variant of SL-resolution described in [BB88]. This algorithm requires that formulas be translated to clausal form (a conjunction of disjuncts), which may be quite costly. It is naturally incremental and yields a "simplified" form of the constraint set. Binary Decision Diagrams [Bry92] offer an efficient, incremental representation of boolean formulas. One of the boolean solvers of CHIP uses a variable elimination algorithm along with these BDDs. A BDD explicitly represents all the solutions of the constraint set. Despite efficient compression methods, it may occupy an exponential amount of memory in the worst-case.

  An important difference with finite domains solvers is that most CLP languages try to offer the user a "legible" representation of the current boolean constraint set (whereas only restricted domains or a solution are available for finite domains). This may define a very difficult problem.

- *non linear equations in* $\mathbb{R}$: several approaches exist. In a first approach, the expression of non linear constraints is possible, but these constraints are simply "frozen" (ignored) until they become eventually linear, when enough variables get assigned (by the user or because their value can be entailed from the constraints). These constraints are then communicated to the linear programming solver. The approach is quite simple and allows the use of polynomials or transcendental functions. If some non-linear constraints remain non linear (and frozen) when a solution is obtained, then the answer given by the CLP language ignore the frozen constraints and may be meaningless[10]. For example, a constraint such as "`X=pow(Y,Z)`" stating that $\mathtt{X} = \mathtt{Y}^{\mathtt{Z}}$, will be delayed until either (1) `Z` is known to be equal to `0` (and `X=1`) or `1` (and `X=Y`) or (2) `Y` is known to be equal to `1` (and `X=1`) or (3) two variables among `X, Y, Z` have a known value and the remaining one can be evaluated.

  Another approach consists in trying to solve the non-linear equations using dedicated algorithms. CAL [ASS+88] uses a specific solver[11] to solve polynomial equations in $\mathbb{C}$ (a relaxation of the original problem in $\mathbb{R}$) to eventually prove its inconsistency.

  A last approach, which is not limited to polynomial equations, is to use extensions of the finite domains local consistency properties to floating point domains (see [Dav87, SH91, Hyv92, Lho93]).

Several other domains, including strings, sets, features trees... have received attention from the CLP community (see [Coh90, JM94]).

[10] CLP($\mathbb{R}$) answers '`Maybe`' when some constraints are still frozen and a solution is found.

[11] Here, Buchberger's Gröbner bases algorithm is used, with a worst-case doubly exponential complexity [Buc85].

4 Some existing tools

In this section, we rapidly present the main features of some well-known CLP language implementations. Most of these languages are actually commercial products whose underlying mechanisms is not precisely documented: the constraint solvers are black boxes and the precise operational semantics of some constraints may be unspecified. A lot of other CLP languages exist and we invite the reader to fetch [JM94, FAQ95] to better perceive the variety of available implementations (often for free).

As in classical logic programming, a constraint logic program is a collection of clauses, possibly involving some constraints.

```
% Example in Prolog III (non standard syntax)
c1(X,Y,Z) -> c2(R), {Z = 2X+Y, X>=2, Y>3} ;
c2(X) -> {X > 4} ;

% Example in Chip V4 (Edinburgh syntax)
c1(X,Y,Z) :- c2(R), Z #= 2X+Y, X #>= 2, Y #> 3.
c2(X) :- X #> 4.
```

To activate a program, one has to express a query, *i.e.,* a sequence of goals and constraints; if no goals are specified, the query is simply to solve the constraint system. The system then behaves like a classical prolog, but each time a clause is used, the associated constraints are added to the current constraint set. Constraints solvers are then used to check the consistency of the current state. Backtrack occurs when either a goal cannot be proved or the current set of constraints becomes inconsistent. If the goals can be proved, the current substitutions of the variables are a solution, provided that they satisfy the set of constraint. Hence the "answer" is given under the form of a set of variable bindings and constraints. Completeness of the solving process depends on the domain of computation used.

4.1 CLP(R)

The language CLP($\mathbb{R}$) [HJM+92] is an implementation of the general CLP($\mathcal{X}$) scheme defined by J. Jaffar and J-L. Lassez in [JL87]. The first release was available around 1986 and the current release 1.2 is available for free from IBM for academic and research purposes only[12].

CLP($\mathbb{R}$) is perhaps the best system to start with if you want to discover what "pure" constraint logic programming is[13]. The only domain tackled, beyond the usual tree domain, is, as the name says, real numbers, approximated using floating point numbers. This keeps the system small, homogeneous and with a simple syntax.

The underlying solver is a Simplex-derived solver, with a specific Gaussian elimination module for linear equalities. Because of the floating point implementation, the strict inequalities, which are available as in PROLOG III, do not have the precise semantics that can be obtained using infinite precision rational numbers.

The system contains a mechanism for delaying non-linear constraints until enough other numerical constraints make them linear. As we said in Section 3.4, if a solution is found when a non-linear constraint is still delayed, CLP($\mathbb{R}$) simply answers '`Maybe`'. Anyway, a large set of

[12] For more information, contact Joxan Jaffar via e-mail (joxan@watson.ibm.com).

[13] However, CLP($\mathbb{R}$) does not include a finite domain solver, one of the main novelty of constraint programming. People interested in finite domains may try to get `clp(FD)`, another free system [CD93] available via anonymous FTP at `ftp.inria.fr:/INRIA/Projects/ChLoE/LOGIC_PROGRAMMING/clp_fd`.

primitives is available to express non-linear constraints (*e.g.,* $\mathsf{Z} = \mathsf{X} * \mathsf{Y}$, $\mathsf{Y} = \log(\mathsf{X})$) and the language has been essentially used to deal with partially non linear problems.

The system contains facilities for "meta-programming" with constraints and allows to explicitly manipulate terms and arithmetical constraints, a coded form being available to the user. These facilities are not available in any of the other languages considered here.

The initial implementation was interpreted. The actual release uses a byte-code compiler for a better efficiency of the underlying Prolog. It is entirely written in the C language and runs on most existing Unix workstations and includes MS-DOS support. CLP($\mathbb{R}$) has been used to solve various real problems, analysis and synthesis of analog circuits [HMS92], stock option analysis [HL88] or problems from computational biology such as DNA sequencing by restriction site mapping [Yap93].

4.2 Prolog III

A. Colmerauer and his team first defined the model of a Prolog with constraints around 1984, as an extension of the PROLOG II model. The commercial version of PROLOG III has been officially announced in 1990.

Compared with other tools (*e.g.,* CHIP), PROLOG III is very homogeneous, providing, in particular, a nice uniform syntax. A PROLOG III program, like a Prolog one, consists of a collection of clauses, which can possibly contain constraints. Constraints are syntactically distinguished from classical Prolog predicates using braces.

PROLOG III supports three domains of computation:

- *rational numbers*: like CHIP, PROLOG III uses exact precision rational number arithmetics. The numerical module handles linear inequations (where every variable is supposed to be positive). It is essentially an incremental Simplex-like solver, which checks that the accumulated numerical constraints can be satisfied (as soon as the resolution process triggers a new constrained clause, the solver is fed with the attached constraints: it ensures a complete resolution of the constraint system). Of course, it is also possible to specify some (linear) criteria to be optimized. Note that in addition to traditional Simplex techniques, much attention has been paid to strict inequalities.

  PROLOG III also allows the specification of non-linear constraints. They are in fact delayed until the binding of some variables makes them linear. If it is not the case, the solving process may give an inconsistent set of constraints as answer, without detecting the inconsistency.

  PROLOG III does not naturally handle discrete domains. Nevertheless, it is possible to express some linear constraints over integers: linear arithmetics constraints over integers are first solved in $\mathbb{Q}$. Enumeration predicates must be used in order to find an integer solution.

- *booleans*: the boolean module is based on clausal forms. The underlying algorithms are incremental versions of the SL-resolution algorithms. They are able to check the consistency of a set of boolean constraints; moreover, they simplify the constraint system into a set of constraints involving a minimal set of variables.

- *finite lists (or finite strings)*: for finite strings, there exists a single function to concatenate two strings, denoted by "." and the only constraint is the equality constraint. PROLOG III delays the evaluation of any string constraint until the length of the string is known. Technically, the string solver is based on a restricted string unification algorithm.

PROLOG III is developed and distributed by PROLOGIA (France). It is an interpreter with possible connections to the C language. It is available on Unix workstations, PC and MacIntosh. While the other constraint-based tools focus on big industrial accounts, PROLOG III was first successfully distributed to the academic and research fields. Let us nevertheless cite two applications: the first one is an expert system analyzing failures in motors (used by Daimler-Benz and Bosch, Germany). The second one (for Delacroix, France), optimizes the cutting of wood panels.

4.3 CHIP V4

Like PROLOG III, CHIP V4 is a Prolog-based constraint tool. It is a direct heir of the ECRC research prototype CHIP, but differs from the prototype in many aspects (notice that the ECRC continues to develop its own system, ECLIPSE, which, unlike CHIP V4, is an open system). In a CHIP program, constraints are not syntactically distinguished from the other predicates. CHIP provides a large number of pre-defined constraint predicates, on different domains of computation:

- *integers / finite domains*: the most important feature of the CHIP system is the introduction of arithmetic constraints over finite domains. In addition, a rich set of symbolic constraints is provided (for instance, the "`alldifferent(Variable_List)`" predicate, which specifies that all the variables in the list must have different values; let us also cite the cumulative constraint, dedicated to scheduling applications, see Section 5).

  Classical constraint propagation methods (*e.g.,* arc-consistency enforcing) originating from the CSP field are used to handle finite domain constraints. Since they are of course incomplete, choice predicates are also provided which allow the generation of values for the finite domain variables.

  Moreover, minimization is done using Branch & Bound: the predefined predicate "`min_max(Goal,C)`" finds a solution which minimizes the maximal value in a list of cost terms `C`. In practice, this is done by finding a first solution, evaluating `C` and then restarting the search with a new constraint requiring `C` to be lower than this value. Nevertheless, as discussed in [vH89], optimization problems resolution is still to be enhanced.

- *rationals*: linear rational constraints are handled by an extended Simplex algorithm associated with some Gaussian elimination. Only linear arithmetic constraints are allowed in CHIP V4. The rational constraint solver is then obviously complete. Note that optimization predicates over rational linear constraints are also provided.

- *booleans*: boolean constraints are solved by a variable elimination based boolean unification algorithm, which is totally deterministic but still complete. Nevertheless, since constraint solving in boolean algebra is NP-complete, CHIP may encounter problems which cannot be solved in a reasonable time (this is of course a general problem, which may also occur in other languages).

Finally, CHIP gives the user the possibility to define its own constraints and control their execution, via the use of demons: these have the effect of re-evaluating a specified goal each time a given triggering event (such as a change in a variable domain) occurs. This mechanism implements local consistency algorithms for user-defined constraints. The declarative "`if-then-else`" construct also allows a (limited) way of communicating information between heterogeneous constraints.

CHIP V4 is developed and distributed by COSYTEC (France). The product also offers graphic capabilities, connections to databases and connections to C language. It runs under Unix (Sun, IBM, HP, Dec) and Dos (PC). Many applications have been developed, for instance, the planning application "Plane" [BCF95] provides 5-year schedules for Dassault-Aviation (France) and the multi-user scheduling tool developed for Monsanto (Belgium).

4.4 ILOG Solver/Schedule

ILOG-SOLVER is often described [JM94] as a C++ library that implements CSP algorithms on finite and floating point domains rather than as a CLP language. Presumably, this is due to the fact that SOLVER does not use the usual Horn clauses language and does not include a solver on rational or finite trees.

Anyway, this description is somewhat unfair, since SOLVER includes an extension of C++ that enables the expression of non-determinism, traditionally offered by Horn clauses in other languages. The domain of finite/rational trees is not included simply because it does not seem useful for the combinatorial problems addressed by SOLVER.

SOLVER provides several classes of variables, each implemented as a C++-class. These classes define the domains handled by the language:

- *integer variables*, whose domain is either an interval or an enumeration of integers. These variables can be involved in linear or non linear arithmetic constraints. Non linear constraints are propagated. According to the authors [Pug94], the handling is similar to the propagation mechanism described in [vH89].
- *floating point variables*, whose domain is an interval. These variables can be involved in linear and nonlinear arithmetic constraints, or monotonic operators such as "log". The underlying solver consists of a limited form of arc-consistency enforcing which, according to the authors, owes much to [Lho93].
- *boolean variables* (domain $\{true, false\}$). These are also handled using constraint propagation.
- *set variables*, whose domain is a set of sets. Initially, the domain of such a variable contains all the subsets of a given initial set of objects. The constraints that may be expressed state that some elements must be in the set, that other should be excluded, or can limit the range of the cardinality of the set.

All these constraints, are handled using a uniform underlying constraint propagation mechanism, related to arc-consistency enforcing. Some additional "symbolic constraints" are provided and the user has the ability to define *ad-hoc* propagation mechanisms for user defined constraints.

Finally, the library tries to offer "object oriented" features: members of user-defined classes may be "CSP" variables and the user may define "constraints of classes", which are inherited by instances.

SOLVER is developed and distributed by ILOG (France). The product, distributed as a C++ library, is fully compatible with all other ILOG C++ software components, including connection to databases, graphical interfaces, rule-based programming... It runs under several Unix-based systems and under Windows (PC). SOLVER has been used to tackle several real combinatorial problems, from locomotive scheduling used by the SNCF [Pug92] to personnel management in the French army [PPMD94].

	Rational trees	Finite trees	Strings	Finite domains	Booleans	Rationals	Floating points	Integers	Sets
CLP($\mathbb{R}$)	−	+	∘	−	−	−	+	∘	−
PROLOG III	+	-	+	−	+	+	∘	∘	−
CHIP	−	+	∘	+	+	+	∘	+	−
SOLVER	−	−	−	+	+	−	+	+	+

+: constraints are handled on this domain
∘: authorized variable type, no constraints
−: non-existent variable type

Table 1: Domains addressed in each language

ILOG-SCHEDULE [Pap94] is a C++ library of scheduling object classes that automates the use of SOLVER for representing finite capacity scheduling problems. It offers tuned constraint propagation mechanisms for the usual constraint types that occurs in scheduling. Typical classes include *activities* and various *resources* types (renewable or not, unary or finite capacity...). See Section 5 for a first approach of job-shop-scheduling problems using a CLP language.

4.5 Summary

Table 1 gives an overview of which domains are addressed by which CLP languages. The language CHIP appears to be the more general CLP language. This judgment should be somewhat tempered. Indeed, all the languages which handle finite domains can also handle booleans and integers, with the ability of handling non convex domains such as $\{1, 2, 3,\ 7, 8, 9\}$. PROLOG III may also handle integers (rationals with a denominator equal to one), but it cannot handle non convex domains. Since the Simplex only proves satisfiability in $\mathbb{Q}$, an expensive enumerating process, as for finite domains, must be used.

Note also that one single problem may often be formalized using different models: a scheduling date may be modeled as integer, rational or floating point numbers.

More importantly, if several constraint solvers exist in the same language, it is often impossible to use the solvers simultaneously on "mixed" problems, where more than one type of variable occurs in the same constraint, because it is usually impossible to relate a variable from one domain to a variable from another domain. For instance, the relation between a rational and its integer part cannot be expressed as a constraint in CHIP. Along the same idea, it is usually not possible to associate a boolean variable to the fact that a given constraint is satisfied, *e.g.,* $\ell = (x < y)$.

Thanks to its underlying uniform constraint propagation mechanism, SOLVER is quite advanced in this direction and can handle some "mixed" constraints. For example, a constraint of equality exists between floating point and integer variables.

5 Application to Job-Shop Scheduling

A scheduling problem arises when a set of interdependent tasks is to be organized in time. Such a situation appears for example in project planning, service activities, manufacturing shops, computer systems or examination timetabling. The problem is to locate a set of tasks in time, each task needing one or several resources during its execution. The constraints to satisfy may be various: technological (sequencing, routing), resource (limited capacity), temporal location (release dates, deadlines), ... Most optimization problems in the scheduling field are NP-hard [GJ79]. To find solutions to these combinatorial problems, operational research developed exact procedures like Branch & Bound, in order to find lower and upper bounds of the optimal solution, or designed heuristics with some refinements specific to the problem at hand for pruning the search space [ML93].

Scheduling problems have been widely studied in the literature [Bak74, Gra81, GOT93, Pin95]. Amongst many different typologies, a basic classification may be based on the resource environment: single machine, parallel machines, flow-shop, job-shop, open-shop or resource-constrained problems.

In this section, an illustration of the application of CLP to scheduling problems is given through the job-shop scheduling problem, as in [Wal94]. The following examples of modeling are written in the same language for the sake of unity. The CHIP language was chosen and more specifically its CSP-based solver over finite domains. A domain variable (the CHIP equivalent of a CSP variable) is defined via "`:: a..b`" where "`::`" is the domain definition operator and "`a..b`" a domain of consecutive set of integers. Furthermore, constraints on finite domains are preceded by the special character "`#`" to distinguish them from other arithmetical constraint types in CHIP.

5.1 Problem Statement

In the job-shop problem a set of n jobs has to be processed using a set of m machines. Each job consists of a set of ordered tasks forming a routing; each task runs on a separate machine. At any time a machine can process only one task. Preemption *i.e.,* interruption of a started task before completion, is not allowed. The objective is often to find a schedule that minimizes the *makespan*, *i.e.,* the total duration.

This section aims at giving information about how to simply model such problem, deriving benefit from assets of CLP. Facing the important number of constraints for large job-shop scheduling problems, an approach using CLP does not have to solve the constraints involved but to propagate them by local consistency techniques, in order to reduce the search space as drastically as possible.

5.2 Precedence constraints

The jobs are considered one after the other. For each job a list of tasks is given in the order of the *routing*. Thus task i (with start time T_i and duration D_i) from the list must precede task $i+1$. To satisfy this precedence constraint, one must post a relative location constraint between tasks i and $i+1$:

$$T_i + D_i \leq T_{i+1}.$$

To each task of a job is also associated a start domain variable, posting in this way an absolute location constraint (`Max` corresponds to a given horizon).

```
job([T1|T],[D1|D],Max) :-
    T1 :: 0..Max,                % Declaration of the first variable
    routing([T1|T],[D1|D],Max).

routing([Tm],[Dm],Max) :-
    Tm + Dm #<= Max.             % Constraint on the latest variable
routing([T1,T2|T],[D1,D2|D],Max) :-
    T2 :: 0..Max,                % Declaration of other variables
    T1 + D1 #<= T2,              % Precedence constraint
    routing([T2|T],[D2|D],Max).
```

5.3 Disjunctive constraints

As it is not possible to process simultaneously two tasks on the same machine, one needs to generate disjunctive constraints between competing tasks. In a general way, a disjunctive constraint (or *competition*) between a pair (i, j) of tasks states that i precedes j or j precedes i. It yields:

$$(T_i + D_i \leq T_j) \text{ or } (T_j + D_j \leq T_i).$$

Different ways for modeling this kind of constraint are presented below.

1. *Choice points*

   The most natural way to model an alternative is made through the introduction of choice points [DSv90, vH89]:

   ```
   competition(Ti,Di,Tj,Dj) :- Ti + Di #<= Tj.     % i precedes j
   competition(Ti,Di,Tj,Dj) :- Tj + Dj #<= Ti.     % j precedes i
   ```

   This implementation is clearly nondeterministic due to its transcription into a disjunction of constraints (one packet of two clauses). Hence this way of modeling a disjunctive constraint is highly inefficient: indeed, for n competing tasks, it develops a search space in $O(2^n)$.

2. *Conditional propagation*

   The conditional propagation of CHIP allows the programmer to avoid the aforementioned choice points. It uses a demon mechanism to limit the nondeterministic behavior of previous implementation by postponing the choices until enough information can be deduced from the constraints. As soon as the `if` condition is true for *all* possible values for the variables, the `then` branch is selected[14]. Thus the principle of the disjunctive constraint implementation is the following: if the minimal finish time of a task j is larger than the maximal start time of another task i, i must be scheduled before j.

   ```
   competition(Ti,Di,Tj,Dj) :-
       if Tj + Dj #> Ti  then Ti + Di #<= Tj,
       if Ti + Di #> Tj  then Tj + Dj #<= Ti.
   ```

   Although more efficient than the previous implementation, this method keeps the inconvenience to have to wake the demon for pruning the search tree.

[14] Using "`if-then-else`", the `else` branch is executed when the condition is always false.

3. *Cardinality operator*

   Using the same principle of conditional propagation, van Hentenryck & Deville [vHD93] introduce the *cardinality operator* to impose the minimum and maximum number of constraints to be satisfied among a set of constraints. This operator is a mathematical abstraction which implements the principle "*Infer simple constraints from difficult ones*", at the operational level. Thus with such an operator exactly equal to 1, the disjunctive constraint may be written as follows:

```
competition(Ti,Tj,Di,Dj) :-
    cardinality(1,1,[Ti+Di #<= Tj, Tj+Dj #<= Ti]).
```

   Note that in the case of *two* disjunctive constraints `C1` and `C2`, the cardinality operator may be expressed thanks to the conditional propagation [Cos93]:

```
or(C1,C2) :-
    if C1  then true  else C2,
    if C2  then true  else C1.
```

   The authors show that non-primitive constraints built with the cardinality operator give comparable (or even better) results to builtin constraints, but really improving the flexibility of utilization and expressiveness of cardinality constraints (in order for example to model more general problems than the disjunctive one).

4. *The cumulative primitive*

   The previous implementations concern the so-called bound propagation also known as "2B-consistency" [Lho93], that is to say these rules adjust the extreme bounds of the domain variables (head and tail of a task). However, the disjunctive constraints could also attempt to achieve arc-consistency, without posting any choice point or waiting for a demon to be awaken.

   This is the goal of the cumulative constraint of CHIP which aims at solving resource-constrained scheduling problems; it can also be used for disjunctive problems such as the job-shop problem where the amounts of resources (intensities or capacity) are all equal to 1.

```
competition(Ti,Di,Tj,Dj) :-
    cumulative([Ti,Tj],[Di,Dj],[1,1],1).
```

   In [AB92], the designers of the cumulative primitive present some results in placement, project management and job-shop scheduling problems. In the latter case and for the famous 10×10 benchmark [FT63], it seems that a simple programming using the cumulative primitive and a labeling procedure based on first-fail principle, allow them to obtain the optimal solution of cost 930 in $25'$ on a Sun4 workstation (12Mb) while a first solution is found with a cost of 1088 in $1''$.

   In the examination timetabling framework [BDP95], this kind of implementation did not realize sufficient pruning to get acceptable results. The authors improved the efficiency by using another builtin predicate ("`distance`"[15]) whose utilization has been possible due to the presence of symmetries in the problem constraints.

   While using the cumulative primitive without generating any solution, one can notice that the impact of the constraint propagation may vary with the order in which constraints

[15] The predicate "`distance`" imposes an absolute distance between two domain variables.

are posted. Hence this builtin predicate, running as a black-box, uses *ad-hoc* propagation and does not seem to achieve arc-consistency.

5. *Arc-consistency*

   The disadvantage raised previously led us to build our own primitive for arc-consistency enforcing. It satisfies proposition 1 ($\underline{T}_i$ and $\overline{T}_i$ stand for the earliest and the latest start times of i, respectively):

   Proposition 1 (see Fig. 4) *If* $\underline{T}_j + D_j + D_i > \overline{T}_j$ *then* $T_i \in [\underline{T}_i, \overline{T}_j - D_i] \cup [\underline{T}_j + D_j, \overline{T}_i]$.

   Proof. *Two sequences are possible between* i *and* j*:* i before j *or* j before i*. If* i before j, $T_i + D_i \leq \overline{T}_j$*. If* j before i, $T_i \geq \underline{T}_j + D_j$*. It yields:* $T_i \in [\underline{T}_i, \overline{T}_j - D_i] \cup [\underline{T}_j + D_j, \overline{T}_i]$*. Thus if both intervals are disjoint* $T_i \notin]\overline{T}_j - D_i, \underline{T}_j + D_j[$ *with* $\overline{T}_j - D_i < \underline{T}_j + D_j$. □

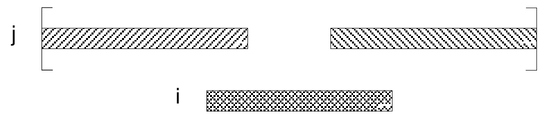


   Figure 4: Inconsistent values for start time of i

   Proposition 1 leads to the removal of inconsistent dates of T_i, *i.e.*, in $]\overline{T}_j - D_i, \underline{T}_j + D_j[$. The implementation uses the "`notin`" predicate which allows the removal of values inside a domain, and the conditional propagation where the same domain variable is used in each hand of the inequality: in the lines commented by `% check proposition 1`, in the left hand the maximum possible value of the variable is examined whereas it is the minimum one in the right hand.

```
revise(Ti,Di,Tj,Dj) :-
    D is Di + Dj,
    if Tj #< Tj + D                          % check proposition 1
        then remove_values(Ti,Di,Tj,Dj),     % remove values  in Ti
    if Ti #< Ti + D                          % check proposition 1
        then remove_values(Tj,Dj,Ti,Di).     % remove values  in Tj

remove_values(Ti,Di,Tj,Dj) :-
    domain_info(Tj,Tjmin,Tjmax),
    Min is Tjmax - Di + 1,
    Max is Tjmin + Dj - 1,
    notin(Ti,Min,Max).
```

   Note that the builtin information predicate "`domain_info`" returns statically the smallest and the largest values in the domain of the variable. It is then interesting to handle demon-propagation into the "`remove_values`" predicate (not detailed here), so as to obtain a complete dynamic behavior.

5.4 Edge finding

The previous disjunctive constraint (whatever its implementation) can deduce strong conclusions for the global sequencing. In practice, time windows can be very large related to processing times, and this rule may be useless and prune no value. It could then be more promising to study the extreme positions of a single task i relatively to a group S of other ones ($i \notin S$).

This technique whose principle is explained through the two following propositions, arises from *constraint-based analysis* [BBD+89, ERV76, ERV80], *immediate selections* [CP94] and has also been studied in [Nui94]; It is called *edge finding* in [AC91].

Proposition 2 (see Fig. 5) *If* $\max_{s\in S}(\overline{T}_s + D_s) - \underline{T}_i < \sum_{s\in S} D_s + D_i$ *then* S *is* non-posterior *to* i*, i.e.,* i *cannot be scheduled before* all *activities of* S*.*

Proof. *Suppose* i *precedes all activities of* S *and let* $z \in S$ *be the task scheduled last. Thus* $(T_z{+}D_z) \geq (\underline{T}_i{+}D_i{+}\sum_{s\in S} D_s) > (\max_{s\in S}(\overline{T}_s{+}D_s)) \geq (\overline{T}_z{+}D_z)$ *which leads to the contradiction* $T_z > \overline{T}_z$. □

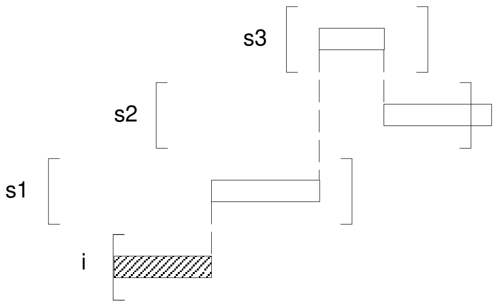


Figure 5: Non-posterior set

If S is non-posterior to i then at least one task of S must precede i. Thus it can be deduced a lower bound of the start time of i; it is updated to the smallest earliest finish time among the tasks of S (the `minimum` and `maximum` predefined predicates are used).

```
non_posterior_set(Ti,Di,S) :-
    set_duration(S,DS),             % Sum of durations of S
    D is Di + DS,
    set_ends(S,Ends),               % List of end variables of S
    minimum(Min_end,Ends),
    maximum(Max_end,Ends),
    if Max_end #< Ti + D
        then Ti #>= Min_end.
```

Proposition 3 (see Fig. 6) *If* $\max_{s\in S}(\overline{T}_s + D_s) - \min_{s\in S} \underline{T}_s < \sum_{s\in S} D_s + D_i$ *then* i *is* non-inserable *into* S*.*

Proof. *Similar to previous one.* □

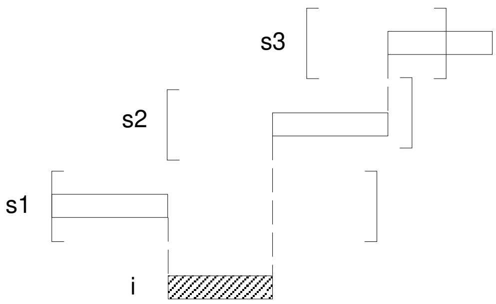


Figure 6: Non-inserable task

If i is non-inserable into S then i must be scheduled either before or after S. Moreover, if S has previously been proved to be non-posterior to i then i is necessarily scheduled after all tasks in S. Thus the start time of i can be adjusted to the earliest finish time of whole set S, value much stronger than with the sole non-posterior condition.

```
non_inserable_task(Ti,Di,S) :-
    set_duration(S,DS), D is Di + DS, set_ends(S,Ends),
    set_starts(S,Starts),             % List of start variables of S
    minimum(Min_start,Starts),
    maximum(Max_end,Ends),
    if Max_end #< Min_start + D
        then Ti #>= Min_start + DS.    % Iff S non-posterior to i
```

An analogous reasoning symmetrically establishes upper bound on the start time of a task from a non-anterior set condition. With non-inserability condition, one can derive that i precedes all tasks in S and then refine this upper bound.

6 Conclusion

The constraint programming framework is mostly known because of noticeable achievement in solving large combinatorial problems [Yap93, BCF95, Pug92, PPMD94]. These results have been obtained both because of the efficiency of the underlying solvers and the extra generality offered by the host non-deterministic Prolog language.

From the end of the eighties to now, the CLP scheme has integrated an increasing amount of solvers and domains. There are now CLP languages that use intricate solvers or which are connected to general systems such as MATHEMATICA[16]. An important shortcoming of most existing systems is the absence of relations between domains: it is not yet possible to think of *hybrid* models where constraints are propagated between several domains, even with incomplete satisfaction mechanisms. For instance, a boolean variable cannot be usually associated to the truth of one inequality between integer variables, achieving by that implicit propagations between boolean and finite domain solvers.

Such hybrid models would however be promising to model both numerical (time) and symbolic (sequencing) constraints of scheduling problems, and to derive inferences from a computation domain to another one. On the contrary, certain languages propose "global" primitives for the solving of some specific problems (*e.g.,* combinatorial problems and more precisely scheduling). Surprisingly, although their underlying mechanisms are not complete, they are often kept top secret and there is no other way for the user than using these primitives as black boxes.

Economical accounts favors more and more the competition between constraint programming tools. It has the advantage to give rise to the application of methods developed by academics but also to bring some new theoretical problems to light. Unfortunately, it also leads to the "closing" of the systems, and plays a great part in missing the primary objectives of CLP, *i.e.,* formalizing and integrating general mechanisms of constraint propagation.

For these reasons, nowadays, the only way to experiment hybrid solvers or to design user-defined propagation mechanisms is either to build your own system from scratch or from constraint programming libraries such as ILOG-SOLVER (which has perhaps the largest set of event-handling primitives), or to implement it as meta-level CLP interpretor [Boc93].

[16]See [FAQ95] for an exhaustive lists of existing CLP implementations.

On the contrary, the authors think that today there still exist good reasons to maintain Logic Programming as a basic natural framework for constraint programming, in particular the need of high-level languages for a concise and declarative representation of problems, and the rapid prototyping of constraint-oriented applications.

These kinds of facilities require a well-founded generalization of the semantical definition of constraints in CLP languages[17] whereas the trend of improving efficiency has led to develop separately then simply juxtapose each of the constraint domains, their solvers and primitives.

References

[AB92] A. Aggoun and N. Beldiceanu. Extending CHIP in order to solve complex scheduling and placement problems. In *Actes des Journées Francophones de Programmation Logique (JFPL'92)*, pages 51–66, Lille, France, 1992.

[AC91] D. Applegate and W. Cook. A computational study of the job-shop scheduling problem. *ORSA Journal on Computing*, 3(2):149–156, 1991.

[ACM92] Special section on logic programming. Communications of ACM, 1992. 33(7).

[AS93] Jean-Marc Alliot and Thomas Schiex. *Intelligence Artificielle et Informatique Théorique (in French)*. Cepadues, Toulouse, France, 2nd edition, 1993. ISBN : 2-85-428-324-4.

[ASS+88] A. Aiba, K. Sakai, Y. Sato, D.J. Hawley, and R. Hasegawa. Constraint logic programming language CAL. In *Proceedings of the International Conference on Fifth Generation Computer Systems (FGCS-88), ICOT, Tokyo*, pages 263–276, December 1988.

[Bak74] K.R. Baker. *Introduction to sequencing and scheduling.* John Wiley & Sons, New-York, 1974.

[BB88] J-M. Boi and F. Benhamou. *Boolean Constraints in Prolog III.* PhD thesis, Groupe d'Intelligence Artificielle, Université d'Aix-Marseille, Luminy, November 1988.

[BBD+89] G. Bel, E. Bensana, D. Dubois, J. Erschler, and P. Esquirol. A knowledge-based approach to industrial job-shop scheduling. In A. Kusiak, editor, *Knowledge-based systems in manufacturing*, pages 207–246. Taylor & Francis, 1989.

[BCF95] J. Bellone, A. Chamard, and A. Fishler. Constraint logic programming decision support systems for planning and scheduling aircraft manufacturing at dassault aviation. In *Third International Conference on the Practical Application of Prolog (PAP'95)*, pages 63–67, Paris, France, 1995.

[BDP95] P. Boizumault, Y. Delon, and L. Péridy. Constraint logic programming for examination timetabling. *Journal of Logic Programming*, 1995. To appear.

[BFR95] C. Bessière, E.C. Freuder, and J.C. Régin. Using inference to reduce arc-consistency computation. In *Proc. of the 14th IJCAI*, Montreal, Canada, August 1995.

[Boc93] A. Bockmayr. Logic programming with pseudo-boolean constraints. In F. Benhamou & A. Colmerauer, editor, *Constraint Logic Programming: Selected research*, Logic Programming Series, pages 327–350. MIT Press, 1993.

[17] Maybe with the brand-new version of PrologIA, PrologIV [BT95]?

[Bry92] R.E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[BT95] F. Benhamou and Touraïvane. Prolog IV : langage et algorithmes. In *Actes des Journées Francophones de Programmation en Logique (JFPL'95)*, pages 51–64, Dijon, France, 1995.

[Buc85] B. Buchberger. Gröbner bases: An algorithmic method in polynomial ideal theory. In N. K. Bose, editor, *Multidimensional Systems Theory*, pages 184–232. Reidel Publishing Co., 1985.

[CD93] P. Codognet and D. Diaz. Boolean constraints solving using `clp(fd)`. In *International Logic Programming Symposium*, pages 529–539, 1993.

[CKvC83] A. Colmerauer, H. Kanoui, and M. van Caneghem. Prolog, bases théoriques et développements actuels. *Techniques et Sciences Informatiques*, 4(2):271–311, 1983. (in French).

[CM81] W.F. Clocksin and C.S. Mellish. *Programming in Prolog.* Springer Verlag, 3rd edition, 1981.

[Coh90] J. Cohen. Constraint logic programming languages. *Communications of the ACM*, 33(7):52–68, July 1990.

[Col82a] A. Colmerauer. Prolog and infinite trees. In New York Academic Press, editor, *Logic Programming*, pages 231–251. K.L. Clark and S.A. Tarnlund, 1982.

[Col82b] A. Colmerauer. PROLOG II reference manual and theoretical model. Technical report, Groupe Intelligence Artificielle, Université Aix-Marseille II, October 1982.

[Coo89] M.C. Cooper. An optimal k-consistency algorithm. *Artificial Intelligence*, 41:89–95, 1989.

[Cos93] Cosytec. CHIP *Reference Manual*, June 1993. COSY/REF/001.

[CP94] J. Carlier and E. Pinson. Adjustment of heads and tails for the job-shop problem. *European Journal of Operational Research*, 78:146–161, 1994.

[Dav87] E. Davis. Constraint propagation with interval labels. *Artificial Intelligence*, 32(3):281–331, July 1987.

[dK89] J. de Kleer. A comparison of ATMS and CSP techniques. In *Proc. of the 11*$^{\text{th}}$ *IJCAI*, pages 290–296, Detroit, MI, August 1989.

[DP60] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):210–215, 1960.

[DSv90] M. Dincbas, H. Simonis, and P. van Hentenryck. Solving Large Combinatorial Problems in Logic Programming. *Journal of Logic Programming*, 8(1-2):74–94, January-March 1990.

[ERV76] J. Erschler, F. Roubellat, and J-P. Vernhes. Finding some essential characteristics of the feasible solutions for a scheduling problem. *Operations Research*, 24(4):774–783, 1976.

[ERV80] J. Erschler, F. Roubellat, and J-P. Vernhes. Characterizing the set of feasible sequences for n jobs to be carried out on a single machine. *European Journal of Operational Research*, 4(3):189–194, 1980.

[FAQ95] FAQ of the `comp.constraints` newsgroup. Available at URL `http://web.cs.-city.ac.uk/archive/constraint/`, 1995.

[Fre78] E.C. Freuder. Synthesizing constraint expressions. *Communications of the ACM*, 21:958–966, November 1978.

[FT63] H. Fisher and G.L. Thompson. Probabilistic learning combinations of local job-shop scheduling rules. In J.F. Muth & G.L. Thompson, editor, *Industrial Scheduling*, pages 225–251. Prentice Hall, Englewood Cliffs, NJ, 1963.

[GJ79] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness.* W.H. Freeman and Company, New York, 1979.

[GOT93] GOThA. Les problèmes d'ordonnancement. *RAIRO-RO*, 27(1):77–150, 1993. (in French).

[Gra81] S.C. Graves. A review of production scheduling. *Operations Research*, 29:646–675, 1981.

[HJM+92] N.C. Heintze, J. Jaffar, S. Michaylov, P.J. Stuckey, and R.H.C. Yap. *The CLP($\mathcal{R}$) Programmer's Manual.* IBM T.J. Watson Research Center, PO Box 704, Yorktown Heights, NY 10598, September 1992. (Version 1.2).

[HL88] T. Huynh and C. Lassez. A CLP($\mathcal{R}$) options trading analysis system. In Robert A. Kowalski and Kenneth A. Bowen, editors, *JICSLP'88: Proceedings 5th International Conference and Symposium on Logic Programming*, pages 59–69, Seattle, Washington, U.S.A., 1988. MIT Press.

[HMS92] N. Heintze, S. Michaylov, and P. Stuckey. CLP($\mathcal{R}$) and some electrical engineering problems. *Journal of Automated Reasoning*, 9:231–260, October 1992.

[Hoo88] J.N. Hooker. A quantitative approach to logical inference. *Decision Support Systems*, 1(4):45–69, 1988.

[Hyv92] E. Hyvönen. Constraint reasoning based on interval arithmetic: the tolerance propagation approach. *Artificial Intelligence*, 58:71–112, December 1992.

[Imb93] J-L. Imbert. Fourier's elimination: Which to choose? In *First Workshop on Principles and Practice of Constraint Programming*, pages 119–131, Newport, April 1993.

[JL87] J. Jaffar and J-L. Lassez. Constraint logic programming. In *POPL'87: Proceedings 14th ACM Symposium on Principles of Programming Languages*, pages 111–119, Munich, January 1987. ACM.

[JM94] J. Jaffar and M.J. Maher. Constraint logic programming: a survey. *The Journal of Logic Programming*, 19(20):503–581, 1994.

[JMSY93] J. Jaffar, M.J. Maher, P.J. Stuckey, and R. Yap. Projecting CLP($\mathfrak{R}$) constraints. *New Generation Computing*, 11:449–469, 1993.

[Kar84] N. Karmarkar. A new polynomial time algorithm for linear programming. *Combinatorica*, 4:373–395, 1984.

[Kha79] L.G. Khachian. A polynomial algorithm in linear programming. *Soviet Math. Dokl.*, 20(1):191–194, 1979.

[Kor85] R.E. Korf. Depth first iterative deepening : An optimal admissible tree search. *Artificial Intelligence*, 27:97–109, 1985.

[Kow79] R.A. Kowalski. *Logic for Problem Solving.* North Holland, 1979.

[Lho93] O. Lhomme. Consistency techniques for numeric CSPs. In *Proc. of the 13*$^{\text{th}}$ *IJCAI*, pages 232–238, Chambéry, France, August 1993.

[LM92] J-L. Lassez and K. McAlloon. A canonical form for generalized constraints. *J. Symbolic Computation*, 13:1–24, 1992.

[Mac77] A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.

[MJPL90] S. Minton, M.D. Johnston, A.B. Philips, and P. Laird. Solving large-scale constraint satisfaction and scheduling problems using a heuristic repair method. In *Proc. of AAAI-90*, pages 17–24, Boston, MA, 1990.

[ML93] B.L. McCarthy and J. Liu. Addressing the gap in scheduling research: A review of optimization and heuristic methods in production scheduling. *International Journal of Production Research*, 31(1):59–79, 1993.

[Mon74] U. Montanari. Network of constraints: Fundamental properties and applications to picture processing. *Inf. Sci.*, 7:95–132, 1974.

[Nad89] B.A. Nadel. Constraint satisfaction algorithms. *Comput. Intell.*, 5(4):188–224, November 1989.

[Nui94] W.P.M. Nuijten. *Time and resource constrained scheduling.* PhD thesis, Eindhoven University of Technology, 1994.

[Pap94] C. Le Pape. Implementation of resource constraints in ILOG-SCHEDULE: A library for the development of constraint-based scheduling systems. *Intelligent Systems Engineering*, 3:55–66, 1994.

[Pin95] M. Pinedo. *Scheduling: Theory, Algorithms and Systems.* Prentice-Hall, Englewood Cliffs, NJ, 1995.

[PPMD94] C. Le Pape, J-F. Puget, C. Moreau, and P. Darneau. PMFP: The use of constraint-based programming for predictive personnel management. In *Proc. of the 11*$^{\text{st}}$ *ECAI*, Amsterdam, The Netherlands, 1994.

[Pug92] J-F. Puget. Object oriented constraint programming for transportation problems. In *Proc. of ASTAIR'92*, 1992.

[Pug94] J-F. Puget. A C++ implementation of CLP. Technical Report 94-01, Ilog, 1994.

[PW78] M.S. Paterson and M.N. Wegman. Linear unification. *JCSS*, 16:158–167, 1978.

[Rob65] J. Alan Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12:23–44, 1965.

[SH91] G. Sidebottom and W.S. Havens. Hierarchical arc consistency applied to numeric processing in constraint logic programming. Technical report, Center for Systems Science, Simon Fraser University, Burnaby, Canada, 1991.

[Ste80] G.L. Steele. The definition and implementation of a computer programming language based on constraints. Technical report, Dept. of Electrical Engineering and Computer Science, M.I.T., August 1980.

[Tsa93] E.P.K. Tsang. *Foundations of Constraint Satisfaction*. Academic Press Ltd., London, 1993.

[Ull66] J.R. Ullman. Associating parts of patterns. *Inform. Contr.*, 9:583–601, 1966.

[vH89] P. van Hentenryck. *Constraint Satisfaction in Logic Programming*. Logic Programming Series. MIT Press, Cambridge, MA, 1989.

[vHD93] P. van Hentenryck and Y. Deville. The cardinality operator: A new logical connective for constraint logic programming. In F. Benhamou & A. Colmerauer, editor, *Constraint Logic Programming: Selected research*, Logic Programming Series, pages 383–403. MIT Press, 1993.

[vHDT92] P. van Hentenryck, Y. Deville, and C. Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57:291–321, 1992.

[Wal72] D. L. Waltz. Generating semantic descriptions from drawings of scenes with shadows. Technical Report AI271, M.I.T., Cambridge MA, 1972.

[Wal94] M. Wallace. Applying constraints for scheduling. In B. Mayoh, E. Tyugu, and J. Penjaam, editors, *Constraint Programming: Proceedings 1993 NATO ASI Parnu, Estonia*, NATO Advanced Science Institute Series, pages 161–180. Springer-Verlag, 1994.

[Yap93] R.H.C. Yap. A constraint logic programming framework for constructing DNA restriction maps. *Artificial Intelligence in Medicine*, 5:447–464, 1993.