Constraint Logic Programming

Marco Gavanelli[1] and Francesca Rossi[2]

[1] Dipartimento di Ingegneria-Università di Ferrara
[2] Dipartimento di Matematica Pura e Applicata - Università di Padova

Abstract. Constraint Logic Programming (CLP) is one of the most successful branches of Logic Programming; it attracts the interest of theoreticians and practitioners, and it is currently used in many commercial applications. Since the original proposal, it has developed enormously: many languages and systems are now available either as open source programs or as commercial systems.

Also, CLP has been one of the technologies able to recruit researchers from other communities to the declarative programming cause. Current CLP engines include technologies and results developed in other communities, which themselves discovered logic as an invaluable tool to model and solve real-life problems.

1 The CLP Paradigm

Constraint Logic Programming (CLP) [7] represents a successful attempt to merge the best features of logic programming (LP) and constraint solving.

Constraint solving [127, 6, 56, 31] includes a variety of expressive modelling frameworks and efficient solving tools for real-life problems that can be described via a set of variables and constraints over them. A constraint is just a restriction imposed over the combination of values of some variables of the problem. Solving a problem with constraints means finding a way to assign values to all its variables such that all constraints are satisfied. Constraint solving methods have been successfully applied to many application domains, such as scheduling, planning, resource allocation, vehicle routing, computer networks, and bioinformatics [137, 127, 51].

Embedding the notion of constraint into a high-level programming language allows for a more flexible and practical constraint processing environment, where constraints can be represented as formulae and can be incrementally accumulated. Moreover, the presence of constraints in a programming language usually augments its expressive power, in the sense that some complex relations can be defined easily by means of constraints, and there are also efficient techniques to prove them.

For these reasons, constraints have been embedded in many programming environments, but some are more suitable than others. For example, the fact that constraints can be seen as relations or predicates, that constraint solving can be seen as a generalized form of unification, that their conjunction can be seen as a *logical and*, and that backtracking search is the base methodology to

A. Dovier, E. Pontelli (Eds.): 25 Years of Logic Programming, LNCS 6125, pp. 64–86, 2010.

solve them, makes them very compatible with logic programming, which is based on predicates, unification, logical conjunctions, and depth-first search.

These observations led to the development of the CLP paradigm, where constraints are embedded in the logic programming paradigm. The main goal is to maintain a declarative programming paradigm while increasing expressivity and efficiency via the use of specific constraint sorts and algorithms.

The first CLP language was Prolog II [42], designed by Colmerauer in the early 80's. Prolog II could treat term equations like Prolog, but in addition could also handle term disequations. After this, Jaffar and Lassez observed that both term equations and disequations were just a special form of constraints, and developed the concept of a constraint logic programming scheme in 1987 [99].

Syntactically, constraints are added to logic programming by considering a specific constraint sort (e.g., linear equations over the reals) and then allowing constraints of this type in the body of the usual logic programming clauses. Beside the classical resolution engine of logic programming, a (complete or incomplete) constraint solving system is added, able to check the consistency of constraints of the considered sort. Moving from LP to CLP, the concept of unification is generalized to constraint solving: the relationship between a goal and a clause (to be used in a resolution step) can be described not only via term equations but via more general statements, i.e., constraints. This allows for a more general and flexible way to control the flow of the computation. Also, the presence of an underlying constraint solver, usually based on incomplete constraint propagation of some sort, allows one to alternate backtracking search (as in classical LP) with efficient constraint propagation, thus generating a more efficient solver, that is nevertheless complete, being based on systematic search.

More precisely, a CLP clause is just like an LP clause, except that its body may contain also constraints of the considered sort. For example, if we can use linear inequations over the reals, a CLP clause could be:

```
p(X,Y) :- X < Y+1, q(X), r(X,Y,Z).
```

Logically speaking, this clause states that `p(X,Y)` is true if `q(X)` and `r(X,Y,Z)` are true, and if the value of x is smaller than that of $y + 1$.

From the operational point of view, in an LP resolution step, we have to check the existence of a most general unifier between the selected subgoal and the head of a clause. In CLP, instead, we also have to check the consistency of the current set of constraints (called the *constraint store*) with the constraints in the body of the clause. Thus two solvers are involved: unification, as usual in LP, and the specific constraint solver for the constraints in use. To make it more efficient, this constraint solver may be not complete, that is, it may fail to discover some inconsistencies.

While in LP a computation state consists of a goal and a substitution, in CLP we have a goal and a constraint store. While in LP we just accumulate substitutions during a computation, in CLP we also accumulate constraints. Given a state $\langle G, S\rangle$, where G is the current goal (the resolvent) and S is the current constraint store, assume G consists of an atom A (that we want to rewrite) and a rest R, i.e., $G = (A, R)$. Then, at each step:

- if A is a constraint, A is added to S and its consistency is checked through a transition that checks if $consistent(A \wedge S)$; if it is, the new state is $\langle R, prop(S \wedge A)\rangle$, where $prop(C)$ is the result of applying some constraint propagation algorithm (like arc-consistency) to the constraint store C;
- if instead A is a literal, and there is a clause $H :- B$ with the same head-predicate as A, then we add the constraint $A = H$ to the constraint store, check its consistency, and replace A with B in the resolvent: the new goal is then $\langle (B, R), prop(S \wedge \{A = H\})\rangle$.

A CLP computation is successful if there is a way to get from the initial state $\langle G, true\rangle$ to the goal $\langle G', S\rangle$, where G' is the empty goal and S is satisfiable.

Derivation trees are defined as in LP, except that each node in the tree now represents both the current goal and the current constraint store. Also, in practical CLP systems, the usual depth-first Prolog traversal mode is retained, with subgoals selected from left to right, and clauses from the first to the last one. Early detection of failing computations is achieved by checking the consistency of the current constraint store. At each node, the underlying constraint system is automatically invoked (via function *prop* above) to check consistency and the computation along this path continues only if the check is successful (although the check itself could be incomplete). Otherwise, backtracking is performed.

Although CLP significantly extends LP in expressive power and application domains, it maintains its semantic properties, such as the existence of equivalent operational, model-theoretic, and fixpoint semantics [99]. Several semantics, describing different observable properties of CLP programs, have been presented in the literature, with significant contributions from Italian researchers [84, 94, 52, 74, 43, 115, 19]. Properties of such semantics, such as fully abstraction, compositionality, and correctness, have been studied in depth. The power of CLP has also been exploited to treat negation in LP, by allowing constraints that are equalities or inequalities over the Herbrand domain [29]. Also, constraint solving in LP was compared with the equivalent notions in automated deduction [8].

Finally, abstract interpretation has been applied to CLP [11], but we will not discuss the issue because it is subject of another chapter of this book [59].

2 Constraint Sorts

CLP is not a programming language, but a programming *paradigm*, which is parametric with respect to the class (sort) of constraints used in the language. Working with a particular CLP language means choosing a specific class of constraints (for example, finite domains, linear, or arithmetic) and a suitable constraint system for that class. Notice also that unification is not *replaced*, rather it is assisted by the specific constraint solver, since every CLP language also needs to perform usual LP-style unification over its variables.

Denoting a CLP language over a constraint class X as CLP(X), we can say that logic programming is just $CLP(Trees)$, where Trees identifies the class of term equalities, with the unification algorithm to solve them. Other examples of instances of the CLP scheme are Prolog III [41], that treats constraints over

terms, strings, booleans, and real linear arithmetic, the language CLP(R) [100], that works with both terms and arithmetic constraints over the reals.

The possibility to instantiate the CLP scheme with many constraint sorts is one of the features that made CLP successful, since in this way the variety of solvers added to a LP language becomes almost unlimited (e.g., [110]).

2.1 Finite Domains

A popular class of constraints used with the CLP scheme is the class of constraints with variables ranging over finite domains. Constraint logic programming using finite domain constraints is a useful language scheme, referred to as CLP(FD). Its applicability is very large, since many real-life problems can be modelled via imposing a set of constraints over variables with finite domains (for example, the wide class of Constraint Satisfaction Problems [56]). Examples can be found in configuration, scheduling, and resource allocation [56, 127, 12, 51].

Finite domain constraints, as used within CLP languages, are usually intended to be arithmetic constraints over finite integer domain variables. Thus a CLP(FD) language needs a constraint system which is able to perform consistency checks and projection over this kind of constraints. Usually, the consistency check is based on some kind of constraint propagation, such as arc-consistency [105], some weaker version, like bound-consistency [20], or, more rarely, path-consistency [117] (see also Section 3.1).

Many CLP(FD) languages or environments have been developed, either in academic or commercial environments. Constraint logic programming over finite domains was first implemented in the late 80's by Pascal Van Hentenryck [135] within the language CHIP. Since then, more sophisticated constraint propagation algorithms have been developed and added to more recent CLP(FD) languages, like ECL^{i}PS^e [37], GNU Prolog [60], CIAO [32], B-Prolog [141], SWI-Prolog [139] and SICStus Prolog [35].

One of the main features of CLP(FD) languages is that they have a specific mechanism for defining the initial finite domains of the variables: usually as an interval over the integers. For example, a typical CLP(FD) syntax to say that the domain of variable x contains all integers between 1 and 10 is `X in [1..10]`, or `X::[1,10]`, or `fd_domain(X,1,10)`.

Another feature of all CLP(FD) languages is the use of a built-in predicate called `labeling` defined over a list of variables, and which finds values for them such that all constraints in the current store are satisfied. The `labeling` predicate provides a mechanism to generate solutions, that is, variable assignments that satisfy all accumulated constraints. More precisely, this predicate triggers backtracking search over a set of variables. For example, the following clause defines a problem with three finite domain variables (x, y, and z), each with domain containing the integers from 1 to 10, and sets a constraint over them ($x+y=9-z$). After this, it triggers backtracking search via predicate `labeling`:

```
p(X,Y,Z) :- [X,Y,Z]::[1,10], X + Y = 9 - Z, labeling([X,Y,Z]).
```

The result of executing the goal `:- p(X,Y,Z).` is any instantiation of the three variables over their domains which satisfies the constraint $x + y = 9 - x$. Notice that without `labeling`, this same goal would return just the new domains obtained after applying constraint propagation (together with the constraint store). E.g., running this goal in the CLP(FD) language GNU Prolog [40] returns the answer `[X,Y,Z]:[1,7]`, meaning that the domains have been reduced from [1..10] to [1..7] via constraint propagation. The clause above presents the typical shape of a CLP(FD) program: first the variable domains are specified, then the constraints are imposed, and finally the backtracking search is invoked via a labeling predicate. A CLP(FD) program can consist of many clauses, but the overall structure of the program always reflects this order, which refers to a methodology called *constrain and generate*, where first variables are constrained and only later (when the domains are smaller) backtracking search is invoked. This corresponds to applying constraint propagation prior to search and therefore avoiding early some dead-ends.

In many CLP(FD) systems, arc-consistency is considered too expensive: for each binary constraint one should (in general) check if for each domain element there exists a support in the other domain. So, for each constraint involving two variables with d elements in the domains, one has to do $O(d^2)$ constraint checks. Since constraints are many, and arc-consistency propagation can wake many times the same constraint, a quicker algorithm is often adopted, at the expenses of a lower pruning. Bound consistency considers only the bounds (minimum and maximum values) of the domains, so the number of checks is drastically reduced. This means that, e.g., the propagation of the $X = 2Y$ constraint will not remove all the odd values from the domain of X, but will have to perform only 4 checks. A powerful feature of CLP(FD) is that for each constraint one can have a different propagation algorithm: if we know an efficient algorithm to perform arc-consistency for a specific constraint, we can use it, even if for other constraints the solver performs only bound consistency.

For example, consider the goal $A :: [-1, 0, 1]$, $B :: [-1, 1]$, $C :: [0, 1]$, $A = B$, $A^2 \leq C$. If all the constraints have bound-consistency propagation, no pruning occurs, in fact all the extreme values in each domain are consistent with some value in each other domain. On the other hand, arc-consistency propagation for the equality constraint is very simple: one has to compute the intersection of the two domains, which has linear complexity, instead of the expensive $O(d^2)$ of the general case. By applying arc-consistency to the $A = B$ constraint we can remove value 0 from the domain of A. Now, the bound-consistency propagation of $A^2 \leq C$ detects that the value 0 in the domain of C is no longer supported and removes it, implicitly assigning 1 to C. So, by strengthening the propagation of a single constraint (in the example, the equality constraint), we can propagate removals also by constraints with a weak bound-consistency propagation.

Global constraints are non-binary constraints that appear often in applications and for which specialized constraint propagation methods are developed. Sometimes those constraints are logically equivalent to the conjunction of a set of binary constraints, but global constraint typically perform stronger propagation

than applying standard arc-consistency to many binary constraints. A typical example is the `alldifferent` constraint [136], which requires that n variables have mutually different values. Although this constraint can be defined with a binary not-equal constraint for each pair of variables, such a representation does not allow for much domain pruning by arc-consistency. Since such a constraint appears very often, it is worthwhile to strengthen its propagation method by employing an ad hoc filtering algorithm. The concept of arc-consistency was suitably extended for non-binary constraints and named *Generalized Arc-Consistency (GAC)*. Most current CLP languages are equipped with a rich taxonomy of global constraints. During a computation, the current constraint store in a CLP computation may contain both binary and global constraints such as `alldifferent`. At each step, when constraint propagation is performed, each constraint propagates with its own algorithm, and achieves arc or bound-consistency. Not all non-binary constraints have a specialized constraint propagation algorithm, just those that occur more frequently in applications.

Other logic languages, such as Answer Set Programming (ASP) [27], address similar types of problems addressed by CLP(FD); there are works comparing the two approaches [63, 109], and also integrating the two [15]. We will not give more details on ASP, since it is the subject of another chapter of this book [27].

2.2 Sets

Various Italian researchers studied the integration of *sets* into logic programming. Sets are widely used in mathematics to define new objects, and they allow for a natural representation of concepts in AI and in software engineering. One of the languages that integrate sets into logic programming is $\{log\}$ [64], that later evolved into the language CLP($\mathcal{SET}$) [65]. In CLP($\mathcal{SET}$), unification is extended to deal with variables representing sets and set objects. Prolog users often represent collections of values as lists, but this is insufficient when one needs a set semantics. Sets intrinsically remove symmetries (see also Section 4.2), since {1,2} and {2,1} are the same set, while for lists [1,2] and [2,1] represent different terms (i.e., they do not unify). In CLP($\mathcal{SET}$), $\{1,2\} = \{2,1\}$ succeeds, as well as $\{1,2,3,2\} = \{3,2,1,1\}$; moreover, one can have variables and non-ground terms as elements of sets, so the unification $\{p(X), p(2)\} = \{Y\}$ succeeds, giving $Y = p(2)$ and $X = 2$. CLP($\mathcal{SET}$) supports sets, possibly partially specified and nested like e.g. $\{X, \{\emptyset\}\} \cup Y$. Moreover, set unification and set constraint solving has been analysed in a modular way so as to easily replace sets with multi sets (and other similar data structures)—see e.g. [67, 66].

CLP($\mathcal{SET}$) has been used for various applications, among which to represent actions [124], and to implement abductive reasoning [89] (see also Section 3.3). Other efforts tried to integrate reasoning on sets with the classical CLP(FD). In one case the starting point was a visual search application [45]. Visual search and image recognition are classical applications of CLP(FD) [45, 75]. Visual search is the task of finding an object (described in some formal way, called the *object model*) in an image. CLP(FD) provides the language for describing the object model: first one decides the visual features (the basic components of the image,

such as lines, points, surface patches, etc.), then he/she defines the object model by means of constraints that relate the visual features (surface s_1 is orthogonal to surface s_2, etc.). Now, before CLP(FD) performs constraint propagation and subsequent search, one has to know all the visual features in the image, as they compose the domains of the variables. This task is performed by a segmentation system, that takes often most of the computing time, since it has to relate the pixels of the image with higher-level information. In order to speed up the acquisition process, one can interleave constraint propagation and value acquisition; in this way only those features actually required for solving the CSP are acquired from the segmentation system. The classical CSP model is then extended to an Interactive CSP [46], with corresponding solving algorithms. A corresponding CLP language [88] uses sets to represent the domains of FD variables. Later on, a general integration of the two sorts was proposed [50,18], which integrates sets and finite domain variables to speedup the CLP($\mathcal{SET}$) computation.

3 Related Frameworks

3.1 Constraint Handling Rules

In classical CLP languages, solvers are embedded in the language in a *hard-wired* way: each language comes with one or more solvers for some constraint sorts. However, defining a new constraint, or even a new solver, is often tricky: one has to know (part of) the implementation of the solver itself, study the interface for defining new constraints, and implement the propagation algorithm. While usually very efficient, this approach is rather operational and not always flexible. Constraint Handling Rules (CHR) [82] represents a successful example of a high-level, logic language for designing constraint solvers. Also, usually solvers adopt arc or bound consistency, that look at one constraint at a time. For example, the constraints $[A, B] : [1..10]$, $A \leq B$, $B \leq A$, $A \neq B$ do not perform any pruning, even if we can easily see that there is no solution. If we looked at pairs of constraints, we could infer from $(A \leq B \wedge B \leq A)$ that $A = B$, and from $(A = B \wedge A \neq B)$ that there is no solution. Intuitively, looking at pairs of constraints allows one to achieve higher levels of consistency, such as path-consistency [117].

CHR is a powerful language for modelling solvers, based on the rewriting of constraints into simpler ones until they are solved. CHR can be seen as a CLP language where clauses are multi-headed guarded rules for constraint rewriting.

CHR rules are of two kinds, based on the notions of *simplification* and *propagation* over user-defined constraints. Simplification rules replace constraints by simpler constraints while preserving logical equivalence. Propagation rules add new, logically redundant constraints, which may cause further simplifications. More precisely, a CHR program is a finite set of CHR rules. A *simplification* CHR rule is of the form $H \Leftrightarrow G|B$ and a *propagation* CHR rule is of the form $H \Rightarrow G|B$. The multi-head H is a conjunction of CHR constraints. The optional guard G is a conjunction of built-in constraints. The body B is a conjunction of built-in and CHR

constraints. An example of a simplification rule is $X \leq Y \wedge Y \leq X \Leftrightarrow X = Y$, while a possible propagation rules is $X \leq Y \wedge Y \leq Z \Rightarrow X \leq Z$.

A state of a computation is a conjunction of built-in and CHR constraints, and states evolve via derivation steps. An *initial state (or query)* is an arbitrary state. In a *final state (or answer)*, either the built-in constraints are inconsistent or no derivation step is possible anymore. A rule with head H and guard G is *applicable* to CHR constraints H' in the context of constraints D, if the underlying constraint theory entails D and $\exists\theta(H\theta = H' \wedge G\theta)$. Notice that the symbol $=$ is to be understood as built-in constraint for syntactic equality and is usually implemented by a (one-way) unification. If H' matches H, we equate H' and H. This corresponds to parameter passing in conventional programming languages, since only variables from the rule head H can be further constrained, and all those variables are new. Finally, using the variable equalities from D and $H' = H$, we check the guard G.

Any of the applicable rules can be applied, but the choice of the rule is a committed choice, thus it cannot be undone.

If an applicable simplification rule $(H \Leftrightarrow G \mid B)$ is applied to the CHR constraints H', H' is removed from the state, and the body B, the equation $H = H'$, and the guard G are added to the state. If a propagation rule $(H \Rightarrow G \mid B)$ is applied to H', we add B, $H = H'$ and G, but do not remove H'.

CHR is now implemented in most major CLP languages (e.g., SICStus, SWI or ECL^{i}PS^e), and the number of applications developed in CHR is impressive (see, e.g., the web page[1] *"The first fifty applications using CHR"*, amongst which we find many works of Italian researchers [4, 126, 22, 61].)

Beside the operational semantics briefly outlined above, several declarative semantics have been defined for CHR programs, and soundness and completeness results have been obtained. The issue of confluence has also been studied in depth, since applicable CHR rules may be applied in any order giving rise to resulting states with the same meaning but not necessarily the same syntax. This may be a problem in terms of constraint solvers, since the ability to detect the inconsistency of the current set of constraints depends also on the syntax. Another important property is compositionality [58]. This property allows to compute the semantics of a conjunctive query from the semantics of its components, and is obviously very desirable since it allows to define incremental and modular analysis and verification tools.

Various extensions of the basic CHR language have been proposed in the literature. For example, CHR has been extended with a probabilistic weighting of the rules, by specifying the probability of their application [83]. In this way, it is possible to formalise various randomised algorithms, such as simulated annealing.

3.2 Concurrent Constraint Programming

In CLP, each computation step adds new constraints to the constraint store, and checks if the resulting store is consistent. However, the constraint store could also be used to check whether it contains enough information to entail certain

[1] `http://www.cs.kuleuven.be/~dtai/projects/CHR/chr-appls.html`

constraints. This is what is done in the concurrent constraint (cc) programming paradigm [130], where several agents work concurrently with a unique constraint store. Each agent can perform two kinds of actions: either to add (called *tell*) a new constraint to the store, and proceed if this produces a consistent new store, or to wait (called *ask*) until the current store entails a certain constraint, and proceed only after this holds. In this paradigm, the concurrent agents communicate via the shared constraint store. CLP can be seen, very abstractly, as a restriction of the cc paradigm where only tell operations are performed.

Many significant results from Italian researchers have been obtained in defining and proving properties of several different semantics for the cc paradigm [53, 73,71]. Also, the cc paradigm has been extended to work with soft constraints [26], with probabilistic actions [123], and with timed operators [54, 23].

We avoid entering into the details of the various research lines related to cc, since it is the subject of another chapter of this volume [85].

3.3 Abductive Constraint Logic Programming

Logic programming is based on deductive reasoning, i.e., if we have a rule with conditions and a conclusion, and we know that the preconditions of the rule are true, we infer that also the conclusion is true. On the other hand, the human mind uses also other types of inference: for example, in medical diagnosis a physician is given a set of symptoms, that are the effects of some illness, and has to infer the illness that possibly caused such effects. The inference rule that allows one to reason from the conclusions to possible causes, or conditions, was called *abduction* by the philosopher Peirce.

Abductive Logic Programming [102,101] is an extension of LP that deals with incomplete information by performing abduction. In ALP, there are some syntactically distinguished predicates that have no definition, and cannot be proven: an abductive proof-procedure will *assume* their possible truth, and provide the abduced literal in the answer. E.g., an abductive program could be:

```
headache :- flu.
```

where flu is declared as an abducible predicate. Given the query `:- headache`, an abductive proof-procedure will provide as answer

```
yes, flu.
```

However, abductive reasoning has a very wide search space, and researchers soon found out that it could be reduced by means of constraints [103]. Obviously the integration also provides more expressivity to the abductive language, as the user can now write constraints in his/her programs. This opened the path to the development of a series of proof-procedures that integrate abductive reasoning with constraint propagation [104, 69, 3]. Abductive constraint programming languages have been used for a variety of applications, including agents, planning, web service composition [2, 1], web sites verification [107] and two-player games [87].

More on Abductive Logic Programming can be found in the chapter [95].

3.4 Soft Constraints and Preferences

Classical constraints are statements that have to be satisfied in order to obtain a feasible solution. Thus the role of a constraint solver is to find a variable assignment that satisfies all constraints. In several real-life scenarios, this approach is too rigid, since there may be no variable assignment that satisfies all constraints. These scenarios often occur when constraints are used to formalize desired properties rather than requirements that cannot be violated. Such desired properties are not faithfully represented by constraints, but should rather be considered as *preferences*, whose violation should be avoided as far as possible. *Soft constraints* [24] provide one way to model such preferences, by extending the classical constraint notion into a more general and flexible one.

A soft constraint is just like a constraint, but instead of being only satisfied or violated, it may have several levels of satisfiability. Historically, first a variety of specific extensions of the basic constraint formalism have been introduced, such as fuzzy constraints [129]. Later, these extensions have been generalized using more abstract frameworks, which have been crucial in proving general properties and in identifying the relationship among the specific frameworks [24, 133]. Moreover, for each of the specific classes, algorithms for solving problems specified in the corresponding formalisms have been defined. In fact, many techniques and approaches to solve classical constraints, included constraint propagation, have been generalized to work also with soft constraints.

In the semiring-based formalism [24], a soft constraint is a cost function, where each assignment of the variables of the constraint is associated to an element coming from an ordered set, whose properties are similar to those of a semiring. This set contains all possible levels of preference (or costs, or quality, etc.), of a variable assignment in the considered constraint class. For example, for fuzzy constraints, the preference levels are values between 0 and 1, and higher values are more preferred. Classical constraints can also be cast in this general framework: in this case the preference set contains just two elements (*true* and *false*, or *satisfied* and *violated*). The preference set also comes with an operation to combine preference levels. This is useful to compute the satisfiability level of a complete variable assignment from those given by the constraints to the portion of the assignment relevant to them. For example, in fuzzy constraints the combination takes the minimum preference level, while in classical constraints it is just a *logical and*, since *all* constraints need to be satisfied. A survey of the various approaches to deal with soft constraints can be found in [113].

The notion of global constraints has been exploited also in the context of soft constraints. For example, in [97] a general method to soften global constraints is presented, which is based on the notion of a flow in a graph, and several global constraints are defined in their soft version. Also, in [140] efficient algorithms are proposed to achieve generalized arc consistency for the soft global cardinality constraint.

Classical CLP handles only standard constraint solving. Thus it is natural to try to extend the CLP formalism in order to handle also soft constraints. A first attempt was the *hierarchical CLP* (HCLP) system [28], a CLP language where

each constraint has a level of importance and a solution of a constraint problem is found by respecting the hierarchy of constraints. The finite domain CLP language clp(fd) [40] has been extended to handle semiring-based constraints, obtaining a language paradigm called clp(fd,S) [93] where S is any semiring, chosen by the user. By choosing one particular semiring, the user uses a specific class of soft constraints: fuzzy, optimized, probabilistic, or even classical hard constraints.

The language SCLP [25] treats in a uniform way, and with the same underlying machinery, all constraints that can be seen as instances of the semiring-based approach: from optimization to satisfaction problems, from fuzzy to probabilistic, prioritized, or uncertain constraints, and also multi-criteria problems, while still being able to handle classical constraints. Syntactically, SCLP extends CLP by allowing the presence of preference levels as the body of a clause. E.g., the clause `p(X,Y,N) :- (X+Y)/N.` states that $(X + Y)/N$ is the preference level to be given to the assignment (X, Y, N) for constraint `p`. The usual three equivalent semantics (model-theoretic, fix-point, and operational) can be defined also for the SCLP paradigm, although suitably generalized to handle soft constraints.

4 Improvements, Solution Techniques

4.1 Integration with Operations Research

CLP(FD) is an effective language to model and solve combinatorial problems. However, there are other frameworks that address the same problems, such as meta-heuristics, integer linear programming, population-based methods, etc. CLP(FD) has unique advantages: there are many types of available constraints, compared to integer linear programming that accepts only linear inequalities. It supports complete solving algorithms, while local search or genetic algorithms are usually incomplete (i.e., they might fail to produce a solution even if it exists). On the other hand, there are some types of problems in which other techniques are more efficient. For this reason, various efforts tried to merge algorithms and solvers, in order to improve on both of them. The fact that CLP(FD) is very general makes it the ideal playground to test the integration of different techniques.

One type of integration, already mentioned, is global constraints. In general, the (generalized) arc-consistency propagation of an n-ary constraint is very expensive (see, e.g., [116]): since an n-ary constraint can encode a whole CSP, removing all values that do not belong to a solution is in general NP-hard. However, despite this worst-case complexity, there exist significant constraints of practical use that have polynomial-time, specific propagation algorithms. For example, the `alldifferent` constraint uses results from graph theory, the global cardinality constraint `gcc` computes the maximum flow of a graph, all techniques borrowed from Operations Research (OR). In OR there are very efficient algorithms to solve very specific tasks, however a slight change in the problem formulation (e.g., a new constraint added by the user) can make a very good algorithm inapplicable. CLP(FD), instead, is very general-purpose. In OR, combining a graph algorithm with a maximum flow is a rather complex task, while

in CLP(FD) it is trivial: just a matter of adding two constraints (`alldifferent` and `gcc`) to the program, and they will automatically communicate through the constraint store and the domains of the variables. The user does not even need to know the details of the propagation algorithm.

Another key observation is that CLP(FD), being based on the concept of consistency, is very oriented to solve satisfiability problems, and optimization problems are often converted into (sequences of) satisfiability ones. OR, instead, has a wide literature focussed on optimization problems, using bounds, relaxations, and cuts, to remove sub-optimal parts of the search space. Moreover, arc-consistency reasons about one constraint at a time, meaning that if no constraint is able to perform pruning alone, no propagation occurs. This can be partially solved using higher levels of consistency, also supported by languages like CHR (Section 3.1), but this is not always a solution, since higher levels of consistency require more computation time. Linear programming algorithms, instead, navigate a polytope focussing only on the vertices carrying the best values of objective function, so they have a more global view.

So, an interesting way to integrate CLP and OR is by trying to exploit both the satisfaction-based techniques of CLP and the optimization-based tools of OR. A simple idea is to use both a linear model and a CLP(FD) model at the same time: if either of the two detects inconsistency, we can fail and backtrack. An important information a linear solver provides is a *bound*: by giving up the integrality constraint, the linear solver is able to compute an over-optimal solution. So, if the linear relaxation of the current node gives a worse bound than the best solution found so far, the current node can be pruned [34]. Moreover, the linear solver is able to provide another piece of information, namely *reduced costs*. For each variable x_i in the linear model, the reduced cost r_i is the derivative of the objective function with respect to x_i. Suppose we have a minimization problem $min(f)$, and that the linear relaxation provides a value LB (Lower Bound). Suppose that we already know a solution with cost UB (Upper Bound). Of course, if $LB \geq UB$, we can fail and backtrack. Otherwise, suppose that there is some variable x_i that in the optimal solution of the linear relaxation takes value 0, and suppose the reduced cost is 10. This means that, if we change the value of x_i to 1, the value of the objective function will increase of at least 10. If $LB + 10 \geq UB$, then I cannot add 1 to x_i, because that would mean going to a worse solution than the current best, so we can remove the value 1 from the domain of x_i. This is called *cost-based filtering* [80, 90].

Other techniques from (integer) linear programming have been adapted to include constraint programming. Column generation is a technique used in linear programming to solve very large problems. The basic idea is that the simplex algorithm uses a tableaux to represent the linear program, and uses reduced costs to drive the search. Since reduced costs are the derivatives of the objective function with respect to the variables in the current solution, if all reduced costs are positive, then there is no way to reduce the value of the objective function, i.e., we are in the optimal solution (global minimum). Otherwise, if there is at least a negative reduced cost, increasing the value of the corresponding

variable will reduce the objective function, and the search continues. However, if the tableaux contains a huge number of columns, finding a negative cost may become a constraint satisfaction problem itself that can be solved with various techniques, including constraint programming [96].

Bender's decomposition is another technique used to solve very large problems. The whole problem is decomposed into a master problem and a subproblem, that will then communicate. One of the two could be more easily solvable by an FD solver, while the other by a linear solver; this gives an interesting pattern to have the two solvers communicate [70, 17, 98].

Finally, various methods exist to integrate local search with CLP [38,112,78,39].

4.2 Symmetry Breaking

In CLP and constraint reasoning in general, there are several techniques that try to change the problem formulation to improve the efficiency of the solution process. For example, some approaches include rewriting (through folding and unfolding steps) a constraint logic program [77], to make it more efficient for a specific instance or a query. We will not go into further details, as the interested reader will find an exhaustive exposition in another chapter of this book [122].

Another interesting and useful idea is to try to remove some symmetrical parts of the search space, by rewriting the constraint program or by adding (by hand or automatically [108], in the CLP program) so-called *symmetry breaking constraints*. In fact, the presence of symmetries can expand exponentially the size of the search space. Consider, for example, a graph coloring problem: each node of a graph should be assigned a color from a finite palette (the same one for all nodes), with the constraint that two nodes connected with an arc should have different colors. Backtracking search will try to assign a value to a first node, for example color red to node N_1. Suppose that, after constraint propagation and a long search, we find out that there is no solution with $N_1 = red$: backtrack search will now choose the second value in the domain of N_1, say *blue*. However, since the colors are symmetric, there is no solution with blue as well. This observation can be used to reduce significantly the search space. Other problems have many more symmetries than the graph coloring. The classical benchmark problem in this research area is the social golfer, which is an abstraction of many real-life scenarios: N golf players want to play golf every week, in groups of M golfers; we have to find a schedule for W weeks such that no two players play in the same group more than once.

A first way to tackle this problem is by changing the constraint model, by switching to a representation with no symmetries, or with a reduced number of symmetries. The first solution to the social golfer problem was implemented by Stefano Novello in CLP [120]. The idea was to use a set representation (see also Section 2.2): the position of elements in a set is immaterial, so the intrinsic symmetry related to the order of the elements no longer exists.

Other solutions include finding the equivalence classes for the symmetries, and adding constraints that are satisfied only by one representative of each equivalence class. In the graph coloring example, one can leave only one element in

the domain of a given node. Of course, this simple constraint will not always remove all the symmetries, but it usually greatly reduces the search space. When the constraint problem is represented by a sequence of symmetric variables (i.e., every permutation of a solution is still a solution), one can impose that the variables are ordered. If the problem contains a matrix of variables, and exchanging two lines or two columns of a solution yields another solution, a lexicographic ordering between the rows/columns can be imposed [81].

In some cases, one has a very powerful heuristics for solving a CSP, and the heuristic can become less effective if we change the constraint model; in particular the heuristic could be deceived by the addition of symmetry breaking constraints. In those cases, one can revert to algorithms that break the symmetries during search: i.e., after exploring (unsuccessfully) some part of the search space, they prune the symmetrical parts of the already explored zones [114, 92, 79, 72].

All these methods assume that the symmetries are already known; however, there are also approaches trying to identify the symmetries from the specifications [108]. In some cases, one tries to detect the symmetries from the general model [33], without looking at the specific instance. E.g., the graph coloring problem has symmetries in general, irrespectively of the particular graph we are considering. In other cases, one tries to detect symmetries that hold only in the given instance we are about to solve [86].

5 Applications

CLP has shown to be successfully used in many application domains. For space reasons, we will just mention few of them, not intending to give a complete survey. The reader can refer to existing surveys on CLP applications [137], as well as on the chapter on applications of LP in this book [51].

In recent years, biology has been the source of interesting application problems for the whole of computer science, due to the large volume of data and the combinatorial nature of many scenarios. CLP, and constraint programming in general, has been recently applied to some of these problems [10]. In particular, CLP has been used to tackle the protein structure prediction problem, which is one of the most challenging problems in biological sciences, and which can be seen as an optimization problem [48, 55]. The complexity of constraint propagation was also studied [49]. The results obtained on small proteins show that CLP can be employed for studying protein simplified models. The advantage of CLP over other approaches lies in the rapid software prototyping, in the easy way of encoding heuristics, and in the several efficient constraint-based techniques, such as constraint propagation, to prune huge search spaces.

Constraint logic programming was also used to reason about spatial and temporal data, and a CLP solver was integrated with a geographical information system. One practical applications was the study of the mating habit of the crested porcupine [125], in which information is gathered through radio-collars and processed by a CLP program.

Planning and scheduling have always been two of the main application areas for constraint-based approaches [12]. Scheduling is the problem of assigning

a timing to the various tasks composing a complex activity, and often, other resources. As such, it has various specializations: in sport scheduling [131] one wants to fix the matches of a tournament; in school timetabling [132, 86] the aim is deciding when and where lessons take place, in crew rostering we have to find a sequencing of a given set of duties into rosters satisfying operational constraints [34], etc. [30, 36]. CLP(FD) has proved to be very successful in this area mainly because of an important global constraint, called `cumulative`. This constraint relates the start times, the durations, and the resource consumptions of a set of tasks, and it ensures that in any instant of time, the total resource consumption of the tasks being executed does not exceed a given limit. So, for a school timetabling, one can state that the rooms are resources: if in a school there are R available rooms, there cannot be more that R lessons at the same time. Teachers can also be considered as resources: two contemporary lessons cannot involve the same teacher, and so on. There are various implementations of the `cumulative` constraint, that give different balances of computational complexity (usually from $O(n^2)$ to $O(n^3)$) and achieved pruning.

Planning, instead, is the problem of finding a sequence of actions that, taken in the correct order, achieve a given goal. Each action has pre-conditions and post-conditions, and the automatic planner must ensure that the post-conditions of some action do not invalidate the pre-conditions of the subsequent actions. CLP(FD) is useful to detect such possible situations, called threats [14, 13], and to implement the temporal reasoning [121]. Also, some notable works propose to implement action description languages in CLP(FD) [62].

A remarkable amount of work in CLP is connected with database theories and applications. Considering the theory, the semantics of the U-Datalog language is cast through a CLP semantics, and, in particular, updates in rule bodies are specified through constraints [118]. Constraints are also used to schedule the transactions in a distributed database [111]. Constraints are also useful to represent incomplete information, e.g., in temporal-probabilistic databases [106].

CLP(FD) was used to find an optimal placement of sirens to alert the population in Venice of the high tide [9]. The map of the city is divided by a grid into cells, and for each cell a number of features is recorded, such as the average and maximum height of the buildings, their density, etc. The authors use a simulator to compute the sound propagation, and they relate the sound propagation with the position of the sirens through constraints. The objective is to find the best placement (that minimizes the number of sirens) such that in each cell the signal strength is greater than or equal to a given threshold.

Other authors [76] tackle the problem of detecting excess of pollution in the Venice lagoon. Every day, information is acquired through sensors, and fed to a decision support system. The system is implemented in CLP, and uses constraints to model the propagation of pollutants in the lagoon; it is able to provide suggestions to the Venice Water Magistracy on which implants to close, which to relocate, etc, to keep the level of pollution within acceptable levels.

The system LODE [91] applies CLP to reason about temporal information in an e-learning software devoted to deaf children. Deaf people can have difficulties

in understanding temporal relations in textual information, and such software helps them by proposing stories and exercises.

Verification is a very important application of CLP, that has been deeply studied by many authors in Italy and abroad. It is also a vast discipline, that includes important applications of theoretical and practical importance, such as security verification [57,44,16]. We will not delve into this fascinating discipline, because it is the subject of another chapter of this book [59].

6 Conclusions

Constraint Logic Programming is a computation paradigm that joins the theoretical features of Logic Programming (declarative semantics, soundness, completeness) with an important range of practical applications. However, additional efforts are needed to make it more widely applicable. Features such as uncertainty, multi-agent reasoning, lack of data, and vast amounts of information, just to cite few examples, should be fully integrated and satisfactorily handled in CLP-style languages if we want CLP to be successfully used also in more modern applications. We also see two other threats to the spreading of the CLP technology into the industrial world. One is the lack of a common syntax: as already hinted in Section 2.1, every CLP(FD) solver has itw own syntax for defining domains, and same constraints can have different names. There are standardisation efforts, and new modelling languages such as (Mini)Zinc [119] become more and more supported by CLP systems, but still the goal of a commonly agreed language seems far away. A second threat comes from imperative and object-oriented languages: many solvers are now available also with C++ (ILOG[2], Gecode [134]) or Java syntax (Choco, Jacop[3], JsetL [128]), giving up the gains coming from logic programming, but with the advantage of an easier integration into already developed applications. To keep up with those solvers, CLP languages should either provide new features, unapplicable to imperative/OOP languages, or have better integration with real world applications, with ability to develop attractive user interfaces, access to web services, and so on. Finally, CLP languages are usually tailored for the experienced user: one can develop extremely efficient search strategies, and heuristics for solving a specific problem, even with integration of different solvers, but these technologies are often out of reach for the naive user. CLP has taken the opposite viewpoint with respect to, e.g., SAT, MIP or ASP solvers: in those languages the user has only to state the problem, and the solver will choose a good strategy to solve it. Research trying to bridge the gap between the unexperienced user and the state-of-the-art technology could really boost the widespreading of the CLP word.

Acknowledgements. This research has been partially funded by PRIN 2008 project *'Innovative and multi-disciplinary approaches for constraint and preference reasoning'*.

[2] ILOG: www.ilog.com/products/cp/
[3] Choco: http://choco.emn.fr/, Jacop: http://jacop.osolpro.com/

References

1. Alberti, M., Cattafi, M., Gavanelli, M., Lamma, E., Chesani, F., Montali, M., Mello, P., Torroni, P.: Integrating abductive logic programming and description logics in a dynamic contracting architecture. In: IEEE Int. Conf. on Web Services (2009)
2. Alberti, M., Chesani, F., Gavanelli, M., Lamma, E., Mello, P., Montali, M.: An abductive framework for a-priori verification of web services. In: PPDP (2006)
3. Alberti, M., Chesani, F., Gavanelli, M., Lamma, E., Mello, P., Torroni, P.: Verifiable agent interaction in abductive logic programming: the SCIFF framework. ACM Transactions on Computational Logics 9(4) (2008)
4. Alberti, M., Lamma, E.: Synthesis of object models from partial models: A csp perspective. In: van Harmelen, F. (ed.) ECAI, pp. 116–120. IOS Press, Amsterdam (2002)
5. Alpuente, M., Sessa, M. (eds.): GULP-PRODE 1995 (1995)
6. Apt, K.R.: Principles of Constraint Programming. Cambridge Univ. Press, Cambridge (2003)
7. Apt, K.R., Wallace, M.G.: Constraint Logic Programming Using ECL^{i}PS^e. Cambridge University Press, Cambridge (2006)
8. Armando, A., Melis, E., Ranise, S.: Constraint solving in logic programming and in automated deduction: A comparison. In: Giunchiglia, F. (ed.) AIMSA 1998. LNCS (LNAI), vol. 1480, pp. 28–38. Springer, Heidelberg (1998)
9. Avanzini, F., Rocchesso, D., Belussi, A., Dal Palù, A., Dovier, A.: Designing an urban-scale auditory alert system. IEEE Computer 37(9), 55–61 (2004)
10. Backofen, R., Gilbert, D.: Bioinformatics and constraints. In: Rossi, et al [127]
11. Bagnara, R., Gori, R., Hill, P.M., Zaffanella, E.: Finite-tree analysis for constraint logic-based languages. In: Cousot, P. (ed.) SAS 2001. LNCS, vol. 2126, pp. 165–184. Springer, Heidelberg (2001)
12. Baptiste, P., Laborie, P., Le Pape, C., Nuijten, W.: Constraint-based scheduling and planning. In: Rossi, et al [127]
13. Barruffi, R., Milano, M., Montanari, R.: Planning for security management. IEEE Intelligent Systems 16(1), 74–80 (2001)
14. Barruffi, R., Milano, M., Torroni, P.: Planning while executing: A constraint-based approach. In: Ohsuga, S., Raś, Z.W. (eds.) ISMIS 2000. LNCS (LNAI), vol. 1932, pp. 228–236. Springer, Heidelberg (2000)
15. Baselice, S., Bonatti, P., Gelfond, M.: Towards an integration of answer set and constraint solving. In: Gabbrielli, M., Gupta, G. (eds.) ICLP 2005. LNCS, vol. 3668, pp. 52–66. Springer, Heidelberg (2005)
16. Bella, G., Bistarelli, S.: Soft constraint programming to analysing security protocols. TPLP 4(5-6), 545–572 (2004)
17. Benini, L., Lombardi, M., Mantovani, M., Milano, M., Ruggiero, M.: Multi-stage Benders decomposition for optimizing multicore architectures. In: Perron, L., Trick, M.A. (eds.) CPAIOR 2008. LNCS, vol. 5015, pp. 36–50. Springer, Heidelberg (2008)
18. Bergenti, F., Dal Palù, A., Rossi, G.: Generalizing finite domain constraint solving. In: Formisano, A. (ed.) CILC 2008 (2008)
19. Bertolino, B., Bonatti, P.A., Montesi, D., Pelagatti, S.: Correctness and completeness of logic programs under the CLP schema. In: Asirelli, P. (ed.) Proc. Sixth Italian Conference on Logic Programming, Pisa, Italy, pp. 391–405 (1991)
20. Bessiere, C.: Constraint propagation. In: Rossi, et al. [127]

21. Bessière, C. (ed.): CP 2007. LNCS, vol. 4741. Springer, Heidelberg (2007)
22. Bistarelli, S., Frühwirth, T.W., Marte, M.: Soft constraint propagation and solving in chrs. In: SAC, pp. 1–5. ACM, New York (2002)
23. Bistarelli, S., Gabbrielli, M., Meo, M., Santini, F.: Timed soft concurrent constraint programs. In: Lea, D., Zavattaro, G. (eds.) COORDINATION 2008. LNCS, vol. 5052, pp. 50–66. Springer, Heidelberg (2008)
24. Bistarelli, S., Montanari, U., Rossi, F.: Semiring based constraint solving and optimization. Journal of the ACM 44(2), 201–236 (1997)
25. Bistarelli, S., Montanari, U., Rossi, F.: Semiring-based constraint logic programming. In: IJCAI 2001, pp. 352–357 (2001)
26. Bistarelli, S., Montanari, U., Rossi, F.: Soft concurrent constraint programming. In: Le Métayer, D. (ed.) ESOP 2002. LNCS, vol. 2305, pp. 53–67. Springer, Heidelberg (2002)
27. Bonatti, P., Calimeri, F., Leone, N., Ricca, F.: Answer Set Programming. In: Dovier, A., Pontelli, E. (eds.) 25 Years of Logic Programming, ch.8. LNCS, vol. 6125, pp. 159–182. Springer, Heidelberg (2010)
28. Borning, A., Maher, M., Martindale, A., Wilson, M.: Constraint hierarchies and logic programming. In: Levi, G., Martelli, M. (eds.) ICLP (1989)
29. Bruscoli, P., Levi, F., Levi, G., Meo, M.: Compilative constructive negation in constraint logic programs. In: Tison, S. (ed.) CAAP 1994. LNCS, vol. 787, pp. 52–67. Springer, Heidelberg (1994)
30. Brusoni, V., Console, L., Lamma, E., Mello, P., Milano, M., Terenziani, P.: Resource-based vs. task-based approaches for scheduling problems. In: Michalewicz, M., Raś, Z.W. (eds.) ISMIS 1996. LNCS, vol. 1079. Springer, Heidelberg (1996)
31. Buscemi, M.G., Montanari, U.: A survey of constraint-based programming paradigms. Computer Science Review 2(3), 137–141 (2008)
32. Cabeza, D., Hermenegildo, M.: Implementing distributed concurrent constraint execution in the CIAO system. In: Lucio, P., Martelli, M., Navarro, M. (eds.) APPIA-GULP-PRODE (1996)
33. Cadoli, M., Mancini, T.: Using a theorem prover for reasoning on constraint problems. In: Bandini, S., Manzoni, S. (eds.) AI*IA. Springer, Heidelberg (2005)
34. Caprara, A., Focacci, F., Lamma, E., Mello, P., Milano, M., Toth, P., Vigo, D.: Integrating constraint logic programming and operations research techniques for the crew rostering problem. Softw. Pract. Exper. 28(1), 49–76 (1998)
35. Carlsson, M., Widen, J.: SICStus Prolog User's Manual. Technical report, Swedish Institute of Computer Science (SICS) (1999)
36. Carraresi, P., Gallo, G., Rago, G.: A hypergraph model for constraint logic programming and applications to bus drivers' scheduling. AMAI 8(3-4) (1993)
37. Cheadle, A., Harvey, W., Sadler, A., Schimpf, J., Shen, K., Wallace, M.: ECLiPSe: a tutorial introduction (2003), `http://eclipse-clp.org/doc/tutorial`
38. Cipriano, R., Di Gaspero, L., Dovier, A.: Hybrid approaches for rostering: A case study in the integration of constraint programming and local search. In: Almeida, F., Blesa Aguilera, M.J., Blum, C., Moreno Vega, J.M., Pérez Pérez, M., Roli, A., Sampels, M. (eds.) HM 2006. LNCS, vol. 4030, pp. 110–123. Springer, Heidelberg (2006)
39. Cipriano, R., Di Gaspero, L., Dovier, A.: A hybrid solver for large neighborhood search: Mixing Gecode and EasyLocal^{++}. In: Sampels, M. (ed.) HM 2009. LNCS, vol. 5818, pp. 141–155. Springer, Heidelberg (2009)
40. Codognet, P., Diaz, D.: Compiling constraints in clp(fd). J. Log. Prog. (1996)

41. Colmerauer, A.: An introduction to Prolog-III. Communication of the ACM (1990)
42. Colmerauer, A.: Prolog II reference manual and theoretical model. Technical report, Groupe Intelligence Artificielle, Universitè Aix-Mareseille II (October 1982)
43. Colussi, L., Marchiori, E., Marchiori, M.: A dataflow semantics for constraint logic programs. In: Alpuente, Sessa [5], pp. 557–568
44. Corin, R., Etalle, S.: An improved constraint-based system for the verification of security protocols. In: Hermenegildo, M.V., Puebla, G. (eds.) SAS 2002. LNCS, vol. 2477, pp. 326–341. Springer, Heidelberg (2002)
45. Cucchiara, R., Gavanelli, M., Lamma, E., Mello, P., Milano, M., Piccardi, M.: Extending CLP(FD) with interactive data acquisition for 3D visual object recognition. In: Proc. PACLP 1999, pp. 137–155 (1999)
46. Cucchiara, R., Gavanelli, M., Lamma, E., Mello, P., Milano, M., Piccardi, M.: From eager to lazy constrained data acquisition: A general framework. New Generation Computing 19(4), 339–367 (2001)
47. Dahl, V., Niemelä, I. (eds.): ICLP 2007. LNCS, vol. 4670. Springer, Heidelberg (2007)
48. Dal Palù, A., Dovier, A., Fogolari, F.: Constraint logic programming approach to protein structure prediction. BMC Bioinformatics 5 (2004)
49. Dal Palù, A., Dovier, A., Pontelli, E.: Computing approximate solutions of the protein structure determination problem using global constraints on discrete crystal lattices. Int'l Journal of Data Mining and Bioinformatics 4(1) (January 2010)
50. Dal Palù, A., Dovier, A., Pontelli, E., Rossi, G.: Integrating finite domain constraints and CLP with sets. In: PPDP 2003, pp. 219–229. ACM, New York (2003)
51. Dal Palù, A., Torroni, P.: 25 Years of Applications of Logic Programming. In: Dovier, Pontelli [68], vol. 6125, ch.14, pp. 298–325 (2010)
52. de Boer, F.S., Di Pierro, A., Palamidessi, C.: An algebraic perspective of constraint logic programming. Journal of Logic and Computation 7(1), 1–38 (1997)
53. de Boer, F.S., Gabbrielli, M.: Infinite computations in concurrent constraint programming. Electr. Notes Theor. Comput. Sci. 6 (1997)
54. de Boer, F.S., Gabbrielli, M., Meo, M.C.: A timed concurrent constraint language. Inf. Comput. 161(1), 45–83 (2000)
55. De Maria, E., Dovier, A., Montanari, A., Piazza, C.: Exploiting model checking in constraint-based approaches to the protein folding. In: WCB 2006 (2006)
56. Dechter, R.: Constraint Processing. Morgan Kaufmann, San Francisco (2003)
57. Delzanno, G., Etalle, S.: Proof theory, transformations, and logic programming for debugging security protocols. In: Pettorossi, A. (ed.) LOPSTR 2001. LNCS, vol. 2372, p. 76. Springer, Heidelberg (2002)
58. Delzanno, G., Gabbrielli, M., Meo, M.: A compositional semantics for CHR. In: PPDP 2005, pp. 209–217. ACM, New York (2005)
59. Delzanno, G., Giacobazzi, R., Ranzato, F.: Analysis, Abstract Interpretation, and Verification in (Constraint Logic) Programming. In: Dovier, Pontelli [68], vol. 6125, ch. 7, pp. 136–158 (2010)
60. Díaz, D., Codognet, P.: GNU Prolog: Beyond compiling Prolog to C. In: Pontelli, E., Santos Costa, V. (eds.) PADL 2000. LNCS, vol. 1753, p. 81. Springer, Heidelberg (2000)
61. Dondossola, G., Ratto, E.: GRF temporal reasoning language. Technical report, CISE, Milano (1993)
62. Dovier, A., Formisano, A., Pontelli, E.: Multivalued action languages with constraints in CLP(FD). In: Dahl, Niemelä [47], pp. 255–270

63. Dovier, A., Formisano, A., Pontelli, E.: An empirical study of constraint logic programming and answer set programming solutions of combinatorial problems. J. Exp. Theor. Artif. Intell. 21(2) (2009)
64. Dovier, A., Omodeo, E., Pontelli, E., Rossi, G.: {log}: A logic programming language with finite sets. In: ICLP, pp. 111–124 (1991)
65. Dovier, A., Piazza, C., Pontelli, E., Rossi, G.: Sets and constraint logic programming. ACM Trans. Program. Lang. Syst. 22(5), 861–931 (2000)
66. Dovier, A., Piazza, C., Rossi, G.: A uniform approach to constraint-solving for lists, multisets, compact lists, and sets. ACM Trans. Comput. Log. 9(3) (2008)
67. Dovier, A., Policriti, A., Rossi, G.: A uniform axiomatic view of lists, multisets, and sets, and the relevant unification algorithms. Fundam. Inform. 36(2-3) (1998)
68. Dovier, A., Pontelli, E. (eds.): 25 Years of Logic Programming. LNCS, vol. 6125. Springer, Heidelberg (2010)
69. Endriss, U., Mancarella, P., Sadri, F., Terreni, G., Toni, F.: The CIFF proof procedure for abductive logic programming with constraints. In: Alferes, J.J., Leite, J. (eds.) JELIA 2004. LNCS (LNAI), vol. 3229, pp. 31–43. Springer, Heidelberg (2004)
70. Eremin, A., Wallace, M.: Hybrid Benders decomposition algorithms in constraint logic programming. In: Walsh [138], pp. 1–15
71. Etalle, S., Gabbrielli, M., Meo, M.: Transformations of CCP programs. ACM Trans. Program. Lang. Syst. 23(3), 304–395 (2001)
72. Fahle, T., Schamberger, S., Sellman, M.: Symmetry breaking. In: Walsh [138]
73. Falaschi, M., Gabbrielli, M., Marriott, K., Palamidessi, C.: Confluence in concurrent constraint programming. Theor. Comput. Sci. 183(2), 281–315 (1997)
74. Falaschi, M., Gabbrielli, M., Marriott, K., Palamidessi, C.: Constraint logic programming with dynamic scheduling: A semantics based on closure operators. Information and Computation 137(1), 41–67 (1997)
75. Farenzena, M., Fusiello, A., Dovier, A.: Reconstruction with interval constraints propagation. In: CVPR, pp. 1185–1190. IEEE Computer Society, Los Alamitos (2006)
76. Festa, G., Sardu, G., Felici, R.: A decision support system for the Venice lagoon. In: Herold, A. (ed.) Handbook of parallel constraint logic programming applications (1995)
77. Fioravanti, F., Pettorossi, A., Proietti, M.: Transformation rules for locally stratified constraint logic programs. In: Bruynooghe, M., Lau, K.-K. (eds.) Program Development in Computational Logic. LNCS, vol. 3049, pp. 291–339. Springer, Heidelberg (2004)
78. Focacci, F., Laburthe, F., Lodi, A.: Local search and constraint programming: LS and CP illustrated on a transportation problem. In: Milano, M. (ed.) Constraint and Integer Programming. Towards a Unified Methodology, pp. 137–167. Kluwer Academic Publishers, Dordrecht (2003)
79. Focacci, F., Milano, M.: Global cut framework for removing symmetries. In: Walsh [138], pp. 77–92
80. Focacci, F., Milano, M., Lodi, A.: Soving TSP with time windows with constraints. In: International Conference on Logic Programming, pp. 515–529 (1999)
81. Frisch, A., Hnich, B., Kızıltan, Z., Miguel, I., Walsh, T.: Propagation algorithms for lexicographic ordering constraints. Artif. Int. 170(10), 803–834 (2006)
82. Frühwirth, T.: Theory and practice of constraint handling rules. Journal of Logic Programming 37, 95–138 (1998)
83. Frühwirth, T., Di Pierro, A., Wiklicky, H.: An implementation of probabilistic constraint handling rules. In: Comini, M., Falaschi, M. (eds.) WFLP (2002)

84. Gabbrielli, M., Dore, G.M., Levi, G.: Observable semantics for constraint logic programs. J. Log. Comput. 5(2), 133–171 (1995)
85. Gabbrielli, M., Palamidessi, C., Valencia, F.D.: Concurrent and Reactive Constraint Programming. In: Dovier, Pontelli [68], vol. 6125, ch. 11, pp. 225–248 (2010)
86. Gavanelli, M.: University timetabling in ECLiPSe. ALP Newsletter 19(3) (2006)
87. Gavanelli, M., Alberti, M., Lamma, E.: Integration of abductive reasoning and constraint optimization in SCIFF. In: Hill, P.M., Warren, D.S. (eds.) ICLP 2009. LNCS, vol. 5649, pp. 387–401. Springer, Heidelberg (2009)
88. Gavanelli, M., Lamma, E., Mello, P., Milano, M.: Dealing with incomplete knowledge on CLP(FD) variable domains. ACM TOPLAS 27(2) (2005)
89. Gavanelli, M., Lamma, E., Mello, P., Torroni, P.: An abductive framework for information exchange in multi-agent systems. In: Dix, J., Leite, J. (eds.) CLIMA 2004. LNCS (LNAI), vol. 3259, pp. 34–52. Springer, Heidelberg (2004)
90. Gavanelli, M., Milano, M.: Cost-based filtering for determining the Pareto frontier. In: Junker, U., Kießling, W. (eds.) Multidisciplinary Workshop on Advances in Preference Handling, in conjunction with ECAI 2006 (2006)
91. Gennari, R., Mich, O.: Constraint-based temporal reasoning for e-learning with LODE. In: Bessiere [21]
92. Gent, I.P., Smith, B.M.: Symmetry breaking in constraint programming. In: Horn, W. (ed.) ECAI, pp. 599–603. IOS Press, Amsterdam (2000)
93. Georget, Y., Codognet, P.: Compiling semiring-based constraints with clp(fd,s). In: Maher, M.J., Puget, J.-F. (eds.) CP 1998. LNCS, vol. 1520, p. 205. Springer, Heidelberg (1998)
94. Giacobazzi, R., Debray, S., Levi, G.: Generalized semantics and abstract interpretation for constraint logic programs. J. Log. Program. 25(3) (1995)
95. Giordano, L., Toni, F.: Knowledge representation and non-monotonic reasoning. In: Dovier, Pontelli [68], vol. 6125, ch. 5, pp. 86–110 (2010)
96. Gualandi, S., Malucelli, F.: Constraint programming-based column generation. 4OR: A Quarterly Journal of Operations Research 7(2), 113–137 (2009)
97. Hoeve, W.-J.V., Pesant, G., Rousseau, L.-M.: On global warming: Flow-based soft global constraints. Journal of Heuristics 12(4-5), 347–373 (2006)
98. Hooker, J.: Logic-Based Methods for Optimization: Combining Optimization and Constraint Satisfaction. John Wiley & Sons, Chichester (2000)
99. Jaffar, J., Lassez, J.-L.: Constraint logic programming. In: Proc. 14th symp. on Principles of programming languages. ACM, New York (1987)
100. Jaffar, J., Michaylov, S., Stuckey, P., Yap, R.: The CLP(R) Language and System. ACM Transactions on Programming Languages and Systems (1992)
101. Kakas, A.C., Kowalski, R.A., Toni, F.: Abductive Logic Programming. Journal of Logic and Computation 2(6), 719–770 (1993)
102. Kakas, A.C., Mancarella, P.: On the relation between Truth Maintenance and Abduction. In: Fukumura, T. (ed.) PRICAI (1990)
103. Kakas, A.C., Michael, A., Mourlas, C.: ACLP: Abductive Constraint Logic Programming. Journal of Logic Programming 44(1-3), 129–177 (2000)
104. Kakas, A.C., van Nuffelen, B., Denecker, M.: $\mathcal{A}$-System: Problem solving through abduction. In: Nebel, B. (ed.) Proc. of IJCAI 2001, pp. 591–596 (2001)
105. Mackworth, A.: Consistency in networks of relations. Artif. Intell. 8(1) (1977)
106. Majkic, Z.: Constraint logic programming and logic modality for event's valid-time approximation. In: 2nd Indian Int. Conf. on Artificial Intelligence (2005)
107. Mancarella, P., Terreni, G., Toni, F.: Web sites verification: An abductive logic programming tool. In: Dahl, Niemelä [47]

108. Mancini, T., Cadoli, M.: Detecting and breaking symmetries by reasoning on problem specifications. In: Zucker, J.-D., Saitta, L. (eds.) SARA 2005. LNCS (LNAI), vol. 3607, pp. 165–181. Springer, Heidelberg (2005)
109. Mancini, T., Micaletto, D., Patrizi, F., Cadoli, M.: Evaluating ASP and commercial solvers on the CSPLib. Constraints 13(4), 407–436 (2008)
110. Manco, G., Turini, F.: A structural (meta-logical) semantics for linear objects. In: Alpuente, Sessa [5], pp. 421–434
111. Mascardi, V., Merelli, E.: Agent-oriented and constraint technologies for distributed transaction management. In: Parenti, R., Masulli, F. (eds.) Proc. Int. ICSC Symposia IIA 1999 and SOCO 1999 (1999)
112. Merelli, E., De Leone, R., Martelli, M., Panti, M.: Embedding constraint logic programming formula in a local search algorithm for job shop scheduling. In: EURO XVI, Bruxelles (July 1998)
113. Meseguer, P., Rossi, F., Schiex, T.: Soft constraints. In: Rossi, et al [127]
114. Meseguer, P., Torras, C.: Exploiting symmetries within constraint satisfaction search. Artificial Intelligence 129(1-2), 133–163 (2001)
115. Mesnard, F., Ruggieri, S.: On proving left termination of constraint logic programs. ACM Trans. Comput. Log. 4(2) (2003)
116. Mohr, R., Masini, G.: Good old discrete relaxation. In: ECAI (1988)
117. Montanari, U.: Networks of constraints: Fundamental properties and applications to picture processing. Information Science 7, 95–132 (1974)
118. Montesi, D., Bertino, E., Martelli, M.: Transactions and updates in deductive databases. IEEE Trans. Knowledge and Data Engineering 9(5), 784–797 (1997)
119. Nethercote, N., Stuckey, P., Becket, R., Brand, S., Duck, G., Tack, G.: MiniZinc: Towards a standard CP modelling language. In: Bessiere [21], pp. 529–543
120. Novello, S.: ECL$^{\text{i}}$PS$^{\text{e}}$ examples (1998), `http://eclipse-clp.org/examples/golf.ecl.txt`
121. Orlandini, A.: Model-based rescue robot control with ECLiPSe framework. In: Oddi, A., Cesta, A., Fages, F., Policella, N., Rossi, F. (eds.) CSCLP (2008)
122. Pettorossi, A., Proietti, M., Senni, V.: The Transformational Approach to Program Development. In: Dovier, Pontelli [68], vol. 6125, ch. 6, pp. 111–135 (2010)
123. Pierro, A.D., Wiklicky, H.: An operational semantics for probabilistic concurrent constraint programming. In: ICCL, pp. 174–183 (1998)
124. Provetti, A., Rossi, G.: Action specifications in {log}. In: Falaschi, M., Navarro, M., Policriti, A. (eds.) APPIA-GULP-PRODE (1997)
125. Raffaetà, A., Ceccarelli, T., Centeno, D., Giannotti, F., Massolo, A., Parent, C., Renso, C., Spaccapietra, S., Turini, F.: An application of advanced spatio-temporal formalisms to behavioural ecology. Geoinformatica 12(1), 37–72 (2008)
126. Raffaetà, A., Frühwirth, T.W.: Spatio-temporal annotated constraint logic programming. In: Ramakrishnan, I.V. (ed.) PADL 2001. LNCS, vol. 1990, pp. 259–273. Springer, Heidelberg (2001)
127. Rossi, F., van Beek, P., Walsh, T. (eds.): Handbook of Constraint Programming. Elsevier, Amsterdam (2006)
128. Rossi, G., Panegai, E., Poleo, E.: JSetL: a Java library for supporting declarative programming in Java. Softw. Pract. Exper. 37(2), 115–149 (2007)
129. Ruttkay, Z.: Fuzzy constraint satisfaction. In: FUZZ-IEEE 1994, Orlando, FL (1994)
130. Saraswat, V.A.: Concurrent Constraint Programming. MIT Press, Cambridge (2003)
131. Schaerf, A.: Scheduling sport tournaments using constraint logic programming. Constraints 4(1), 43–65 (1999)

132. Schaerf, A.: A survey of automated timetabling. Artif. Intell. Review 13(2) (1999)
133. Schiex, T., Fargier, H., Verfaillie, G.: Valued constraint satisfaction problems: hard and easy problems. In: IJCAI 1995, pp. 631–637 (1995)
134. Schulte, C., Stuckey, P.: Efficient constraint propagation engines. In: ToPLaS 2008 (2008)
135. Van Hentenryck, P.: Constraint Satisfaction in Logic Programming. MIT, Cambridge (1989)
136. van Hoeve, W.-J.: The all different constraint: a survey. In: Sixth Annual Workshop of the ERCIM Working Group on Constraints (2001)
137. Wallace, M.: Practical applications of constraint programming. Constraints (1996)
138. Walsh, T. (ed.): CP 2001. LNCS, vol. 2239. Springer, Heidelberg (2001)
139. Wielemaker, J., Huang, Z., Van der Meij, L.: SWI-Prolog and the web. Theory and Practice of Logic Programming 8(3), 363–392 (2008)
140. Zanarini, A., Milano, M., Pesant, G.: Improved algorithm for the soft global cardinality constraint. In: Beck, J.C., Smith, B.M. (eds.) CPAIOR 2006. LNCS, vol. 3990, pp. 288–299. Springer, Heidelberg (2006)
141. Zhou, N.-F.: Programming finite-domain constraint propagators in action rules. Theory and Practice of Logic Programming 6(5), 483–507 (2006)