Università degli Studi di Bologna
DEIS

Reasoning on Constraints in Constraint Logic Programming

Evelina Lamma Paola Mello Michela Milano

DEIS Technical Report no. DEIS-LIA-96-006 *LIA Series no. 11*

Reasoning on Constraints in Constraint Logic Programming

Evelina Lamma [1] Paola Mello [2] Michela Milano [1]

[1] *DEIS, Università di Bologna*
Viale Risorgimento 2, 40136 Bologna, Italy
Tel: +39 51 6443033, Fax: +39 51 6443073
{elamma,mmilano}@deis.unibo.it
[2] *Istituto di Ingegneria, Università di Ferrara*
Via Saragat 1, 41100 Ferrara, Italy
pmello@ing.unife.it

Abstract. Constraint Logic Programming solvers on finite domains use constraints to prune those combinations of assignments which cannot appear in any consistent solution. There are applications, such as temporal reasoning or scheduling, requiring some form of qualitative reasoning where constraints can be changed (restricted) during the computation or even chosen when disjunction occurs. We embed in a CLP(FD) solver the concept of constraints as first class objects. In the extended language, variables range on finite domains of integers and *relation variables* range on finite domains of relation symbols. We define operations and constraints on the two sorts and one constraint linking the two. Programming examples are given, showing the advantages and the expressive power of the extension proposed.

Keywords: *Constraint Logic Programming, Meta-Level Reasoning*

Contents

1 Introduction

Constraint Logic Programming (CLP) [8] is a programming paradigm combining the advantages of Logic Programming and the efficiency of constraint solving. In this paper, we are interested in CLP over finite domains, CLP(FD), as introduced by the CHIP programming language [3]. CLP(FD) variables range over a finite domain of integers, and are linked by means of constraints. During the computation, constraints are used in order to actively prune the search space, by reducing the variable domains. For this reason, CLP(FD) has been proved to be a suitable tool for solving hard combinatorial problems such as graph coloring, planning, scheduling, sequencing and assignment problems, to name a few (see for instance [4]). Many of these problems deal with temporal constraints which can be either *quantitative*, e.g., they link the temporal location of points (intervals) along the time line, or the distance between two points (intervals), [2], or *qualitative*, e.g., they link the relative positions of points and intervals, [1, 18].

While the quantitative reasoning can be easily handled with CLP(FD), the qualitative one leads to some problems. For example, suppose we have three temporal points T_1, T_2 and T_3 ranging from 1 to 10 linked by the relations $T_1 \leq T_2$, $T_2 < T_3$. CLP(FD) constraint propagation reduces the size of variable domains, thus leading to $T_1 :: [1..9]$, $T_2 :: [1..9]$ and $T_3 :: [2..10]$. However, by using CLP(FD) solvers, it is not possible to infer the relation between T_1 and T_3. This happens because CLP(FD) uses constraints in order to restrict variable domains, but it does never reason on constraint as domain elements, thus computing new (tighter) constraints during the computation.

As another example, suppose we have two variables X and Y with an undefined domain. CLP(FD) solvers consider the default domain as $[-max, max]$. Therefore, the constraints $X < Y$ and $X > Y$ are recognized as inconsistent after a number of propagations proportional to max. By reasoning on relations, we are able to determine the inconsistency in one propagation step.

In this paper, we present an extension of the CLP framework that makes it possible to reason on constraints. Intuitively, we allow the CLP(FD) language to treat relations as domain elements. In this way, we are able to perform operations and to pose constraints on them and change relations during the computation. For instance, in the first above mentioned example, the relation between T_1 and T_2 can be represented by a *relation variable* called $R_{T_1T_2}$, which ranges on a finite domain of relation symbols $[<, =]$ and the relation between T_2 and T_3 can be represented by another *relation variable* called $R_{T_2T_3}$, which is instantiated to the value $<$. Therefore, the relation between T_1 and T_3 can be computed as $<$ by composing the first two variables.

In the second example, the relation between X and Y can be represented by a relation variable R_{xy} whose value is $<$. When the second constraint is considered, an inconsistency arises since $>$ is not contained in the domain of R_{xy}.

Based on this intuition, we present a two sorted first order language on integers and relations. On integers we have the usual operations and constraints, while on relations we have composition, union operations and equality, inequality and disequality constraints. Beside the increased expressive power, we obtain an improvement in the propagation of disjunctive constraints when reasoning on relations.

The paper is organized as follows: in section 2, we concentrate on the language extension. We provide the declarative and operational semantics of the resulting language and give some implementation details. Programming examples are given in section 3 showing the advantages and the expressive power of the extension proposed. Related works are presented in section 4. Conclusions and future works follow.

2 Extending the CLP(FD) Language

In this section we describe a constraint solver which makes it possible to reason on relations. The solver is built on a two sorted first order language (on integers and relations). We have to define operations and constraints on integers (they are the usual operations and constraints of CLP(FD) solvers) and operations and constraints on relations (like composition, intersection and union).

A constraint on finite domains can be in the form: $x :: [a_1, \ldots, a_n]$ or $t_1 \; R \; t_2$ where t_1 and t_2 are finite terms, i.e., variables, natural numbers and usual expressions built from the addition and multiplication operations, and R is one of the constraints $\{>, \geq, <, \leq, =, \neq\}$. Intuitively, we want to extend this language so as to make it possible to express constraints like $rel(X, R_{xy}, Y)$ which holds if and only if the *relation variable* R_{XY} represents the relations between X and Y. This constraint links integer variables X and Y and a relation variable R_{xy}. Therefore, we have to work on two sorts: integers Z and relations Rel. For the sake of clarity, some points should be discussed about the Rel sort. The first point concerns the relations $\leq$, $\geq$ and $\neq$. Consider for example the relation $\leq$. The same considerations apply for the other two. The symbol $\leq$ is a shorthand for the disjunction $<$ or $=$. Therefore, a relation variable whose domain contains the symbol $\leq$ leads to the same propagation and the same consistency check of one with a finite domain containing both the relations, i.e., $[<, =]$. The main difference concerns the labeling step. In fact, when a consistent solution is found in the first case, we have an instantiation of the relation variable to the unique symbol $\leq$ while in the second case we possibly have two different solutions: one for the relation $<$ and the other for the symbol $=$. Therefore, by using the symbol $\leq$ we allow a more concise but less precise representation of the solutions. In the following, we use also the symbols $\leq$, $\geq$ and $\neq$ as a shorthand for the corresponding disjunctive relations to maintain the same constraint symbols as in CLP(FD). However, in practice they can be avoided since they do not increase the expressive power.

The second point concerns the fact that the *Rel* sort is not a partially ordered set since it lacks of a *greatest lower bound*, referred to as $\perp$ in the following, to complete the lattice. The symbol $\perp$ represents the inconsistent relation. It is always false for any pair of variables. In the following, we avoid the use of the inconsistent relation in relation variable domains and we fail if a relation variable domain becomes empty. However, the symbol $\perp$ can be used by the user for explicitly handling the failure and recovering from it.

On the contrary, the lattice has the *least upper bound*, referred to as $\top$ in the following, representing the most relaxed constraint, i.e., the disjunction $<$ or $=$ or $>$. It is always true for any pair of variables. This relation can be used as default for instantiating relation variables with undefined domains.

2.1 Syntax and Declarative Semantics

We consider a two sorted first order language Σ with two associated sorts, S_1 and S_2, where S_1 represents the integer sort, and S_2 the relation sort. The carriers of the two sorts are integer numbers Z and the relation set $Rel = \{<, \leq, >, \geq, \neq, =, \top\}$.

For the integer sort, unary constraints link a variable to range over a finite domain of integers $X :: [a_1, \ldots, a_n]$. The meaning of this constraint is:

$$X :: [a_1, \ldots, a_n] \ \leftrightarrow\ X = a_1 \vee \ldots \vee X = a_n$$

As usual, binary constraints are $>, \geq, <, \leq, =, \neq$ and operations are addition (+) and multiplication (*).

As concern the relation sort, unary constraints link each relation variable to range over a finite domain of relations $R :: [r_1, \ldots, r_n]$. The meaning of this constraint is:

$$R :: [r_1, \ldots, r_n] \ \leftrightarrow\ R =_r r_1 \vee \ldots \vee R =_r r_n$$

On the relation sort, equality and inequality constraints can be defined as usual. We will refer to $=_r$ and $\neq_r$ when talking about constraints between relations, i.e., constraints between relation variables. A partial order relation $<_r$ can be defined on the relation sort as well. The partial order relation on *Rel* is defined by the lattice (described in section 2) where the top is the constraint $\top$, i.e., the most relaxed constraint, called *universal* constraint, and the bottom is the inconsistent relation $\perp$ which is not included in the carrier *Rel*. The ordering relation is represented as the constraint

$$R_1 <_r R_2 \ \leftrightarrow\ R_1 =_r lb(R_2)$$

where the symbol lb means lower bound[1]. For example, the constraint $<$ is a lower bound of either $\leq$, or $\neq$ or $\top$.

[1] The lower bound is not uniquely defined. We cannot use the greatest lower bound since the relation $<_r$ should be symmetrical. Thus, we can give also another definition of the partial order relation $<_r$: $R_1 <_r R_2 \leftrightarrow \forall x, y\ R_1(x, y) \rightarrow R_2(x, y)$.

Operations on relations are composition and union. We can define them as two predicates: *comp/3* and *union/3*. The signature of the predicate *comp* is $\langle Rel, Rel, Rel\rangle$. The declarative meaning of *comp* is the following:

$$comp(R_1, R_2, R_3) \;\leftrightarrow\; R_1 \otimes R_2 =_r R_3$$

The operation $\otimes$ is defined by a transitivity table described in [18] and reported here for the sake of completeness. For example, if $R_1 :: [<, =]$ and $R_2 :: [<, =]$, then $R_3 :: [<, =]$.

$\otimes$	$<$	$\leq$	$>$	$\geq$	$=$	$\neq$	$\top$
$<$	$<$	$<$	$\top$	$\top$	$<$	$\top$	$\top$
$\leq$	$<$	$\leq$	$\top$	$\top$	$\leq$	$\top$	$\top$
$>$	$\top$	$\top$	$>$	$>$	$>$	$\top$	$\top$
$\geq$	$\top$	$\top$	$>$	$\geq$	$\geq$	$\top$	$\top$
$=$	$<$	$\leq$	$>$	$\geq$	$=$	$\neq$	$\top$
$\neq$	$\top$	$\top$	$\top$	$\top$	$\neq$	$\top$	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$

Table 1: Composition of Constraints

The signature of the predicate *union* is $\langle Rel, Rel, Rel\rangle$ and its declarative meaning is:

$$union(R_1, R_2, R_3) \;\leftrightarrow\; R_1 =_r R_3 \;\vee\; R_2 =_r R_3$$

For example, if $R_1 :: [<, =]$ and $R_2 :: [<, >]$, then $R_3 :: [<, =, >]$.

Now, we define the $rel(X, R_{xy}, Y)$ constraint linking the two sorts, whose signature is $\langle Z, Rel, Z\rangle$. The declarative meaning of the $rel/3$ is the following:

$$\begin{aligned} rel(X, R_{xy}, Y) \quad \leftrightarrow \quad & (R_{xy} =_r \text{‘}<\text{‘}) \wedge \; X < Y \;\vee \\ & (R_{xy} =_r \text{‘}\leq\text{‘}) \wedge \; X \leq Y \;\vee \\ & (R_{xy} =_r \text{‘}>\text{‘}) \wedge \; X > Y \;\vee \\ & (R_{xy} =_r \text{‘}\geq\text{‘}) \wedge \; X \geq Y \;\vee \\ & (R_{xy} =_r \text{‘}\neq\text{‘}) \wedge \; X \neq Y \;\vee \\ & (R_{xy} =_r \text{‘}=\text{‘}) \wedge \; X = Y \;\vee \\ & (R_{xy} =_r \text{‘}\top\text{‘}) \end{aligned}$$

2.2 Operational semantics

The operational semantics of the extended language is defined in [8] based on transition rules, i.e., *resolution*, *constraining*, *inference* and *satisfiability* check, which allow the system to switch from different states. Two points should be considered for specializing this general semantics scheme for a particular language: the predicate *consistent* and the function *infer*.

In the extended language, the *consistent* transition fails if and only if one (relation or integer) variable domain is empty, i.e., no consistent value exists either for a integer variable or for a relation variable.

The transition *infer* defines the propagation of constraints. In our language, the function $infer$ performs an arc-consistency, as well as in CLP(FD) solvers. In fact, we perform arc-consistency on finite domain variables, on relation variables and between the two. Obviously, the constraint $rel(X, R_{xy}, Y)$ is arc consistent if for each value of X (resp. Y), there exists a value for R_{xy} and Y (resp. X) which is consistent with the constraint, and also if for each value of R_{xy} there exists a value for X and Y consistent with the constraint. As concerns implementation details, in the next section we will see the detailed steps performed by the *infer* transition.

2.3 Implementation

In this section, we sketch the implementation scheme of our language. We have implemented the extension proposed on top of the finite domain library of ECL^{i}PS^e [5]. Relation variables can be defined as usual finite domain variables. Operations and constraints can be defined as user-defined operations and constraints. ECL^{i}PS^e, in fact, makes it possible to write user-defined constraints by means of low level predicates for suspension and domain manipulation.

We present now the three basic steps performed by the $infer(C, S)$ transition.

1. Arc-consistency on constraints between relation variables

The first step deals with constraints and operations among relation variables like $comp(R_{xy}, R'_{xy}, R''_{xy})$, $union(R_{xy}, R'_{xy}, R''_{xy})$, $R'_{xy} <_r R''_{xy}$, $R'_{xy} \neq_r R''_{xy}$, $R_{xy} =_r R'_{xy}$. For example, if $R_{xy} :: [<, =]$ and $R'_{xy} :: [<, >]$ then the propagation of the constraint $R_{xy} =_r R'_{xy}$ leads to the instantiation of both variables to the symbol $<$.

2. Domain reduction of relation variables according to integer variables

Given the constraint $rel(X, R_{xy}, Y)$, the propagation reduces relation variable domains according to variable domain values. The check performed is the following: if C is the current store for each $r \in D_{R_{xy}}$ if $C \models X\ r\ Y$, then r is left in the domain of R_{xy}. Otherwise, it is deleted. For example, if we have the following goal:

```
:- X::[1..5], Y::[6..10], R::[<, >], rel(X, R, Y)
```

the symbol $>$ is no longer *supported* by `X` and `Y` domain values. Therefore, the propagation leads to the instantiation $R =_r <$.

3. Arc-consistency on integer variables subject to *least upper bound constraints*

The third step is the usual arc-consistency propagation on domain variables, subject to "least upper bound constraints". For each constraint in S of the form $rel(X, R_{xy}, Y)$, where R_{xy} has been reduced according to the previous step, the system applies the arc-consistency to the constraint $X \ lub(D_{R_{xy}}) \ Y$. If we have the following program:

```
:- rel(X, R, Y), R::[<, >]
```

the constraint added to the set of passive constraints is simply the least upper bound of the two constraints, i.e., $X \neq Y$. In this way, we exploit the propagation mechanism of the usual CLP(FD) solver. This propagation is the same we obtain if we consider the result of the propagation of each constraint separately and perform a union on variable domains.

3 Examples

In this section, we present some examples that show the advantages of the extension proposed. Our extension can be useful in many fields of Artificial Intelligence such as temporal reasoning, scheduling, and planning. In section 3.1, we describe a point based temporal reasoning problem and an extended CLP(FD) program which solves it. In section 3.2, we discuss the application of the extended framework to disjunctive constraints.

3.1 Temporal Reasoning

Let us consider the point algebra introduced by Vilain and Kautz [18]. Let us start with an example where temporal points T_1, T_2 and T_3 range over a finite domain of integers (temporal locations). Suppose our "beginning of the world" is at 7.00 a.m. and we know that T_1 happens between 7.10 and 7.20 a.m., T_2 between 7.14 and 7.28 and T_2 between 7.14 and 7.32. In the example, T_1, T_2 and T_3 represent the events: *John enters the door*, *Mary telephones* and *John meets Helen*. These temporal events are linked by the following relations: T_1 $(after \vee equal)$ T_2, T_2 $(after \vee equal)$ T_3[2]. These relations mean that John enters the door after or while Mary is telephoning and Mary telephones after or at the same time John meets Helen. If the relation between John enters the door (T_1) and meets Helen (T_3) is unknown and the user wants to compute it, he can raise the following query:

```
:- T1 :: [10..20], T2 :: [14..28], T3 :: [14..32],
   R12 :: [after,equal], R23 :: [after,equal],
   rel(T1, R12, T2), rel(T2, R23, T3), rel(T1, R13, T3),
   comp(R12, R23, R13).
```

[2] For a uniform treatment of point-point relation with [18], we replaced the symbol $>$ with `after` and $=$ with `equal`.

The propagation in a classical CLP(FD) is just the reduction of the domain variables to `14..20`, but the relation between T_1 and T_3 cannot be computed starting from domain values. In the extended language, we can compute the above mentioned relation thanks to the operation `comp/3` that performs the composition of the first two variables (see table 2.1). The result will be, therefore, a reduction of the variable domains to `14..20`, as in a pure CLP(FD) language, but also `R13::[after,equal]` with the suspended constraints `comp(R12, R23, R13)`, `rel(T1, R12, T2)`, `rel(T2, R23, T3)` and `rel(T1, R13, T3)`.

Another example, based on the same temporal points, concerns the problem of finding feasible qualitative temporal scenario [15], i.e., an instantiation of relation variables which is consistent with constraints. In the above mentioned program, suppose the relation between T_1 and T_3 is defined by the user to T_1 $after \vee equal$ T_3. The network has three *qualitative* solutions which can be found by the following program:

```
:- rel(T1, R12, T2), rel(T2, R23, T3), rel(T1, R13, T2),
   T1 :: [10..20], T2 :: [14..28], T3 :: [14..32],
   R12 :: [after,equal], R23 :: [after,equal], R13 :: [after,equal],
   comp(R12, R23, R13),
   labeling([R12, R23, R13]).
```

where the labeling predicate instantiates relation variables to a consistent value. The solutions corresponding to feasible scenarios are:

- `T1, T2, T3::[14..20], R12, R13, R23 = equal`;
- `T1::[15..20], T2, T3::[14..19], R12, R13 = after, R23 = equal`;
- `T1, T2::[15..20], T3::[14..19], R12 = equal, R23, R13 = after`;

The expressive power of CLP(FD) languages is thus increased by the extension proposed. In [11] we have presented the application of this extension to the Interval Algebra [1] and to the STP framework [2].

3.2 Disjunctive Constraints

In this section, we show how to handle disjunction in our framework. This example is motivated by the need to achieve a more global pruning for disjunctive constraint than the one offered by disjunctive clauses in CLP(FD) solvers. We will see two problems where disjunctive constraints arise: *scheduling* and *packing* problems. An extended paper about scheduling problems solved with this approach can be found in [12].

3.2.1 Scheduling Applications

Consider for instance two tasks a and b competing for the same single-capacity resource. Of course the two tasks cannot be executed at the same time, i.e., they cannot overlap. Given S_a and S_b the starting points of the tasks and D_a and D_b their duration, the `no-overlap` constraint can be expressed as:

```
no_overlap(Si,Sj,Di,Dj):- Si + Di #<=Sj.
no_overlap(Si,Sj,Di,Dj):- Sj + Dj #<=Si.
```

where the symbol `#<=` represent the constraint less or equal in the ECL^{i}PS^e [5] syntax. The main problem with this formulation comes from the fact that no pruning is performed in order to reduce the search space. Constraints are only used as choices.

In our framework, we can reason on relations and we can represent the same problem as:

```
no_overlap(Si,Sj,Di,Dj):-
   R1 ::[<=, >=], R2 ::[<=, >=], R1 ## R2,
   rel(Si + Di, R1,Sj), rel(Sj + Dj, R2,Si).
```

where symbol `##` is the disequality constraint between relation variables. This representation of the problem can be explained as follows: we have two relation variables between the starting and ending points of the tasks. These relations can assume only two values, i.e., $\mathtt{R}_1 =_r \leq$ and $\mathtt{R}_2 =_r \geq$ if task i is scheduled before task j. Viceversa, if task j is scheduled before task i $\mathtt{R}_2 =_r \leq$ and $\mathtt{R}_1 =_r \geq$. Therefore, the constraint $\mathtt{R}_1$ `##` $\mathtt{R}_2$ states these two possible configurations. We have to impose these constraints between starting and ending points of all not overlapping tasks.

A more global propagation can be performed by computing the (qualitative) transitive closure of the network. This closure can be realized by means of the `comp/3` operator thus composing the relation variables. The idea is to find a consistent ordering between tasks competing for the same resource by imposing new *precedence* constraints, as suggested in [14]. The first step is to impose precedence relations (i.e., ground relation variables), then collect all the tasks that compete for the same resource in a list and impose *no_overlap* constraints. Finally by working on a list of the kind (Task$_i$, Rel, Task$_j$) we can compute the transitive closure in the following way.

```
transitive([(Ti, Rij, Tj) | RestTasks]):-
   findall((Ti, Rik, Tk), member((Ti, Rik, Tk), RestTasks), List),
   closure((Ti, Rij, Tj), RestTasks, List),
   transitive(RestTasks).

closure(_, _, []).
closure((Ti, Rij, Tj), RestTasks, [(Ti, Rik, Tk)|RL]):-
   member((Tj, Rjk, Tk), RestTasks), !,
   comp(Rij, Rjk, Rik),
   closure ((Ti, Rij, Tj), RestTasks, RL).
```

The complexity of the transitive closure propagation can be computed as follows: if we have n tasks competing for a resource, we have a number of compositions equal to the combinations of n elements in k-tuples with $k = 3$ (we compose T_i, T_j and T_k independently from the order). Therefore the number of compositions is proportional to n^3. In fact:

$$\frac{n(n-1)\ldots(n-k+1)}{k!} = \frac{n(n-1)\ldots(n-2)}{6}$$

This propagation is quite expensive but in many cases very powerful since constraint `rel/3` propagates from values of tasks starting and ending points to relations and viceversa, while the transitive closure removes inconsistent relations. Consider the following simple example: we have five tasks , named respectively a_1, a_2, b_1, b_2 and b_3, belonging to two jobs a and b competing for the same resource. Tasks of the same job are linked by precedence relations, i.e., $a_1 \leq a_2$, $b_1 \leq b_2$, $b_2 \leq b_3$. Consider tasks a_1 and b_3: they have a starting time S_{a_1} :: [45..85], S_{b_3} :: [7..30] and a duration $d_{a_1} = 20$, $d_{b_3} = 10$. Obviously, all the tasks competing for the same resource are linked by a relation variable R whose domain is $[\leq, \geq]$. The starting and ending points determine an ordering for tasks a_1 and b_3 where b_3 precedes a_1. This propagation can be achieved by means of the constraint $rel/3$ which instantiates the relation between the two tasks, as shown in figure 1.a where all the missing arcs represents relations whose domain is $[\leq, \geq]$. At this point the transitive closure of the network propagates precedence relations and this new ordering instantiates all the other relations as described in figure 1.b. A standard CLP(FD) approach does not recognize that the ordering among all the tasks is defined by the consideration that b_3 comes before a_1 and tries the assignment of an ordering in a *standard backtracking* way. We perform a *forward checking* propagation since as soon as a relation is known all the other relations can be propagated accordingly. In [14] the authors suggest some heuristics which can be effectively used in order to further reduce the search space. We are currently testing these heuristics on the prototype.

Based on these consideration, we have checked the number of nodes generated on different scheduling benchmarks starting from simple problems to more difficult problems. The nodes considered are nodes generated by choosing an ordering between two tasks. The average number of nodes generated with the relation variable propagation is less than the 20% of nodes generated in the standard CLP(FD) approach. For the set of problems, we have tested that the transitive closure propagation determines after 3 attempts on average (`indomain` on relation variables) all the consistent orderings over a set of variables sharing the same resource. We have considered problems with 5, 7, 10, 30 tasks sharing the same resource. The orderings are determined after 3 or 4 attempts (labeling of relation variables) for each set of tasks sharing the same resource. The number depends also from the number of precedence relations among the tasks. However, for an increasing number of

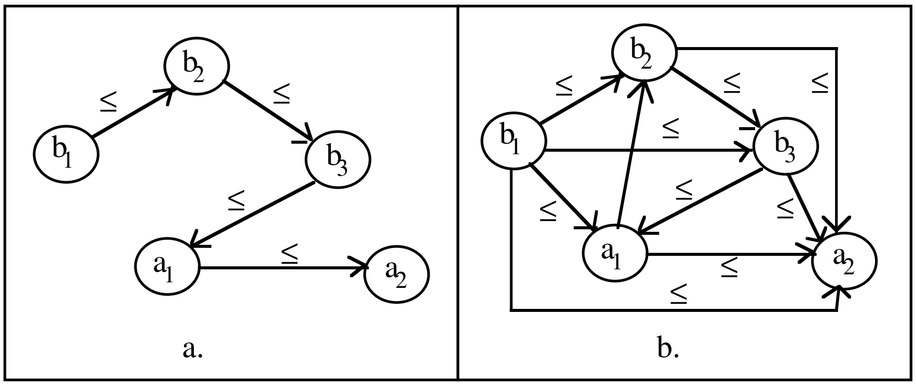

Figure 1: Example of ordering relations

tasks the transitive closure is quite hard. Therefore, we are currently comparing the above mentioned approach with the transitive closure against the same approach without the transitive closure with some heuristics on the relation variable ordering and value ordering. We have to say also that all these constraints have been implemented on top of the ECL^{i}PS^e finite domain constraint solver. A more efficient low level implementation of the constraints could greatly increase the performances.

3.2.2 Packing Applications

Another problem we consider is the packing problem where we have to pack small squares in a big one. As shown in [17] the structure of the program solving the problem is the following:

```
packSquares(Xs,Ys):-
   generateSquares(Xs,Ys,Sizes,Size),
   stateNoOverlap(Xs,Ys,Sizes),
   stateCapacity(Xs,Sizes,Size), stateCapacity(Ys,Sizes,Size),
   labeling(Xs), labeling(Ys).
```

Basically, the `generateSquares` predicate defined the domains of the variables corresponding to square coordinates. We are here interested in the `stateNoOverlap` predicate stating no-overlap constraints on two dimensions. This constraint can be expressed as:

$$\{(X1 + S1 \leq X2) \cup (X2 + S2 \leq X1) \cup (Y1 + S1 \leq Y2) \cup (Y2 + S2 \leq Y1)\}$$

This disjunction expresses that the first square (with bottom-left corner coordinates $(X1, Y1)$ and edge $S1$) must be on the left, on the right, below or above the second square (with bottom-left corner coordinates $(X2, Y2)$ and edge $S2$).

We can express this disjunction with relation variables on “two levels”.

```
no_overlap(X_i,Y_i X_j, Y_j,S_i,S_j):-
    R_ijX ::[<=, >=], R_jiX ::[<=, >=],
    R_ijY ::[<=, >=], R_jiY ::[<=, >=],
    rel(X_i + S_i, R_ijX,X_j), rel(X_j + S_j, R_jiX,X_i),
    rel(Y_i + S_i, R_ijY,Y_j), rel(Y_j + S_j, R_jiY,Y_i),
    MetaR_X::[=, ≠], MetaR_Y::[=, ≠], MetaR_X## MetaR_Y,
    rel(R_ijX, MetaR_X, R_jiX), rel(R_ijY, MetaR_Y, R_jiY).
```

The relation variables link starting and ending points of squares on X and Y axes. However, we have to impose the non overlapping constraint on one direction if and only if on the other direction they overlap. This is achieved by meta relation variables.

4 Related Works

A previous approach to the idea of reasoning on relations has been presented in [10] by means of a meta architecture. The object level is a finite domain constraint solver on integers, while the meta level is a finite domain constraint solver on relation symbols. The meta level reasons on constraints of the underlying level. The main problem with this kind of approach is that the user can not specify at language level which relations should be reified and how to propagate constraints on them. In fact, the user writes CLP(FD) programs and the solver performs a more powerful propagation. In the extended CLP(FD) framework here proposed, the user has greater flexibility and the language provides greater expressive power.

A related work is that by T. Frühwirth [6] on Constraint Handling Rules (CHRs) as an extension of Constraint Logic Programming languages. The author basically uses guarded rules in order to perform operations such as composition and intersection on constraints. CLP programs can be extended by CHRs that operationally describe the behavior (in terms of the propagation to be performed) of the constraint solver when it encounters a constraint. We do not define operational rules (that are embedded in the constraint solver) but we have a constraint that declaratively links integer and relation variables, and constraints among relation variables in the CLP(FD) language. However, CHRs can be a suitable tool for implementing our extension.

As far as disjunction in CLP is concerned, we have to mention the *constructive disjunction* [17]. The idea is to remove from variable domains those values which are not supported by any disjuncts of the disjunction. This is a very powerful way of coping with disjunction which is not alternative to our method. In fact, it can be integrated in our language in order to further increase the expressive power and the efficiency of the solver. Our extension is mainly devoted to the qualitative reasoning. One example can be found by considering the following constraint store:

$\{X \leq Y, Y \leq Z, (X \geq Z \cup X \leq Y\}$, where variables X, Y and Z are defined on a domain [1..10]. Even by considering the constructive disjunction, i.e., propagating each disjunct separately and considering the union of the resulting domains, no inference can be done on constraints in the disjunction. In our framework, even thought we do not reduce variable domains, we choose the constraint $X \leq Y$ since the only way to guarantee the consistency of the constraint $\geq$ is to have three equal values which is subsumed by $X \leq Y$.

Another related approach, as far as disjunction is concerned, is the cardinality operator by Van Hentenryck and Deville [16]. The cardinality operator is a non primitive constraint for *inferring simple constraints from difficult ones.* The syntax of the cardinality operator is the following:

$\#(l, u, [c_1, c_2, \ldots, c_n])$.

Declaratively, it means that the number of satisfied constraints in the set $[c_1, c_2, \ldots, c_n]$ is greater than l and smaller that u. The above mentioned problem of disjunction is thus solved in the following way:

$\#(1, 1, [T_{j1} < T_{i2}, T_{i1} > T_{j2}])$

The cardinality operator can be implemented on top of the so called *ask and tell* languages that support the *entailment* of constraints [13]. Our approach can be built on top of any CLP(FD) language without the entailment, i.e., *tell* languages.

Another way of coping with disjunction in CLP is by means of the *cumulative* constraint [3]. This constraint is often used when expressing capacity constraints.

Its syntax is `cumulative(Start, Duration, Resource, Max)` where `Start` represents a list of tasks starting times, `Duration` a list of durations, `Resource` a list of resources used by different tasks, and `Max` the maximum capacity of a set of resources. The first three parameters are lists of domain variables. The semantics of the constraint is that the sum of the resources used by single tasks, in each time point, cannot be greater than the threshold `Max`. In order to express the $before \vee after$ disjunction we can write:

cumulative([S_i, S_j], [D_i, D_j], [1, 1], 1).

In this way, we avoid the two tasks to overlap. However, this constraint cannot be used in order to represent more complex qualitative constraints such as $Task_i$ $before \vee$ *overlapped_by* $Task_j$ [1]. We can express this kind of constraints in our framework, by properly combining relations and operations between meta variables.

As far as the transitive closure of the network is concerned (as described in section 3.2), we mention [9]. In this paper the authors propose an efficient and incremental propagation algorithm for (linear) *Two-Variables-Per-Inequality* (TVPI) constraints. This algorithm can be embedded in a constraint solver thus increasing its performance on this kind of constraints. The transitive closure of the network can be computed also in our frameworks with some differences with respect to [9]:

- we perform a qualitative closure on binary constraints while in [9] they perform a more complex closure that introduces more constraints containing operators such as sum, difference, and so on.

- our transitive closure is guided by the user that imposes, at language level, the composition operator among relation variables. On the other hand, in [9] the transitive closure is embedded in the constraint solver.

This paper, even thought presenting these differences, suggests us an interesting future direction for our framework for treating operators. For example, given the constraint store $\{X < Y + 5, Y < Z\}$ we would like to deduce the constraint $X < Z + 4$. A possibility could be to represent a relation R as a triple (Rel, Val, Op) where Rel is an ordering relation, Val is a value and Op an operator. The meaning of the relation $rel(X, R, Y)$ is that the application of the operator Op to the variable X and the value Val is in relation Rel with Y.

5 Conclusions and Future Works

We have presented an extension of the CLP framework for reasoning on constraints. We have a two sorted first order language on integers and relations with a clear declarative and operational semantics. We provided some examples that show the increased expressive power and the effectiveness of the approach in dealing with disjunction. We have implemented the extension proposed on top of the finite domain library of ECL^{i}PS^e. Obviously, the efficiency of the resulting solver cannot be equal to a special purpose solver for the domain of relations.

Future works are aimed to:

- define heuristics on relation variables and value ordering during the labeling step, as those suggested in [14];
- extend the language in order to cope with operators as briefly suggested in section 4;
- apply the extended language to real life application in the field of scheduling and planning. In particular, we will solve a real job shop scheduling application in the railway traffic regulation field.

Acknowledgments

We would like to thank Pascal Van Hentenryck for useful discussions during our visit at Brown University.

References

[1] J.F.Allen. Maintaining Knowledge About Temporal Intervals. in *Communications of the ACM*, 26:832–843, 1983.

[2] R.Dechter, I.Meiri, J.Pearl. Temporal Constraint Networks. in *Artificial Intelligence*, 49:61–95, 1991.

[3] M.Dincbas, P.Van Hentenryck, M.Simonis, A.Aggoun, T.Graf, F.Berthier. The Constraint Logic Programming Language CHIP. in *Proceedings of the International Conference on Fifth Generation Computer System (FGCS88)*, pp.693–702,1988.

[4] M.Dincbas, P.Van Hentenryck, M.Simonis, Solving Large Combinatorial Problems in Logic Programming. in *Journal of Logic Programming*, 8 (1-2) pp.75–93.

[5] ECL^{i}PS^e User Manual Release 3.3, ECRC 1992.

[6] T. Frühwirth. Temporal reasoning with constraint handling rules *Tech. Report* ECRC-94-05, ECRC, 1994, Germany.

[7] J.Jaffar , J.L.Lassez. Constraint Logic Programming. in *Proceedings of the Conference on Principle of Programming Languages*, Munich 1987.

[8] J.Jaffar, M.J.Maher. Constraint Logic Programming: a Survey. in *Journal of Logic Programming on 10 years of Logic Programming*, 1994.

[9] J.Jaffar, M.J. Maher, P.J. Stuckey, R.H.C. Yap. Beyond Finite Domains. in *Second International Workshop on Principles and Practice of Constraint Programming*, PPCP94, 1994.

[10] E.Lamma, P.Mello, M.Milano. Combining Solvers in a Meta Constraint Logic Programming Architecture in Proceedings of the First International Workshop *Frontiers of Combining Systems*, "Applied Logic", Kluwer, 1996.

[11] E.Lamma, P.Mello, M.Milano. Temporal Reasoning in a Meta Constraint Logic Programming Architecture. will appear in *Annals of Mathematics and Artificial Intelligence* special issue on *Temporal Representation and Reasoning*, Baltzer.

[12] E.Lamma, P.Mello, M.Milano. Consistent Orderings for Constraint Satisfaction Scheduling. DEIS Technical Report 97-001, 1997.

[13] V.A. Saraswat. Concurrent Constraint Logic Programming *PhD Thesis*, Carnegie-Mellon University, 1989.

[14] S.F.Smith, C.Cheng. Slack-based Heuristics for Constraint Satisfaction Scheduling. in *Proceedings of AAAI93*, 1993.

[15] P.Van Beek. Reasoning on Qualitative Temporal Information. in *Artificial Intelligence*, 58:297–326, 1992.

[16] P.Van Hentenryck, Y.Deville. The Cardinality Operator: a new Logical Connective for Constraint Logic Programming in *Constraint Logic Programming: selected research*, F.Benhamou, A.Colmerauer eds., MIT Press,1993.

[17] P.Van Hentenryck, V. Saraswat, Y. Deville. Design, Implementation and Evaluation of the Constraint Language cc(FD). Technical Report CS-93-02, Brown University, 1993.

[18] M.B.Vilain, H.Kautz, P. Van Beek. Constraint Propagation Algorithms for Temporal Reasoning: A Revised Report. in *Readings in Qualitative Reasoning about Physical Systems*, D.S.Weld, J.De Kleer, Morgan Kaufmann, 1990, pp.373–381.