ELSEVIER

The Journal of Logic Programming 38 (1999) 93–110

THE JOURNAL OF LOGIC PROGRAMMING

Technical Note

Reasoning on constraints in CLP(FD)

Evelina Lamma [a,*], Michela Milano [a,1], Paola Mello [b,2]

[a] *Dipartimento di Elettronica, Informatica e Sistemistica, Università di Bologna, Viale Risorgimento 2, 40136 Bologna, Italy*

[b] *Dipartimento di Ingegneria, Università di Ferrara, Via Saragat 1, 44100 Ferrara, Italy*

Received 30 July 1997; received in revised form 21 January 1998; accepted 28 May 1998

Abstract

Constraint Logic Programming solvers on finite domains (CLP(FD) solvers) 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 over finite domains of objects (e.g., integers) and *relation variables* range over finite domains of relation symbols. We define operations and constraints on the two sorts of variables and one constraint linking the two. We first present the extension as a general framework, then we propose two specializations on finite domains of integers and of sets. Programming examples are given, showing the advantages of the extension proposed from both a knowledge representation and an operational viewpoint.

Keywords: Constraint logic programming extensions; Qualitative reasoning; Disjunctive constraints

1. Introduction

Constraint Logic Programming (CLP) [19] is a programming paradigm combining the advantages of Logic Programming and the efficiency of constraint solving. In this paper, we focus on CLP over finite domains, CLP(FD). CLP(FD) variables range over a finite domain of objects (e.g., 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

* Corresponding author. Tel.: +39 51 643033; fax: +39 51 6443073; e-mail: elamma@deis.unibo.it.

[1] E-mail: mmilano@deis.unibo.it.

[2] E-mail: pmello@ing.unife.it.

0743-1066/99/$ – see front matter
PII: S0743-1066(98)10016-X

coloring, planning, scheduling, sequencing and assignment problems, to name a few [9]. Many of these problems require some form of *qualitative* reasoning. For instance, many applications 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) [8], or *qualitative*, e.g., they link the relative positions of points and intervals [1,29].

While the quantitative reasoning can be easily handled by 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 $T_1 \leqslant T_2$, $T_2 < T_3$. CLP(FD) constraint propagation, arc-consistency [25], reduces the size of variable domains, thus leading to T_1::[1..9], T_2::[1..9] and T_3::[2..10], but it does not infer the relation between T_1 and T_3. This happens because CLP(FD) uses constraints in order to restrict variable domains, but it never does reason on constraint as domain elements, thus inferring qualitative relations and 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, pose constraints on them and change relations during the computation. 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 in one propagation step since $>$ is not contained in the domain of R_{XY}.

Based on this intuition, we present a general language extension which can be applied, in theory, to any finite domain constraint language and enables to reason on constraints. We present two specializations of the extension proposed on finite domains of integers and on finite domains of sets (as defined in the Conjunto language [14]). For instance, in the first specialization on integers, we have the usual operations $(+,*)$ and constraints $(<, >, \leqslant, \geqslant, \neq, =)$, while on sets, we have the usual operations $(\cup, \cap, /)$ and constraints like set inclusion, disjointness and equality $(\subseteq, \supseteq, \neq^0, =)$. In both cases, on relations we define composition, union, least upper bound operations, and equality, inequality and disequality constraints.

Beside the increased expressive power in knowledge representation when reasoning on relations, we deal with disjunctive constraints more effectively than with pure CLP(FD) languages. However, current CLP(FD) commercial solvers such as CHIP [10] or ILOG [18], making use of global symbolic constraints [4], outperform our relation based approach since they use sophisticated propagation techniques tailored on the specific constraint. The purpose of this paper is to show the effectiveness

and the generality of the approach when compared with pure CLP(FD) solvers making use of arc-consistency propagation techniques, and not to outperform the state of the art commercial CLP solvers.

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 general language, two specializations on integers and on sets, and give some implementation details. Programming examples are given in Section 3, showing the advantages of the extension proposed. Related works are presented in Section 4. Conclusion and future work is given in Section 5.

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 objects (e.g., integers) and relations. We define operations and constraints on finite terms (they are the usual operations and constraints of CLP(FD) solvers) and operations and constraints on relations.

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, finite domain objects and usual expressions, and R is one of the constraints defined on the domain of discourse (e.g., for integers we have the usual relations: $>, \geqslant, <, \leqslant, =, \neq$). Intuitively, we want to extend the 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 relation between *finite domain variables* X and Y.

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, whose carrier is *FD*, represents a finite domain sort, and S_2, whose carrier is *Rel*, the relation sort.

For finite domains, unary constraints link a variable to range over a finite domain of objects belonging to *FD*, $X::[a_1,\ldots,a_n]$. The meaning of this constraint is

$$X::[a_1,\ldots,a_n] \leftrightarrow_f X = a_1 \vee \cdots \vee X = a_n.$$

As usual, binary constraints and operations can be defined on finite domain variables.

For the relation sort, unary constraints link each relation variable to range over a finite domain of relations belonging to *Rel*, $R::[r_1,\ldots,r_n]$. 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, we always have equality and inequality constraints $\{=_r, \neq_r\}$ which can be defined as usual. [3] In addition, we assume that a partial order relation

[3] The letter r in relation symbols refers to the *Rel* sort. However, with abuse of notation, we omit this letter in the programming examples since the propagation performed for $=_r$ and $\neq_r$ is the same as for the standard CLP(FD) constraints $=$ and $\neq$.

on the *Rel* sort exists. We introduce it in the language as a constraint through the symbols $\{\leqslant_r, \geqslant_r\}$ in the following way:

$$R_1 \leqslant_r R_2 \leftrightarrow \forall x, y\ R_1(x,y) \rightarrow R_2(x,y).$$

Operations on relations are *composition*, *union* and *least upper bound* (for the latter we assume that the operation is well defined, i.e., the least upper bound of any couple of elements exists). We can define them as three predicates: *comp*/3, *union*/3 and *lub*/3. The signature of the predicate *comp* is $\langle Rel, Rel, Rel\rangle$ and its declarative meaning is

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

where $\otimes$ is defined by a transitivity table providing the composition between primitive relations.

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

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

The signature of the *lub* operation is $\langle Rel, Rel, Rel\rangle$ and its declarative meaning is

$$lub(R_1, R_2, R_3) \leftrightarrow R_3 =_r lub\_lattice(R_1, R_2),$$

where *lub_lattice* is defined by the lattice defining the partial order on the *Rel* sort.

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

$$\forall r \in Rel\ (rel(X, R_{xy}, Y) \leftrightarrow (R_{xy} =_r r) \wedge X\ r\ Y).$$

This constraint substitutes the usual CLP(FD) constraints when some form of reasoning on relations is required. It is worth noting that the proposed extension can be expressed in the classical first order framework.

2.2. Specialization on integers

In this section, we present the specialization of the proposed framework on finite domains of integers. We have to define the *Rel* sort, *Rel_int*, and operations and constraints on *Rel_int*. The resulting language is a two sorted language: the carriers of the two sorts are integer numbers, i.e., $FD = Z$, and the relation set $Rel\_int = \{<, \leqslant, >, \geqslant, \neq, =, \top\}$.

On integers, we have the usual constraints $>, \geqslant, <, \leqslant, =, \neq$ and operations are addition (+) and multiplication (*).

As concerns the relation sort, unary constraints link each relation variable to range over a finite domain of relations $R::[r_1, \ldots, r_n]$, where $r_1, \ldots, r_n$ belong to *Rel_int*. On the relation sort, equality and inequality constraints can be defined as usual.

The partial order relation on *Rel_int* is defined by the lattice depicted in Fig. 1, where the top is the constraint $\top$, i.e., the most relaxed constraint, which is always

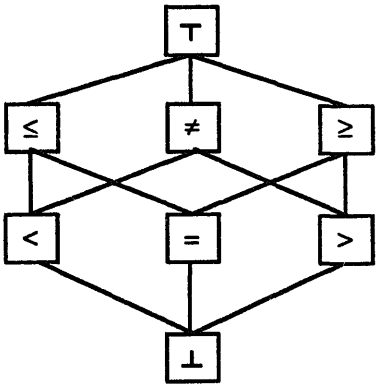

Fig. 1. Lattice on the *Rel_int* sort.

true for any pair of variables, and the bottom is the inconsistent relation $\perp$ which is not included in the carrier *Rel_int*. [4]

As a consequence, the *lub_lattice* operation is defined on the lattice of Fig. 1. For example, if $R_1::[\leqslant,\neq]$, $R_2::[\leqslant,\neq]$, $R_{lub}::[\leqslant,\neq,\top]$, then $lub(R_1,R_2,R_{lub})$ holds.

The operation $\otimes$ is defined by a transitivity table described in [29] and reported in Table 1 for the sake of completeness. For example, if $R_1::[<,=]$ and $R_2::[<,=]$, $R_3::[<,=]$, then $comp(R_1,R_2,R_3)$ holds.

Union operation does not need to be specialized because it is general for any kind of relation domain. In the integer specialization, for example, if $R_1::[<,=]$, $R_2::[<,>]$ and $R_3::[<,=,>]$, then $union(R_1,R_2,R_3)$ holds.

The mixed computation domain constraint *rel* links two integer variables and one relation variable. In this case, it can be specialized as follows:

$$
\begin{aligned}
rel(X,R_{xy},Y) \leftrightarrow\ & (R_{xy} =_r \text{`}<\text{`}) \wedge X < Y \vee \\
& (R_{xy} =_r \text{`}\leqslant\text{`}) \wedge X \leqslant Y \vee \\
& (R_{xy} =_r \text{`}>\text{`}) \wedge X > Y \vee \\
& (R_{xy} =_r \text{`}\geqslant\text{`}) \wedge X \geqslant Y \vee \\
& (R_{xy} =_r \text{`}\neq\text{`}) \wedge X \neq Y \vee \\
& (R_{xy} =_r \text{`}=\text{`}) \wedge X = Y \vee \\
& (R_{xy} =_r \text{`}\top\text{`}).
\end{aligned}
$$

2.3. Specialization on sets

In this section, we present the specialization of the framework proposed on finite domains of sets as introduced in the Conjunto language [14] (provided as a library of *ECL^{i}PS^e*). A number of works have been proposed on constraint programming on sets, such as, for example, {*log*} [11] and CLPS [23] which are more expressive than

[4] The symbol $\perp$ represents the inconsistent relation. It is always false for any pair of variables. In the following, we omit the inconsistent relation in relation variable domains and we fail if a relation variable domain becomes empty. However, the symbol $\perp$ can be used in the language for explicitly handling failures and recovering from them.

Table 1
Composition of constraints on integers

$\otimes$	$<$	$\leqslant$	$>$	$\geqslant$	$=$	$\neq$	$\top$
$<$	$<$	$<$	$\top$	$\top$	$<$	$\top$	$\top$
$\leqslant$	$<$	$\leqslant$	$\top$	$\top$	$\leqslant$	$\top$	$\top$
$>$	$\top$	$\top$	$>$	$>$	$>$	$\top$	$\top$
$\geqslant$	$\top$	$\top$	$>$	$\geqslant$	$\geqslant$	$\top$	$\top$
$=$	$<$	$\leqslant$	$>$	$\geqslant$	$=$	$\neq$	$\top$
$\neq$	$\top$	$\top$	$\top$	$\top$	$\neq$	$\top$	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$

Conjunto since they allow to treat non-ground sets. We adopt Conjunto because it works on finite domain of sets.

The aim of this section is to show the capability of our framework to be specialized on other domains than integers. The Conjunto language is mainly targeted to express and solve combinatorial problems on sets on which it has been proved to be extremely effective, guaranteeing good expressiveness and considerably speed up in execution time over traditional logic programming solutions.

We have to define the *Rel* sort, *Rel_set*, and operations and constraints on *Rel_set*. The resulting language is a two sorted language: the carriers of the two sorts are $FD = FD(HU)$ which is the set of finite (ground) sets in 2^{HU}, HU is the Herbrand Universe, and the relation set $Rel\_set = \{\subseteq, \supseteq, \neq^0, =, \top\}$, where $\neq^0$ represents set disjointness.

On sets, we have unary constraints defined by specifying the greatest lower bound, *glb*, and the least upper bound, *lub*, i.e., $X::glb..lub$, binary constraints $\subseteq, \supseteq, \neq^0, =$, and operations are intersection, union and complementary difference.

As concerns the relation sort, unary constraints link each relation variable to range over a finite domain of relations $R::[r_1, \ldots, r_n]$, where $r_1, \ldots, r_n$ belong to *Rel_set*. On the relation sort, equality and inequality constraints can be defined as usual.

The partial order relation on *Rel_set* is defined by the lattice reported in Fig. 2 as well as the *lub_lattice* operation. As for integers, in Fig. 2, the top is the constraint $\top$, i.e., the most relaxed constraint, which is always true for any pair of variables, and the bottom is the inconsistent relation $\bot$ which is not included in the carrier *Rel_set*. The operation $\otimes$ is defined by a transitivity table (see Table 2).

The mixed computation domain constraint *rel* links two set variables and one relation variable. In this case it can be specialized as follows.

$$
\begin{aligned}
rel(X, R_{xy}, Y) \leftrightarrow\ & (R_{xy} =_r \text{`}\subseteq\text{'} \wedge X \subseteq Y) \vee \\
& (R_{xy} =_r \text{`}\supseteq\text{'} \wedge X \supseteq Y) \vee \\
& (R_{xy} =_r \text{`}\neq^0\text{'} \wedge X \neq^0 Y) \vee \\
& (R_{xy} =_r \text{`}=\text{'} \wedge X = Y) \vee \\
& (R_{xy} =_r \text{`}\top\text{'}).
\end{aligned}
$$

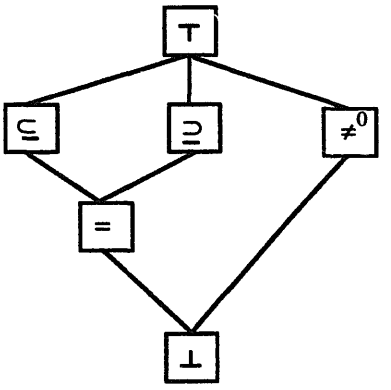

Fig. 2. Lattice on the *Rel_set* sort.

Table 2
Composition of constraints on sets

⊗	⊆	⊇	=	$\neq^0$	⊤
⊆	⊆	⊤	⊆	$\neq^0$	⊤
⊇	⊤	⊇	⊇	⊤	⊤
=	⊆	⊇	=	$\neq^0$	⊤
$\neq^0$	⊤	$\neq^0$	$\neq^0$	⊤	⊤
⊤	⊤	⊤	⊤	⊤	⊤

2.4. Operational semantics

The operational semantics of the extended language can be defined by specializing the operational semantics presented in [19] based on transition rules, i.e., *resolution*, *constraining*, *inference* and *satisfiability* check, which allows the system to switch from different states. States are represented by tuples $\langle A, C, S\rangle$ where A is a multiset of atoms and constraints, C is the constraint store of active constraints, i.e., constraints that are *awake*, and S is the constraint store of passive constraints, i.e., constraints that are *asleep*. The failure of the system is represented by a state called *fail*. The transition rules can be found in [19], and reported in Fig. 3 for the sake of completeness. 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 proposed, the *consistent* transition fails if and only if one (relation or finite domain) variable domain is empty, i.e., no consistent value exists either for a finite domain 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

Resolution: $\langle A \cup a, C, S\rangle \rightarrow_r \langle A \cup B, C, S \cup (a = h)\rangle$
where a is an atom selected by the computation rule
$h \leftarrow B$ is a rule of the program P renamed to new variables
h and a have the same predicate symbol
fails otherwise.

Constraining: $\langle A \cup c, C, S\rangle \rightarrow_c \langle A, C, S \cup c\rangle$
where c is a constraint selected by the computation rule

Infer: $\langle A, C, S\rangle \rightarrow_i \langle A, C', S'\rangle$.
where $(C', S') = \mathit{infer}(C, S)$

Satisfiability: $\langle A, C, S\rangle \rightarrow_s \langle A, C, S\rangle$
if $\mathit{consistent}(C)$,
fails otherwise.

Fig. 3. CLP operational semantics.

consistent with the constraint. As concerns implementation details, in the next section we will see the detailed steps performed by the *infer* transition.

2.5. Implementation

In this section, we sketch the implementation scheme of our language. We have implemented the two specializations of the proposed extension on integers and on sets respectively on top of the finite domain library and of the Conjunto library of ECL^{i}PS^e [12]. In both cases relation variables have been defined as usual finite domain variables. Operations and constraints have been defined as ECL^{i}PS^e user-defined operations and constraints, by means of low level predicates for suspension and domain manipulation.

It is worth noting that implementation of a new specialization is modular since just by changing the transitivity table and the definition of the lattice induced by the partial order relation, we can tailor the language on whatever relation domain.

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 $\mathit{comp}(R_{xy}, R'_{xy}, R''_{xy})$, $\mathit{union}(R_{xy}, R'_{xy}, R''_{xy})$, $\mathit{lub}(R_{xy}, R'_{xy}, R''_{xy})$, $R'_{xy} \leqslant_r R''_{xy}$, $R'_{xy} \neq_r R''_{xy}$, $R_{xy} =_r R'_{xy}$. An arc-consistency is performed on relation variables. For example, in the integer specialization, given the query:

$$:- \texttt{R1::[<, =]}, \ \texttt{R2::[<, >]}, \ \texttt{R1} = \texttt{R2}.$$

The arc-consistency on relation variables instantiates both variables R1 and R2 to the symbol <.

2. *Domain reduction of relation variables according to finite domain variables*: Given the constraint $\mathit{rel}(X, R_{xy}, Y)$, the propagation reduces relation variable domains according to domain values of finite domain variables. The check performed is the following: if C is the current store, for each $r \in D_{R_{xy}}$ if $C \cup (X\ r\ Y)$ does not produce failure, then r is left in the domain of R_{xy}. Otherwise, it is deleted. For example, in the integer specialization, 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 instantiates R to the value <. This behavior is achieved through a propagation based on the cardinality of the intersection of variable domains and on domain upper and lower bounds. For more details on this propagation, see [26].

3. *Arc-consistency on finite domain 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 query:

```
:- 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 underlying CLP(FD) solver. We would obtain the same propagation if we consider the result of the propagation of each constraint separately and perform the union of the resulting variable domains.

3. Examples

In this section, we present some examples that show the advantages of the extension proposed from both a knowledge representation and an operational viewpoint. In Section 3.1, we describe a point based temporal reasoning problem and an extended CLP(FD) program on integers which solves it. In Section 3.2, we discuss the application of the extended framework to disjunctive constraints. We present an example of the integer specialization and one example of the set specialization.

3.1. Temporal reasoning

Let us consider the Point Algebra introduced by Vilain and Kautz [29]. 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 a.m. and T_3 between 7.14 and 7.32 a.m. 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* ∨ *equal*) T_2, T_2 (*after* ∨ *equal*) T_3. [5] 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],
```

[5] For a uniform treatment of point–point relation with [29], in the current example, we replace the symbol > with `after`, = with `equal`, and < with `before`.

```
rel(T1, R12, T2), rel(T2, R23, T3), rel(T1, R13, T3),
comp(R12, R23, R13).
```

The propagation in pure 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 specialized over integers, we can compute the above mentioned relation thanks to the operation comp/3 that performs the composition of the first two variables (see Table 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 [2], 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 as T_1 *before* $\vee$ *equal* T_3. Obviously, the network is inconsistent if we consider the left side of the disjunctions, i.e., T1 after T2, T2 after T3, T1 before T3. However, there is a solution if we consider the relations T1 equal T2, T2 equal T3, T3 equal T1.

This solution can be found if we assign values to relation variables by means of a labeling step:

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

where the labeling clause instantiates each variable to a value of its domain. The extended CLP(FD) solver on integers gives the solution:

```
T1, T2, T3::[14..20], R12, R13, R23 = equal
```

corresponding to the only feasible scenario where the temporal events represented by T_1, T_2, T_3 happen at the same time. They can happen in all temporal locations between 14 and 20.

As another example, consider the operation *union*, which is a very effective operator from a knowledge representation viewpoint since it embeds the concept of disjunction. We can state complex constraints such as: the relation between temporal events T_1 and T_2 can be equal to the relation between events T_3 and T_4 or between events T_5 and T_6. The resulting code is:

```
:-rel(T1, R12, T2), rel(T3, R34, T4), rel(T5, R56, T6),
  R34::[after], R56::[before],
  union(R34, R56, R12).
```

where variable R12 is instantiated to the domain [after, before].

The expressive power of CLP(FD) languages is thus increased by the extension proposed. In [22] we have presented the application of this extension to the Interval Algebra [1] and to the Simple Temporal Problem (STP) framework [8].

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 constraints than the one offered by disjunctive clauses in pure CLP(FD) solvers. We

will see two applications of our extended language: the first concerns *scheduling* problems, while the second *bin packing* problems.

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 directly in a CLP(FD) language by using two disjunctive clauses:

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

where the symbol `#<=` represents the constraint less or equal in the ECL^{i}PS^e [12] 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 a conjunction of constraints on relation variables:

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

where symbol `##` is the inequality constraint between relation variables. This representation of the problem can be explained as follows: we have two relation variables between the starting and end points of the tasks. [6] Only two configurations of values are allowed:

$R_1 =_r \leqslant$ and $R_2 =_r \geqslant$ if task i is scheduled before task j,
$R_2 =_r \leqslant$ and $R_1 =_r \geqslant$ if task j is scheduled before task i.

The constraint `R1 ## R2` allows only these two possible configurations. We have to impose these constraints between starting and end 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. The idea is to find a consistent ordering between tasks competing for the same resource by imposing new *precedence* constraints, as suggested in [28]. 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 triplets (`Task`$_i$, `Rel`, `Task`$_j$), representing respectively the name of `Task`$_i$ the relation between `Task`$_i$ and `Task`$_j$ and the name of `Task`$_j$, we can compute the transitive closure by selecting (through the `findall` predicate) all those couples of tasks (Task$_i$, Task$_k$) whose first task unifies with `Task`$_i$ in the clause head, and then by performing the closure on the triplet of tasks.

[6] It is worth noting that, for instance, a domain containing the relation symbol $<=$ is different from a domain containing both symbols $[<, =]$. In fact, although they lead to the same propagation (point 3 in Section 2.5) on integer variables linked by the *rel* constraint, they have a different behavior in the labeling step. In the first case, when a consistent solution is found, we have an instantiation of the relation variable to the unique symbol $\leqslant$, while in the second case we possibly have two different solutions: one for the relation $<$ and the other for the symbol $=$.

```
transitive([(Task_i, R_ij, Task_j) | RestTasks]):-
    findall((Task_i, R_ik, Task_k), member((Task_i,
     R_ik, Task_k), RestTasks), List),
    closure((Task_i, R_ij, Task_j), RestTasks, List),
    transitive(RestTasks).
closure(_, _, [ ]).
closure((Task_i, R_ij, Task_j), RestTasks, [(Task_i, R_ik, Task_k)|RL]):-
    member((Task_j, R_jk, Task_k), RestTasks), !,
    comp(R_ij, R_jk, R_ik),
    closure((Task_i, R_ij, Task_j), 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 $O(n^3)$. In fact

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

This propagation is quite expensive, but in many cases very powerful since constraint `rel/3` propagates from values of tasks starting and end points to relations and vice versa, 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., $a_1 \leqslant a_2$, $b_1 \leqslant b_2$, $b_2 \leqslant b_3$. 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 $[\leqslant, \geqslant]$. The starting and end 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 Fig. 4(a) where all the missing arcs represents relations whose domain is $[\leqslant, \geqslant]$. The transitive closure of the network propagates precedence relations and the one inferred by the *rel*/3 constraint between b_3 and a_1. As a consequence, all the other relations are instantiated as described in Fig. 4(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 sort of *forward checking* propagation since as soon as a relation is known all the other relations can be propagated accordingly.

We have evaluated the performance of our approach based on relation variables, on five sets of randomly generated disjunctive scheduling problems of, respectively, 18 and 20 tasks on two resources, 20 and 25 on three resources and of 40 and 50 on five resources. We have considered two parameters: the number of nodes generated for the non-deterministic choice of conflicting task ordering, and the computational time.

In Table 3, the number of nodes generated by the relation-based approach (`indomain` on relation variables), reported in column *Rel Nodes*, is significantly less than the number of nodes generated by a pure CLP approach reported in column *CLP Nodes* (number of calls to the `no_overlap` disjunctive choice where an ordering between conflicting tasks is decided).

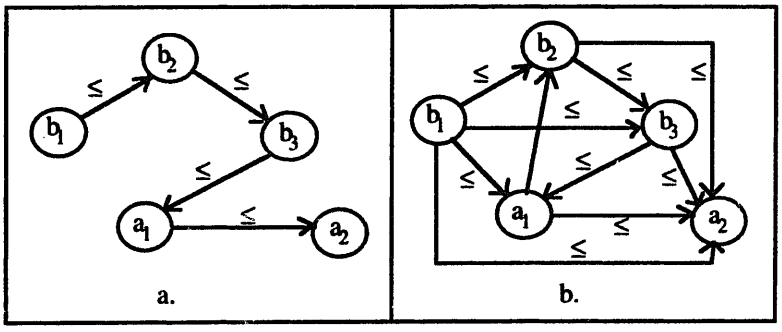

Fig. 4. Example of ordering relations.

Table 3
Experimental results

Tasks	Resources	*Rel Nodes*	*CLP Nodes*	*Rel Time*	*CLP Time*
18	2	133	12 278	22	58
20	2	336	23 716	297	1042
20	3	416	169 368	139	100
25	3	741	131 229	322	541
40	5	1020	255 402	870	1690
50	5	1540	266 710	1200	1873

Another comparison parameter is the computational time. The extended language generally outperforms a pure CLP(FD) solver on disjunctive scheduling problems. Some applications, however, do not benefit from the use of the transitive closure. In particular, for applications where domain variables do not restrict relation variable domains, or applications with few precedence constraints, the tradeoff between the overhead introduced by relation variables and the propagation benefit is not worthy. In addition, note that current CLP(FD) commercial solvers such as CHIP [10] or ILOG [18], making use of global symbolic constraints [4], outperform our relation based approach since they use sophisticated propagation techniques tailored on the specific constraint. In fact, global symbolic constraints represent suitable abstractions that enable a declarative statement of the problem and an operational behavior matching the best available pruning techniques. Our approach strength is the generality and expressivity since it can be applied to any kind of disjunctive constraints and does not rely on special purpose propagation techniques. For this reason, we compare our extended solver with a pure CLP(FD) solver making use of arc-consistency propagation techniques.

3.2.2. Bin packing problem

In this section, we present an example of the extended language specialized on sets. All the considerations from a knowledge representation viewpoint made on integers hold also for sets.

Consider the *bin packing problem*: given a multiset of n integers $\{w_1, \ldots, w_n\}$, representing weights of a set of objects to partition, the goal is to find a partition of the objects into a minimal number m of bins or sets (to be determined) such that in each

bin the sum of all weights does not exceed a given capacity W_{max}. The Conjunto abstract formulation of the problem is the following:

$$s_i :: \{(1, w_1), \ldots, (n, w_n)\},$$

$$s_i \cap s_j = \{\ \} \quad \text{for all } i \neq j,$$

$$s_1 \cup s_2 \cup \cdots \cup s_m = \{(1, w_1), \ldots, (n, w_n)\},$$

$$\sum_{i|(i,w_i) \in glb(s_j)} w_i \leqslant W_{max} \quad \text{for all } s_j.$$

The goal is to minimize the number m of bins. Now, suppose that we have the following constraint: object i and object j cannot be inserted in the same bin. Therefore, we have a disjunctive constraint of the kind:

$$\{i\} \subseteq s_k \vee \{j\} \subseteq s_k \vee (\{i\} \neq^0 s_k \wedge \{j\} \neq^0 s_k) \quad \text{for all } k = 1..m.$$

In the extended language, we express this disjunction by creating, for each bin, two relation variables R_{i,s_k} and R_{j,s_k} (representing the relations between sets $\{i\}$ and $\{j\}$, respectively, and set s_k) whose domain contains both symbols $[\subseteq, \neq^0]$. The allowed configurations are

$$R_{i,s_k} =_r \subseteq \quad \text{and} \quad R_{j,s_k} =_r \neq^0,$$
$$R_{i,s_k} =_r \neq^0 \quad \text{and} \quad R_{j,s_k} =_r \subseteq,$$
$$R_{i,s_k} =_r \neq^0 \quad \text{and} \quad R_{j,s_k} =_r \neq^0.$$

Thus, we can impose that the least upper bound of the two variables can assume only two values: $R_{lub} :: [\top, \neq^0]$ (i.e., $lub(R_{j,s_k}, R_{i,s_k}, R_{lub})$). In fact, the first two allowed configurations give $\top$ as a least upper bound, while the third configuration gives $\neq^0$ as a least upper bound. The only not allowed configuration would give as a least upper bound the symbol $\subseteq$ which is not included in the domain of R_{lub}. Experimental results are encouraging. For example, a set partitioning problem with 81 objects to be placed in at most 28 bins with disjunctive constraints can be solved by our extension in 19 s, while the Conjunto language, exploiting clause disjunction, finds the same solution in 42 s.

An interesting point is that if we change the order in which disjunctive clauses are selected, performances change drastically: Conjunto does not even find a solution in 1 h, while if we change the order in which values are inserted in the domain of relation variables, the same solution can be found in almost the same time (22 s). Thanks to constraint propagation on relation variables.

The same considerations made on symbolic constraints in Section 3.2.1 also apply for this case. Modern Constraint Programming systems [10,18] have special purpose constraints for bin packing applications more efficient than Conjunto and our proposed approach.

4. Related work

A work related to our approach is described in [24] and concerns the integration into a finite domain constraint solver of the Interval Algebra qualitative constraints.

The authors represent Interval Algebra constraints as variables and compute the transitive closure of the network. Our approach extends and formalizes this idea since it embeds not only qualitative but also quantitative reasoning and the integration of both.

A work similar to ours is that by Frühwirth [13] on Constraint Handling Rules (CHRs) as an extension of Constraint Logic Programming languages. The author 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. Nevertheless, CHRs can be a suitable tool for implementing our extension.

A common way of coping with disjunctive constraints in CLP systems such as Oz [17], AKL(FD) [5] and Prolog IV [6] relies on the use of boolean variables associated with constraints, i.e., $(C \leftrightarrow X = 1 \wedge X::[0,1])$, called *reified constraints*. A similar concept has been applied for solving disjunctive scheduling problems in [7]. The idea is that the boolean value of variable X corresponds to the truth value of the constraint C. Disjunction is handled by associating two boolean variables with different disjuncts and imposing an exclusive or between them. From this perspective, we achieve the same results. The main difference with our work concerns the fact that in our language the user can express operations and constraints among relation variables thus changing constraints during the computation. This feature can be very useful and expressive in temporal reasoning applications and in general from a knowledge representation point of view.

As far as disjunction in CLP is concerned, we have to mention also the *constructive disjunction* [15]. 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 efficiency of the solver. Our extension is mainly devoted to the qualitative reasoning. One example can be found by considering the following constraint store: $\{X \leqslant Y, Y \leqslant Z, (X \geqslant Z \cup X \leqslant Y\}$, 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 $X \leqslant Y$ since the only way to guarantee the consistency of the constraint $\geqslant$ is to have three equal values which is subsumed by $X \leqslant Y$.

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 than u. Disjunction is 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 [27]. 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 [10]. 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 relation variables, see [22]. From this perspective, our approach is more general since it can be applied to any kind of disjunctive constraints. Obviously, the propagation performed by the `cumulative` constraint is much more powerful since it is based on the best available pruning techniques.

As far as the transitive closure of the network is concerned (as described in Section 3.2.1), in [20] 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 these kinds of constraints. The transitive closure of the network can be computed also in our framework with some differences with respect to [20]: first, we perform a qualitative closure on binary constraints, while in [20] they perform a more complex closure that introduces more constraints containing operators such as sum, difference, and so on; second, our transitive closure is guided by the user that imposes, at language level, the composition operator among relation variables, while in [20] the transitive closure is embedded in the constraint solver.

This paper, even though presenting these differences, suggests to us an interesting future direction for treating operators in our framework. 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.

Other works, like Refs. [21,3] treat relations as matrices of admitted sets of values and in that sense they reason on constraints. In particular, the work presented in [21] is based on a relation algebra with operations on relations which are basically the same used in our paper. However, we work on an intensional representation of relations as relation symbols and not on admitted values.

5. Conclusion and future work

We have presented an extension of the CLP framework for reasoning on constraints. The extension has been first introduced as a general framework, then specialized on integers and sets. We provided some examples that show the increased expressive power of the extended CLP(FD ... nguage and the effectiveness of the approach in dealing with disjunction.

Future works are aimed to:

- implement the relation variable constraints at lower level thus increasing the efficiency of the relation based propagation;
- define heuristics on relation variables and value ordering during the labeling step, as those suggested in [28];
- extend the language in order to cope with operators as briefly suggested in Section 4;
- specialize the extension proposed on other relation domains;
- apply the extended language to real life applications in the field of scheduling and planning and compare the proposed approach with a pure CLP one.

Acknowledgements

Authors' work has been partially supported by CNR, Committee 12 on Information Technology (Project SCI*SIA). We would like to thank Pascal Van Hentenryck for useful discussions during our visit at Brown University and Carmen Gervet for providing the Conjunto code of the bin packing problem. In addition, we would like to thank the anonymous referees for their helpful comments on earlier versions of this paper.

References

[1] J.F. Allen, Maintaining knowledge about temporal intervals, Communications of the ACM 26 (1083) 832–843.
[2] P. Van Beek, Reasoning on qualitative temporal information, Artificial Inteiligence 58 (1992) 297–326.
[3] P. Van Beek, R. Detcher, Constraint tightness and looseness versus local and global consistency, Journal of the ACM 44 (4) (1997) 549–584.
[4] N. Beldiceanu, E. Contejean, Introducing global constraints in CHIP, Mathematical Computer Modelling 20 (12) (1994) 97–123.
[5] B. Carlson, S. Haridi, S. Janson, AKL(FD) – A concurrent language for FD programming, in: Proceedings of the International Logic Programming Symposium, MIT Press, Cambridge, MA, 1994.
[6] A. Colmerauer, An introduction to prolog III, ACM Communication 33 (7) (1990) 70–90.
[7] Y. Colombani, Constraint programming: An efficient and practical approach to solving the job-shop scheduling, in: Proceedings of the Second International Conference on Principles and Practice of Constraint Programming, 1996, pp. 149–163.
[8] R. Dechter, I. Meiri, J. Pearl, Temporal constraint networks, Artificial Intelligence 49 (1991) 61–95.
[9] M. Dincbas, P. Van Hentenryck, M. Simonis, Solving large combinatorial problems in logic programming, Journal of Logic Programming 8 (1–2) (1990) 75–93.

[10] 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, 1988, pp. 693–702.
[11] A. Dovier, E.G. Omodeo, E. Pontelli, G. Rossi, {*log*}: A language for programming in logic with finite sets, Journal of Logic Programming 28 (1) (1996) 1–44.
[12] ECRC *ECL^{i}PS^e* User Manual Release 3.5, 1992.
[13] T. Frühwirth, Temporal Reasoning with Constraint Handling Rules, ECRC-94-05, ECRC, 1994.
[14] C. Gervet, Propagation to reason about sets: Definition and implementation of a practical language, Constraints 1 (1997) 191–244.
[15] 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.
[16] P. Van Hentenryck, Y. Deville, The cardinality operator: A new logical connective for constraint logic programming, in: F. Benhamou, A. Colmerauer (Eds.), Constraint Logic Programming: Selected Research, MIT Press, Cambridge, MA, 1993.
[17] M. Henz, M. Mehl, M. Müller, T. Müller, J. Niehren, R. Scheidhauser, C. Schulte, G. Smolka, R. Treinen, J. Würtz, The Oz Handbook, DFKI, RR-94-09, 1994.
[18] ILOG, ILOG Solver 4.2 Reference Manual, 1998.
[19] J. Jaffar, M.J. Maher, Constraint logic programming: A survey, Journal of Logic Programming 19/20 (1994) 503–582.
[20] J. Jaffar, M.J. Maher, P.J. Stuckey, R.H.C. Yap, Beyond finite domains, in: Proceedings of the Second International Workshop on Principles and Practice of Constraint Programming, 1994.
[21] P.B. Ladkin, R.D. Maddux, On binary constraint problems, JACM 41 (3) (1994) 435–469.
[22] E. Lamma, P. Mello, M. Milano, Temporal reasoning in a meta constraint logic programming architecture, Annals of Mathematics and Artificial Intelligence 22 (1998) 139–158.
[23] B. Legeard, E. Legros, CLPS: A set constraint logic programming language, Laboratoire d'Automatique de Besancon Institut de Productique, 1991.
[24] J. Lever, B. Richards, R. Hirsh, Temporal Reasoning and Constraint Solving, IC-Park, Deliverable CHIC ESPRIT Project EP5291, 1992.
[25] A.K. Mackworth, Consistency in networks of relations, Artificial Intelligence 8 (1977) 99–118.
[26] M. Milano, Reasoning on constraints in CLP(FD), University of Bologna, 1998.
[27] V.A. Saraswat, Concurrent Constraint Logic Programming, Carnegie-Mellon University, 1989.
[28] S.F. Smith, C. Cheng, Slack-based heuristics for constraint satisfaction scheduling, Proceedings of AAAI93, 1993.
[29] M.B. Vilain, H. Kautz, P. Van Beek, Constraint propagation algorithms for temporal reasoning: A revised report, in: D.S. Weld, J. De Kleer (Eds.), Readings in Qualitative Reasoning about Physical Systems, Morgan Kaufmann, Los Altos, CA, 1990, pp. 373–381.