Università degli Studi di Bologna
DEIS

Consistent Orderings for Constraint Satisfaction Scheduling

Silvia Buzzi Evelina Lamma Paola Mello
Michela Milano

April 15, 1997

DEIS Technical Report no. DEIS-LIA-97-001 *LIA Series no. 18*

Consistent Orderings for Constraint Satisfaction Scheduling

Silvia Buzzi [2] 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] *Dipartimento di Ingegneria, Università di Ferrara*
Via Saragat 1, 41100 Ferrara, Italy
pmello@deis.unibo.it

April 15, 1997

Abstract. The job shop scheduling problem has been formulated in Constraint Logic Programming as one of finding a consistent assignment of start times for each operation of each job. Nonetheless, in some proposals within the Artificial Intelligence area, the problem has been alternatively formulated as one of establishing sequencing constraints between pairs of operations contending for the same resource over time. In this work, we follow this latter approach within a Constraint Logic Programming (CLP) framework. CLP solvers on finite domains use constraints to prune those combinations of assignments which cannot appear in any consistent solution. Scheduling problems require 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 to represent ordering relations between pairs of tasks and apply the resulting language to scheduling problems.

Keywords: *Scheduling, Constraint Logic Programming*

Contents

1 Introduction

Scheduling problems arise in domains like manufacturing, transportation, computer processing, production planning and so on. Scheduling deals with the allocation of resources to activities over time.

A convenient formulation of scheduling problems is that based on Constraint Satisfaction Problems (CSPs). A CSP can be defined on a set of variables, each ranging on a finite domain, and a set of constraints. A solution to a CSP is an assignment of a value (belonging to variable's domain) to each variable which is consistent with the constraints involved in the problem.

Usually, in the formulation of scheduling problems as CSPs, variables of the problem are tasks' starting points and the constraints of the problem include precedence relations linking the operations' starting and ending times, capacity constraints that prevent resources from being allocated to more activities than they can process at one time.

The Constraint Logic Programming (CLP) [7] paradigm 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 [3, 15]). In particular, for scheduling applications the adoption of CLP on finite domains (CLP(FD)) results in a direct implementation of the original CSP problem also providing an embedded constraint satisfaction mechanism.

Recently, other proposals within the Artificial Intelligence area (e.g. [10, 14]) formulated the scheduling problem as one of establishing sequencing constraints between pairs of operations contending for the same resource over time. The basic idea is to post additional precedence constraints between pairs of tasks competing for the same resource so as to ensure feasibility with respect to time and capacity constraints. As shown in [14] this second formulation of the problem provides a more convenient search space in which to operate.

In this work, we follow this latter approach to solve scheduling problems within a CLP(FD) framework. In particular, we treat constraints as first-class objects (i.e., relations as domain elements) which can be changed (restricted) during the computation or even chosen when disjunction occurs, in order to support some form of *qualitative reasoning*. The resulting language is a two sorted first order language: we have *object* variables ranging on a finite domain of integers and *relation* variables ranging on finite domains of relation symbols.

In order to face scheduling problems, we start by defining both object variables (representing starting and ending times of tasks) and relation variables (each representing possible orderings between pairs of tasks). Then, we apply a transitive closure of the relations among tasks competing for the same resource. A labeling step then determines a consistent ordering between tasks, if it exists. Finally, on the basis of the new sequencing constraints between tasks (i.e., the value of relation variables) new constraints are imposed on the starting and ending time of tasks.

The paper is organized as follows: in section 2, we introduce the approach to scheduling based on task sequencing. In section 3, we present the CLP(FD) extension to cope with task sequencing. In section 4, we show how to apply the extended language to scheduling problems. Section 5 presents some experimental results. Related works are presented in section 6. Conclusions and future works follow.

2 Problem modeling and search strategy

We are considering job-shop scheduling where tasks of the same job are linked by precedence relations, capacity constraints prevent resources from being allocated to more activities than they can process at one time, and ready time and deadlines define the admissible locations of tasks.

Usually, in the formulation of scheduling problems as CSPs, variables of the problem $S_1, \ldots, S_n$ are tasks starting points whose domain represents the possible starting times according to ready times and deadlines. On the basis of this representation, constraints are modeled in the following way:

- **sequencing constraints**: precedence relations constrain a task i to be executed before another task j. The corresponding constraint is $S_i + d_i \leq S_j$, where d_i is the duration of the task i.
- **resource capacity constraints**: if two operations i and j require the same single capacity resource, they are linked by the disjunctive relation $(S_i + d_i \leq S_j) \vee (S_j + d_j \leq S_i)$.
- **ready times and deadlines**: ready times and deadlines can be considered on the job or on the task. The general form of these constraint is the following: $S_i \geq ready$ and $S_i + d_i \leq deadline$.

An alternative approach, recently proposed in the Artificial Intelligence field [10, 14], can be formulated as one of establishing sequencing constraints between each pair of variables competing for the same resource. In this case, each pair of tasks i and j requiring the same resource is linked by a relation variable R_{ij}, i.e., a variable ranging on a finite domain of constraint symbols. The assignment of the value $\leq$ to variable R_{ij} implies the ordering i before j, while the assignment of the value $\geq$ to variable R_{ij} implies the ordering j before i. Therefore, the instantiation of a relation variable corresponds to post a new precedence relation. If the relation variable has a domain containing both constraint symbols, it means that either sequencing decision is still possible. An empty domain for a relation variable results in a failure.

Relation variables can be linked by constraints for ensuring the temporal consistency. Constraint propagation possibly computes new ordering constraints subsumed by already established sequencing relations. Our problem solving framework adopts a backtrack search procedure in which a solution is incrementally extended through the intertwining of a selection (and instantiation) of a relation variable and constraint propagation process.

We have basically three kinds of constraint propagation:

- from tasks' starting and ending points to relation variables;
- among relation variables;
- from relation variables to starting and ending points.

In the following, we refer to E_i as the ending point of the task i, i.e., $S_i + d_i$. In addition, we refer to X_{min} and X_{max} as the minimum and the maximum value of the domain of variable X which is represented as $[X_{min}..X_{max}]$. As concern the first kind of propagation, for any unordered pair of operations i and j contending for the same resource we have four possible situations:

- Task i can be scheduled only before task j, i.e., R_{ij} is instantiated to $\leq$: this case happens if the domain of the starting time of i has no intersection with the domain of the ending time of j and $E_{i_{max}} < S_{j_{min}}$ and $S_{i_{min}} < E_{j_{max}}$. For example, given variables $S_i :: [5..9]$, $E_i :: [11..17]$, $S_j :: [20..27]$, $E_j :: [35..42]$, the task j cannot be scheduled before task i.
- Task j can be scheduled only before task i, i.e., R_{ij} is instantiated to $\geq$: this case happens if the domain of the starting time of j has no intersection with the domain of the ending time of i as in the previous case and $E_{j_{max}} < S_{i_{min}}$ and $S_{j_{min}} < E_{i_{max}}$. The previous example can be applied also to this case by inverting pedexes i and j.
- Both sequencing relations are possible, i.e., $R_{ij} :: [\leq, \geq]$: this case happens if the domain of the starting time of i has a non empty intersection with the domain of the ending time of j and the domain of the staring time of j has a non empty intersection with the domain of the ending time of i, i.e., $S_{j_{min}} \leq E_{i_{max}}$ and $S_{i_{min}} \leq E_{j_{max}}$. For example, given variables $S_i :: [1..6]$, $F_i :: [5..10]$, $S_j :: [4..8]$, $E_j :: [5..9]$ both orderings are still consistent.
- No consistent ordering is possible i.e., $R_{ij} :: []$: this case happens if $S_{j_{min}} > E_{i_{max}}$ and $S_{i_{min}} > E_{j_{max}}$. For example if $S_i :: [1..10]$, $E_i :: [14..21]$, $E_j :: [12..20]$, $S_j :: [5..13]$, no consistent assignment can be performed.

As concerns the propagation of constraints among relation variables, a powerful but quite heavy propagation is the (qualitative) transitive closure of the network by composing the relation variables. The complexity of the transitive closure propagation can be computed as follows: if we have n tasks competing for a resource, we have

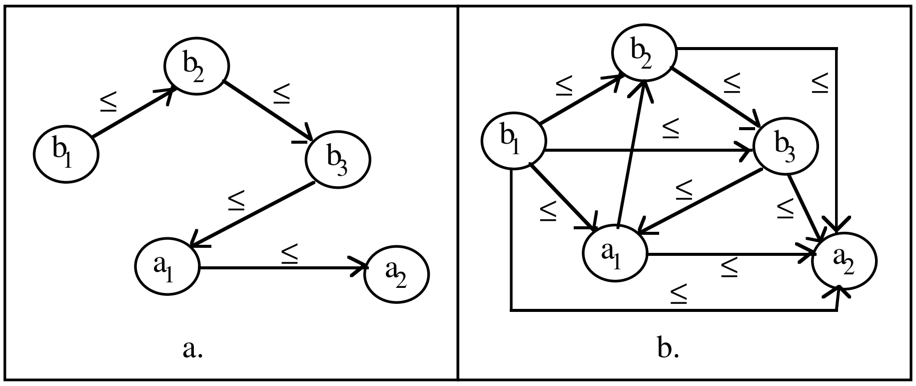

Figure 1: Example of ordering relations

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:

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

This propagation is quite expensive but in many cases very powerful, especially if used together with the propagation form starting and ending points to relation variables. 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. In fact, being the domain of the ending point of task b_3 [17..40] and the domain of the starting point of a_1 [45..85], the only possible ordering is b_3 before a_1 ($E_{b_3max} < S_{a_1min}$ and $S_{b_3min} < E_{a_1max}$). At this point, we have the situation depicted 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 precedes 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.

The third kind of propagation, i.e., from relation variables to starting and ending points, is easier. The instantiation of a relation variable R_{ij} to the value $\leq$ corresponds to impose the following constraint between starting and ending times: $S_i + d_i \leq S_j$. The instantiation of a relation variable R_{ij} to the value $\geq$ corresponds to impose the following constraint between starting and ending times: $S_j + d_j \leq S_i$. If a relation variable is still unbound, no propagation can be performed.

3 Extending the CLP(FD) Language

In this section, we describe the language which makes it possible to reason on relations. An extended paper describing the language, its declarative and operational semantics can be found in [9]. The language is a two sorted first order language (on integers and relations). Operations and constraints on integers are the usual operations and constraints of CLP(FD) solvers and operations and constraints on relations are equality, inequality constraints, composition, and union operations.

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 $\{>, \geq, <, \leq, =, \neq\}$. We 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. Relation variables are defined on the Rel sort i.e., $\{>, \geq, <, \leq, =, \neq, \top\}$. The relation $\top$ represents the most relaxed constraint. Scheduling applications usually make use of symbols $\leq$ and $\geq$ only.

For the integer sort, unary constraints link a variable to range over a finite domain of integers $X :: [a_1, \ldots, a_n]$. As usual, binary constraints are $>, \geq, <, \leq, =, \neq$ and operations are addition (+) and multiplication (*).

As concern the relation sort, unary constraint links each relation variable to range over a finite domain of relations $R :: [r_1, \ldots, r_n]$. On the relation sort, equality and inequality constraints can be defined as usual. We 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 but we don't use it in the paper.

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 $R_3 :: [<, =, >]$. We do not use this operation in the following.

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:

$$\begin{array}{lcl} rel(X, R_{xy}, Y) & \leftrightarrow & (R_{xy} =_r `<`) \wedge\; X < Y \;\vee \\ & & (R_{xy} =_r `\leq`) \wedge\; X \leq Y \;\vee \\ & & (R_{xy} =_r `>`) \wedge\; X > Y \;\vee \\ & & (R_{xy} =_r `\geq`) \wedge\; X \geq Y \;\vee \\ & & (R_{xy} =_r `\neq`) \wedge\; X \neq Y \;\vee \\ & & (R_{xy} =_r `=`) \wedge\; X = Y \;\vee \\ & & (R_{xy} =_r `\top`) \end{array}$$

The interesting point concerns the operational semantics of the extended language. In particular, the propagations needed for solving the scheduling problems are easily achieved by the newly introduced constraints. In section 2, we have identified three basic propagations: from tasks' starting and ending points to relation variables, among relation variables and from relation variables and starting and ending points. The first and the third propagation are executed by the $rel/3$ constraint. 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: given the constraint store C 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 $R =_r <$.

This propagation can be easily achieved in a CLP framework by performing a kind of bound propagation on the minimum and maximum value of starting and ending points. We consider the intersection among the domains of variables X and Y. There are three cases:

[1] the intersection contains more than one element;

[2] the intersection contains only one element;

[3] the intersection is empty.

In the first case the relation variable between X and Y will not change because in the intersection there are some values that entail all the constraints. For example, if $D_X = [1..10]$ and $D_Y = [5..14]$ are the domains associated with X and Y, then $D_X \cap D_Y = [5..10]$. If it is the case, we can infer nothing on the constraints between X and Y and, therefore, the domain of R_{XY} does not change.

Another situation occurs when the intersection contains only one value. In this case, the constraint "equal" is entailed by the intersection while all the other values entail either the relation "before" or the relation "after" but not both. For example, if $D_X = [1..10]$ and $D_Y = [10..14]$ are the domains associated with X and Y, then $D_X \cap D_Y = 10$. Therefore, if between the variables X and Y there is the constraint $X < Y \vee X = Y \vee X > Y$ (i.e., $R_{XY} :: [<, =, >]$), the constraint $X > Y$ is no longer entailed by the values of the single variables and $R_{XY} :: [<, =]$.

The third case happens when the intersection is empty. Obviously, the constraint "equal" is no longer entailed. Moreover, the values of the variables entail only one of the relations "before" or "after". For example, if $D_X = [1..10]$ and $D_Y = [11..14]$ are the domains associated with X and Y, then $D_X \cap D_Y = \emptyset$. Therefore, if between the variables X and Y there is the constraint $X < Y \vee X = Y \vee X > Y$ (i.e., $R_{XY} :: [<, =, >]$), then the constraint $X = Y \vee X > Y$ is no longer entailed by the values of the single variables and $R_{XY} =_r <$.

The inverse propagation (from relation variables to integers variables) can be done in two ways: we can constrain integer variables subject to *least upper bound constraints*. For each constraint in S of the form $rel(X, R_{xy}, Y)$ 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 then we perform a union on variable domains.

A weaker propagation very useful in scheduling application is simply achieved by a *coroutining* mechanism where the propagation on integer variables representing starting and ending points can be done only when the relation variable is instantiated.

The second kind of propagation mentioned in section 2 is a simple arc-consistency on relation variables.

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 makes it possible to process domain variables and their domains and to write user-defined constraints by means of low level predicates. An example of the implementation of the `comp` operator is given in the Appendix.

4 Scheduling in the extended language

In this section, we explain how to use the extended language in order to represent and solve scheduling problems. In particular, we focus on capacity constraints. 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.
```

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_rel(Si,Sj,di,dj):-
   R1 ::[<=, >=], R2 ::[<=, >=], R1 ## R2,
   rel(Si + di, R1,Sj), rel(Sj + dj, R2,Si).
```

where symbol `##` represents the disequality constraint[1]. The two relation variables between the starting and ending points of the tasks can assume only two values, i.e., $R_1 = \leq$ and $R_2 = \geq$ if task i is scheduled before task j. Viceversa, $R_2 = \leq$ and $R_1 = \geq$. Note that R_1 corresponds to variable R_{ij} in section 2, while R_2 corresponds to the inverse relation R_{ji}. Therefore, the constraint R_1 `##` R_2 states these two possible

[1] We omit the pedex r in the real syntax since the constraint `##` works as well for any other finite domain variable.

configurations. We have to impose these constraints between all the not overlapping tasks starting and ending points.

The `rel/3` constraint performs both propagations from starting and ending points to relations and vice versa.

A possible propagation on relation variables 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_rel` 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).
```

Given the relation variable R$_{ij}$, the predicate `findall` collects in the list `List` all the relation variables R$_{ik}$ which can be composed with R$_{ij}$. The predicate `closure` performs the transitive closure of the network by finding the relation R$_{kj}$ in the list `RestTasks`. The constraint `comp/3` performs the composition of relation variables and has been described in section 3. The implementation of this constraint is reported in the appendix.

5 Experimental Results

In this section, we evaluate the performance of the *relation-based* method, i.e., our approach based on relation variables, on a suite of randomly generated job shop scheduling problems.

We have considered five sets of problems of respectively 15, 18 and 20 tasks on 2 resources, and of 20 and 25 on 3 resources. We have considered two parameters: the number of nodes generated for the non-deterministic choice of conflicting task ordering, and the computational time.

We can see in table 5 that the number of nodes generated by the relation-based approach (`indomain` on relation variables) is significantly less than the number of nodes generated by the classical CLP approach (number of calls to the `no_overlap` disjunctive choice where an ordering between conflicting tasks is decided).

Another comparison parameter is the computational time even if this parameter is not significant since the relation variables and constraint propagation on them has been implemented on top of a CLP on finite domain solver. Therefore, the efficiency of the extended language could further increase if we implement the relation variable constraints at solver level.

Tasks	Resources	CLP Nodes	Rel. Nodes	CLP Time	Rel. Time
15	2	9440	40	1,9	35
18	2	12278	133	58	64
20	2	23716	336	1042	364
20	3	169368	416	100	278
25	3	131229	741	541	322

Table 5: Experimental Results

We have noted that two scheduling problems with the same number of tasks behave in different ways. In particular, problems with less tasks per job are less efficient in our approach with respect to problems with more tasks per job. This can be explained by the fact that more precedence relations between tasks of the same job produce a more efficient propagation in terms of a transitive closure. However, for an increasing number of tasks the transitive closure is quite hard. Therefore, we are currently comparing the same approach without the transitive closure with some heuristics on the relation variable ordering and value ordering with an approach with the transitive closure. We have to say also that all these constraints have been implemented on top of the ECL^{i}PS^e finite domain constraint solver. We think that a more efficient way of implementing the constraints could greatly increase the performances.

6 Related Works

Related papers concern both the field of constraint programming and its application to scheduling problems, and the field of Artificial Intelligence scheduling.

Many related works in the field of AI scheduling should be mentioned. Here, for the sake of conciseness, we mention only some of them. The PhD thesis by Sadeh [13] proposes a look-ahead analysis of the problem in order to identify critical operations and promising reservations for these operations. Critical operations are identified by analyzing contentions between operations for a resource over some time interval. Therefore, Sadeh's work relies on a dynamic variable ordering heuristic that maintains profiles of resource demand over time and focuses attention on operations that most contribute to *bottlenecks*. Attention is still focussed on tasks, but has shifted on resource contention.

In [4] the structure of resource capacity constraints is exploited to define dominance conditions for pruning the set of feasible sequencing alternatives. More recently, in [10] sequencing constraints are repeatedly posted between sets of conflicting operations until resource capacity analysis indicates no further resource contentions.

Finally, the paper by Smith and Cheng [14] claims that by repeatedly posting new ordering constraints between pairs of tasks contending for the same resource provides a more convenient search space in which to operate. Here we apply all these considerations but in a CLP framework by providing the user relations as first class objects and constraints between them in order to directly implement this problem modelization and search strategy. The only difference with the approach proposed in [14] is that we compute a transitive closure on relation variables in order to provide a more powerful but less efficient propagation on relation variables. Within the extended language, we are currently comparing a relation based approach relying on variable ordering heuristics with no propagation on relation variables with one exploiting the transitive closure operation.

As concerns the application of CLP to scheduling problems, many attempts have been performed in order to create special purpose constraints with more efficient propagations in order to effectively cope with disjunctive ordering constraints.

First, 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.

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 $[c_1, c_2, \ldots, c_n]$ 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 [12]. Our approach can be built on top of any CLP(FD) language without the entailment, i.e., *tell* languages.

Another way of coping with capacity constraints in CLP is by means of the *cumulative* constraint [2].

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.

One of our current work is to compare the efficiency of our methodology not only with classical CLP based schedulers but also with all the above mentioned CLP based approaches.

7 Conclusions and Future Works

We have presented an alternative way of modeling scheduling problems in an enriched CLP(FD) language. Instead of finding a consistent assignment of task starting points, we iteratively impose new ordering relations between tasks competing for the same resource. This approach is based on propagations from starting and ending points to relation variables and viceversa. We have implemented the CLP(FD) 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.

We have tested the system on scheduling benchmarks and we have obtained encouraging results. Further developments are aimed to:

- define heuristics on relation variables and value ordering during the labeling step, as those suggested in [14], and eventually test whether they can replace the transitive closure of the network;
- apply the extended language to real life applications in the field of scheduling and planning. In particular, we will solve a real job shop scheduling application in the railway traffic regulation field as the one described in [8].

References

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

[2] 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.

[3] 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.

[4] J. Erschler, F. Roubellat, J.P. Vernhes, " Characterizing a set of feasible sequences for n jobs to be carried out on a single machine", European Journal of Operational Research, 4, pp. 189–194, 1980.

[5] ECL^iPS^e User Manual Release 3.3, ECRC 1992.

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

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

[8] E.Lamma, P.Mello, M.Milano. "A distributed Constraint-Based Scheduler", in *Artificial Intelligence in Engineering*, Vol 11, pp 91-105, Elsevier Science Ltd., 1997.

[9] E.Lamma, P.Mello, M.Milano. "Reasoning on Constraints in Constraint Logic Programming" DEIS - University of Bologna - Technical Report 96-006. `http://www-lia.deis.unibo.it/Research/TechReport.html`

[10] N. Muscettola, "Scheduling by iterative Partition of Bottleneck Conflicts", in *Proceedings of 9th IEEE Conference on AI Applications*, 1993.

[11] S. Kambhampati, "Using Disjunctive orderings instead of Conflict Resolution in Partial Order Planning", Tech. Report, ASU CSE TR96-002, Arizona State University, 1996.

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

[13] N.Sahed, “Look-Ahead Techniques for Micro-Opportunistic Job Shop Scheduling”, Tech. Report CMU-CS-91-102, Carnegie Mellon University, 1991.

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

[15] P.Van Hentenryck, “Constraint Satisfaction in Logic Programming”, MIT Press, 1989.

[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.

8 Appendix

In the appendix, we report an example of implementation of the *comp* constraint. It is implemented as a user defined constraint. ECL^{i}PS^e [5] provides the mechanisms for suspending and awaking constraints (such as `make_suspension` and `insert_suspension`) in order to perform the propagation and built-in predicates in order to manipulate variable domains. All predicates used in the `comp` constraints are built-in predicates for the manipulation of domains. For example, predicate `dvar_domain(Z,DZ)` extracts the domain `DZ` from variable `Z`, predicate `dvar_update(Z,DZnew)` updates the domain of variable `Z`. The predicate `composition` performs the composition of all the domain elements by using the table 3.

```
% X, Y and Z are three domain variables

comp(X,Y,Z):-
    dvar_domain(X,DX),
```

```
    dvar_domain(Y,DY),
    dvar_domain(Z,DZ),!,
    composition(DX,DY,DZtemp),
    list_to_dom(DZtemp,DZ1),
    (dom_compare(=,DZ,DZ1)->true;
        (dom_intersection(DZ,DZ1,DZint,_),
        (dom_size(DZ,1)-> true;
         dvar_update(Z,DZint)),
         composition(DX,DYtemp,DZint),
         list_to_dom(DYtemp,DY1),
        (dom_compare(=,DY,DY1) ->true;
            dom_intersection(DY,DY1,DYint,_),
               (dom_size(DY,1)-> true;
                dvar_update(Y,DYint)),
                composition(DXtemp,DYint,DZint),
                list_to_dom(DXtemp,DX1),
                (dom_compare(=,DX,DX1)->true;
                    dom_intersection(DX,DX1,DXint,_),
                   (dom_size(DX,1)->true;
                    dvar_update(X,DXint),
                    comp(X,Y,Z) ))))),
    ((var(X);var(Y);var(Z)) ->
    (make_suspension(comp(X,Y,Z),3,Susp),
     insert_suspension((X,Y,Z),Susp,any of fd,fd));
     true),
     wake.

% X and  Y are domain variables

comp(X,Y,Z):-
    dvar_domain(X,DX),
    dvar_domain(Y,DY),!,
    composition(DX,DY,DZ1),
    list_to_dom(DZ1,DZ),
    var_fd(Z,DZ),
    ((var(X);var(Y);var(Z)) ->
        (make_suspension(comp(X,Y,Z),3,Susp),
        insert_suspension((X,Y,Z),Susp,any of fd,fd));
        true),
        wake.
```

```
% X and Z are domain variables

comp(X,Y,Z):-
    dvar_domain(X,DX),
    dvar_domain(Z,DZ),!,
    composition(DX,DY1,DZ),
    list_to_dom(DY1,DY),
    var_fd(Y,DY),
    ((var(X);var(Y);var(Z)) ->
        (make_suspension(comp(X,Y,Z),3,Susp),
        insert_suspension((X,Y,Z),Susp,any of fd,fd));
        true),
        wake.

% Y and Z are domain variables

comp(X,Y,Z):-
    dvar_domain(Y,DY),
    dvar_domain(Z,DZ),!,
    composition(DX1,DY,DZ),
    list_to_dom(DX1,DX),
    var_fd(X,DX),
    ((var(X);var(Y);var(Z)) ->
        (make_suspension(comp(X,Y,Z),3,Susp),
        insert_suspension((X,Y,Z),Susp,any of fd,fd));
        true),
        wake.
```