Enhancing Set Constraint Solvers with Bound Consistency

J. Correas[a], S. Estévez Martín[b], F. Sáenz-Pérez[c]

[a] jcorreas@fdi.ucm.es
[b] s.estevez@fdi.ucm.es
[c] fernan@sip.ucm.es
Faculty of Computer Science
Complutense University of Madrid, Spain

Abstract

Constraint solvers have proved to be a powerful tool for both expressing and solving complex problems in many contexts, in particular in Automated Planning and Scheduling. In this work, we propose new constraints to enhancing an implementation of a constraint solver for sets of integers (ic_sets) embedded in the ECL^{i}PS^e system. This way, we confer a new vision of modelling that in particular allows a task to be planned on alternate days. In addition, these new constraints make finite domain and finite sets domains to cooperate, reducing the search space and improving the efficiency of the existing set solvers for some specific cases. We report on a prototype implementation and apply these constraints to modelling scheduling problems as benchmarks for a performance analysis. Experiments reveal a relevant improvement with respect to the standard ic_sets solver which only features cardinality as cooperation mechanism between domains. Both enhanced expressiveness and performance are key points for practical applications of such solvers.

Keywords: Constraint Programming, Scheduling, Integer Set Solvers, Finite Domain Constraints

1. Introduction

Constraint Programming (CP) (Apt, 2003, Rossi et al., 2006) offers an abstract way for specifying constraints among objects of some domain. Example of domains are reals, integers, enumerated and sets. Finite sets ($\mathcal{FS}$) domains and solvers have been studied in a number of scientific works (Azevedo, 2007, Bergenti et al., 2009, Bessiere et al., 2013, Fernández & Hill, 2004, Gervet, 1997, Hawkins et al., 2011, Sadler & Gervet, 2008, Yip & Hentenryck, 2011), and implemented in several systems (ECL^{i}PS^e (Niederliński, 2014), Mozart (Mozart Consortium, 2013), B-Prolog (Zhou, 2011), Choco (Prud'homme et al., 2016), and Gecode (Schulte et al., 2017)) over the last years. Systems implementing $\mathcal{FS}$ solvers can be found either as integrated in a host language (as ECL^{i}PS^e) or as a constraint programming library (as Choco). The $\mathcal{FS}$ domain has clear advantages in terms of conciseness and neat formulation for expressing problems in a wide range of areas (Azevedo & Barahona, 2000). Nevertheless, in order to make $\mathcal{FS}$ more widely used for complex problems, additional improvements must be done to enhance the efficiency of the solvers.

In the context of Constraint Logic Programming (CLP) (Jaffar & Lassez, 1987), several works (Gervet, 1994, 1997, Sadler & Gervet, 2004, 2008) set a framework for the $\mathcal{FS}$ domain which has been implemented

in the ECL^{i}PS^e system. To define this new framework, they defined the set variable domain by a lattice of ground lower and upper bounds. Thus, a set variable ranges over *finite set domains* specified by ground bounds for the set inclusion ($\subseteq$). The constraint propagation uses consistency techniques based on interval reasoning over set constraints.

Enhancing the performance of $\mathcal{FS}$ constraint solving in particular resorts to the cooperation among solvers on different domains (Demoen et al., 1999, Gervet, 1997, Monfroy et al., 1995). Finite domain ($\mathcal{FD}$) solvers are revealed as an attractive mate for cooperation because of its inherent techniques to deal with non-linear constraints. One of these approaches (Apt & Wallace, 2007) defines the cooperation between $\mathcal{FD}$ and $\mathcal{FS}$ by means of the cardinality of a set. Constraints in $\mathcal{FD}$ limit the cardinality of sets with integer lower and upper bounds. However, there are other pieces of information available in a domain for sets of integers that can be projected to the $\mathcal{FD}$ domain as we introduce later in our proposal.

Application examples of these techniques are scheduling problems, which have been the subject of research from several approaches. For example, many specific techniques have been applied to the case of personnel scheduling, as summarized by den Bergh et al. (2013). Among them, CP has been proved to be a relevant technique that in many cases has been used in combination with other approaches as Integer Programming and Variable Neighbourhood Search. Remarkable works in this respect are those related to the nurse rostering problem: Qu & He (2009), Rahimian et al. (2017), Stølevik et al. (2011). Another research area in which Constraint Programming has been used is Production Scheduling (Harjunkoski et al. (2014)). As in personnel scheduling, Constraint Programming has been extensively used in combination with Mathematical Programming and heuristic methods. In these works, very well-known CP domains such as $\mathcal{FD}$ and interval domains are used.

Scheduling problems reveal as a quite important application area in many different real-life scenarios. Here, we mention some of such scenarios for large companies operating nowadays and involving huge amounts of data. The crew assignment problem for airlines (Zeghal & Minoux (2006)) which consists of optimally assigning the required crew members to each flight segment of a given time period, while complying with a variety of work regulations and collective agreements. Logistics is another critical area for scheduling problems for which there can be identified several scenarios: Supply chains (Mallik (2010)) which involve moving loads, delivery speed, service quality, operation costs, the usage of facilities and energy saving (Hijaz et al. (2015)), transportation (Yung-yu TSENG (2005)), and customer service (Mallik (2010), Naoui (2014)). Finally, we mention timetabling problems (Babaei et al. (2015), Jaya Pandey (2016)) which embraces schools, universities, employee allocation in companies, and sports tournaments.

We believe that the use of $\mathcal{FS}$ domain may contribute a great deal to the solvability of such complex problems by reducing both the number of required variables and constraints. Nevertheless, to the best of our knowledge there has been no much research on the application of constraints over sets of integers to these kinds of problems, possibly due to the limited cooperation of the $\mathcal{FS}$ domain with other domains, which limits the applicability to real-world case studies. Thus, our aim in this work is to contribute to the

applicability of intermixing $\mathcal{FS}$ and $\mathcal{FD}$ domains to tackle such case studies.

In particular, we propose in this paper the addition of new types of set constraints as well as a richer cooperation between $\mathcal{FS}$ and $\mathcal{FD}$ domains with new hybrid constraints to facilitate the use of set constraints in practical scheduling cases like the ones aforementioned. So, on the one hand, and motivated by the need to enhance performance, we propose a new cooperation between $\mathcal{FS}$ and $\mathcal{FD}$ domains based on the minimum and maximum values of a set. To this end, we present the new constraints `minSet` and `maxSet` to relate variables of both domains, therefore acting as cooperation bridges between both domains. This cooperation enables solving intermixing in both domains because pruning due to one solver is communicated to the other, and *vice versa*, making the finding of solutions faster as we demonstrate via examples. This effectively improves the previous approach (Apt & Wallace, 2007) and might reveal the use of $\mathcal{FS}$ solvers (combined with $\mathcal{FD}$ solvers) as an effective technique for solving constraint problems as the ones aforementioned. On the other hand, and motivated by improving expressiveness in planning and scheduling problems, we develop a new precedence constraint: the constraint operator `<<`. Here, S_1 `<<` S_2 means that all the elements in the set S_1 must be smaller than any element in the set S_2. From an expressiveness point-of-view, this constraint can be understood as a global constraint that allows users to specify with a single constraint what otherwise would have to be stated with primitive constraints and reflexive constraint functions. Such an alternative approach would obscure problem specification and would not use specific propagation rules (thus also enhancing performance) as we do. To the best of our knowledge, this new restriction has not been defined previously. Table 1 in the next page summarizes the most relevant advantages that can be obtained using the approach proposed in this research, as well as its usage and the importance in this scope.

From here on, this paper is organized as follows. We start in Section 2 showing the profits of the cooperation of solvers $\mathcal{FD}$ and $\mathcal{FS}$ by means of a motivating scheduling example. Then, Section 3 introduces the novel primitive constraints `minSet` and `maxSet`, together with the precedence constraint `<<`. Section 4 extends solver cooperation with the projection of the new $\mathcal{FS}$ constraints `minSet` and `maxSet`, and the existing cardinality constraint (#) in ECL^{i}PS^e as communication channels, named *bridges*, with the $\mathcal{FD}$ solver. Implementing this proposal is described in Section 5, and a performance evaluation is presented in Section 6. Finally, Section 7 concludes and provides hints for future work.

2. Motivating Example

This section presents a particular instance of a scheduling problem which is representative for other scenarios. Indeed, modelling for different applications resorts to analogous structures in terms of variables and constraints. This instance has been kept simple for illustration purposes though Section 5 involves a larger number of planned days.

Let us suppose that we need to plan the project of building a house, a modified version of the classical problem exposed in (Marriott & Stuckey, 1998) (page 17). This problem is split into smaller tasks as: laying foundations, build walls and a chimney, place doors and windows, and putting the roof tiles. Each task

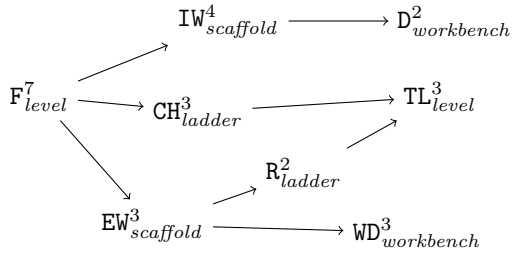
Advantages	
Traditional CP solving and domains	**Proposed approach**
Scheduling problems are traditionally modeled with interval and $\mathcal{FD}$ domains solvers.	Higher level domains such as sets can be efficiently used for those problems.
$\mathcal{FD}$ constraints for ensuring non-overlapping tasks consist of logic combinations of conjunctions and disjunctions.	Task overlapping restrictions can be modelled at a higher level by means of a set-based constraint `disjoint`.
$\mathcal{FD}$ domain variables and constraints naturally fit for problems with tasks running on consecutive days; modelling a more general case requires more complex structures.	General formulations for both consecutive and non-consecutive tasks.
Traditional $\mathcal{FS}$ domain implementations are not widely used for scheduling problems because they lack means for setting precedences among tasks.	The new constraint `<<` allows setting precedence constraints between set variables.
Variable labelling in traditional $\mathcal{FS}$ domains is an expensive computation.	$\mathcal{FS}$ domain cooperates with $\mathcal{FD}$ domain. This allows pruning the $\mathcal{FS}$ search tree earlier. As a result, it allows checking in advance if the scheduling critical path meets the problem constraints.

Usages
Personnel scheduling
Nurse rostering
Production scheduling
Crew assignment problem for airlines
Logistics: Supply chains
Timetabling problems

Relevance
The proposed approach can reduce a great deal the size of the search tree of CP problem formulations, allows the use of a combination of constraint domains, and eases a more compact formulation of the solution. This facilitates the extensive use of constraint programming in a wider set of scheduling problems.

Table 1: Advantages, usages and relevance of this proposal.

has a certain duration that is measured in a number of days and is performed in full days, but may be on non-consecutive days. Some tasks precede others and others require the same resources and therefore cannot be performed simultaneously. Figure 1 shows a precedence graph for this problem where superscripts stand for the number of days needed to complete the tasks, and subscripts for the resource required for performing the task. For instance, task $\texttt{CH}^3_{ladder}$ that corresponds to building the chimney takes three days and requires the use of the ladder. Therefore, it cannot be performed simultaneously with $\texttt{R}^2_{ladder}$, which requires the ladder as well. Furthermore, task `F` (foundations) must be finished before starting the chimney, and `CH` must finish before starting task `TL`.



F: Foundations
IW: Interior Walls
CH: Chimney
EW: Exterior Walls
WD: Windows
TL: Tiles
D: Doors
R: Roof

Figure 1: Building a house

When tasks are performed on consecutive days, coding this problem in the $\mathcal{FD}$ domain is simple. But if we allow tasks to be performed on non-consecutive days (i.e., interruptions are allowed), then the task representation and the overall problem modelling become more complicated.

An alternative model is to use a representation of the problem based on sets of integers. Each task can be represented by a variable containing a set whose elements are the days in which the task is executed. For instance, the set {1,3,4} means that the task is performed in the days 1, 3 and 4.

The issue we encountered when modelling this problem with sets is to establish precedence constraints with the basic set operations ($\subseteq$, $\cup$, $\cap$, ...). To overcome this issue we have designed a new type of constraint Set_1 << Set_2 to force all elements of Set_1 to be less than all elements of Set_2. If we use this new type of constraint, modelling this problem is simple in the finite integer sets domain.

To model this problem, each task is represented as a set variable whose domain ranges between 1 and the number of available days. The duration of a task is the cardinal of the corresponding set, i.e., the number of days the task will take to complete. Tasks that require the same resource are restricted to be disjoint, and precedence constraints are established by the new constraint <<. Next, an ECL^{i}PS^e code excerpt implementing this is reproduced:

```
sched1(Days,Sets) :-

    % List of tasks
    Sets =[F,IW,CH,EW,D,R,TL,WD],

    % Each task in the list Sets is a set of integer numbers from 1 to Days
    intsets(Sets,8,1,Days),

    % The duration of a task corresponds to the cardinal of its set
    #(F,7),#(IW,4),#(CH,3),#(EW,3),#(D,2),#(R,2),#(TL,3),#(WD,3),

    % Precedence constraints
    F<<IW, F<<CH, F<<EW, IW<<D, CH<<TL, EW<<R, EW<<WD, R<<TL,

    % Do not overlap tasks requiring the same resource
    all_disj([F,TL]), all_disj([IW,EW]),
    all_disj([D,WD]), all_disj([CH,R]),

    % Enumerating solutions
    label_sets(Sets).
```

In this code, we use the ECL^{i}PS^e library predicate intsets/4 for declaring a list of set variables, the constraint operator # for setting the cardinal of a set variable, and the predicate all_disj/1 that constrains set variables to be disjoint. Observe that # is the communication bridge between $\mathcal{FS}$ and $\mathcal{FD}$ domains provided by ECL^{i}PS^e. Ground values for set variables that satisfy all constraints in the problem are obtained by means of the predicate label_sets/1. For this example, one of the results of the execution of the goal sched1(16,Sets) corresponds to binding the variable Sets to the list [{1,2,3,4,5,6,7}, {8,12,13,14}, {8,9,10}, {9,10,11}, {15,16}, {12,13}, {14,15,16}, {12,13,14}], which corresponds to the Gantt diagram depicted in Figure 2. Observe that task IW is performed in non-consecutive days. Other solutions produced

by `label_sets` can be obtained using the ECL^{i}PS^e backtracking mechanism.

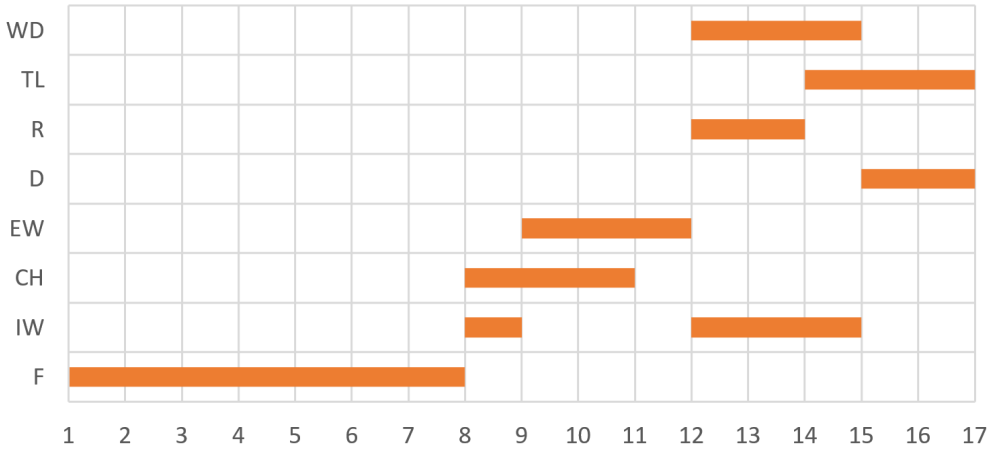


Figure 2: Gantt diagram for a solution to the motivating example.

In addition to modelling this example in a simple way, it is possible to improve the runtime of the goals if the finite domain solver further cooperates with the set solver. To this end, we extend the communication between the $\mathcal{FD}$ and $\mathcal{FS}$ solvers with new constraints besides the cardinal to relate variables between different domains, acting as bridges. For example, let us suppose that the number of days available for all tasks is 14. In this case it is easy to see that tasks `F`, `EW`, `R` and `TL` can not be planned in 14 days because of their precedence constraints, since the sum of the durations of these tasks (15) exceeds the available number of days (14). These tasks form a path which is the *critical path* of the schedule: the longest sequence of tasks in the precedence graph which must be completed on time for the project to complete on due date. Observe that, in the general case of the problem being described, the computation of the critical path is more complicated than the standard problem of finding the longest path, as tasks may split in non-consecutive days and there are resource constraints in addition to precedence constraints. With the constraints in the code excerpt above, the finite set solver is not able to infer in advance that there is no solution and it therefore tests all possible combinations, unless explicit code is added to deal with cases like this one. However, exploring all combinations can be avoided if the information of precedence of these tasks is projected to the $\mathcal{FD}$ domain with some sort of bridge constraint. Let us assume that we could use in $\mathcal{FD}$ constraints the 'function' $min(S)$ that relates its integer outcome with the minimum value in the set variable S. Then, we could write the following $\mathcal{FD}$ constraints in order for the $\mathcal{FD}$ solver to participate in the solving:

$$\textcircled{1} 1 \leq min(F) \qquad \textcircled{2} min(F)+7 \leq min(EW) \qquad \textcircled{3} min(EW)+3 \leq min(R)$$
$$\textcircled{4} min(R)+2 \leq min(TL) \qquad \textcircled{5} min(TL)+3 \leq 14+1$$

In this system of inequalities, the minimum and maximum of each set ranges between 1 and 14. Besides, from the inequalities ①, ② and ③ the inequality $11 \leq min(R)$ is inferred. In ④ the value $min(TL)$ is

therefore at least 13, which does not satisfy the inequality $min(TL) + 3 \leq 15$. So, the solver $\mathcal{FD}$ detects the inconsistency and anticipates failure. This paper develops this idea by introducing the bridge constraints `minSet` and `maxSet` for limiting the minimum and maximum values in set variables.

3. The Primitive Constraints `minSet`, `maxSet`, and `<<`

These constraints are defined by an operational semantics in which the inference rules describe the behaviour of the corresponding constraint propagation. We will use $\mathcal{S}t, \mathcal{S}t'$ to refer to constraint stores. The operational semantics style is based on the works (Azevedo, 2007, Sadler & Gervet, 2008) and the inference rewrite rules have the form:

$$\frac{\textit{Conditions}}{\mathcal{S}t \Rightarrow \mathcal{S}t'}$$

stating that the store changes from $\mathcal{S}t$ to $\mathcal{S}t$' when *Conditions* hold. As in (Sadler & Gervet, 2008), stores contain both $\mathcal{FS}$ and $\mathcal{FD}$ constraints.

We represent set variables by set intervals with a lower and an upper bound considering set inclusion as a partial ordering. In what follows, set domain variables are represented with the letter S and finite domain variables with letters L, M, and N, all of them possibly subscripted. We overload the symbol $\in$ to represent with $S \in [l_S, u_S]$ that a set domain variable S lies within the lattice defined by the interval $[l_S, u_S]$, where l_S and u_S are ground sets ordered by set inclusion, representing the greatest lower bound (abbreviated *lwb*) and the least upper bound (abbreviated *upb*) of S, respectively. Similarly, we use $M \in [l_M, u_M]$ to denote that $\mathcal{FD}$ variable M lies within integer values l_M and u_M.

In the following subsections we present inference rules for the bridge constraints `minSet` and `maxSet` that represent the minimum and maximum bounds for a set, and the precedence constraint `<<`.

3.1. The Bound Constraints `minSet` *and* `maxSet`

These constraints are defined with two parameters: a set of integers S, and an integer M. When the domain of the set is pruned, then the domain of the minimum (maximum, resp.) is pruned accordingly, and vice versa. To obtain this behaviour, the variable M has been defined as an $\mathcal{FD}$ variable whose domain ranges between the minimum of the *upb* of the set variable S and the minimum of the *lwb* if it is not empty. For instance, if S is a set with *lwb* of the form $\{l_1 \ldots l_m\}$ and *upb* of the form $\{u_1 \ldots u_n\}$, the $\mathcal{FD}$ variable M that represents the minimum value in S is defined by the integer interval $[u_1, l_1]$. The integer interval for the $\mathcal{FD}$ variable representing the maximum is $[l_m, u_n]$. If the *lwb* is empty, then the domain of the $\mathcal{FD}$ variables for both the minimum and maximum range between the minimum and maximum values of u_S. Figure 3 graphically illustrates this, where each l_i (u_i, respectively) is an integer value.

Obviously, an empty *upb* for a set implies an empty *lwb*, denoting that the set is the empty set. Usually, this situation corresponds to a failure in the computation, that is, there is no possible element in the set to build a solution. This will be covered later by failure rules.

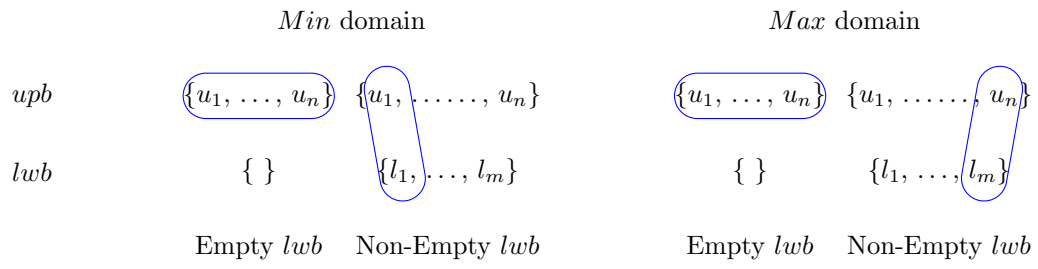


Figure 3: Domain of the variables that represent the minimum and maximum of a set

Next, we present our inference rules for the constraint $\texttt{minSet}(S, M)$. The inference rules for `maxSet` are symmetrical to the ones described for `minSet`.

$$\frac{\texttt{minSet}(S, M) \in \mathcal{S}t}{\{S{\in}[l_S, u_S], M{\in}[l_M, u_M]\} \cup \mathcal{S}t \Rightarrow \{S{\in}[l'_S, u'_S], M{\in}[l'_M, u'_M]\} \cup \mathcal{S}t}$$

where

(1) $l'_S = \begin{cases} l_S \cup \{l_M\} & if\ l_M = u_M \\ l_S & otherwise \end{cases}$ (2) $u'_S = \{x | x \in u_S, x \geq l_M\}$

(3) $l'_M = max(min(u_S), l_M)$ (4) $u'_M = \begin{cases} min(min(l_S), u_M) & if\ l_S \neq \emptyset \\ min(max(u_S), u_M) & otherwise \end{cases}$

Functions *min* and *max* are overloaded to return the minimum and maximum elements of a ground set of integers, respectively. Expression (1) means that the *lwb* of the set is modified only to add the minimum when it is a ground value, that is, when $l_M = u_M$. In (2) the inference rule removes from the set bound u_S the elements smaller than l_M, as the values of the set S cannot be smaller than the *lwb* of the minimum M. In (3) the domain of M is constrained to be at least the minimum value that S can contain, i.e., the minimum element of the ground set, which is the *upb* of S. Finally, in (4) the new value of the *upb* of M, u'_M, is defined as the minimum value between the minimum element of the l_S if it is different from the empty set, or the maximum element of u_S otherwise.

Example 3.1. Given the constraint $\texttt{minSet}(S, M)$, $S \in [\{2, 3, 4\}, \{1, 2, 3, 4, 5\}]$ and $M \in [2, 3]$, the previous rule reduces the lattice of S to $\{2, 3, 4\}, \{2, 3, 4, 5\}$ and the variable M takes the value 2. However, if the domain of M is $[0, 3]$, then its domain changes to $[1, 2]$ and the lattice remains the same. This behavior is graphically displayed in Figure 4.

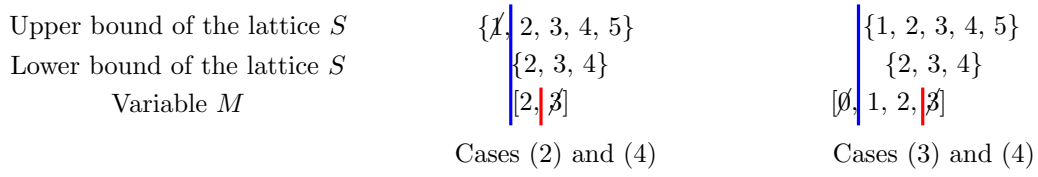


Figure 4: Elements that disappear from the lattice S and the variable M after applying the constraint $\texttt{minSet}(S, M)$

Case (1) arises in the following example: Let S have the domain $[\{2,3,4\},\{1,2,3,4,5\}]$ and let M be defined in domain $[1,3]$. Given the constraints $\texttt{minSet}(S,M)$, $M\# < 2$, the both variables are constrained to be defined in the following domains: $S \in [\{1,2,3,4\},\{1,2,3,4,5\}]$ and $M \in [1,1]$.

Failure rules for $\texttt{minSet}(S,M)$ are defined as follows:

$$\frac{\texttt{minSet}(S,M)\in\mathcal{S}t, u_S = \emptyset}{\{S\in[l_S,u_S], M\in[l_M,u_M]\}\cup\mathcal{S}t \Rightarrow \mathit{fail}} \qquad \frac{\texttt{minSet}(S,M)\in\mathcal{S}t, u_M < min(u_S)}{\{S\in[l_S,u_S], M\in[l_M,u_M]\}\cup\mathcal{S}t \Rightarrow \mathit{fail}}$$

$$\frac{\texttt{minSet}(S,M)\in\mathcal{S}t, l_M > min(l_S)}{\{S\in[l_S,u_S], M\in[l_M,u_M]\}\cup\mathcal{S}t \Rightarrow \mathit{fail}}$$

The case of the first rule corresponds to a set with an empty upper bound. In the second rule, the failing case occurs when all values of the domain of M are smaller than any value of the *upb* of the set S. For example, $\texttt{minSet}(S,M)$ fails if $S \in [\{\},\{4,5,6\}]$ and $M \in [1,2]$. And finally, the third rule states that there is some element in the *lwb* of S that is smaller than any value in the domain of M. For example, when $S \in [\{2,3\},\{1,2,3,4\}]$ and $M \in [3,4]$.

3.2. The Precedence Constraint <<

As already mentioned above, the constraint $S_1 << S_2$ means that all elements of the set S_1 must be less than any element of the set S_2. Note that if one of the two sets is the empty set, then the constraint trivially holds. The operator << is related to the strict partial order between two finite set variables. The next rule states that, given a constraint $S_1 << S_2$, if the *lwb* of S_1 is not empty, then the *upb* of S_2 must be further constrained so that all elements in the *upb* of S_2 are greater than any element in the *lwb* of S_1.

$$\frac{S_1 << S_2 \in \mathcal{S}t, l_{S_1} \neq \emptyset}{\{S_1 \in [l_{S_1},u_{S_1}], S_2 \in [l_{S_2},u_{S_2}]\}\cup\mathcal{S}t \Rightarrow \{S_1 \in [l_{S_1},u_{S_1}], S_2 \in [l_{S_2},u'_{S_2}]\}\cup\mathcal{S}t}$$

where $u'_{S_2} = \{x | x \in u_{S_2}, x > max(l_{S_1})\}$. The following rule is symmetrical to the previous one:

$$\frac{S_1 << S_2 \in \mathcal{S}t, l_{S_2} \neq \emptyset}{\{S_1 \in [l_{S_1},u_{S_1}], S_2 \in [l_{S_2},u_{S_2}]\}\cup\mathcal{S}t \Rightarrow \{S_1 \in [l_{S_1},u'_{S_1}], S_2 \in [l_{S_2},u_{S_2}]\}\cup\mathcal{S}t}$$

where $u'_{S_1} = \{x | x \in u_{S_1}, x < min(l_{S_2})\}$.

Finally, the next rule for $S_1 << S_2$ checks that if there is any element in the *lwb* of S_1 that is larger than some element in the *lwb* of S_2, then the constraint must fail.

$$\frac{S_1 << S_2 \in \mathcal{S}t, l_{S_1} \neq \emptyset, l_{S_2} \neq \emptyset, max(l_{S_1}) > min(l_{S_2})}{\{S_1 \in [l_{S_1},u_{S_1}], S_2 \in [l_{S_2},u_{S_2}]\}\cup\mathcal{S}t \Rightarrow \mathit{fail}}$$

Example 3.2. Assume the sets S_1 and S_2 are defined by the lattices $[\{2\},\{1,2,3,4,5\}]$ and $[\{4\},\{1,2,3,4,5\}]$, respectively. The constraint $S_1 \texttt{<<} S_2$ prunes the domains of S_1 and S_2 to the lattices $[\{2\},\{1,2,3\}]$ and $[\{4\},\{3,4,5\}]$, respectively. However, if the set domains are $[\{3\},\{1,2,3\}]$ and $[\{2\},\{1,2,3\}]$ then the constraint $S_1 \texttt{<<} S_2$ makes constraint propagation to fail.

4. Projecting $\mathcal{FS}$ Constraints into $\mathcal{FD}$

The constraints `minSet` and `maxSet` defined above are *hybrid*, as they relate both $\mathcal{FS}$ and $\mathcal{FD}$ domains, and are used as a communication channel for the cooperation between these domains. As shown in the motivating example, the cooperation between domains is established by creating new constraints on the mate domain using such hybrid constraints that serve as bridges. This process is known as *projection* ((Estévez-Martín, 2015, Estévez-Martín et al., 2012, Estévez-Martín et al., 2009)) and take place whenever a constraint is posted to its corresponding solver.

Specifically, we focus on the projection of the $\mathcal{FS}$ domain into the $\mathcal{FD}$ domain by considering the constraints `minSet` and `maxSet` as communication channels, as well as the cardinality constraint (#) existing already in ECL^{i}PS^e. We propose an additional rule to perform the projection for some $\mathcal{FS}$ primitive constraints. We suppose that, for any set S_i, for any i in the rule, the constraints $\texttt{minSet}(S_i, M_i)$ and $\texttt{maxSet}(S_i, N_i)$ are in $\mathcal{S}t$. We use the syntax $tell(C)$ which indicates that the constraint C has been newly added to the store. The rule is of the form:

$$\frac{tell(C) \in \mathcal{S}t,\ C_{Op} \in \{\subseteq, \cap, \cup, \texttt{<<}, \texttt{all\_disj}\}}{\mathcal{S}t \Rightarrow proj(C) \cup \{C\} \cup \mathcal{S}t \setminus \{tell(C)\}}$$

where C_{Op} represents the constraint operator of C, and *proj* is a function defined for specific cases of C as follows. For each set variable S_i, we use M_i to refer to its minimum and N_i to refer to its maximum:

$$\begin{array}{lcl}
proj(S_1 \subseteq S_2) & = & \{tell(M_1 \geq M_2), tell(N_1 \leq N_2)\} \\
proj(S_1 \cap S_2 = S_3) & = & \{tell(M_3 \geq M_1), tell(M_3 \geq M_2), tell(N_3 \leq N_1), tell(N_3 \leq N_2)\} \\
proj(S_1 \cup S_2 = S_3) & = & \{tell(M_3 \leq M_1), tell(M_3 \leq M_2), tell(N_3 \geq N_1), tell(N_3 \geq N_2)\} \\
proj(S_1 \texttt{<<} S_2) & = & \{tell(N_1 < M_2), tell(M_1 + L_1 \leq M_2), tell(N_1 \leq N_2 - L_2), \\
 & & tell(\#(S_1, L_1)), tell(\#(S_2, L_2))\} \\
proj(\texttt{all_disj}([S_1, \ldots, S_n])) & = & \{tell(\cup_n^{i=1} S_i = S), tell(\#(S_i, C_i)), \forall 1 \leq i \leq n, \\
 & & tell(\#(S, C)), tell(\sum_{i=1}^{n} C_i = C)\}
\end{array}$$

In the first case, the constraint $S_1 \subseteq S_2$ requires that all elements in S_1 are between the extreme values in S_2, and in particular the minimum M_1 and the maximum N_1. The second case projects similar constraints, given that the intersection of two sets S_1 and S_2 must be a subset of both sets. The third case is analogous. The projection for `<<` implements the intuition described in the motivating example: the maximum of S_1 must precede the minimum of S_2, and the difference between the minimum (resp., the maximum) of both sets must be at least the number of elements in S_1 (resp., in S_2). The projection of `all_disj` means: the cardinal

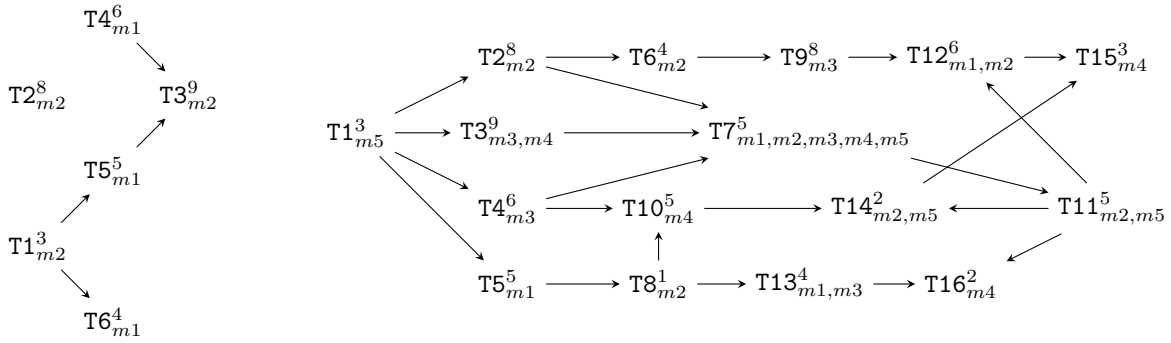


Figure 5: Precedence graphs for sch2 and sch3 benchmarks, respectively.

of the union of the disjoint sets is equal to the sum of their respective cardinal. An interesting application of the last projection is shown below.

Example 4.1. Figure 5 shows the graphs of two additional precedence graphs that will be used in Section 6 as benchmarks. The graph to the left of this figure is composed by six tasks that use two resources evenly distributed among them. The sums of the cardinalities of the tasks of resources m_1 and m_2 are 20 and 15, respectively. If we use for this graph the same approach as described in the motivating example of Section 2, the constraint system will not succeed for 19 days. However, the projection of `<<` on its own is not able to anticipate the failure immediately, because the longest path marked by the precedence constraints amounts 17 days. Fortunately, the projection of `all_disj` anticipates failure much earlier since it takes into account the sum of the cardinals of the sets that share the same resource. As we will see in the experiments (Table 2), the time spent in this example for 18 and 19 days is much longer if `all_disj` projections are disabled.

5. Implementation

We have implemented the constraints described in this work on top of the ECL^{i}PS^e system ((Apt & Wallace, 2007)), an open-source Prolog system for constraint logic programming. Primitive $\mathcal{FS}$ constraint solving is done by the solvers available in ECL^{i}PS^e libraries `ic` and `ic_sets`. We have implemented the additional constraints `minSet`, `maxSet` and `<<` using *demons*, a special low-level mechanism available in ECL^{i}PS^e (Aggoun & et al., 2017). This CLP system extends the standard left-to-right goal selection rule of traditional Prolog systems with a priority-based selection algorithm, providing as a core feature the ability to suspend the execution of a goal at some point during execution, and activate it at a later stage under certain conditions. After its execution, it is automatically re-suspended to be executed again when the firing event occurs anew. Let us describe the `minSet` constraint to illustrate this feature with the help of the following code excerpt:

```
minSet(Set,Min) :-
  #(Set,C), C #> 0,  % to guarantee that Set is not empty.
  suspend(minSet_demon_Min(Min,Set,SMin),0,[Min->constrained],SMin),
  suspend(minSet_demon_Set(Min,Set,SSet),0,[Set->constrained],SSet),
  ...

:- demon(minSet_demon_Min/3).
minSet_demon_Min(Min,Set,Susp) :- ...

:- demon(minSet_demon_Set/3).
minSet_demon_Set(Min,Set,Susp) :- ...
```

This constraint is composed of two demons: one demon for the case in which the integer argument is further constrained (`minSet_demon_Min/3`), and the other one for the case of constraining the set argument (`minSet_demon_Set/3`). When a `minSet` constraint is used in a program, both demons are initially suspended until the domain of the corresponding argument is constrained during the computation, by means of a term such as `[Min->constrained]` in the case of `minSet_demon_Min/3`. Each demon performs the reduction of the domains of `Set` and `Min`, respectively, as described in Section 3 (this code has been omitted in the previous excerpt.) When one of the arguments becomes ground, the corresponding suspended goal is re-activated for the last time and finally removed from the computation.

The precedence constraint `<<` has been implemented in a similar way. Nevertheless, the corresponding demons are awaken just once, when one of the arguments becomes ground, as it has been checked that the activation of these demons introduces a relevant overhead in the overall computation if they are invoked more than once. In addition, the projection mechanism described in Section 4 has been implemented as well, adding to the $\mathcal{FD}$ store the constraints showed in that section, and also `minSet` and `maxSet` constraints when they are required by other constraints generated by the projection mechanism.

The corresponding implementation has been designed to support two modes of use: A *disabled projections* mode which allows to solve hybrid constraints and constraints are resolved in their domains, but do not send complementary information to the mate domain; and an *enabled projections* mode that solves all constraints and projects different kinds of additional information, depending on the user selection, to improve the efficiency of programs. For each particular problem, the user can analyze the trade-off between communication flow and performance gain, and therefore can choose the best option to execute a goal in the context of a given program.

6. Experimental Evaluation

The primitive constraints and projections described in Sections 3 and 4 have been implemented in ECL^{i}PS^e, and such an implementation can be downloaded from `http://gpd.sip.ucm.es/sonia/Eclipse.html`. This implementation has been tested with the scheduling example introduced in Section 2 and three different graphs: the example depicted in Figure 1, and two graphs shown in Figure 5, implemented with predicates similar to the one described in Section 2. The graph to the left of Figure 5 illustrates a simple example with a small number of constraints among tasks, whereas the graph to the right of the same figure shows a more complex graph with a good number of interdependences among tasks, caused by both precedence and resource constraints. Our objective is to evaluate how the primitive constraints and projections presented in this paper improve the efficiency of the execution with respect to a system with no such projections.

The examples have been executed in a system with an Intel© Core™ i7-4510U CPU (2 cores) at 2.00GHz with 8 GB of physical memory and running Ubuntu 14.04.5 and ECL^{i}PS^e 6.1. Numbers correspond to execution times in milliseconds, after computing the average of five executions of the experiments.

Benchmark	No Proj.	Projections				#Sols.
		Precedence	Speedup	Prec.+Disj.	Speedup	
sch_house(14)	142.24	1.88	75.66	1.94	73.32	0
sch_house(15)	343.60	2.45	140.24	2.47	139.11	0
sch_house(16)	195.95	3.23	60.67	3.64	53.83	1
sch_house(17)	390.2	3.71	105.18	4.05	96.35	1
sch_house(18)	575.66	3.89	147.98	4.18	137.72	1
sch2(15)	81.21	1.15	70.62	1.16	70.01	0
sch2(16)	456.81	1.17	390.44	1.15	397.23	0
sch2(17)	1769.49	1.54	1149.02	1.37	1291.60	0
sch2(18)	8553.54	30.33	282.02	1.38	6198.22	0
sch2(19)	43358.66	319.6	135.67	1.44	30110.18	0
sch2(20)	0.69	2.29	-	2.70	-	1
sch3(10)	4.63	0.55	-	0.37	-	0
sch3(11)	6.25	0.61	10.25	0.66	9.47	0
sch3(12)	8.47	1.23	6.89	1.36	6.23	0
sch3(16)	158.01	2.01	78.61	1.90	83.16	0
sch3(20)	72396.21	3.03	23893.14	2.89	25050.59	0
sch3(21)	$> 2\ min$	3.10	-	2.89	-	0
sch3(34)	$> 2\ min$	645.11	-	13.43	-	0
sch3(35)	$> 2\ min$	12245.85	-	14.02	-	0
sch3(36)	$> 2\ min$	$> 2\ min$	-	14.09	-	0
sch3(37)	$> 2\ min$	$> 2\ min$	-	15.10	-	0
sch3(38)	$> 2\ min$	$> 2\ min$	-	$> 2\ min$	-	0
sch3(43)	1.88	34.50	-	37.51	-	1
sch3(44)	2.06	35.18	-	37.98	-	1
sch3(45)	2.03	35.74	-	39.25	-	1

Table 2: Results of experiments: First solution (execution times in milliseconds).

Benchmark	No Proj.	Projections				#Sols.
		Precedence	Speedup	Prec.+Disj.	Speedup	
sch_house(16)	1648.3	18.24	90.37	18.51	89.05	72
sch_house(17)	8632.3	1064.27	8.11	1305.1	6.61	11592
sch_house(18)	69034.45	46988.34	1.47	61132.16	1.13	600992
sch2(20)	$> 2\ min$	5300.71	-	2279.88	-	20790

Table 3: Results of experiments: All solutions (execution times in milliseconds).

Tables 2 and 3 collect preliminary results when computing the first solution and all solutions, respectively, where the column **Benchmark** contains the benchmark and the number of days for the tasks to be scheduled enclosed in parentheses. Column **No Proj.** correspond to the time spent in computing the first solution (all solutions in the second table) of the goal, or until failure if the goal has no solutions, when the projections to the $\mathcal{FD}$ solver are disabled. Columns under **Projections** contain the execution times of the goals enabling some kinds of projections and the gain with respect to the execution time in **No Proj.** Column **Precedence** corresponds to the execution of the goals enabling precedence (`<<`) projections; Column **Prec.+Disj.** corresponds to the execution of the goals enabling both precedence and disjoint (`all_disj`) projections when evaluating the goals. Column **Speedup** contains the speed-up obtained with projections enabled in the column to the left as we propose in this paper. Finally, column **#Sols.** displays, as a reference, the number of solutions that each goal computes. In Table 2 the value 0 corresponds to a failing goal and the

value 1 corresponds to a goal that produces at least one solution. Table 3 shows in this column the number of solutions obtained. Results for `sch3` benchmark are omitted as the first succeeding case ran out of time producing millions of solutions.

The results contained in Table 2 show that the projection of constraints to the $\mathcal{FD}$ domain clearly improves the efficiency of the computation, as it constrains the bounds of the sets early in the computation process. In most cases of failing goals, it checks very fast the insatisfiability of the goal, while in a goal that provides solutions, it prunes the lattices of the set variables dramatically. If projections are disabled, the constraint `<<` is only processed when any of the sets are instantiated and the domains of set variables are constrained mostly at the invocation of `label_sets`. That forces checking the precedence between tasks during labeling, which produces a relevant efficiency loss, as the second column of Table 2 shows.

Columns corresponding to the computation of all solutions in Table 2 show that constraint propagation produces a relevant improvement in efficiency, although it decreases as the number of solutions increases. In such cases, most of the time is spent generating solutions and therefore the gain obtained in pruning the search space is not so evident as in other cases. However, if the number of solutions is very large, the constraints added by projection might even penalize as they must be checked for each solution, but for times in these benchmarks that would go beyond 20 minutes. All in all, for optimization problems (for which many solutions are expected to be checked) this approach might be useful.

The three programs used correspond to different graph configurations. In addition to the running example `sch_house`, the second graph (`sch2`) is a very simple case of scheduling in which nevertheless a relevant amount of computation if projections are disabled, as it is showed in column **No Proj.** In particular, it is remarkable that the execution time grows exponentially for failing goals as the number of days is closer to the smallest successful value (20 in this case). This happens because all computation is performed when `label_sets` tries to fit all the tasks in the available schedule. If precedence projections are enabled, depicted in column **Precedence**, execution times are reduced drastically with very relevant gains. In some cases the gain obtained is more than 10 times with respect to the experiments without projections, as it can be seen in Table 2 in the rows for experiments `sch_house(15)` and `sch_house(18)`, `sch2(16)` to `sch2(19)`, and `sch3(20)` to `sch3(37)`. Nevertheless, when the number of days approaches the smallest successful value, execution times apparently grow exponentially as well. When all projections are enabled (Column **Prec.+Disj.**), the execution time is mainly flat, as the additional constraints allow the system to detect the inconsistency very early in the computation process.

Benchmark `sch3` that corresponds to the graph to the right of Figure 5 has been tested for a larger number of failing cases. These cases are shown graphically in Figure 6. Although there are some cases in which the three experiments behave exponentially and run out of the time limit of 2 minutes, enabling projections makes the constraint system to detect failure very early in the computation for most of the cases. We can see that both **Precedence** and **Prec.+Disj.** produce almost the same results for all cases before 33 days: that behaviour demonstrates that adding disjoint projections does not produce any relevant overhead with

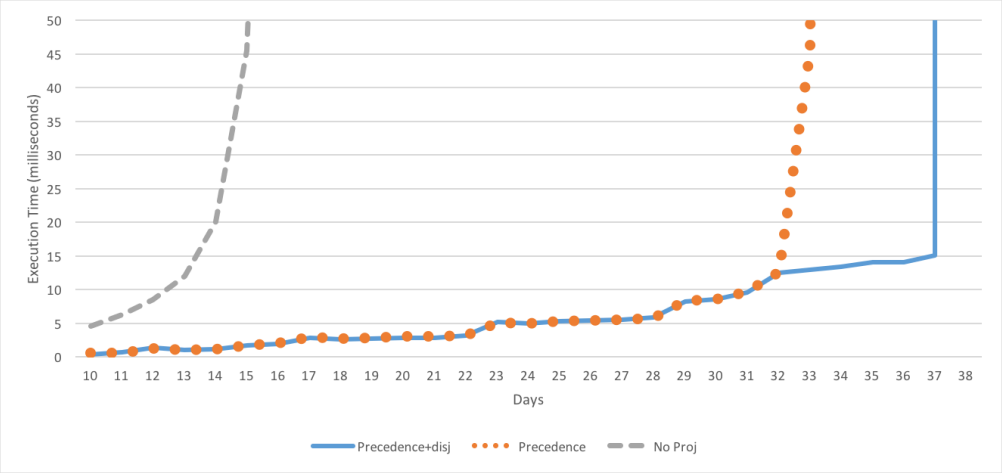


Figure 6: Comparison of execution times for `sch3` benchmark.

respect to enabling precedence projections only. This benchmark also illustrates, with the cases for 33 to 37 days, that `all_disj` projections are very useful for anticipating failure in some cases. In contrast, in the cases which produce solutions, from 43 days on (see Table 2), enabling projections introduces some overhead. Nevertheless, this overhead is paid off by the failing cases.

7. Conclusions and Future Work

We have presented, first, an extension of the toolbox of $\mathcal{FS}$ solvers by adding precedence constraints and, second, how to enhance its solving performance by allowing the cooperation with $\mathcal{FD}$ solvers. As shown in this paper, we have applied our proposal to the constraint solvers available in ECL^{i}PS^e, and, moreover (although not shown in this paper), we have applied this proposal to the constraint functional logic language $\mathcal{TOY}$ that provides a general framework for domain cooperation Estévez-Martín et al. (2012), Estévez-Martín et al. (2009). Different scheduling benchmarks including precedence and resource constraints have been described to test our proposal. Results reveal that enabling projections between domain solvers clearly improves the efficiency of the computation thanks to pruning in advance. Thus, failing branches are detected earlier and discarded for further computations. However, in a number of cases, enabling projections may downgrade performance. So, the user has the ability to either enable or disable them depending on the problem at hand. Further work includes testing our proposal in the $\mathcal{TOY}$ system, which represents a high-level, general-purpose language for solving different optimization problems. In particular, such a system supports model reasoning in the context of such optimization problems. Testing its ability to both neatly expressing models and adapting them along solving is an interesting path to explore. Another industrial application area we will work on is the application of our proposal to formal verification Boulanger (2013), which rely on SMT (e.g., Microsoft's Z3 as used in this setting in Veanes et al. (2017)) and constraint solvers on different domains. One of our current research projects includes the integration of cooperative solvers

for testing pre- and post-conditions on functions, therefore assessing their correctness with respect to such axiomatic conditions.

Acknowledgements

We would like to thank the anonymous referees for helping us in improving this paper with their suggestions. Also, thanks are given to the institutions supporting our work in the form of projects and grants. In particular, this research has been partially supported by the Spanish projects CAVI-ART (TIN 2013-44742-C4-3-R), LOBASS (TIN2015-69175-C4-2-R), MINECO/FEDER project DArDOS (TIN2015-65845-C3-1-R) and Madrid regional projects N-GREENS Software-CM (S2013/ICE-2731) and SICOMORO-CM (S2013/ICE-3006).

References

References

Aggoun, A., & et al. (2017). *ECLiPSe User Manual. Release 6.1*.

Apt, K. (2003). *Principles of Constraint Programming*. New York, NY, USA: Cambridge University Press.

Apt, K. R., & Wallace, M. (2007). *Constraint Logic Programming using Eclipse*. New York, NY, USA: Cambridge University Press.

Azevedo, F. (2007). Cardinal: A finite sets constraint solver. *Constraints*, *12*(1), 93–129.

Azevedo, F., & Barahona, P. (2000). Applications of an extended set constraint solver. In *In Proc. of the ERCIM / CompulogNet Workshop on Constraints*.

Babaei, H., Karimpour, J., & Hadidi, A. (2015). A survey of approaches for university course timetabling problem. *Computers & Industrial Engineering*, *86*(Supplement C), 43 – 59. Applications of Computational Intelligence and Fuzzy Logic to Manufacturing and Service Systems.

Bergenti, F., Dal Palú, A., & Rossi, G. (2009). Integrating finite domain and set constraints into a set-based constraint language. *Fundam. Inf.*, *96*, 227–252.

den Bergh, J. V., Beliën, J., Bruecker, P. D., Demeulemeester, E., & Boeck, L. D. (2013). Personnel scheduling: A literature review. *European Journal of Operational Research*, *226*(3), 367–385.

Bessiere, C., Kiziltan, Z., Rappini, A., & Walsh, T. (2013). A framework for combining set variable representations. In A. M. Frisch, & P. Gregory (Eds.), *Proceedings of the Tenth Symposium on Abstraction, Reformulation, and Approximation, SARA 2013, 11-12 July 2013, Leavenworth, Washington, USA*. AAAI.

Boulanger, J.-L. (Ed.) (2013). *Industrial Use of Formal Methods: Formal Verification*. ISTE & John Wiley and Sons.

Demoen, B., García de la Banda, M. J., Harvey, W., Marriott, K., & Stuckey, P. J. (1999). An overview of HAL. In *Proceedings of CP 1999* (pp. 174–188). Springer volume 1713 of *LNCS*.

Estévez-Martín, S. (2015). *Cooperación entre dominios de restricciones y estrategias de cooperación en el contexto CFLP*. Ph.D. thesis Departamento de Sistemas Informáticos y Computación, Universidad Complutense de Madrid.

Estévez-Martín, S., Correas-Fernández, J., & Sáenz-Pérez, F. (2012). Extending the $\mathcal{TOY}$ System with the ECL^{i}PS^e Solver over Sets of Integers. In *Proceedings of FLOPS 2012* (pp. 120–135). Springer volume 7294 of *LNCS*.

Estévez-Martín, S., Fernández, A. J., Hortalá-González, M. T., Rodríguez-Artalejo, M., Sáenz-Pérez, F., & Vado-Vírseda, R. D. (2009). On the Cooperation of the Constraint Domains $\mathcal{H}$, $\mathcal{R}$ and $\mathcal{FD}$ in $CFLP$. *Theory and Practice of Logic Programming*, *9*, 415–527.

Fernández, A. J., & Hill, P. M. (2004). An interval constraint system for lattice domains. *ACM Transactions on Programming Languages and Systems*, *26*(1), 1–46.

Gervet, C. (1994). Conjunto: constraint logic programming with finite set domains. In *Proceedings of LP 1994* (pp. 339–358). MIT Press.

Gervet, C. (1997). Interval propagation to reason about sets: Definition and implementation of a practical language. *Constraints*, *1*(3), 191–244.

Harjunkoski, I., Maravelias, C. T., Bongers, P., Castro, P. M., Engell, S., Grossmann, I. E., Hooker, J. N., Méndez, C. A., Sand, G., & Wassick, J. M. (2014). Scope for industrial applications of production scheduling models and solution methods. *Computers & Chemical Engineering*, *62*, 161–193.

Hawkins, P., Lagoon, V., & Stuckey, P. J. (2011). Solving set constraint satisfaction problems using ROBDDs. *CoRR*, *abs/1109.2139*.

Hijaz, S., Al-Hujran, O., Al-Debei, M. M., & Abu-Khajil, N. (2015). Green supply chain management and smes: A qualitative study. *Int. J. Bus. Inf. Syst.*, *18*(2), 198–220.

Jaffar, J., & Lassez, J.-L. (1987). Constraint logic programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* POPL '87 (pp. 111–119). New York, NY, USA: ACM.

Jaya Pandey, A. K. S. (2016). Survey on university timetabling problem. In *3rd International Conference on Computing for Sustainable Global Development (INDIACom)*. IEEE.

Mallik, S. (2010). Customer service in supply chain management. *The Handbook of Technology Management: Supply Chain Management, Marketing and Advertising, and Global Management*, *2*.

Marriott, K., & Stuckey, P. J. (1998). *Programming with Constraints: An Introduction*. MIT Press.

Monfroy, E., Rusinowitch, M., & Schott, R. (1995). *Implementing non-linear constraints with cooperative solvers*. Research Report 2747 CRI Nancy.

Mozart Consortium (2013). The mozart programming system. URL: `http://mozart.github.io`.

Naoui, F. (2014). Customer service in supply chain management: a case study. *Journal of Enterprise Information Management*, *27*(6), 786–801.

Niederliński, A. (2014). *A Gentle Guide to Constraint Logic Programming via ECLiPSe*. (3rd ed.). Jacek Skalmierski Computer Studio.

Prud'homme, C., Fages, J.-G., & Lorca, X. (2016). *Choco Documentation*. TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S. URL: `http://www.choco-solver.org`.

Qu, R., & He, F. (2009). A hybrid constraint programming approach for nurse rostering problems. In T. Allen, R. Ellis, & M. Petridis (Eds.), *Applications and Innovations in Intelligent Systems XVI: Proceedings of AI-2008, the Twenty-eighth SGAI International Conference on Innovative Techniques and Applications of Artificial Intelligence* (pp. 211–224). London: Springer London.

Rahimian, E., Akartunali, K., & Levine, J. (2017). A hybrid integer and constraint programming approach to solve nurse rostering problems. *Computers & OR*, *82*, 83–94.

Rossi, F., Beek, P. v., & Walsh, T. (2006). *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. New York, NY, USA: Elsevier Science Inc.

Sadler, A., & Gervet, C. (2004). Hybrid set domains to strengthen constraint propagation and reduce symmetries. In *Proceedings of CP 2004* (pp. 604–618). Springer volume 3258 of *LNCS*.

Sadler, A., & Gervet, C. (2008). Enhancing set constraint solvers with lexicographic bounds. *Journal of Heuristics*, *14*(1), 23–67.

Schulte, C., Mears, C., Rijsman, D., Duchier, D., Konvicka, F., Szokoli, G., Blindell, G. H., Crosswhite, G., Tack, G., Kjellerstrand, H., Scott, J., Moric, L., Lagerkvist, M., Pekczynski, P., Reischuk, R., Lozano, R. C., Gualandi, S., Guns, T., & Barichard, V. (2017). Gecode system. URL: `http://www.gecode.org`.

Stølevik, M., Nordlander, T. E., Riise, A., & Frøyseth, H. (2011). A hybrid approach for solving real-world nurse rostering problems. In *Principles and Practice of Constraint Programming - CP 2011 - 17th International Conference Proceedings* (pp. 85–99).

Yung-yu TSENG, M. A. T., Wen Long YUE (2005). The role of transportation in logistics chain. In *Proceedings of the Eastern Asia Society for Transportation Studies, Vol. 5* (pp. 1657 – 1672).

Veanes, M., Bjørner, N., Nachmanson, L., & Bereg, S. (2017). Monadic decomposition. *J. ACM*, *64*(2), 14:1–14:28.

Yip, J., & Hentenryck, P. V. (2011). Checking and filtering global set constraints. In J. H. Lee (Ed.), *Principles and Practice of Constraint Programming - CP 2011 - 17th International Conference, CP 2011, Perugia, Italy, September 12-16, 2011. Proceedings* (pp. 819–833). Springer volume 6876 of *LNCS*.

Zeghal, F., & Minoux, M. (2006). Modeling and solving a crew assignment problem in air transportation. *European Journal of Operational Research*, *175*(1), 187–209.

Zhou, N. (2011). The language features and architecture of b-prolog. *CoRR*, *abs/1103.0812*.