A CFLP Approach for Modeling and Solving a Real Life Employee Timetabling Problem

Ignacio Castiñeiras and Fernando Sáenz-Pérez
Complutense University of Madrid, Spain
ncasti@fdi.ucm.es and fernan@sip.ucm.es

Abstract

In last years the number of applications of timetabling has grown spectacularly, and different paradigms have risen to tackle these problems. In this paper we present a Constraint Functional Logic Programming (CFLP) approach for modeling and solving a real life optimization employee timetabling problem. We describe the language supported by a particular implementation of the CFLP paradigm. Then, we present the concrete model followed to solve the problem, and we enumerate the advantage our framework provides w.r.t. other approaches. Running results are also reported.

1. Introduction

The Nurse Rostering Problem (NRP) has been extensively studied for more than forty years (Burke et al. 2004). Due to its relevance as a real life problem, it represents the most paradigmatic example of the family of employee timetabling problems. As it is a complex problem, a wide set of techniques has been applied to tackle it, both on formalization design and solving techniques terms. Here we mention Integer Programming (Burke, Li, and Qu 2010), Evolutionary Algorithms (Moz and Pato 2007) and Tabu Search (Burke, Kendall, and Soubeiga 2003) as some approaches. However, due to the constraint oriented nature of the problem, the Constraint Satisfaction Problem (CSP) paradigm (Tsang 1993) becomes a quite suitable framework to the formulation and solving of NRPs. Several programming paradigms have risen to tackle CSPs. First is Constraint Programming (CP) (Marriot and Stuckey 1998), which provides expressive languages for describing the constraints and powerful solver mechanism for reasoning with them. While a CP formulation becomes algebraic (a very abstract programming paradigm) it lacks the benefits of a general constraint programming language. As CP search space can become huge, different techniques as decomposition (Meisels and Kaplansky 2002) and relaxation (Métilvier, Boizumault, and Loudni 2009) has been applied in the solving of NRPs. Another programming paradigm is Constraint Logic Programming (CLP) (Jaffar and Maher 1994), which combines Logic Programming (LP) and CP, providing general purpose languages also equipped with constraint solving. As constraints are basically true relations among domain variables, its integration in the logic field became in a quite natural way.

In this paper we focus on Constraint Functional Logic Programming (CFLP) as another approach to solve employee timetabling problems. CFLP adds constraint solving to the Functional Logic Programming (FLP) framework (Hanus 1994), and attempts to be an adequate framework for the integration of the main properties of Functional Programming (FP) and CLP. Adequation of CFLP(FD) to meet timetabling problems was proposed in (Brauner et al. 2005). The CFLP language presented in (Fernández et al. 2007) combines functional and relational notation, curried expressions, higher-order functions, patterns, partial applications, non-determinism, backtracking, lazy evaluation, logical variables, types, polymorphism, domain variables and constraint composition as some of its features. While its declarative semantics is based on a Conditional Term-Rewriting Logic (CRWL) (González-Moreno et al. 1999), its operational semantics is based on a constrained demanded narrowing calculus (López-Fraguas, Rodríguez-Artalejo, and Vado-Vírseda 2004), making possible to solve syntactic equality and disequality constraints over the Herbrand Universe (H domain constraints), as well as FD constraints, relying on the use of an external solver. An implementation of the system, named TOY(FD), was also presented. Another approach to this framework is Curry (Hanus 1999), and its particular implementation PAKCS (PAKCs), also supporting FD constraint solving by relying on an external solver. As it is quite similar to the TOY(FD) approach, a benchmark comparison between both systems was done in (Fernández et al. 2007).

This paper presents an application of TOY(FD) to the modeling and solving of a real life optimization employee timetabling problem, whose formulation can be seen as a particular instance of the NRP. The structure of the paper is the following: while sections 2 and 3 concern both with language and problem description, Section 4 presents our approach to the modeling of the problem. Section 5 points out the advantages a CFLP(FD) approach offers to face up to this problem, in contrast to using CP(FD) or CLP(FD). Section 6 presents running results for several instances of the problem. Finally, Section 7 reports some conclusions.
2. The TOY(FD) Language

We use constructor-based signatures Σ = {DC, FS}, where DC = ∪∈A DCn resp. FS = ∪∈A FSn are respectively sets of data constructors and defined function symbols with associated arities. As notational conventions, we will assume c, d ∈ DC, f, g ∈ FS and h ∈ DC ∪ FS. We also assume that many countable variables (noted as X, Y, Z, etc.) are available. Given any set X of variables, we will consider the set ExpX(X) of all terms built from symbols in X ∪ DC ∪ FS, and also the set TermX(Σ) of all terms built from symbols in X ∪ DC. Terms (r, e ∈ ExpX(X)) will be called expressions, while terms s, t ∈ TermX(Σ) will be called constructor terms or also data terms. Expressions without variables will be called ground or closed. Moreover, we will say that an expression e is in head normal form iff e is a variable X or has the form c(tn) for some data constructor c ∈ DCn and some n-tuple of expressions tn.

A TOY program consists of datatype, type alias and infix operator definitions, and rules for defining functions. Its syntax is mostly borrowed from Haskell (Peyton-Jones 2002), with the remarkable exception that variables begin with upper-case letters whereas constructor symbols use lower-case, as function symbols do. In particular, functions are curried and the usual conventions about associativity of application hold.

Datatype definitions like data nat = zero | suc nat (which stands as a possible approach to define the natural numbers), define new (possibly polymorphic) constructor types and determine a set of data constructors for each type.

Types τ, τ′,... can be constructed types, tuples (t1,...,tn), or functional types of the form τ → τ′. As usual, τ → associates to the right. TOY provides predefined types such as [A] (the type of polymorphic lists, for which Prolog notation is used), bool (with constants true and false), int for integer numbers, or char (with constants ’a’, ’b’,...).

A TOY program defines a set FS of functions. Each f ∈ FSn has an associated principal type of the form τ1 → ... → τm → τ (where τ does not contain →). Number m is called the type arity of f and well-typedness implies that m ≥ n. As usual in functional programming, types are inferred and, optionally, can be declared in the program.

We distinguish two important syntactic domains: patterns and expressions. Patterns can be understood as denoting data values, i.e., values not subject to further evaluation, in contrast to expressions, which can be possibly reduced by means of the rules of the program. Patterns t, s,... are defined by t := X | (t1,...,tn) | c t1...tn | f t1...tn, where c ∈ DCm, n ≤ m, f ∈ FSn, n < m, and t1 are also patterns. Notice that partial applications (i.e., application to less arguments than indicated by the arity) of c and f are allowed as patterns, which is then called a higher order (HO) pattern, because they have a functional type. Therefore function symbols, when partially applied, behave as data constructors. HO patterns can be manipulated as any other patterns; in particular, they can be used for matching or checked for equality. With this intensional point of view, functions become “first-class citizens” in a stronger sense that in the case of ‘classical’ FP.

Expressions are of the form e := X | c | f | (e1,...,en) | {e1}2, where c ∈ DC, f ∈ FS, and ei are also expressions. As usual, application associates to the left and parentheses can be omitted accordingly. Therefore, e1...en is the same as (...((e1 e2)...) en). Of course, expressions are assumed to be well-typed. First order patterns are a special kind of expressions which can be understood as denoting data values, i.e., values not subject to further evaluation, in contrast to expressions, which can be possibly reduced by means of the rules of the program.

Each function f ∈ FSn is defined by a set of conditional rules f t1...tn = e ≜ l1 = r1,...,lk = rk where (t1...tn) form a tuple of linear (i.e., with no repeated variable) patterns, and l1, r1 are expressions. No other conditions (except well-typedness) are imposed to function definitions. Rules have a conditional reading: if t1...tn can be reduced to e if all the conditions l1 = r1,...,lk = rk are satisfied. The condition part is omitted if k = 0.

TOY includes a polymorphic version of the primitive equality constraint seq :: A → A → bool. The language provides the equality and disequality constraints == and /= to abbreviate seq t s =! true and seq t s =? false (resp.) both constraints first request their arguments to be computed to head normal form, obtaining a variable or a total term. Thus, the symbol == stands for strict equality, which is the suitable notion (see e.g. (Hanus 1994)) for equality when non-strict functions are considered. With this notion, a condition e == e′ can be read as: e and e′ can be reduced to the same pattern. When used in the condition of a rule, == is better understood as a constraint (if it is not satisfable, the computation fails).

A distinguished feature of TOY is that no confluence properties are required for the programs, and therefore functions can be non-deterministic, i.e., return several values for a given (even ground) arguments. For example, the rules coin = heads and coin = tails constitute a valid definition for the 0-ary non-deterministic function coin. Two reductions of coin are allowed, which lead to the values heads and tails. The system try in the first place the first rule, but, if backtracking is required by a later failure or by request of the user, the second rule is tried. Another way of introducing non-determinism in the definition of a function is by adding extra variables in the right side of the rules, as in z: list = [0|l]. Any list of integers starting by 0 is a possible value of z: list. Anyway, note that in this case only one reduction is possible.

The repertoire of FD constraints contains == and /\, that are truly polymorphic. Table 1 includes some of the predefined FD functions and operators supported, where relational constraints support reification (Marriot and Stuckey 1998). The propositional constraint implication posts to the solver a logical implication between the relational constraints A and B, belongs supports domain initialization to a set of values instead of a range as in domain. Finally, sum List Op B imposes a relational constraint between B and the sum of the elements of the list, and count A List Op B imposes a relational constraint between B and the number of elements in the list to be equal to A.
3. Problem Description

Once introduced the TOY(FD) system, the rest of the paper describes a real life optimization employee timetabling problem that can be seen as a particular instance of the NRP. Modeling and solving of this problem motivates the TOY(FD) usability. The problem comes from a technical department of the Spanish public television, where the employed workers must be scheduled to the requested shifts for $n$ days. While the problem was formulated before in (R. González-del-Campo and F. Sáenz-Pérez 2007), here we present new requirements and problem formulation also embodying optimization.

Each day, several workers work at the company. On a working day three workers must work, covering shifts of 20, 22 and 24 hours, respectively. On a weekend two workers must work, covering two shifts of 24 hours. The company employs thirteen workers. Twelve of them are regular workers, and they are divided into three teams of four workers: $\{w_1, \ldots, w_4\}$ belong to $t_1$, $\{w_5, \ldots, w_8\}$ belong to $t_2$ and $\{w_9, \ldots, w_{13}\}$ belong to $t_3$. The extra worker $e$ belongs to no team, and he is only selected by demand for coping with unexpected absences or other circumstances. Optimization arises in the problem because the company must pay regular workers for each extra hour they work, and extra worker hours are paid twice. Optimal schedule minimizes the extra hour payment.

The requirements any valid schedule must hold are the following: Each of the $n$ days of the timetabling must be known as either working days or weekends. Each absence of worker $w_i$ on day $d$ must be provided. Each team must work each three days, working one day and resting two. Any shift of a day must be assigned to a unique worker. Each day, any worker must be assigned to either zero or one of the available shifts. Assigning no shift can be seen as assigning a shift of 0 hours. The extra worker can work any day $d$, but then he must rest on days $d+1$ and $d+2$. $Tight_{t,w}$ is a measure related to the shifts of kind $s$ ($0$, 20, 22 or 24 hours) assigned to the regular workers of team $t$. Let us suppose that, by scheduling the timetabling, workers of $t$ are assigned to $k_1, k_2, k_3$ and $k_4$ shifts of kind $s$. Then, $Tight_{t,w}$ represents the difference between the maximum and the minimum of these $k_s$. Shift distribution is constrained by $Tight$, representing the maximum value any $Tight_{t,w}$ can take. Scheduled timetabling contains $Total_{t,w}$ can take. Schedules timetabling contains $Total/12$ hours per worker implies no extra hour payment, minimizing the optimization function.

4. Problem Implementation

In this section we present our modeling approach for solving the problem, which can be found at http://gpd.sip.ucm.es/ncasti/TOY(FD).zip. Due to its differences with (R. González-del-Campo and F. Sáenz-Pérez 2007) (problem decomposition, optimization, tightness of the shifts, etc.) modeling has been started from scratch. The data of each instance of the problem can be introduced by the user at command line, as well as to be included in a TOY file. First, we devote a subsection to describe the identification of decision variables. Then, next subsection presents a four stage process where the nature of the problem is exploited, decreasing the search space to be explored when looking for an optimal solution.

4.1. Decision Variables

Decision variables are represented by $FD$ variables in a $13 \times n$ table where the columns represent the concrete days and the rows the concrete workers. We refer to this table as timetable. Each variable $timetable_{i,j}$ represents the shift assigned to worker $w_i$ on day $d_j$, and it is initially assigned to the domain values $\{0, 20, 22, 24\}$.

The problem requirements requesting only one team to work each day, and each team to work each three days produce strong dependencies between the timetable variables. First, there is a dependency between the variables of day $d$, as if, for example, $t_1$ works on $d$, the other two teams are precluded from working this day. Thus, their regular workers can be assigned to shifts of 0 hours. Second, there is a dependency between the variables of different days. In the previous example, $t_1$ is also going to work on days $d+3$, $d+6$, etc., binding the variables for the workers of the other two teams to 0. And, also, $t_1$ is not going to work on days $d+1$, $d+2$, $d+4$, $d+5$, etc., so their workers can be bound to 0 on those days. These connections reveal an independency between the different regular working teams. If a variable $v_1$ is susceptible to be bound to 0 by $v_2$ assignment to a shift, then we can model it with an implication constraint. However, this approach does not exploit the independency of the different teams, but treat the whole table as a single problem, making search space to remain at $4^{13\times n}$. Thus, to model and solve an instance of the problem we are going to manage a smaller table, in order to reduce the search space to be explored by reducing both the number of variables and the size of their domains.

4.2. Solving Process

Our modeling approach follows a four stage process to schedule timetable. First stage is Team Assignment, and it concerns with assigning regular workers teams to days.
It is equivalent to trigger the implication constraints associated to the variable dependencies explained before. It allows us to detect the workers susceptible of being assigned to shifts for each day. To explore the whole search space, each feasible assignment of teams to days must be computed. For each feasible assignment, second, third and forth stages are performed. Second stage is Timetabling Generation, and it concerns with the creation of the simpler problem to be solved. On the one hand, after a team assignment, each day only five workers are susceptible of being scheduled to cover the shifts: the team regular workers and e. Thus, this stage creates tt, a 5 × n table which can be mapped to the sub-table of timetable associated to the concrete team assignment performed on first stage. On the other hand, team independence is exploited in tt, obtaining the three sub-problems tt1, tt2 and tt3, which can be solved independently. Third stage is Timetabling Solving, and it concerns with scheduling each tt. Finally, fourth stage is Timetabling Mapping, and it concerns with using the team assignment and the scheduled tt to construct the original timetable, which is outcome as a result to the user.

The main function schedule performs the four stage process to solve each particular instance of the problem. It contains six parameters. First four (N, Abs, DClass and Tight) represent necessary timetabling input data. Last two (W, SS) specify directives to the FD solver. N represents the number of days. Abs is a list of pairs (w, d) representing regular workers planned absences. DClass is a list of N dayType elements representing day classification. W is a labelingOrder representing the order of the variables to be labeled (dayOrder or workerOrder). SS is a labelingStrategy representing a particular variable selection strategy for labeling (firstFail or firstUnbound). Function schedule returns the tuple (Timetabling, ExtraHours) as a result, where ExtraHours represents the number of extra hours associated to the optimal schedule. As to explore the whole search space each team assignment must be computed, collect is applied to function schedule to get all solutions. We devote next four sections to describe each stage of the process.

**Team Assignment** In this stage no shift is assigned to a worker, but regular teams are assigned to days. This allows us to explore a subset of the timetable search space. In concrete, as there are three teams working each three days, there are up to six possible assignments of teams to days (each of them representing 1/6th of the timetable search space). Departing from a team assignment, tt is constructed, scheduled and mapped to timetable. In order to find the optimal schedule, all team assignments must be explored. This behavior is only possible because TOY(FD) allows reasoning with models, where backtracking and multiple search strategies (placed at any point of the problem description) are supported. Let us explain it in detail. Departing from a team assignment solution, the rest of the stages imply the use of new variables, constraints and search strategies.

**Performance of this stage possibly obtain a sub-optimal schedule** (sub-optimal in the sense that it is optimal for that 1/6th timetable search space). Then, next feasible team assignment must be performed, implying backtracking to the Team Assignment stage. This also implies the removing of the variables, constraints and search strategies posted by second, third and fourth stages. Stage Team Assignment takes part on reducing the size of the search space of the timetabling schedule by acting in three different ways:

- **By managing tt instead of timetable**, up to eight variables per day are directly saved, which represent the 60% of the total timetable variables.

- **By associating tt creation and solving to a team assignment**, detection of non-feasible team assignments avoids exploring 1/6th of the remaining search space. Our main aim with this stage is to filter only those team assignments possibly leading to solutions. That is, those that, at least, provide for each day enough workers to accomplish the shifts. Thus, we have to take into account both regular worker absences and the two days resting constraint of e. On the one hand, let us suppose that Abs = [(w1, 4), (w2, 4), (w3, 4)] and that day 4 is classified as workingDay, where 20, 22 and 24 shifts must be assigned to workers. Then, it is for sure that t1 can not be assigned to days 1, 4, etc, as on day 4 there would be only two available workers (w3 and e) to cover three shifts. On the other hand, let us suppose that day 4 is classified as a weekendDay, where two shifts of 24 hours must be assigned. In this setting, t1 can be assigned to days 1, 4, etc, assigning the two shifts of day 4 to w3 and e. But, as t1 requests e for day 4, the latter can not be requested on days 2, 3, 5, 6, due to its two days resting constraint. Two of these days are assigned to team t2 and the other two to team t3. So, for requesting e one day, t1 disallows teams t2 and t3 to request two days from e. If, due to Abs planned absences, e needs to be requested on one of those days, then t1 can not be assigned to days 1, 4, etc. Last but not least, as for each feasible team assignment (at least) two of each three days e must be 0, we can not save but bound (at least) another 5% of the timetable variables. Thus, tt will only contain as much as the 35% of the timetable variables.

- **As e is susceptible of working any day**, it acts as a link between the three working teams. But, as any feasible team assignment entails the resting constraint of e in the whole schedule, we can freely split it into three independent sub-problems, and solve independently each one. As each sub-problem only contains 33% of the variables, we have replaced the effort of solving a complex problem to the effort of solving three exponentially simpler problems.

In summary, initial 4k search space is reduced to 40.35k. But, as we solve the three independent sub-problems independently, the search space becomes 3 × 40 126. As we need to solve the six configurations to find the one minimizing the extra hour payment, the total search space of our approach is 6 × (3 × 40 126) ≡ 18 × 40 126.

Let us now present our approach for modeling this stage. First, our main aim is to assign a concrete team to each day
of the timetabling. Second, we need to ensure that a minimal number of regular workers are available to cover all shifts for each day. Third, we need to ensure the resting constraint. Thus, we need three lists of N new variables: [D1, D2, D3, ... , Dn], where Di represents the concrete team assigned to day i, [A1, A2, A3, ... , An], where Ai represents the number of absences of the regular workers of the selected team to day i and [E1, E2, E3, ... , En], where Ei represents if i is requested to work by selected team to day i. At this point we can see that each Ai and Ei are related to the concrete value Di takes. In particular, for Ai we are only interested in the number of total absences of the regular workers of the selected team Di states. Also, Ei would be bound to 1 only if this team does not provide enough workers to accomplish shifts of this day. To build the list of FD variables, the function genVList is used:

\[
\text{genVList}:: [A] \rightarrow \text{int}_{\text{List}}
\]

workEach3Days:: [A] -> bool
workEach3Days L = true <-=
lengthL < 4
workEach3Days [L1,L2,L3,L1|R] =
workEach3Days [L2,L3,L1|R]

It generates a polymorphic infinite list, and contains a unique rule with extra variables on its right side. By applying take N genVList, lazy evaluation is performed to compute only N new variables. To ensure that each team (D variables) works each three days, function workEach3Days is applied. It recursively checks if a list contains more than three elements, to unify via matching the first and the fourth ones. Then, D becomes [D1, D2, D3, D1, D2, ...]. Finally, a domain constraint attributes (D1, D2, D3) as FD variables with domain value set {1, 2, 3}. An allDifferent constraint (D1, D2, D3) ensures each team work each three days.

Now, information contained in Abs is reorganized to fit it into a new data structure more suitable for posting constraints on A and E. Function orderAbs:: int \rightarrow ([int],int) \rightarrow ([int]) converts Abs in OAbs, a list of lists of integers [[int]] data structure. For each day it contains a list indicating which regular workers are absent. To create OAbs, the elements of tuples in Abs are firstly swapped and then sorted by days. Function sortList orders (with quicksort) its input list:

\[
\text{sortList}:: ([\text{int},\text{int}]) \rightarrow ([\text{int},\text{int}])
\]

sortList [] = []
sortList [(X,Y)|Xs] =
  sortList (filter (ord (X,Y)) Xs)
++ [(X,Y)]
++
sortList (filter (not . ord (X,Y)) Xs)

Its second rule uses first element (X,Y), and relies on both ord and functional composition not . ord to compute lower and greatest elements (resp.):

\[
\text{ord}:: (\text{int},\text{int}) \rightarrow (\text{int},\text{int}) \rightarrow \text{bool}
\]

ord (X,Y) (Z,T) = true <-=
(X > Z)\lor ((X == Z)\land (Y > T))
ord (X,Y) (Z,T) = false <-=
(X < Z)\lor ((X == Z)\land (Y <= T))

In particular, this function is non-deterministic, as its two rules receive the same argument to compute different results (depending on the conditions). Once Abs is sorted, it is filtered by days to compute OAbs.

On the one hand, DClass is used to initialize the domain of A variables. While workingDay implies a domain 0,...,2, weekendDay implies a domain 0,...,3. With this, enough workers to accomplish required shifts is ensured. On the other hand, E variables are initialized to the domain 0,...,1, and the constraint \(\sum (E_i, E_{i+1}, E_{i+2}) <= 1\) is posted on each three consecutive elements to ensure the resting constraint. To relate D with A and E two new [[int]] data structures are built, ATD and ETD, computing absences per team and day and ensuring \(e\) per team and day (resp.). Then, implication constraints relate each possible value \(D_i\) takes (1, 2 or 3) with the binding of \(A_i\) to ATD[i,1], ATD[i,2] or ATD[i,3] (resp.).

Finally, labeling [] (take 3 D) is used to start searching for feasible team assignments.

**Timetabling Generation** This stage performs three actions: create \(tt\), bind to 0 as much as possible of its variables, and split it into \(tt_1, tt_2\) and \(tt_3\) (in order to reduce the effort of solving the problem to the effort of solving three exponentially simpler problems).

\[
tt:: ([\text{int}],\text{int}) \rightarrow (\text{int},\text{int})
\]

tt L = L

tt L R =
  (\text{take 5 genVList}) (N - 1) <= N > 0

Second, to bind as much variables as possible to 0 we create the [[int]] extra data structure ZeroCal. We distinguish between the regular worker absences and the days that \(e\) is not requested. To identify regular worker absences OAbs is filtered to deliver only that absences related to the regular workers of the selected team. To this end, list D is used. Also, for each day for which \(e\) is not requested (\(E_i = 0\)), ZeroCal is increased by 5. Finally, once \(tt\) and ZeroCal are built, both structures are mapped with function putZeros, which zeroes selected variables.

Third, once \(tt\) has been built with the minimum number of variables, it is splitted into the three different sub-problems, one per team. As both N and DClass are requested for solving each sub-problem, they must also be split. Finally, the standard number of hours each regular worker should ideally accomplish is also requested by third stage to solve a single sub-problem. It is computed with the expression \(#/12\) (foldl workHours 0 DClass) == Hours. On the one hand, \(TCY(FD)\) capability of using both carried functions and constraints is used. In this case the constraint \(#/12\) is waiting to be applied to an int. On the other hand, it is applied to the
total number of hours of \(tt\), which is computed by using the FP standard function \(foldl\), and is presented as a higher order application to make the expression more compact. This \(foldl\) uses an initial 0 as accumulator. Function \(work\) adds 66 or 48 to the accumulator, depending on whether it deals with a \(working\) or a \(weekend\) shift. It is important to remark that a common number of standard hours for the twelve regular workers is maintained, instead of creating a specific number for the regular workers of each team.

**Timetabling Solving** As \(tt\), \(N\) and \(DClass\) are split into three independent sub-problems, they can be solved separately. So, to improve performance, it could be possible to send the three sub-problems to different threads. Going even one step further, our solving process could be changed, making Team Assignment stage to collect first all the feasible team assignments, and second stage compute all its \(tt\)'s and sub-problem tables. At that point, we would contain as much as 18 different sub-problems to be solved, and third stage could send them to different threads.

However, \(TOY\) does not support multi-threading up to now. Thus, we solve \(tt\) by solving sequentially \(tt_1\), \(tt_2\) and \(tt_3\). In any case, as \(tt\) optimal schedule is the one which minimizes the extra hour payment, by monotonicity it could be computed as the sum of the optimal schedules for \(tt_1\), \(tt_2\) and \(tt_3\).

Solving each \(tt\) implies several tasks. First, an initial domain on its variables must be imposed. That is, \(\{0, 20, 22, 24\}\) for \(working\) and \(\{0, 24\}\) for \(weekend\). Then, we impose an allDifferent constraint for \(working\), and a count constraint for setting two variables to 24 for \(weekend\). But, if \(e\) is requested this can be simplified. \(e\) is only requested when team working does not provide enough regular workers to accomplish all shifts. In those cases we can ensure that both \(e\) and remaining regular workers are going to work (a value different from 0). Thus, for \(working\) and \(weekend\) the case on which either \(e\) is requested (an unbound variable) or not (bound to value 0) is distinguished. If \(e\) is requested, on \(weekend\) regular workers rows are traversed and the unique variable not bound, say \(X\), is collected. Then, both \(X\) and \(e\) can be bound to 24 without the need of posting \(FD\) constraints.

Second, we take into account \(Tight\), to ensure generated schedules to maintain an homogeneous distribution of shifts. Let us suppose that we have two different schedules implying 36 extra hours. However, while in the first one the 36 extra hours are accomplished by a single regular worker, the second one divides it into 9 hours for the four regular workers of a working team. From the point of view of optimality, both solutions are equal. However, the second one is fairer with the distribution of the work. We let \(Tight\) to be an input parameter to be introduced by the user in order to make homogeneous the distribution of the work. As we said in problem description, we measure \(Tight\) by each working team and each shift. If we make \(Tight = 0\) then all the regular workers of all the teams must be assigned to the same number of 0, 20, 22 and 24 shifts. If we make \(Tight = 1\), then we allow each regular working team \(\{w_i, w_{i+1}, w_{i+2}, w_{i+3}\}\) to assign different shifts to their workers. For example, let us suppose that the maximum number of 20 hours shifts assigned to a worker is \(k_1\), and the minimum is \(k_2\). Then \(Tight = 1\) constraints that the difference between \(k_1\) and \(k_2\) is not greater than 1. It is important to remark that we have decided to include \(Tight\) as an input parameter, instead of wrapping it within the cost function. Our objective is to minimize the extra hour payment, not \(Tight\).

Function \(ts\) is used to constrain with \(Tight\) a single working team and a single shift:

\[
\begin{align*}
\text{ts} & \quad \text{T} \quad \text{S} = \text{true} \iff \text{count} \quad \text{S} \\
& \quad \text{W1L} \quad \text{(#=)} \quad \text{A} \\
& \quad \text{count} \quad \text{S} \quad \text{W2L} \quad \text{(#=)} \quad \text{B} \\
& \quad \text{count} \quad \text{S} \quad \text{W3L} \quad \text{(#=)} \quad \text{C} \\
& \quad \text{count} \quad \text{S} \quad \text{W4L} \quad \text{(#=)} \quad \text{D} \\
& \quad \text{domain} \quad \text{[A#-B, A#-C, A#-D, B#-C, B#-D, C#-D]} \quad \text{(-T)} \quad \text{T}
\end{align*}
\]

In particular, expression \(ts\) \(T\) (transpose \(SubTT\)) \(WS\) is used to, first, transpose the list \(tt\), in order to access its variables by workers, instead of by days. \(ts\) creates four new \(FD\) variables \(SW1, ..., SW4\) representing the amount of shifts assigned to each of the regular workers of the working team. Second, to impose an homogeneous distribution we impose the difference of this variables to be in the domain \((-T), ..., T\). Here we want to remark the \(TOY(FD)\) functional notation capabilities, that allow us to directly express the subtraction of each pair of variables in the domain constraint, instead of creating new variables.

Third, we need to generate the cost function \(EHours = \sum (X_1 \# + X_2 \# + X_3 \# + X_4 \# + (2 \# \ast X_5))\). While \(X_1, ..., X_4\) represent the extra hours of the regular workers, \(X_5\) is the number of extra hours of \(e\). Again we need to use \(transpose\) \(SubTT\) as we need to access its variables by workers. In this setting, \(X_5\) is computed with a \(sum\) constraint of \(e\) working hours. In the case of regular workers, we need four more \(sum\) constraints. But, in this case the generated \(FD\) variables must be compared with standard number of working hours. Thus, we use \(implication\) constraints. If a worker has done extra hours, then \(X_i\) represents that number of extra hours. But, if the worker has done less than standard hours, then \(X_i = 0\).

Finally, a new \(labeling\) over \(tt\), is applied. The input parameters \(\text{\#}\) and \(SS\) are taken into account to follow a particular search strategy. \(SS\) selects the particular variable selection strategy: \(firstUnbound\) or \(firstFail\). \(\#\) enumerates the \(tt\) variables by days or by workers. The \(labelingOrder\) is relevant for both labeling strategies. As \(tt\) variables contain few values in their domain, many ties will be produced when selecting variables by first fail, and the variable enumerated first will be the one selected. All the cases contain the option to \(Minimize\) \(EHours\), ensuring that optimal schedule (minimizing the extra hour payment) is selected as a solution.

**Timetabling Mapping** By scheduling the three \(tt\), (binding all their variables) all the variables of \(tt\) are indirectly bound. We recall here that the user is requested to use
collect to trigger exploration of all feasible team assignments. Users can then either check each sub-optimal solution or map a minimum function to the collected solutions to select the optimal one.

5. Paradigm Comparison

Let us discuss about the benefits of modeling this problem in CFLP(FD). First, we summarize the advantages the logic component offer us in CLP(FD) and CFLP(FD) in contrast to CP(FD). Then, we point out the advantages the functional component offer us in CFLP(FD) in contrast to CLP(FD).

CP(FD) is not thought to reason with models. First, the constraint set is imposed and then several labeling search strategies over disjoint variable sets can be declared. On the one hand, both tasks cannot be mixed. On the other hand, the set of imposed constraints is static and not meant to change. Both CLP(FD) and CFLP(FD) add to CP(FD) allow this task to be dynamic. In addition, the logic component eased the modeling and solving of the employee timetabling problem. We have pointed out the benefits of splitting this problem: Problem division, early detection of unfeasible solutions and reduction of FD variables. Our approach consists of posting a subset of the constraints of the problem and performs a labeling search strategy (obtaining a feasible team assignment). Then, departing from each obtained partial solution, we dynamically create the sub-problems to be solved. Finally, we post the rest of the constraint set over these sub-problems, and we use parameterized labeling strategies to find the optimal schedule. Under this scenario, our model can not be directly translated into a classical CP approach with a requested isolation between constraint and search descriptions. To follow our CFLP(FD) approach described before, in CP(FD) we would need to create several CP models, and a general script to coordinate them. If we decide to keep it in a single model, then we need to create the table of 13 × n FD variables to represent assigned shifts. But, only by doing that, we reach again the 4^{13*n} initial search space we depart from in our CFLP(FD) implementation (and that we reduce dramatically with our approach).

CFLP(FD) provides a sugaring syntax for LP predicates and thus any pure CLP(FD)-program can be straightforwardly translated into a CFLP(FD) program (Fernández et al. 2007). But, in addition, CFLP(FD) includes a functional component. This has helped us easing the task of modeling the calendar problem, as some of the extra capabilities CFLP(FD) enjoys have been used: (a) Types. Our system is strongly typed, and it has eased the modeling development and maintenance, by finding bugs at compile-time. (b) Poly-morphic variables. (c) Lazy evaluation. Function arguments are evaluated to the required extent (the call-by-value used in LP vs. the call-by-need used in FP) (Peyton-Jones 1987). By this, we have managed infinite structures. (d) Call-time choice, allowing shared terms to be computed just once. (e) Higher order. Its use has simplified modeling. (f) Currying. Both curried functions and constraints have been used. (g) Functional notation, by which we have saved some FD variables. (h) Non-determinism rules have been used in some functions. (i) Function composition. It has allowed us to use more compact expressions.

6. Performance

In this section we present results for solving three instances of the problem. Each instance is assumed to start on Monday, where any week contains five working days followed by weekend. The three instances solve one, two and three weeks respectively, where the planned absences are shared among the instances: \( \text{Abs} \equiv \{ (w_1, 1), (w_2, 1), (w_3, 1), (w_5, 6), (w_6, 1), (w_7, 1), (w_7, 6), (w_{10}, 1), (w_{10}, 6), (w_{11}, 1), (w_{12}, 1), (w_{12}, 6) \} \). Thus, there will be only two feasible team assignments, as on day 1 only t1 can work. The two feasible team assignments correspond with permutation of t2 and t3 over days 2 and 3, respectively. Each assignment will request e on days 1 and 6. Tight \( \equiv 1 \) is used allow little differences between the regular workers of each team. The last two parameters specify the search strategy to be accomplished, as defined in Section 4.

Section 5 pointed out the benefits of using CFLP(FD) for modeling the problem. Now we are also interested in measuring the performance impact of its underlying lazy narrowing mechanism, in order to test if the framework maintains a good trade-off between expressivity and efficiency. Lazy narrowing elapsed time is computed via subtracting original instance elapsed time to the FD constraint solving elapsed time. To compute the latter one we isolate the concrete set of FD constraints being sent to the solver. Once obtained this set we re-formulate the instance via a new program, which only contains a function explicitly posting it. Next table summarizes the results. First column represents the number of weeks of the instance. As second and third columns represent the elapsed time of the original and constraint-only programs respectively (measured in milliseconds), the lazy narrowing overhead is presented on fourth column by simply computing their ratio. While for each measurement we have applied our four possible combinations of variable order and strategy selection explained in Section 4, in this table we only present the results of the fastest, to which we devote last column. Benchmarks have been run in a machine with an Intel Dual Core 2.4GHz processor and 4GB RAM memory and Windows XP SP3. \( TOY(FD) \) has been developed with SICStus Prolog, version 3.12.8, and ILOG CP 1.6, with ILOG Concert 12.2 and ILOG Solver 6.8 libraries. Microsoft Visual Studio 2008 tools were used for compiling and linking the \( TOY(FD) \) interface to ILOG CP.

<table>
<thead>
<tr>
<th>Weeks</th>
<th>( TOY(FD) )</th>
<th>( FD )</th>
<th>Overhead</th>
<th>Strat.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.027</td>
<td>859</td>
<td>20%</td>
<td>dO</td>
</tr>
<tr>
<td>2</td>
<td>3.404</td>
<td>3.100</td>
<td>10%</td>
<td>wO</td>
</tr>
<tr>
<td>3</td>
<td>199,359</td>
<td>195,609</td>
<td>2%</td>
<td>wO</td>
</tr>
</tbody>
</table>
largest instance being measured (which is solved in more than three minutes). So, as problem scales, the impact of the lazy narrowing decreases, becoming very few or instead irrelevant for difficult CSPs involving long time during search.

7. Conclusions

CP is a suitable paradigm to tackle timetabiling problems, as it provides expressive languages for describing the constraints and powerful solver mechanism for reasoning with them. However, it neither provides general purpose programming features nor a modeling reasoning framework, as CLP and CFLP frameworks do. In this paper we have presented an employee timetabling problem that puts some evidence of the benefits these two properties provide for the modeling and solving of this particular problem. On the one hand, users can benefit from the extra expressivity those frameworks provide, without losing constraint solving capabilities. On the other hand, modeling framework allows us to dramatically reduce the search space to be explored for obtaining the optimal schedule, without supposing a big overload due to its underlying operational mechanism.

While our problem can not be directly translated to CP using a classical approach (due to the isolation between constraints and search strategies), there exist several ways of exploiting problem independence, relying on a distributed framework (Meisels and Kaplansky 2002). As future work we plan to make an in-depth comparison of $CP(FD)$, $CLP(FD)$ and $CFLP(FD)$ for solving our problem. We plan to model the problem in CP, both with a distributed approach and with the classical approach. Also, we plan to model the problem in CLP. It would put much more evidence between the trade-off of expressivity and efficient our framework is. On the one hand, when modeling in $CLP(FD)$ we can not take advantage of the expressivity of our system. On the other hand, as our system is built on top of SICStus, we can directly assess the impact of our lazy narrowing operational system.

References