Pascal Van Hentenryck

Constraint Programming

Volume 5, nº 2-3 (2024), p. 139-159.

https://doi.org/10.5802/roia.76

© Les auteurs, 2024.

Cet article est diffusé sous la licence
Creative Commons Attribution 4.0 International License.
http://creativecommons.org/licenses/by/4.0/

La Revue Ouverte d'Intelligence Artificielle est membre du Centre Mersenne pour l'édition scientifique ouverte
www.centre-mersenne.org
e-ISSN : 2967-9672

Revue Ouverte d'Intelligence Artificielle
Volume 5, n° 2-3, 2024, 139-159

Constraint Programming

Pascal Van Hentenryck[a]

[a] NSF AI Institute for Advances in Optimization, H. Milton School of Industrial and Systems Engineering, Georgia Institure of Technology, Atlanta, Georgia (USA)
E-mail: pvh@gatech.edu.

Résumé. — Cet article relate l'histoire de la programmation par contraintes, une des contributions scientifiques les plus remarquables et durables de la carrière d'Alain Colmerauer. C'est une histoire personnelle, pour avoir vécu et contribué aux évenements rapportés ici.

Mots-clés. — Programmation logique, Prolog, programmation par contraintes.

It is with tremendous pleasure that I am writing this article about Alain Colmerauer and Constraint Programming (CP). Alain's scientific vision, his originality, and his relentless pursuit of elegance and simplicity had a profound impact on my career and has remained with me to this day. It has become second nature for me to rewrite computer programs until they are beautiful, to think declaratively, and to pursue what is now called use-inspired fundamental research.

This article is a story, in fact multiple stories. It tells the story of how, I believe, constraint programming came to be. It may not be the real story (since I do not pretend that I could read Alain's mind) but it is probably close. It tells this story using some of Alain's words, some pictures (Alain loved pictures), and, of course, some of the most elegant programs ever written. It is an unfinished story because the impact of Alain is broad and endures despite new trends and scientific developments. The article captures this legacy through some recent examples that somehow Alain had anticipated. It is also the story of a man, who gave opportunities to so many, because he was incredibly generous with his ideas.

The article tells this story in reverse, as it should be with Prolog and its descendants. It starts with the fundamental impact of constraint programming, and some examples and applications of this unique technology. It then goes back in time, to explain how constraint programming came to be, starting with Prolog obviously and Prolog II. Duncan Watts wrote a book called "Everything is obvious once you know the answer" [37]. The invention of constraint programming may indeed seem obvious retrospectively. But it required the genius and search for elegance of Alain to reveal it (and, may be, a bit of luck but that bit is pure speculation of my part). The article finishes with some broad impact of Alain's research and some final words that convey my deep appreciation to a remarkable man.

1. The Impact of Constraint Programming

Constraint programming is now used over the world to solve problems in areas as diverse as aircraft assembly, semiconductor fabrication plant scheduling, port logistics, manufacturing (steel, textile, metals, assembly, paint shops ...), workforce scheduling, integrated facility management, and automated bio-labs. An 2010 issue of IEEE software stated that "*the application and importance of CP has grown remarkably in the past two decades*". In August 2011, the OR/MS magazine, which serves the Operations Research community, asserted that constraint programming is a "*must-have tool for any O.R. Practitioner's toolkit*". Prior to that, in 2008, Michael Trick, who was at that time a professor in the business school at CMU, wrote that "*if you only knew optimization from 10 years ago, you probably don't have the techniques needed to solve real-world sport scheduling problems. The following do make a big difference (and are much more recent ideas*

- *Complicated variables*;
- *Large neighborhood search*;
- *Constraint programming (ideally combined with integer programming).*"

Note that large neighborhood search can also be traced to the constraint programming community [27], although it has become a general-purpose optimization technique as discussed later.

The constraint programming community has been a fertile ground for innovation for several decades. In the 2000s, various efforts in the community started to integrate techniques to learn from failures. They led to impressive results in scheduling. For instance, for the Resource Constrained Project Scheduling Problem (RCPSP), (probably) the most studied scheduling problems, constraint programming closed 71 open problems, and solved more problems in 18s then the previous state of the art could in 1800s [25]. For the RCPSP/Max problem, which features more complex precedence constraints, constraint programming closed 578 open instances of 631 recreates or betters all best-known solutions by any method on 2340 instances except 3.

These are remarkable achievements, but the way they are obtained is even more impressive and captures the beauty of a scientific domain that has now bridged computer science and operations research. The rest of this section briefly describes some of the modeling features of constraint programming and how they can be used to solve industrial applications.(1) A short introduction to constraint programming can be found in [35] and some of the material here is adapted from that review.

1.1. Modeling with Constraint Programming

The Steel Mill Slab problem consists in producing n orders using slabs of specified sizes. Each order o has a color c_o and a weight w_o that represents the capacity it takes.

(1)The rest of this section can be skipped by those readers familiar with constraint programming and interested only in the historical perspective.

```
int nbOrders = ...;
range Orders = 1..nbOrders;
int nbSlabs = ...;
range Slabs = 1..nbSlabs;
int nbColors = ...;
range Colors = 1..nbColors;
int nbCaps = ...;
range Caps = 1..nbCaps;
int capacities[Caps] = ...;
int weight[Orders] = ...;
int color[Orders] =...;
int maxCap = max(i in Caps) capacities[i];
int loss[c in 0..maxCap] =
   min(i in Caps: capacities[i] >= c) capacities[i] - c;

dvar int x[Orders] in Slabs;
dvar int load[Slabs] in 0..maxCap;

minimize sum(s in Slabs) loss[load[s]]
subject to {
  packing(x,weight,load));
  forall(s in Slabs)
     sum(c in Colors) (or(o in Orders: color[o] == c) (x[o] == s)) <=
         2);
}
using {
  forall(o in Orders) by (x[o].getSize(),-weight[o]) {
     int ms = max(0,maxBound(x));
     tryall(s in Slabs: s <= ms + 1}
        x[o] = s;
}
```

Figure 1.1. A Constraint Programming Model for the Steel Mill Slab.

Each slab has a size that must be chosen from a finite set $\{u_1, u_2, \ldots, u_k\}$. A solution is an assignment of orders to slabs such that (1) the total weights of the orders in a slab must not exceed the slab size and (2) the orders in a slab can be of at most two different colors. The objective is to minimize the sum of losses, i.e., the space left in the slabs used in the solution.

Figure 1.1 presents a constraint programming model for the Steel Mill Slab problem following [13]. The model uses two sets of decision variables: variable $x[o]$ specifies the slab assigned to order o, while variable $load[s]$ represents the load of slab s, i.e., the sum of the sizes of all orders assigned to s. Once the load of a slab is known, it is easy to compute its loss: simply take the smallest capacity supporting the load. The instruction

```
int loss[c in 0..maxCap] =
   min(i in Caps: capacities[i] >= c) capacities[i] - c;
```

computes an array of losses for each possible capacity, while the objective function uses the element constraint to compute the loss of each slab. It is important to point out that a slab with no order incurs no loss. The global `packing` constraint is used to compute the load of the slabs: this constraint is equivalent to

```
forall(s in Slabs)
  sum(o in Orders) weight[o] * (x[o] == s) == load[s]);
```

which uses reified constraints of the form $x[o] == s$ (also called an indicator constraints in mathematical programming). The global `packing` constraint performs a much stronger pruning by reasoning about the loads of all slabs and all the orders globally. The second set of constraints are reified constraints specifying that the orders in a slab can be of at most two different colors.

A SIMPLE SEARCH PROCEDURE. — A search procedure with symmetry breaking is included in the model, in the syntax of the original OPL system: it uses the nondeterministic control structure `tryall` to try all the possible slabs for each order o. It builds on the the traditional variable assignment of constraint programming: iterate over all variables and assign them a value in a nondeterministic fashion:

```
forall(o in Orders)
   tryall(s in Slabs}
      x[o] = s;
```

When a traditional depth-first is used, the search assigns a value to each variable. Upon backtracking, the search considers the next possible value for the variable. The search procedure can be upgraded to specify the order in which to assign the variables and the values. For instance, the search procedure

```
forall(o in Orders) by (x[o].getSize(),-weight[o])
   tryall(s in Slabs}
      x[o] = s;
```

considers first the order variable with the smallest domain and, in case of ties, the order whose weight is the largest. Choosing the variable with the smallest domain is called the *first-fail principle*, which captures the fact that the variables with small domains are likely to be the hardest to assign and labeling them first results in smaller search trees. This heuristic is feasibility-based and contrasts with heuristics in mathematical programming which are often driven by the objective function (through the linear relaxation). Note also that the search procedure tells the solver what to focus on: it is not necessary to assign values to the *load* variables.

Breaking Symmetries. — The model as presented has many symmetries: indeed, all the slabs are equivalent initially and the search spends considerable time exploring them. The search procedure presented in the model breaks those symmetries. For each order to assign, this search procedure only considers slabs in which some orders have already been placed as well as one additional empty slab. The instruction

```
int ms = max(0,maxBound(x));
```

computes the already used slabs (i.e., $1..ms$), while the `tryall` instruction now considers only the slabs in $1..ms{+}1$. This elimination of value symmetries has been studied theoretically by [32] and used in many constraint programs, e.g., for graph coloring, scene allocation, deployment of serializable services to name only a few. Note also that there are other symmetries that could be broken: two slabs with the same capacities and the same colors are also symmetric, but it is not necessary to break these symmetries to achieve good performance in this model.

Large Neighborhood Search (LNS). — Constraint programming solvers often support LNS (mentioned earlier by Michael Trick), either natively or through search combinators that can be used to express specific incarnations of this search technique. Here is an example of how this can be expressed using some of the search combinators in Objective-CP [34]:

```
repeat {
  limitFailures(failureLimit)
    forall(o in Orders) by (x[o].getSize(),-weight[o]) {
      int ms = max(0,maxBound(x));
      tryall(s in Slabs: s <= ms + 1}
        x[o] = s;
    }
onRepeat {
     s = getBestSolution();
     if (s != nil)
       forall(o in Orders)
         if (random() <= 0.2)
           x[o] = getValue(s,x[o]);
}
```

The overall search is a repeat loop, whose body executes the search procedure presented earlier. Upon reaching its limit on the number of failures or solving the sub-problem optimality, the search executes the `onRepeat` body, before the next iteration. The onRepeat code retrieves the best solution found so far, say s, and fixes the value of each order variable to its value in s with a probability of 0.2. This defines the next sub-problem to solve. In practice, this repeat loop is executed until it reaches a specific time limit.

1.2. Scheduling with Constraint Programming

The second model in this section illustrates some of the scheduling and routing capabilities of constraint programming. Modern CP solvers have dedicated features to model scheduling problems at a high level. These modeling abstractions are then implemented through sophisticated global constraints that includes edge-finder and energy-reasoning for disjunctive and cumulative constraints. It is impossible to do justice to all the modeling and computational contributions of constraint programming in this domain. The following example should simply give a flavor of the type of scheduling and routing abstractions available in constraint programming.

Robot Scheduling and Routing. — The problem arises in manufacturing where a robot must visit a number of stations. Each station is associated with a service time (the time the robot must spend to service the station). Each station is also associated with a time window `[es,ls]` that specifies when service can start (i.e., the robot cannot start service before `es` or after `ls`). The goal is to minimize the travel time taken by the robot to visit all the stations. There is an asymmetric matrix available that gives the travel time between every two stations. The asymmetric travel times complicate the optimization.

Figure 1.2 depicts a high-level constraint programming model for this problem. The first couple of instructions describe the data. The instruction

```
{triplet} transition = {<i,j,time[i,j]> | i in Stations, j in Stations};
```

defines the transition matrix between every two stations as a set of tuples. This will be useful to state some of the constraints.

The decision variables in this model are particularly interesting. The instruction

```
dvar interval service[s in Stations] size data[s].service;
```

associates an *interval variable* with each station. Interval variables in constraint programming have a start date, an end date, and a duration (all decision variables), which are linked by a constraint specifying that the end date must be equal to the start date plus the duration. In this case, the interval variables represent when the robot serves each station, i.e., from the beginning to the end of the service. The next decision variable is even more interesting. The instruction

```
 dvar sequence route in all(s in Stations) service[s]
              types all(s in Stations) s;
```

defines a *sequence variable* that specifies the order in which the robot visits all the stations. The sequence variable collects all the interval variables and associates each station with a type which is simply its index `s` in this case. The goal of the model will be to determine the sequence in which the robot visits all the stations. The constraint

```
int nbStations = ...;
range Stations = 1..nbStations;
tuple stationData { int service; int start; int end; };
stationData data[Stations] = ...;
int time[Stations,Stations] = ...;

tuple triplet {int station1; int station2; int time;};
{triplet} transition = {<i,j,time[i,j]> | i in Stations, j in Stations};

dvar interval service[s in Stations] size data[s].service;
dvar sequence route in all(s in Stations) service[s]
              types all(s in Stations) s;

minimize sum(s in Stations) time[s,typeOfNext(route,service[s],s)];

constraints {
  noOverlap(route,transition);
  forall(s in Stations)
     data[s].start <= startOf(service[s]) <= data[s].end;
}
```

Figure 1.2. A Constraint Programming Model for Robot Scheduling and Routing in a Manufacturing Plant.

```
forall(s in Stations)
  data[s].start <= startOf(service[s]) <= data[s].end;
```

specifies that the start of service for each station is within the prescribed time window. The constraint

```
noOverlap(route,transition);
```

has two purposes. First, it specifies that the services of each station cannot overlap in time (since the robot can only be in one place at any time). Second, it imposes the transition constraints between the interval variables using the transition matrix. In particular, if the robot serves a station i at time t and then goes to station j next, the service at station j cannot start before the end of the service at i plus the transition time `time[i,j]`. Finally, the objective function

```
minimize sum(s in Stations) time[s,typeOfNext(route,service[s],s)];
```

minimizes the travel time of the robot. It uses an element constraint again and the function `typeOfNext` which is particularly useful for modeling sequence-dependent

costs. Indeed, the expression `typeOfNext(route,service[s],s)`, where `route` is the sequence variable associated with the robot, returns a decision variable that represents the type of the interval variable that follows interval variable `service[s]` in the sequence. When used in CP Optimizer, this model uses large neighborhood search to find high-quality solutions quickly, as well as various relaxations to find lower bounds.

Scheduling Autonomous Trucks. — This section reviews an application of constraint programming to a complex application and case study. Self-driving trucks are expected to fundamentally transform the freight transportation industry. Morgan Stanley estimates the potential savings from self-driving trucks at $168 billion annually for the United States alone [15]. Additionally, autonomous transportation may improve on-road safety, and reduce emissions and traffic congestion [28, 29]. The most likely scenario for the adoption of autonomous trucks by the industry, according to some of the major players, is the *transfer hub business model* [23, 26, 36]. An Autonomous Transfer Hub Network (ATHN) makes use of autonomous truck ports, or *transfer hubs*, to hand off trailers between human-driven trucks and driverless autonomous trucks. Autonomous trucks then carry out the transportation between the hubs, while regular trucks serve the first and last miles (see Figure 1.3). Orders are split into a first-mile leg, an autonomous leg, and a last-mile leg, each of which served by a different vehicle. A human-driven truck picks up the freight at the customer location, and drops it off at a nearby transfer hub. A driverless self-driving truck moves the trailer to a transfer hub close to the destination, and another human-driven truck performs the last leg. The ATHN applies automation where it counts: Monotonous highway driving is automated, while more complex local driving and customer contact is left to humans. A global Consultancy firm [23] estimated that operational cost savings range between 22% and 40% in the transfer hub model, based on cost estimates for three example trips. A recent white paper [24] studies whether these savings can be attained for actual operations and realistic orders. It models ATHN operations as a scheduling problem and uses a constraint programming model to minimize empty miles and produce savings from 27% to 40% on a case study in the Southeast of the United States.

A recent paper [21](2) conducted a realistic case study based on data from Ryder System, Inc. (Ryder), one of the largest transportation and logistics companies in North America. It presented a method that optimizes ATHN operations and improves hub utilization even for large-scale systems by combining the strengths of MIP and CP. The MIP model is used to generate routes and an initial schedule, while the CP model is used to shift the schedule and minimize the hub capacities. The rest of this section presents the CP model reusing the presentation from [21].

The CP model is based on $\mathcal{K}$; the set of routes obtained from MIP. Every route $k \in \mathcal{K}$ is split into an sequence of jobs $\mathcal{J}_k = \{1, 2, \ldots\}$. For convenience, the jobs are numbered sequentially by the order in which they are performed, rather than using

(2)This paper is published in the 29th Conference on the Principles and Practice of Constraint Programming. The first conference in the series was organized by Alain in Cassis.

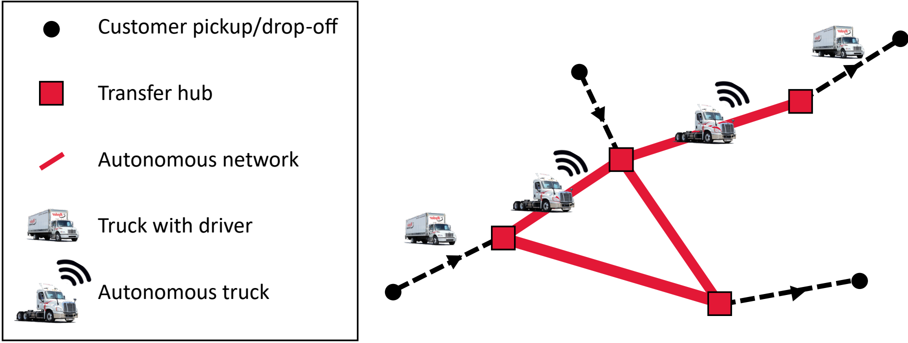


Figure 1.3. Example of an Autonomous Transfer Hub Network.

the original task numbers. Figure 1.4 provides a visualization, which will serve as a running example. Each job $j \in \mathcal{J}_k$ has up to four properties:

- $\text{type}(j) \in \{\texttt{l}, \texttt{d}, \texttt{u}, \texttt{r}, \texttt{p}\}$; type of job: load, drive, unload, relocate, park, respectively.
- $\text{duration}(j)$ (only for types l, d, u, r); duration of this job.
- $\text{task}(j) \in T$ (only for types l, d, u); task associated with this job.
- $\text{hub}(j) \in H$ (only for types l, u, p); hub associated with this job.

Each route is modeled with the same repeating sequence of loading at the origin hub (l), waiting before driving (p), driving (d), waiting at the destination hub before

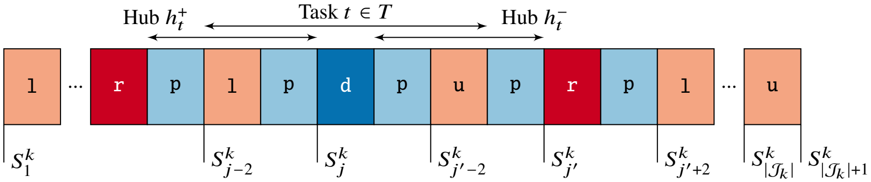


Figure 1.4. Jobs $\mathcal{J}_k$ for Route $k \in \mathcal{K}$ (jobs $j, j' \in \mathcal{J}_k$ are used as examples in the main text).

$$\min \quad \sum_{h \in H} C_h, \tag{1.1a}$$

$$\text{s.t.} \quad I_j^k = \texttt{Interval}([S_j^k, S_{j+1}^k]) \qquad \forall k \in \mathcal{K}, j \in \mathcal{J}_k, \tag{1.1b}$$

$$\texttt{Cumulative}\left([I_j^k | k \in \mathcal{K}, j \in \mathcal{J}_k, \text{type}(j) \in \{\texttt{l}, \texttt{u}\}, \text{hub}(j) = h], C_h\right) \quad \forall h \in H, \tag{1.1c}$$

$$S_{j+1}^k = S_j^k + \text{duration}(j) \qquad \forall k \in \mathcal{K}, j \in \mathcal{J}_k, \text{type}(j) \in \{\texttt{l}, \texttt{d}, \texttt{u}, \texttt{r}\}, \tag{1.1d}$$

$$\text{dom}(S_j^k) = [\underline{s}_j^k, \overline{s}_j^k] \qquad \forall k \in \mathcal{K}, j \in \mathcal{J}_k. \tag{1.1e}$$

Figure 1.5. Constraint Programming Model for Minimizing Required Hub Capacities.

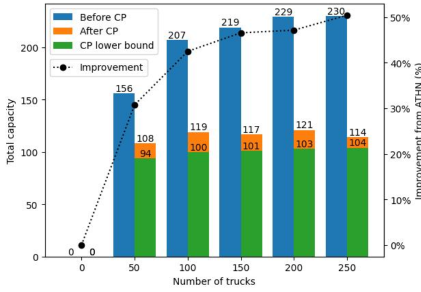


Figure 1.6. Hub Capacities.

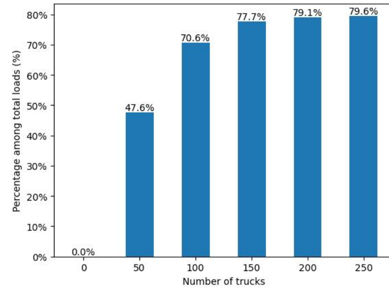
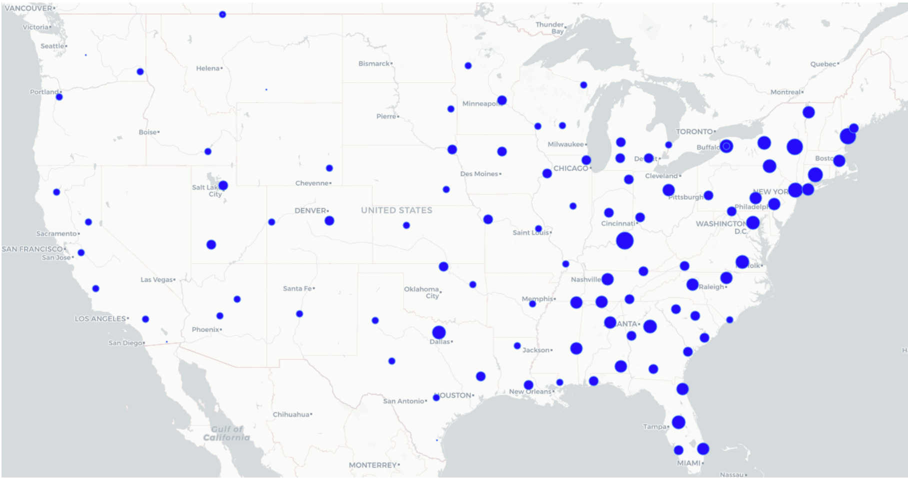


Figure 1.7. Loads Served Autonomously.

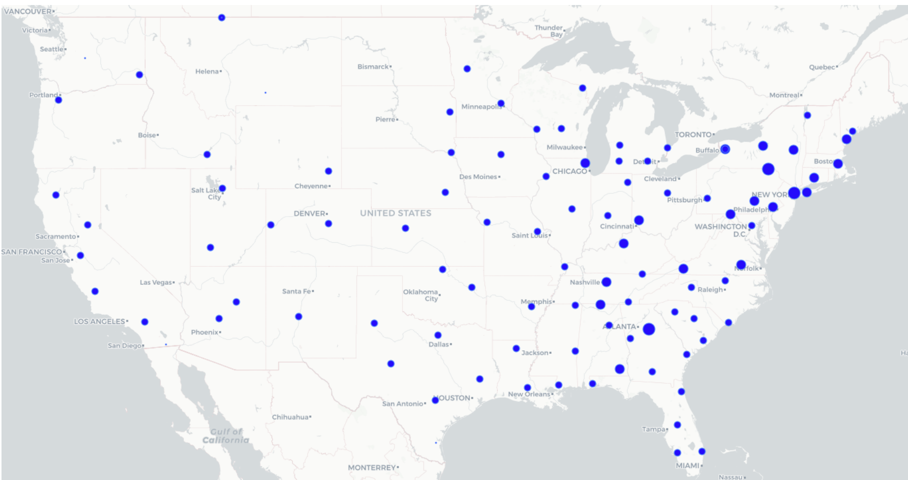
unloading (p), unloading (u), waiting before relocating to the next task if any (p), relocating (r), and waiting at the origin hub of the next task until loading (p).

Note that jobs of type l, d, u, r have a fixed duration, while the duration of p jobs is flexible and can be zero. When no relocation is necessary (the previous destination is equal to the next origin), the r job is still defined with duration zero for convenience. Every l, d, u job is trivially associated with an original task $t \in T$. Jobs l, u, p for which the vehicle is standing still are associated with a hub as described above.

The CP Model (1.1) is based on variables S_j^k that indicate the start time of job $j \in \mathcal{J}_k$ in route $k \in \mathcal{K}$, and variables C_h that indicate the necessary loading/unloading capacity at hub $h \in H$. For convenience, $S^k_{|\mathcal{J}_k|+1}$ is defined to represent the time at which the final job u is completed (see Figure (1.4)). Objective (1.1a) minimizes the total necessary hub capacity. To calculate the hub capacity, the model first defines an interval variable I_j^k for every job (Equation (1.1b)), which implicitly enforces $S_j^k \leqslant S_{j+1}^k$. Equation (1.1c) defines a cumulative constraint for every hub $h \in H$ to collect the intervals of the jobs with type l and u at that hub, and to assign the neccesary hub capacity to C_h. Note that this cumulative constraint has a variable as the capacity. It is also worth mentioning that Constraints (1.1c) span all the vehicles. Constraints (1.1d) ensure the correct job duration when a duration is defined (p jobs are flexible). Finally, Equation (1.1e) defines the domains of the S-variables, where constants $\underline{s}_j^k, \overline{s}_j^k$ remain to be defined. This paper focuses on loading/unloading capacity, but note that CP Model (1.1) is easily adapted to other objectives, such as minimizing parking space. To ensure that the time flexibility Δ is respected, the domains of the S-variables need to be defined accordingly. For job $j \in \mathcal{J}_k$ of route $k \in \mathcal{K}$ and $\text{type}(j) = \texttt{l}$, the domain is defined around the desired pickup time of the task to match the MIP: $\text{dom}(S_j^k) = [p_{\text{task}(j)} - \Delta, p_{\text{task}(j)} + \Delta]$. This domain is translated to the following d and u jobs to make sure that the flexibility is not exceeded until the task is complete. With slight abuse of notation, this gives the translated domains $\text{dom}(S_j^k) = \text{dom}(S_{j-2}^k) + \text{duration}(j-2)$ for jobs of $\text{type}(j) \in \{\texttt{d}, \texttt{u}\}$ (see Figure 1.4).



(a) Before CP Optimization



(b) After CP Optimization

Figure 1.8. Capacity Reduction for the 100 Truck Case (circle area proportional to hub capacity).

Figure 1.6 summarizes the results of using the CP model to minimize the necessary hub capacity for loading and unloading trucks. “Before CP” shows the required capacity if the MIP solution were implemented immediately, while “After CP” shows the results after applying the CP model. All instances were solved to optimality in under 30 seconds. The figure shows that the CP model is highly effective in reducing the total hub capacity for all instances, reducing the necessary capacity by 31% up to 50%. For the base case of $K = 100$ trucks, the CP model can reduce capacity by 42%. Note that

this may correspond to significant monetary savings: If each unit of loading/unloading capacity requires the assistance of a mechanic around the clock (three shifts of $57,557 per year [14]), the CP model reduces the annual labor cost by $15.2 million. Figure 1.6 also shows that the resulting capacity is close to the lower bound *for any ATHN solution*, which is obtained by removing the time constraints between subsequent tasks, i.e., treat every task as if it is the only task on the route.

2. Everything is Obvious

The previous section presented an overview of modeling with constraint programming, one of the beautiful descendants of Prolog. This section recounts how, I believe, it came to emerge. It is an interesting story, which highlights some of Alain's most impactful scientific contributions.

2.1. Prolog

Prolog is this unique combination of *elegance* and *practical use*, the trademark of Alain's research. The best way to present Prolog, and I believe that Alain would agree, is to show some truly beautiful programs. The first one is a Prolog program that holds if an element X is member of a list.

```
member(X,[X|_]).
member(X,[_|Y]) :- member(X,Y).
```

It is a non-deterministic program: it can be used to test whether a given element X is inside a given list L, but it can also be used to enumerate all the elements of a list. This Prolog program, although amazingly simple, features the two essential elements of Prolog: nondeterminism and the logic variable. In the words of Alain, "*the existence of variables provides Prolog with great expressive power because it permits reasoning about unknown objects.*" This is a key element of why CP emerged from Prolog: optimization is about finding unknown objects that satisfy some properties.

The power of the logic variable is best illustrated by difference lists and the ability in Prolog to concatenate lists in constant time and declaratively.

```
append(H1-T1,T1-T2,H1-T2).
```

This is one of the Prolog gems that will stay with everyone for a lifetime. Another gem is the use of the *not* operator in the following program.

```
test(G) :- not not G.
```

This program blew my mind the first time I saw it: it allows to test whether a goal G succeeds but without binding any of its variables. It started the entire area of "negation by failure" [3], and negation in general.

Prolog was born out of a marriage between elegance and practicality. It has its theoretical roots in formal languages and nondeterministic grammars, and its design

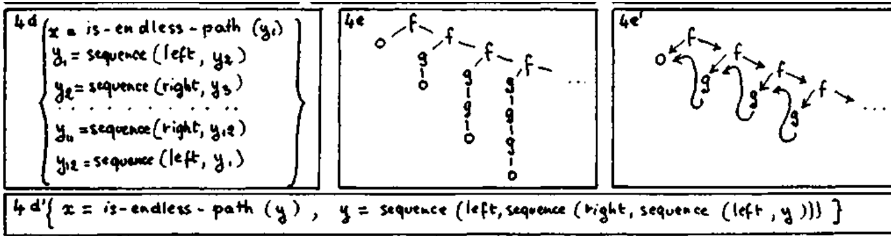


Figure 2.1. An Illustration of Rational Trees (from [5]).

was driven by applications in natural language processing. The link between the two came from theorem proving and the collaboration between Alain and Bob Kowalski, that led to the field of logic programming.

The programs shown above is why I fell in love with Prolog during my undergraduate studies and why I decided to pursue a PhD. But before talking about that, there is another important story to tell.

2.2. Prolog II

The 1973 version of Prolog [1] contains the so-called *diff* operator $\neq$, but it was dropped in the "official" 1974 version. That Prolog implementation also omitted the occur check during unification: it was a pragmatic decision for efficiency reasons but this omission had amazing consequences.

Indeed, as I alluded to already, Alain was always concerned with elegance and specifying precisely what his systems were computing. The omission of the occur check was not meeting his definition of elegance. So Alain started to investigate how to reconcile elegance and practical use, and how to design a new theoretical framework that would explain what is computed by Prolog. It led to the concept of rational trees (instead of finite trees), a concept that is illustrated in Figure 2.1. Rational trees may be infinite but they have a finite number of subtrees. Based on this concept, Alain designed the Prolog II system whose theoretical foundations were based on *solving constraints over rational trees*. Moreover, Prolog-II reintroduces the *diff* operator, thus supporting both equations and disequations over rational trees. Alain developed the theory of these disequations, showing that they can be handled independently. Alain published these theoretical results in 1984, in the Proceedings of the International Conference on Fifth Generation Computer Systems [6], at a time where the fifth generation project, and later the ECRC research centre in Munich, were building Prolog machines.

These theoretical results however opened a completely new perspective for the field, introducing Prolog II [4, 11] as the first constraint programming language. In the words of Alain, "*in much the same way, a Prolog programmer can refer to a tree x and a tree y without knowledge of the tree themselves–the only requirement is that they satisfy certain equalities and inequalities. These equalities and inequalities constitute a set of equations and inequations that the computer must be able to solve in the*

domain of trees–a task that is easier to perform in that domain than in the domain of numbers. These systems of inequations are called <u>*constraints*</u> *because they limit the values variables can assume.*"

3. Pandora's Box

Pandora's box was open and, obviously, Alain saw the tremendous potential of programming with constraints.

3.1. Prolog III

He started to work with a team of talented researchers (more on this later) and collaborators to design Prolog III [8, 9], often considered as the first constraint programming language. Prolog III includes equations and disequations over rational trees, linear constraints over rational numbers, and Boolean constraints. Alain, always generous with his ideas, gave talks about Prolog III over the world, sharing his research well before it was published. The introduction to Prolog III was only published in the Communications of the ACM in 1990. The linear constraints over rational numbers of Prolog III are solved with the simplex algorithm.

Alain was always searching for beautiful examples to showcase his systems. The square problem, depicted in Figure 3.1, is a perfect illustration of the power of Prolog III. Its goal is to place a given number of squares, all of different unknown sizes, to form a larger square (see the left side of Figure 3.1). It consists of two parts: a constraint part

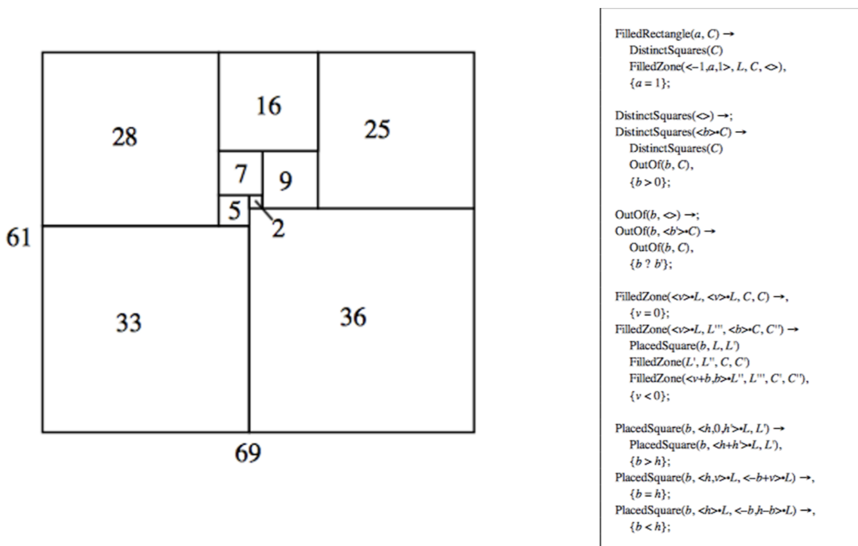


Figure 3.1. The Square Problem in Prolog III (from [9]).

that states that all squares must be of different sizes (procedure `DistinctSquares`) followed by a search component (procedure `FilledZones`) to place the smaller squares into the larger square. This search procedure adds constraints during the search, a key aspect of constraint programming.

It is hard to capture how much innovation came from this combination of stating constraints and expressing search procedures. Nowadays, many constraint programming systems provide black-box search to let users focus on the modeling of their applications. However, it is still the case that dedicated search is desirable in the most challenging applications. This combination of "Constraints + Search" was critical in the early days of constraint programming: it allows us to overcome the limitations of languages like Alice, breaking symmetries (as shown earlier) and, more generally, exploiting the structure of complex applications. These capabilities emerge naturally from introducing constraints into Prolog. They were key (in my opinion) to the (fast) pace of innovation that took place at the time and certainly in the work we did on the CHIP system (to be described shortly).

3.2. Constraint Logic Programming

A key development in the field came from Joxan Jaffar and Jean-Louis Lassez. They introduced the framework of "Constraint Logic Programming", demonstrating that all the theory underlying logic programming could be generalized to accommodate constraints [17]. This result was not only important scientifically, but also as a vehicle for the community to think about this new class of programming languages. Their group at the University of Melbourne and Monash University in Australia also started their own implementation of constraint logic programming [18].

3.3. CHIP

I started my PhD in August 1985 at ECRC in Munich, under the direction of Mehmet Dincbas and Hervé Gallaire, who was adamant that we should investigate the topic of "constraints" (we were also working on the merging of functional and logic programming). I had a stack of papers to read: they contained slides of Alain on Prolog III [7], an article of Jean-Louis Lauriere on his Alice system [20], and a paper by Haralick and Elliott on forward checking [16]. Alain and Jean-Louis were pioneers of artificial intelligence in France, but could not be more different. They came from different communities and had very different styles. In the last months of 1985, I was trying to see whether their work could be reconciled. Alain's quote

The Power of Prolog III = constraint solver + nondeterminism

was the driving force behind my research, as was the desire to design declarative systems. But I also wanted to solve the applications showcased in the 1978 paper about Alice, which were all about combinatorial optimization. Jean-Louis Lauriere was no longer working on Alice but rather on his expert system Snark, which had me puzzled obviously. I was lucky to read, sitting on the floor in the library of the University

of Namur, the seminal 1977 paper on "Consistency in Networks of Relations" by Alan Mackworth [22]: it contained a footnote, saying consistency techniques could be integrated in programming languages. Consistency techniques, also a declarative concept, were the link I was missing and, that winter, I wrote my first paper entitled "Domains in Logic Programming" [31] about what would become known as "constraint programming over finite domains" subsequently [30]. I started the implementation of the CHIP system [12] and I was fortunate to have an incredible first user and office mate at ECRC, Helmut Simonis, who pushed me to make the system applicable to industrial applications.

4. Disequations over Linear Constraints

I would like to mention another story that highlights Alain's approach to research. Prolog III supports the *diff* operator over linear constraints. This was again part of Alain's fascination for the *diff* operator, but also an integral part of making the language "complete": if equations are supported, disequations must be supported. Implementing this functionality raised interesting algorithmic questions and was one of the reasons I met Alain for the first time. Indeed, at ECRC, Thomas Graf and I were looking at Alain's slides on Prolog III, had encountered this technical challenge when designing a solver, and were searching for a way to implement this functionality. Alain gave no detail about how this was implemented in Prolog III. Our solution was to define a *syntactic* normal form for linear equations and disequations, that generalizes the simplex normal form and could be preserved by pivoting [33]. Alain saw the paper and was intrigued by the elegance of this normal form. I met Alain at the International Conference on Logic Programming in 1990 and he invited me to come to Marseilles in 1992. For a young researcher, this was an incredible honor and opportunity: Alain always invited scientists to Marseilles and, sometimes, at his home. I had the opportunity to swim in the Mediterranean sea with Alain, have lunches, and discusses technical topics with him (sometimes on the beach). He studied our syntactic normal form in all details, and subsequently adopted it for Prolog IV [10]. During his career, Alain was always driving his own agenda, with a relentless focus. But, at the same time, he was also open to ideas that would improve his systems, their elegance, or their functionalities.

5. CLP(Interval)

I spent my sabbatical in Marseilles in the Spring of 1994. Alain was an incredible host, finding a rental house in Cassis for my young family and making a car at our disposal. Constraint programming over intervals was becoming an important topic in the community at the time. Frédéric Benhamou, a key part of the talented team behind Prolog III, had visited Bill Older and André Vellino in Canada where they were implementing BNR-Prolog. I had just finished the first part of my sabbatical at MIT, where David McAllester and I had been working on using the combination of interval Newton's method and constraint propagation to solve nonlinear problems globally. Deepak Kapur was also visiting. This group worked intensively on constraint programming over intervals during that period, leading to a paper that won a test of time

award (20 years) and a paper in the SIAM Journal of Optimization. Frédéric brought Alain's rigor to the project, Deepak connected us to the right applications, and David and I refined the computational and implementation aspects. This is an example of the research environment that Alain created at Luminy over and over again: a stimulating intellectual atmosphere where ideas were exchanged freely, in the beautiful setting of Marseilles and Cassis. This work led to numerous follow-up contributions by Frédéric and his team in Nantes, and at Brown where we designed the Numerica system. Many of the techniques we designed then are integrated in mathematical and global optimization systems.

Prolog IV. — Constraint programming over intervals was of course the subject of much thinking by Alain at the time. I believe that he wanted to reconcile the beauty of Prolog III and the applications of the CHIP system (my book was on many desks at Luminy when I visited the first time). Intervals, narrowing operators, and fixed point theory became the way the Prolog IV team reconciled these two systems. Alain and Frédéric worked hard in defining the semantics of Prolog IV, leveraging intervals and fixpoint theory. Figure 5.1 depicts a snapshot of one of Frédéric's papers, starting with a definition of an approximation function for a single domains and its generalization to Carthesian products of domains, before defining the narrowing operator. Prolog IV [10] is an impressive system that supports constraints over rational trees, Booleans, linear constraints over rational numbers, intervals viewed as relations, and an integer constraint to specify that a variable must take integer value.

As a young researcher, I was always struck by how fascinated the community was with Alain's research. Researchers were always asking questions such as "Is Alain working on Prolog IV", "will there be a Prolog V", and "Is Alain writing a book". His impact on the community was tremendous although he wrote few papers, did not give many talks, and rarely attended conferences (unless invited). But his systems and their elegance had an impact that is hard to quantify: they directly led to the creation of the logic programming and the constraint programming communities, but also provided the seeds for many other innovations.

6. The Broad Impact of Alain

The following quote by Alain summarizes his overall remarkable vision for problem solving:

> "*Solving a problem using Prolog amounts to first describing the universe in which the problem is placed and then asking pertinent questions, instead of giving a recipe for computing the solution. This approach will eventually allow us to acquire a finer and more synthesis-oriented method of problem solving, and will hopefully inspire others to consider domains as varied as database systems, natural-language interfacing, expert systems, and computer-aided design.*"

Definition 1 *Let D be a set. Let $\underline{apx}$ be a function defined from $\mathcal{P}(D)$ into $\mathcal{P}(D)$. The function $\underline{apx}$ is an approximation over D iff the four following properties hold:*

1. $\underline{apx}(\emptyset) = \emptyset$,
2. $\forall \rho \in \mathcal{P}(D), \rho \subset \underline{apx}(\rho)$,
3. $\forall \rho, \rho' \in \mathcal{P}(D), \rho \subset \rho'$ *implies* $\underline{apx}(\rho) \subset \underline{apx}(\rho')$,
4. $\forall \rho \in \mathcal{P}(D), \underline{apx}(\underline{apx}(\rho)) = \underline{apx}(\rho)$.

Definition 2 *Let $\underline{apx}$ be an approximation function over a set D. Let ρ be an n-ary relation over D. Then,*

$$\underline{apx}(\rho) = (\underline{apx}(\pi_1(\rho))) \times \ldots \times (\underline{apx}(\pi_n(\rho)))$$

where $\pi_i(\rho)$, the *ith projection* of ρ, is defined as follows:

$$\pi_i(\rho) = \{x_i \in D \mid (\exists x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n \in D) \text{ such that } (x_1, \ldots, x_n) \in \rho\}$$

Definition 3 *Let ρ be an n-ary relation on a set D. The* narrowing function *of ρ is the function $\overrightarrow{\rho}_{\underline{\text{apx}}} : \mathcal{P}_{\underline{\text{apx}}}(D^n) \longrightarrow \mathcal{P}_{\underline{\text{apx}}}(D^n)$, such that for every u,*

$$\overrightarrow{\rho}_{\underline{\text{apx}}}(u) = \underline{apx}(u \cap \rho).$$

Figure 5.1. The Semantics of Interval Constraint Logic Programming (from [2]).

During his career, Alain anticipated many developments (e.g., in constraint databases, natural language interfaces, and CAD) and articulated a vision that is still relevant today in our world. I cannot list all the applications that build, directly or indirectly, on Alain's work, but I would like to list some of them and highlight the broad impact of his research.

The developers of IBM Watson, the software system that defeated the "champion" of Jeopardy, stated: *We required a language in which we could conveniently express pattern matching rules over the parse trees and other annotations (such as named entity recognition results), and a technology that could execute these rules very efficiently. We found that Prolog was the ideal choice for the language due to its simplicity and expressiveness.* [19].

In 2018, Bob Bixby, who wrote the world-leading mathematical optimization software, told the audience of the Annual Informs meeting that *the biggest improvements in Mixed-Integer Programming solvers have come from constraint programming*.

Probabilistic programming languages (PPLs) is one of the most active research areas at the intersection of artificial intelligence and programming languages. The wikipedia page for probabilistic programming includes the following text: *PPLs often extend from a basic language. The choice of underlying basic language depends on*

the similarity of the model to the basic language's ontology, as well as commercial considerations and personal preference. For instance, Dimple and Chimple are based on Java, Infer.NET is based on .NET Framework, while PRISM extends from Prolog.

Almost 50 years after the release of the first Prolog, its legacy lives on in one of the most exciting areas of AI: building a bridge between deep learning and inference.

7. Conclusion

It is impossible to conclude an article about Alain. He was an exceptional scientist and an endearing man. He was incredibly generous with his ideas, amazingly open, and accessible. Prolog is a truly beautiful language, Prolog III had profound ramifications on computer science and operations research, and both of them provided a fertile ground for numerous innovations. I hope that I was able to convey some of these in this article and to describe Alain's unique qualities through the history of constraint programming. I will leave you with a comment made by the owner of the rental house Alain found for us in Cassis. A few days after we were settled in the house in Cassis, the lady who owned the place asked me (in French) "and the elegant man who came to visit the house first, is he coming back some time?" Alain had a real sense of elegance in everything he did. Together with the communities they inspired, one of the lasting scientific legacies of Alain is the elegance, simplicity, and expressiveness of his programming systems.

Acknowledgments

I would like to thank Colette Colmerauer for her kindness and hospitality over the years, for the beautiful evenings we spent together in Cassis with Alain and Alice, and for her perseverance in organizing the events celebrating Alain in the middle of a pandemic.

Bibliography

[1] G. Battani & H. Meloni, "Interpreteur du Langage de Programmation Prolog", Internal report, Groupe Intelligence Artificielle, Universite Aix-Marseille II, September 1973.

[2] F. Benhamou, "Interval constraint logic programming", in *Constraint Programming: Basics and Trends* (Berlin, Heidelberg) (A. Podelski, ed.), Springer Berlin, Heidelberg, 1995, p. 1-21.

[3] K. Clark, "Negation as Failure", in *Logic and data bases* (H. Gallaire & J. Minker, eds.), Plenum Press, New York, 1978.

[4] A. Colmerauer, "PROLOG II : Manuel de Reférence et Modèle Théorique", Tech. report, GIA – Faculté de Sciences de Luminy, March 1982.

[5] ———, "Prolog in 10 Figures", in *IJCAI-83* (Karlsruhe), 1983, p. 487-499.

[6] ———, "Equations and Inequations on Finite and Infinite Trees", in *Proceedings of the International Conference on Fifth Generation Computer Systems (FGCS-84)* (Tokyo, Japan), ICOT, November 1984, p. 85-99.

[7] ———, "PROLOG III: Copy of slides", in *IEEE International Symposium on Logic Programming* (Atlantic City, New Jersey), February 1984.

[8] ———, "Opening the Prolog-III Universe", *BYTE Magazine* **12** (1987), no. 9, p. 177–182.

[9] ———, "An Introduction to Prolog III", *Commun. ACM* **33** (1990), no. 7, p. 69–90.

[10] ———, "Spécification de Prolog IV", Tech. report, Laboratoire d'informatique de Marseille, 1996.

[11] A. Colmerauer, H. Kanoui & M. Van Caneghem, "Prolog, bases théoriques et développements actuels", *T.S.I. (Techniques et Sciences Informatiques)* **2** (1983), no. 4, p. 271-311.

[12] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf & F. Berthier, "The Constraint Logic Programming Language CHIP", in *Proceedings of the International Conference on Fifth Generation Computer Systems* (Tokyo, Japan), December 1988, p. 693-702.

[13] A. Gargani & P. Refalo, "An Efficient Model and Strategy for the Steel Mill Slab Design Problem", in *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming (CP'07)*, Sep 2007, p. 77-89.

[14] Glassdoor, "How much does a Truck Mechanic make?", https://www.glassdoor.com/Salaries/truck-mechanic-salary-SRCH_KO0,14.htm, 2023, Accessed: 2023-04-28.

[15] W. Greene, "Autonomous Freight Vehicles: They're Heeeeere!", in *Autonomous Cars – Self-Driving the New Auto Industry Paradigm* (R. Shanker, A. Jonas, S. Devitt, K. Huberty, S. Flannery & W. Greene et al., eds.), Morgan Stanley & Co. LLC, 2013, p. 85-89.

[16] R. Haralick & G. Elliot, "Increasing Tree Search Efficiency for Constraint Satisfaction Problems", *Artif. Intell.* **14** (1980), no. 3, p. 263-313.

[17] J. Jaffar & J.-L. Lassez, "Constraint logic programming", in *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Munich, Germany), POPL-87, Association for Computing Machinery, January 1987, p. 111–119.

[18] J. Jaffar & S. Michaylov, "Methodology and Implementation of a CLP System", in *Fourth International Conference on Logic Programming* (Melbourne, Australia), May 1987, p. 196-218.

[19] A. Lally & P. Fodor, "Natural Language Processing With Prolog in the IBM Watson System", Association for Logic Programming Newsletter, March 2011.

[20] J.-L. Lauriere, "A Language and a Program for Stating and Solving Combinatorial Problems", *Artif. Intell.* **10** (1978), no. 1, p. 29-127.

[21] C. Lee, W. Boonbandansook, V. E. Akhlaghi, K. Dalmeijer & P. Van Hentenryck, "Constraint Programming to Improve Hub Utilization in Autonomous Transfer Hub Networks", in *Proceedings of the 29th International Conference on Principles and Practice of Constraint Programming*, vol. 280, Leibniz International Proceedings in Informatics (LIPIcs), August 2023, p. 46:1-46:11.

[22] A. K. Mackworth, "Consistency in Networks of Relations", *Artif. Intell.* **8** (1977), no. 1, p. 99-118.

[23] Roland Berger, "Shifting up a gear – Automation, electrification and digitalization in the trucking industry", https://www.rolandberger.com/publications/publication_pdf/roland_berger_trucking_industry.pdf, 2018.

[24] Ryder System, Inc. & *Anonymized for double blind*, "The Impact of Autonomous Trucking: A Case-Study of Ryder's Dedicated Transportation Network", Ryder Newsroom, 2021, https://newsroom.ryder.com/news/news-details/2021/Ryder-Teams-Up-with-Georgia-Tech-for-Industrys-First-Data-Driven-Study-on-Impact-of-Autonomous-Trucking/.

[25] A. Schutt, T. Feydy, P. J. Stuckey & M. G. Wallace, "Explaining the cumulative propagator", *Constraints* **16** (2011), no. 3, p. 250-282.

[26] M. Shahandasht, B. Pudasaini & S. L. McCauley, "Autonomous Vehicles and Freight Transportation Analysis", Tech. report, The University of Texas at Arlington, 2019.

[27] P. Shaw, "Using Constraint Programming and Local Search Methods to Solve Vehicle Routing Problems", in *Proceedings of Fourth International Conference on the Principles and Practice of Constraint Programming (CP'98)* (Berlin, Heidelberg), Springer Verlag, October 1998, p. 417-431.

[28] J. Short & D. Murray, "Identifying Autonomous Vehicle Technology Impacts on the Trucking Industry", American Transportation Research Institute, 2016, http://atri-online.org/2016/11/15/identifying-autonomous-vehicle-technology-impacts-on-the-trucking-industry/.

[29] P. Slowik & B. Sharpe, "Automation in the long haul: Challenges and opportunities of autonomous heavy-duty trucking in the United States", The International Council on Clean Transportation, 2018, https://theicct.org/publications/automation-long-haul-challenges-and-opportunities-autonomous-heavy-duty-trucking-united.

[30] P. Van Hentenryck, *Constraint Satisfaction in Logic Programming*, The MIT Press, Cambridge, MA, 1989.

[31] P. Van Hentenryck & M. Dincbas, "Domains in Logic Programming", in *Proceedings of the Fifth AAAI National Conference on Artificial Intelligence* (Philadelphia, PA), AAAI-86, AAAI Press, August 1986, p. 759–765.

[32] P. Van Hentenryck, P. Flener, J. Pearson & M. Ågren, "Tractable Symmetry Breaking for CSPs with Interchangeable Values", in *Proceedings of the 18th International Joint Conference on Artificial Intelligence* (Acapulco, Mexico), IJCAI'03, Morgan Kaufmann Publishers Inc., 2003, p. 277–282.

[33] P. Van Hentenryck & T. Graf, "Standard forms for rational linear arithmetics in constraint logic programming", *Ann. of Math. and Artif. Intel.* **5** (1992), no. 2-4, p. 303-320.

[34] P. Van Hentenryck & L. Michel, "The Objective-CP Optimization System", in *Proceedings of the* 19th *International Conference on Principles and Practice of Constraint Programming*, Sep 2013.

[35] P. Van Hentenryck & W. van Hoeve, "Constraint Programming", Springer, 2023.

[36] S. Viscelli, "Driverless? Autonomous Trucks and the Future of the American Trucker", Center for Labor Research and Education, University of California, Berkeley, and Working Partnerships USA. `http://driverlessreport.org/`, 2018.

[37] D. J. Watts, *Everything Is Obvious: How Common Sense Fails Us*, Random House LLC, 2012.

Abstract. — This article recounts the story of constraint programming, one of the most remarkable and enduring scientific contributions of Alain Colmerauer. It is a personal story, having lived through, and contributed to, the events that are reported here.

Keywords. — Logic programing, Prolog, Constraint programming.

Manuscrit reçu le 27 mai 2024, accepté le 12 juillet 2024.