Dedicated to
Professor Jakub Gutenbaum
on his 70th birthday

Control and Cybernetics

vol. **29** (2000) No. 1

Declarativity in modelling and problem solving

by

Antoni Niederliński

Instytut Automatyki, Politechnika Śląska
PL-44-100 Gliwice, Akademicka 16, Poland
e-mail aniederlinski@ia.polsl.gliwice.pl

Abstract: The paper discusses a new trend in the modelling software for combinatorial and mixed combinatorial-continuous decision problems. The trend, aiming at solving those problems by the simple activity of properly describing them, is best exemplified by a constantly increasing spectrum of Constraint Logic Programming (CLP) languages. The first such language was Prolog. After a short historical survey concentrating mainly on Prolog, main characteristics of a modern, commercially successful CLP language - CHIP - are presented, discussed and illustrated. The CLP approach to problem solving is compared with traditional Operation Research approaches.

Keywords: CHIP, CLP, constraint solving, combinatorial optimisation, declarative programming, Prolog.

1. Introduction

Declarativity in modelling and problem solving means roughly that the system analyst models the problem and declares the goal (a feasible solution, all feasible solutions, or an optimum solution), while the computer (hardware *cum* software) does the rest, i.e. chooses and parameterises a suitable algorithm, performs necessary computations and presents the results. In a nutshell, declarativity is recognised by the fact that *the model is the program is the solution.* The trend towards declarativity is just part and parcel of the general trend to make more and more computer power available to a wider and wider group of users who cannot be expected to be theoretically and programmatically knowledgeable to an extent allowing them to write their own procedures and put them into their

own code. The group of users meant are people seeking decision support, who as a rule know well their decision problems but are unable to cope with sophisticated mathematical techniques and complicated programming environments.

The trend has been fuelled not only by the demand of users, but also by the supply of fresh ideas from people working in artificial intelligence (AI), whose ultimate dream is to make computers more and more powerful and user friendly. This is exemplified by one of those powerful ideas from AI research - object oriented programming; it contributed towards large progress in declarativity for dynamic continuous- and discrete-time system modelling, best illustrated by such popular tools as Simulink /Matlab.

Of course, the questions may well be asked whether the trend to make sophisticated model building and decision support tools generally available to the more or less profane on a *black box basis* is something save and something to rejoice about in *academia* and whether *academia* should this trend support by popularising it and contributing towards it. These questions are, however, themselves of academic nature: our daily and professional lives get steadily more and more saturated by various complicated black boxes we have to master, simply because we cannot afford the alternatives like a personal car driver, a typist, a typesetter, a secretary, a personal tax consultant, a professional computer programmer, a professional photographer etc. etc. There seem to be no legitimate reasons why highly professional activities, like modelling and computer aided decision support, should be exempted from the overwhelming trend of cutting everything down *ad usum delphini* and remain a reserve of *academia*. What remains to be done is thus best expressed by the saying: *If you can't beat them, join them!*

The most successful declarative modelling tools so far are restricted to the domain of dynamic continuous- and discrete-time system. The challenge seems to be to extend this approach to the more difficult, analysis-proof combinatorial and mixed (combinatorial-continuous) decision and optimisation problems. Substantial progress in the combinatorial domain has been already made. This has been done mainly under the banner of *Constraint Logic Programming* (CLP). CLP may be defined as a body of techniques used for solving problems with constraints. The idea is: problems to be solved are modelled using elementary logic, in a way that turns the model into a main part of the problem-solving program. The activity of exploring constraints, which must be satisfied by the solutions, generates the solutions. More precisely, CLP may be considered as a programming paradigm, tightly integrating traditional Logic Programming as given by Prolog, Constraint Satisfaction/ Solving as developed in Artificial Intelligence, and Optimisation as developed in Operations Research.

2. In the beginning there was Prolog

Prolog means not only *Programming in Logic*, but denotes something preliminary, to be developed further. Those that coined this name showed a lot of

foresight. Prolog was indeed an introduction, it was the first popular, powerful but scope-limited, CLP language.

Prolog (see Bartak, 1999, Sterling and Shapiro, 1996) is what is known as *declarative language.* This means that given the necessary facts and rules, Prolog will use deductive reasoning to solve the problem described by the program. This is in contrast to traditional computer languages, such as C, BASIC and Pascal, which are procedural languages. In a procedural language, the programmer must provide step by step instructions that tell the computer exactly how to solve a given problem. In other words, *the programmer must know how to solve the problem before the computer can do it.* The Prolog programmer, on the other hand, only needs to supply a description of the problem and state the goal to be achieved (i.e. solved, accepted, or rejected). A Prolog program contains no algorithm, it just describes the problem (or some part of the world) we want to reason about. The very problem description is the program solving the problem. Separating problem description from reasoning, which aims at drawing conclusions from this description, is at the root of this important property: the reasoning is done by the Prolog compiler/interpreter. As a result there is no need to construct algorithms while working with Prolog. The problem is described using rather elementary logic: facts and rules (Horn clauses) of the form $p : -a, b, c$, where p is the head of the rule (the conclusion) and a, b, c are the body of the rule (a conjunction of premises). Horn clauses are a versatile model: it has been shown by Kowalski (1989), that any problem of logic may be expressed using Horn clauses. Prolog facts and rules are supplemented by the Closed World Assumption, which may be stated like this: *if something does not follow from facts and rules of the Prolog program, it is considered by the reasoning system to be false.*

The use of Horn clauses is instrumental for Prolog's declarativity: supplemented by lists as the basic data structure and recursion as the basic way to define relations, they form a very handy tool to model an important set of real-world problems, amenable to description by terms from the Herbrand universe. The terms are just strings with no semantic meaning. This is at the root of Prolog's strength as well as its weakness: Prolog makes it easy to formulate very general statements, but difficult if not impossible to describe relations from some less general universe, like the universe of integers or reals. However, Prolog was the first language deserving to be called *constraint logic programming language,* because of its capacity to solve constraints defined in the Herbrand universe. The algorithm needed to reason is universal and provided by the Prolog compiler. Prolog's compiler has a built-in inference mechanism *combining standard (chronological) backtracking with term unification, modus ponens inference* and the *closed-world assumption* in order to retrieve automatically answers to queries based on rules and facts of the program. Consider e. g. the following small

Prolog program, declaring constraints between variables X and Y:

```
a(X,Y) :- b(X), c(X,Y).
b(1).
b(&).
c(&, "A").
```

together with the goal `a(X,Y)`. The solution `X=&, Y="A"` for this goal is found by Prolog's search using the Standard Backtracking algorithm, incorporated in its compiler/interpreter, and forming, together with the mechanism for Herbrand term unification, the heart of the reasoning system. Standard Backtracking (SB) is a universal search algorithm for determining values of variables so that some constraints are satisfied. SB attempts incrementally to extend a partial consistent solution with values of some of the uninstantiated variables, checks the consistency of the extended solution and when not consistent, drops the last instantiation and tries another one for this variable. This is summarised by the name "generate and test" given sometimes to Standard Backtracking: backtracking is initiated by the violation of some constraint caused as a result of variable instantiation. The actual *constraint solving* is done in Prolog by the unification algorithm, which has an important limitation: in the Herbrand universe only terms syntactically equivalent are unificable. Another important Prolog feature is that lists (basic Prolog data structures) and recursion support synergistically each other while serving to formulate short but powerful definitions for a very broad range of operations. A list is defined recursively as having the structure `[Head|Tail]`, where `Head` is the first element of the list and `Tail` is the list with `Head` removed. A simple example of a recursively defined property is the property "member" describing list membership:

```
member(M, [M|_]).
member(M, [_|T] :-member(M, T).
```

The first clause states that a list `Head` is a list member. The second clause states that M is a list member if it is a member of the lists `Tail`. The conciseness of this expression is due both to the recursive nature of lists as such as to the recursive nature of the clause `member(_,_)`. Paraphrasing Shakespeare it can be said that *Brevity is the soul of Prolog.*

Notwithstanding its advantages, Prolog has some serious weaknesses: the main is perhaps its domain of computation, which is not allowing to solve constraints in the integer or real variable domains. Also, Standard Backtracking leaves much to be desired because of its inefficiency: in Prolog backtracking is initiated by constraint violation only!

3. Prolog extension and modification

A number of attempts have been made to remove Prolog weaknesses. For a historical review the reader is referred to Jaffar and Maher (1996). For a general

discussion of CLP ideas the books by Tsang (1995) and Mariott and Stuckey (1998) are recommended. The most technically advanced and successful attempt to create a CLP language is CHIP (for *Constraint Handling In Prolog*). CHIP was initiated at ECRC (European Computer Industry Research Centre) in Munich, in the early 1980s, and was subsequently developed and commercialised by COSYTEC, France. In the sequel, reference is made to CHIP V5.2 for Windows NT. CHIP extends Prolog by introducing three new domains of computation: the integer (finite), rational and Boolean domains. For each of them a specialised constraint handling technique has been introduced:

- for finite domains - constraint satisfaction performed by advanced backtracking techniques relying upon various consistency checking algorithms;
- for rational domains - constraint solving done by a symbolic simplex-like algorithm;
- for Boolean domains - equation solving in Boolean algebra.

Those constraint handling techniques play the role of the Prolog unification mechanism for non-Herbrand domains. For finite domains and rational domains CHIP offers powerful optimisation predicates:

- for finite domains it is *branch-and-bound* supplemented with advanced backtracking to deal with constraints;
- for rational domains it is *linear programming* using incremental simplex.

Both algorithms are - in the form of predicates - well integrated with all the other predicates. As a result the embedding of optimisation into a set of constraints is done in a rather obvious, intuitive and simple way. What's more - the performance index to be optimised may be computable not as a direct function of the decision variables (like e. g. in linear programming), but as an undirect function of the decision variables; sometimes, while using standard CHIP constraints like `cumulative()` or `cycle()`, the dependence remains partially hidden behind those very constraints. The discussion in the sequel will concentrate on finite domain (combinatorial extension). For rational domains see CHIP V.5.2 (1998) and Niederliski (1999).

3.1. Basic finite domain extensions

The most important extension is doubtless that for finite domains. Finite domain variables in CHIP have to be characterised by *domains*, given by sets of discrete items, e. g. natural numbers. Domain variables are defined in CHIP with the primitive ::, e. g.:

$X :: 0..20$, $X :: 0:20$, $X :: [2,4,6,8,10]$, $[X,Y,Z] :: 0..20$.

They appear as initial data in all constraint satisfaction problems, formulated as follows: given the following data:

- a set of finite variables;
- a set of possible values for each variable (their domains);
- a set of constraints restricting values that variables can simultaneously take,

the problem is to determine sets of values for each variable fulfilling all constraints. The instantiation of variables to values in their domains is done by the nondeterministic predicate `indomain(Element)`, which instantiates the variable *Element* to the smallest value in its domain and upon backtracking, to the next smallest value, and so on. It is usually used in the structure:

```
labeling([ ]).
labeling([X|Y]) :-
      indomain(X),
      labeling(Y).
```

The predicate `labeling(_)` is searching for instantiations of a list of domain variables and is, while backtracking, instantiating all the variables.

The introduction of finite domain paved the way for advanced backtracking techniques. The main weakness of Standard Backtracking used by Prolog is that *testing* (i. e. checking for consistency) is done after a new extended solution is generated. This weakness has been removed from the advanced backtracking algorithm as implemented in CHIP by enhancing it with some *predictive* power. It uses the basic form of constraint propagation: a new variable being instantiated, from the domains of all remaining variables values are removed inconsistent with the instantiated variable. As a result, backtracking may be initiated not only by the violation of some constraint, but also by two different *predictive* mechanisms:

1. *Forward checking*: backtracking is initiated by the appearance of an empty domain.
2. *Looking ahead*: backtracking is initiated by the appearance of a non-empty domain with no feasible values.

Both mechanisms turn out to be computationally less costly than checking for constraint violation only while using Standard Backtracking.

The advanced backtracking algorithm has been supplemented by *labelling strategies*, which aim at instantiating variables. Labelling strategies consists of *variable-* and *value-ordering strategies*:

- *variable-ordering strategies* decide which variable to choose for instantiation;
- *value-ordering strategies* decide which value to give the instantiated variable; the process of attaching a value to a domain variable is referred to as *labelling*.

CHIP relies upon the following variable-ordering strategies:

1. *First Fail* chooses for instantiation the variable with the smallest domain.
2. *Smallest* chooses for instantiation the variable having the smallest value in its domain.
3. *Largest* chooses for instantiation the variable having the largest value in its domain
4. *Max Regret* chooses for instantiation the variable having the largest difference between its smallest and second smallest values in its domain.

The value-ordering strategies are as follows:

1. *Smallest* chooses the smallest value in the domain.
2. *Largest* chooses the largest value in the domain.
3. *Succeed first* chooses the value which previously had succeede.

Constraint propagation, which is in CHIP automatic (i.e. no programming needs to be done to activate it) and incremental (i. e. new constraints are simply applied to the domains obtained up to now), works as illustrated by the following example: given variable X with initial domain $[1, 2, \cdots, 10]$, variable Y with initial domain $[1, 2, \cdots, 10]$, and variable Z with initial domain $[1, 2, \cdots, 10]$, the appearance of constraint 1: $Y < Z$ propagates into the domains, resulting in: variable X getting domain $[1, 2, \cdots, 10]$, variable Y getting domain $[1, 2, \cdots, 9]$, and variable Z getting domain $[2, 3, \cdots, 10]$. Next, the appearance of constraint 2: $X = Y + Z$ propagates into the domains, resulting in: variable X getting domain $[3, 4, \cdots, 10]$, variable Y getting domain $[1, 2, \cdots, 8]$, variable Z getting domain $[2, 3, \cdots, 9]$. Further on, the new constraint 3: $X = Z + 3$ results in variable X getting domain $[5, 6, \cdots, 10]$, variable Y getting domain $[1, 2, \cdots, 6]$, and variable Z getting domain $[2, 3, \cdots, 7]$. Obviously, at any stage of the constraint propagation, for any domain value of any variable there are domain values of all other variables fulfilling all constraints.

A most welcomed addendum in CHIP is combinatorial optimisation, considered as special case of constraint satisfaction. Given a set of domain variables, a set of their domains, a set of constraints, a performance index depending (directly or indirectly) upon the domain variables, the values of domain variables optimising the performance index are to be determined. The method used is Branch-and-Bound with advanced backtracking for constraint handling and domains being either explicitly enumerated or given as intervals, constraints being either explicitly enumerated or given as linear functions, and performance index being either explicitly enumerated or given as linear function.

Both constraint satisfaction and combinatorial optimisation as realised by CHIP offer an additional practically important bonus: there is no need to transform problems solved by CHIP into some canonical form. This is a substantial advantage considering that any transformation into canonical forms is as a matter of fact destroying declarativity and introducing a *semantic gap* between problem formulation and the problem-solving program. This is so because the transformation is usually apt to aggregate the problem data, usually having some important technical or economical meaning, into compounds notoriously difficult to interpret in terms of the original problem formulation. This absence of canonical forms for problem solution is another much welcomed step on the way towards complete declarativity.

3.2. Powerful finite-domain predicates

To safeguard the Prolog spirit of declarativity and simplicity, CHIP designers had to design a number of high-level predicates (or rather constraints), adapted

to the special needs of combinatorial systems. The results obtained are excellent. Such constraints as `cumulative()`, `diffn()`, `cycle()` and `among()` seem to be more than just constructs of a programming language. They seem to fulfil in a splendid way the role of basic concepts describing combinatorial processes. This has been e. g. tacitly acknowledged by Mariott and Stuckey (1998) who, while claiming independence of any actual constraint programming language, introduced and freely used the `cumulative()` constraint in a slightly less general form than it is done in CHIP.

The most useful and powerful is beyond doubt the constraint `cumulative()`, invoked in the most general case as:

```
cumulative([S1,···,Sm],[D1,···,Dm],[R1,···,Rm],[E1,···,Em],
           [V1,···,Vm],L,E,[Reference,Overshoot])
```

Here `Si` are starting times of activities, `Di` are durations of activities, `Ri` are units of a common resource used by activities, `Ei` are activity ends, `Vi` denotes the surface-of-usage of resource by activity (equal `Ri * Di`), `i` is the number of activity, `L` is the overall amount of the common resource available, `E` is the overall end, `Reference` denotes a level of common resource utilisation and `Overshoot` is the overall surface-of-usage of the resource above the value `Reference`.

Consider the following example: For seven jobs using the same resource with limited overall availability of 13, job 4 should follow job 2, job 5 should follow job 4 and the following durations and resource demands are given:

Job	Duration	Resource demand
1	16	2
2	6	9
3	13	3
4	7	7
5	5	10
6	18	1
7	4	11

Determine a job sequence that minimises the overall time of job completion. The corresponding CHIP program looks as follows:

```
top:-
  LO   =   [O1,O2,O3,O4,O5,O6,O7],    %list of starting times
  LD   =   [16, 6,13, 7, 5,18, 4],    %list of durations
  LE   =   [E1,E2,E3,E4,E5,E6,E7],    %list of completion times
  LR   =   [2, 9, 3, 7, 10, 1,11],    %list of resource demands
 LO :: 1..100,              % domain of starting times
 End :: 1..100,             % domain of overall time
 LE :: 1..100,              % domain of completion times
 High :: 1..13,             % domain of resource demands
 O4#>=O2+6, O5=#>=O4+7,     % precedence constraints
```

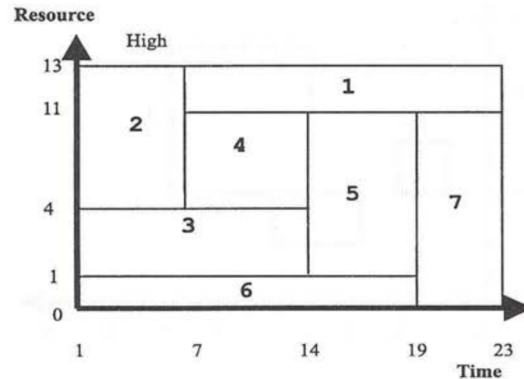

Figure 1. Gantt diagram for cumulative() example

```
cumulative(LO,LD,LR,LE,unused,High,End,unused),
  min_max(labeling(LO),End),     % that's the way to ask for mi-
                                 % nimisation of END by proper
                                 % choice of LO elements
  write('LO = '),         writeln(LO),
  write('LE = '),         writeln(LE),
  write('High = '),       writeln(High),
  write('End = '),        writeln(End).
  labeling([]).           % Enumerating variables.  The predicate
  labeling([X|Y]):-       % is searching for instantiations of a
     indomain(X),         % list of domain variables and is, while,
                          % backtracking, instantiating all variables.
     labeling(Y).
```

The solutions is:

```
  LO    =    [7,1,1,7,14,1,19]
  LE    =    [23,7,14,14,19,19,23]
             High = 13
              End = 23
```

It corresponds to the Gantt diagram form Fig. 1. It should be noticed that the same program is solving the packing problem of seven rectangles into a rectangle of minimum length given its height.

It should be noted that the performance index `End` need not be explicitly expressed as function of the decision variables `[O1,O2,O3,O4,O5,O6,O7]`. `End` is bounded to `[O1,O2,O3,O4,O5,O6,O7]` *via* the `cumulative()` constraint.

Besides `cumulative()`, another combinatorial predicate (constraint) of CHIP contributes much to its power. This is the `diffn()` constraint, which checks for

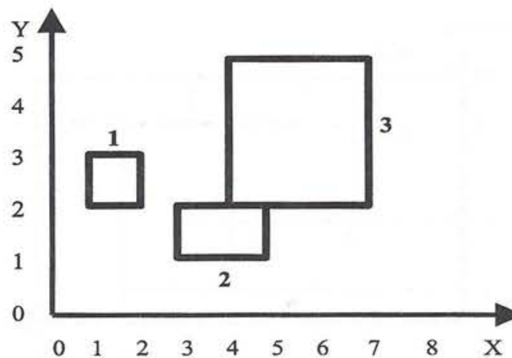

Figure 2. Diagram for the diffn() example

overlapping of m hyper-rectangles, each defined in a space of dimension n. Its basic arguments are lists defining those rectangles and including origins (O) and lengths (L), as well as lists of maximum and minimum volumes of rectangles:

```
diffn([[O11.,,,.O1n,L11,···,L1n],···,[Om1.,,,.Omn,Lm1,···,Lmn]],
[Vmax1,···,Vmaxm],[Vmin1,···,Vminm],unused,unused,unused)
```

It is useful for the scheduling and placing applications. Consider the following example with the thrid 2-dimensional rectangle defined by its volume and contraints on origin and length:

```
top:-
  LX :: 1 .. 4, LY :: 1 .. 4, X :: 1 .. 9, Y :: 1 .. 9,

  diffn([[1,2,1,1],[3,1,2,1],[X,Y,LX,LY]],[1,2,9],[1,2,9],
        unused,unused,unused),
  labeling([X,Y,LX,LY],0,first_fail,indomain),

  write('X = '), write(X),nl, write('Y = '), write(Y), nl,
  write('LX = '), write(LX),nl, write('LY = '), write(LY).
```

The solution is: X=1,Y=3,LX=3,LY=3, and corresponds to the diagram in Fig. 2.

Both predicates will be used to solve the following combinatorial optimisation problem: There are four students, Algy, Bertie, Charlie and Dingby, who share a flat. Four newspapers are delivered to the flat: the *Financial Times (FT)*, the *Guardian*, the *Daily Express* and the *Sun*. Each of the students reads all of the newspapers, in a particular order and for a specified amount of time (see below). The readings are not interrupted. Given that Algy gets up at 8.30,

Bertie and Charlie at 8.45, Dingby at 9.30, what is the earliest time that they can all set off for college?

Order	Algy, 8:30	Bertie, 8:45	Charlie, 8:45	Dingby, 9:30
1	FT 60 minutes	Guardian 75 minutes	Express 5 minutes	Sun 90 minutes
2	Guardian 30 minutes	Express 3 minutes	Guardian 15 minutes	FT 1 minute
3	Express 2 minutes	FT 25 minutes	FT 10 minutes	Guardian 1 minute
4	Sun 5 minutes	Sun 10 minutes	Sun 30 minutes	Express 1 minute

This is modelled and solved by the following CHIP program:

```
top:-
  A=[AFT,AGu,AEx,ASu],            % start list of reading times for Algy
  B=[BGu,BEx,BFT,BSu],            % start list of reading times for Bertie
  C=[CEx,CGu,CFT,CSu],            % start list of reading times for Charlie
  D=[DSy,DFT,DGu,DEx],            % start list of reading times for Dingby
  END=[AEnd,BEnd,CEnd,DEnd],      % end list of reading times for students

  A ::  30..300,       % domain declarations for list elements
  B ::  45..300,
  C ::  45..300,
  D ::  105..300,
  END ::  90..300,
  Final_Time ::  90..300,

  AGu#>=AFT+60,     % reading order constraints for Algy
  AEx#>=AGu+30,
  ASu#>=AEx+2,
  AEnd#>=ASu+5,

  BEx#>=BGu+75,     % reading order constraints for Bertie
  BFT#>=BEx+3,
  BSu#>=BFT+25,
  BEnd#>=BSu+10,

  CGu#>=CEx+5,      % reading order constraints for Charlie
  CFT#>=CGu+15,
  CSu#>=CFT+10,
  CEnd#>=CSu+30,

  DFT#>=DSu+90,     % reading order constraints for Dingby
  DGu#>=DFT+1,
  DEx#>=DGu+1,
  DEnd#>=DEx+1,
```

```
  diffn([[AFT,60],[BFT,25],[CFT,10],[DFT,1]],
        unused,unused,unused,unused,unused),
  diffn([[AGu,30],[BGu,75],[CGu,15],[DGu,1]],
        unused,unused,unused,unused,unused),
  diffn([[AEx,2],[BEx,3],[CEx,5],[DEx,1]],
        unused,unused,unused,unused,unused),
  diffn([[ASu,5],[BSu,10],[CSu,30],[DSu,90]],
        unused,unused,unused,unused,unused)
      % only one student may read at any time a given newspaper

  cumulative(A,[60,30,2,5],[1,1,1,1],unused,unused,1,AEnd,unused),
  cumulative(B,[75,3,25,10],[1,1,1,1],unused,unused,1,BEnd,unused),
  cumulative(C,[5,15,10,30],[1,1,1,1],unused,unused,1,CEnd,unused),
  cumulative(D,[90,1,1,1],[1,1,1,1],unused,unused,1,DEnd,unused),
      % each student may read at any time only a single newspaper

  maximum(Max_Time,END), % determine the maximum reading time Max_Time
  min_max     (labeling([AFT,AGu,AEX,ASu,BGu,BEX,BFT,BSu,CEX,CGu,
              CFT,CSu,DSu,DFT,DGu,DEX],0, first_fail,
              indomain),Max_Time),
                 % minimize Max_Time
              write('A = '),write(A),nl,
              write('B = '),write(B),nl,
              write('C = '),write(C),nl,
              write('D = '),write(D),nl,
              write('Final_Time = '),write(Final_Time),nl,
  show_schedule
  ([AFT,AGu,AT,ASU,BGu,BT,BFT,BSU,CT,CGu,CFT,CSU,DSU,DFT,DGu,DT],
  [Algy, Financial_Times,60,Algy,Guardian,30,Algy,Express,2,
  Algy,Sun,5,Bertie,Guardian,75,Bertie,Express,3,Bertie,
  Financial_Times,25,Bertie,Sun,10,Charlie,Express,5,Charlie,
  Guardian,15,Charlie,Financial_Times,10,Charlie,Sun,10,
  Dingby,Sun,90,Dingby,Financial_Times,1,Dingby,Guardian,1,
  Dingby,Express,1]).
  show_schedule     ([ ],[ ]).
  show_schedule     ([H1|T1],[H21,H22,H23|T2]) :-
        determine_time          (H1, FG, FM),
        determine_time          (H1+H23, TG, TM),
        printf("  %s ---->      %s from %u:%u to %u:%u{}n",
        show_schedule           [H21,H22,FG,FM,TG,TM]),(T1,T2).
%% Time is given by minutes after 08:00

determine_time (Time,Hours,Mins) :-
      Hours is (Time/60)+8, Mins is mod(Time,60).
```

```
The program produces the following result:
Found schedule which ends at 11:45
Found schedule which ends at 11:30

  min_max      ->       proven optimality
  'Dingby'     ---->    'Express' from 11:17 to 11:18
  'Dingby'     ---->    'Guardian' from 11:16 to 11:17
  'Dingby'     ---->    'FT' from 11:15 to 11:16
  'Dingby'     ---->    'Sun' from 9:45 to 11:15
  'Charlie'    ---->    'Sun' from 9:15 to 9:25
  'Charlie'    ---->    'FT' from 9:5 to 9:15
  'Charlie'    ---->    'Guardian' from 8:50 to 9:5
  'Charlie'    ---->    'Express' from 8:45 to 8:50
  'Bertie'     ---->    'Sun' from 11:15 to 11:25
  'Bertie'     ---->    'FT' from 10:23 to 10:48
  'Bertie'     ---->    'Express' from 10:20 to 10:23
  'Bertie'     ---->    'Guardian' from 9:5 to 10:20
  'Algy'       ---->    'Sun' from 11:25 to 11:30
  'Algy'       ---->    'Express' from 10:50 to 10:52
  'Algy'       ---->    'Guardian' from 10:20 to 10:50
  'Algy'       ---->    'FT' from 9:15 to 10:15
```

The beauty of this program lies in its declarativity which is directly reflecting the mechanism of newspaper sharing and constraint satisfaction. It is not obscured by the need to put the problem into the straightjacket of some canonical formulation.

4. CLP versus the OR tradition

CLP languages are tools competing with two traditional and general operations research approaches towards constraint solving and optimisation:

1. Specialized solvers
2. Specialised custom-tailored algorithm implemented by some custom-tailored procedural program (C, Pascal).

Solvers make powerful algorithms available at low cost. However, problems to be solved by solvers need to be transformed into the straightjacket of some canonical form. This is destroying declarativity at the outset and is a constant source of frustration for the user interested in analysing the influence of various technical or economic parameters upon the final solution. Problem transformation breeds thus a semantic gap between the original problem (OP) and transformed problem (TP): the TP has more variables and is difficult to understand and modify. Solvers may be resistant to some problem-specific knowledge: they may be even unfriendly when it comes to accommodating some unusual constraints. Most surprising of all - some solvers - although enjoying the reputation of being

optimised with respect to solving a particular class of problems, turn out to be rather ineffective when compared with CLP languages like CHIP. The design of custom-tailored algorithms and its implementation by some custom-tailored procedural program may take into account all problem-specific knowledge. However, the semantic gap still exists; it is inexorably connected with *procedural* programming. The customised solution takes usually a long developing time; the result of all this effort is as a rule a rather long program because of the need to program things like, e.g., various backtracking and branch and bound algorithms. The program is difficult to modify by anybody but its designer; modifications done be him/her are usually time-intensive and costly.

Both of the traditional approaches share the truly irrational property that the more constraints the more difficult the search and the longer the time necessary to get the solution.

The technology offered by CLP languages like CHIP is truly refreshing. CHIP contains IP and LP solvers, well integrated with the rest of the package. There is no need for the problems to be transformed into some canonical forms. There are no difficulties with expressing unconventional constraints. Because of its declarative nature, the semantic gap created by CHIP is very narrow: the statement of the problem is almost the program solving the problem. For professional programs it is necessary to master the art of defining various lists (of lists of lists ...) with variables subject to constraints, which is a slight extension of techniques popular in Prolog. In CHIP a custom-tailored program may be designed without designing the algorithm and without going into all those details that make life difficult when programming procedurally. The logic of the program is more rational: the more constraints the easier the search. The development time is short because problem constraints may be used directly and there is no need to program things like backtracking and branch and bound. The resulting program is short and transparent. The developing cost is reasonable. The program is easy to modify and as (or even more) effective as custom-tailored procedural solutions.

5. Conclusion

The importance of constraint logic programming (CLP) is well reflected by a saying (see Barták, 1999a) attributed to Eugene C. Freuder: *Constraint Programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming: the user states the problem, the computer solves it.* Alas, the Holy Grail has still to be searched for, although enormous progress has been made on the way toward it.

References

Aggoun, A. and Beldiceanu, N. (1993) Extending CHIP in order to solve complex scheduling and placement problems. *Mathl. Comput. Modelling,*

17, 7, 57-73.

BARTÁK, R. (1999A) On-Line Guide to Constraint Programming. `http://kti.msmff.cuni.cz/~ bartak/constraints/`

BARTÁK, R. (1999B) Guide to Prolog Programming. `http://kti.msmff.cuni.cz/~ bartak/prolog/`

BELDICEANU, N. AND E. CONTEJEAN, (1994) Introducing Global Constraints in CHIP. *Mathl. Comput. Modelling*, **20**, 12, 97-123.

CHIP (1998) V5.2 compact disc, Cosytec, France.

DEVILLE, Y. AND VAN HENTENRYCK, P. (1992) Construction of CLP Programs. In: D.R. Brough, ed., *Logic Programming.* Kluwer Academic Publ., Dordrecht, 112-135.

DINCBAS, M., SIMONIS, H. AND VAN HENTENRYCK, P. (1990) Solving Large Combinatorial Problems in Logic Programming. *Journal of Logic Programming*, **8**, 75-73.

JAFFAR, J. AND MAHER,M. J. (1996) Constraint Logic Programming - A Survey. *Journal of Logic Programming*, **19/20**, 503-581.

KOWALSKI, R. (1989) *Logika w rozwiązywaniu zadań.* WNT, Warszawa.

MARRIOTT, K. AND STUCKEY, P. J. (1998) *Programming with Constraints: an Introduction.* The MIT Press, Cambridge, Mass.

NIEDERLIŃSKI, A. (1999) Constraint Logic Programming - from Prolog to CHIP. *Proceedings of the Workshop on Constraint Programming for Decision and Control CPDC'99*, Gliwice.

STERLING, L. AND SHAPIRO, E. (1996) *The Art of Prolog.* The MIT Press, Cambridge, Mass.

TSANG, E. (1995) *Foundations of Constraint Satisfaction.* Academic Press, London.

VAN HENTENRYCK, P. (1989) *Constraint Satisfaction in Logic Programming.* The MIT Press, Cambridge, Mass.

VAN HENTENRYCK, P., MICHEL, L. AND DEVILLE, Y. (1998) *Numerica. A modeling language for global optimization.* The MIT Press, Cambridge, Mass.