The Knowledge Engineering Review, Vol 6:3, 1991, 151–194

Constraint logic programming

PASCAL VAN HENTENRYCK

Brown University, Box 1910, Providence, RI 02912, USA

Abstract

Constraint logic programming (CLP) is a generalization of logic programming (LP) where unification, the basic operation of LP languages, is replaced by constraint handling in a constraint system. The resulting languages combine the advantages of LP (declarative semantics, nondeterminism, relational form) with the efficiency of constraint-solving algorithms. For some classes of combinatorial search problems, they shorten the development time significantly while preserving most of the efficiency of imperative languages.

This paper surveys this new class of programming languages from their underlying theory, to their constraint systems, and to their applications to combinatorial problems.

1 Introduction

Combinatorial problems are ubiquitous in computer science. They appear in areas as diverse as operations research (e.g., scheduling), hardware design (e.g., circuit verification), biology (e.g., DNA sequencing), finance (e.g., option trading), and software design (e.g., simulation and testing of protocols), to name a few. Many of these problems are of high complexity ($\mathcal{NP}$-complete or worse), which means that there is no efficient algorithm for solving them. Much research, however, has been spent on designing algorithms to tackle these problems, and one of the interesting outcomes has been the development of constraint-solving algorithms for various classes of problems.

Logic programming (Colmerauer et al., 1973; Kowalski, 1974), on the other hand, is an appealing language to state combinatorial search problems thanks to its relational form and its nondeterminism. The relational form makes it convenient for stating constraints, while the nondeterminism precludes the need for programming a search procedure. Hence it is not surprising that problems such as the 8-queens are part of the folklore of logic programming. Unfortunately, logic programming languages can also be very inefficient when presented with a natural formulation of combinatorial search problems. The main reason comes from the passive use of constraints which only test potential values instead of pruning the search space in an active manner (Gallaire, 1985). As a consequence, logic programming languages (e.g., Prolog) often lead to "generate and test" or "standard backtracking" approaches which exhibit a pathological behaviour known as thrashing (Mackworth, 1977). Much research has been devoted in the logic programming community to improve the computational model of Prolog by introducing, for instance, intelligent backtracking, coroutining and meta-level reasoning. These extensions, while important, are not sufficient to achieve a reasonable efficiency for combinatorial search programs.

Constraint Logic Programming (CLP) (Colmerauer, 1987; Jaffar & Lassez, 1987) can be viewed as a radically different solution to the above problem. Instead of improving the computational model of Prolog, CLP generalizes its basic operation, i.e., unification (a two-way pattern-matching operation), and replaces it by constraint solving over some computation domain (e.g., the integers or the reals). A CLP program is still a set of clauses, but each clause may now contain constraints in its body. At each derivation step, instead of unifying two atoms, the computation checks the satisfiability of a set of constraints. CLP is not a unique language, but rather it defines a class of languages. An instance of the class can be obtained by considering a specific constraint system (or computation domain). More precisely, defining a CLP language amounts to selecting a compu-

tation domain (i.e., a set of allowed constraints and their associated semantics), and to providing a constraint solver for these constraints. Various CLP languages, such as CHIP (Dincbas et al., 1988; Van Hentenryck, 1989b), CLP($\mathcal{R}$) (Jaffar et al., 1990), Prolog III (Colmerauer, 1990), have been defined in recent years, including computation domains such as integers, reals, rationals, and Booleans, to name a few.

From a theoretical standpoint, the move from unification to constraint solving is natural, as unification can be regarded as a simple form of constraint solving (solving equations among first-order terms). Moreover, it turns out that the main theoretical results on the semantics of logic programming carry over naturally to CLP, as shown in the seminal paper of Jaffar & Lassez (1987).

From a practical standpoint, the move to constraint logic programming substantially increases the applicability of logic programming. By providing efficient constraint-solving methods from algebra, artificial intelligence, operations research and logic, CLP languages support, in a declarative way, computational paradigms such as partial enumeration, constraint satisfaction and branch and bound. The resulting languages combine the advantages of logic programming (declarative semantics, relational form, nondeterminism) with the efficiency of special-purpose algorithms. New application areas, including various classes of combinatorial search problems, can now be tackled by these languages, providing a short development time and a competitive efficiency. Especially attractive is the combination of constraint solving and nondeterminism.

The purpose of this paper is to survey this new class of programming languages from their underlying foundations to their constraint solvers and applications. We do not try to provide an historical account on the development of CLP (see Cohen, 1990; Jaffar & Lassez, 1987). Rather, we try to convey the basic ideas behind CLP to a wide audience.

The rest of the paper is organized in the following way. Section 2 presents a simple CLP program and studies its behaviour. The purpose of the example is to give the reader an informal understanding of the computation model of CLP. It is a small problem and the reader should not conclude that CLP is restricted to the solving of small problems. Industrial applications will be considered subsequently in the paper. Section 3 reviews the foundations of CLP, i.e., the declarative and operational semantics of this class of languages. The presentation has been kept as simple as possible, often at a loss of mathematical rigor. However, the basic idea should be accessible to a large audience with little or no effort. Section 4 presents a generalization of the CLP framework: *ask* and *tell* languages. This class originated from the work of Maher (1987) on ALPS, and of Saraswat (1989) on the cc framework. It generalizes the work on coroutining and introduces a new basic operation: constraint entailment. Section 5 studies various computation domains that we consider successful in the solving of industrial problems. Section 6 discusses a number of significant applications, including formal verification of digital circuits (Simonis et al., 1988), car-sequencing (Dincbas et al., 1988c), and simulation of hybrid circuits (Graf et al., 1990). Section 7 contains a description of some other important areas of CLP, not covered in the paper, and directions for further readings.

There are various ways of reading the paper depending on the interest of the reader. Readers interested in applications should read sections 2 and 3 and go directly to section 6. Parts of section 4 are necessary for section 6, but can be read in a lazy manner. Readers interested in CLP techniques in general should read sections 3 and 4. Those interested in the constraint systems should read section 5. Readers familiar with CLP may scan the first three sections and devote more time to the remaining sections. Readers not familiar with CLP and the logic programming literature may skip in a first reading the more technical parts. Those parts are always preceded by an informal presentation that should enable them to read the paper without loss of continuity.

2 A motivating example

As an introductory example, we present a CLP program solving the "send + more = money" puzzle. The problem amounts to finding an assignment of different digits to the letters s, e, n, d, m, o, r, y such that the addition

```
  s e n d
+ m o r e
---------
m o n e y
```

holds. Figure 1 depicts a CLP program over finite domains solving the problem (Van Hentenryck, 1989b).

As the reader can notice, a CLP clause is an expression of the form

$$H \leftarrow c_1, \ldots, c_m \diamond B_1, \ldots, B_n$$

where $H, B_1, \ldots, B_n$ are atoms and $c_1, \ldots, c_m$ are constraints. The symbol $\diamond$ is used in the body of the clause to separate the constraint part from the goals and can be read as "and" in a similar way as ",".

The program illustrates a methodology common to many CLP programs: first state the constraints, then make choices (if necessary).

Stating the constraints is a simple matter: first the domain constraints specifying the possible range of variables; second the disequations expressing that all variables must be distinct and that S and M must be different from 0; finally, the equation specifying the addition constraint. The domain constraints are generated by Procedure `state_domains` which is a simple recursive procedure adding a constraint on each variable. The disequations are generated by the recursive procedure `alldifferent` which generates a disequation between each two variables. Finally, individual constraints are used to state that S and M must be distinct from 0 and to express the

```
sendmory([S,E,N,D,M,O,R,Y]) ←
  state_domains([S,E,N,D,M,O,R,Y]),
  state_constraints([S,E,N,D,M,O,R,Y]),
  generate_values([S,E,N,D,M,O,R,Y]).

state_domains([]).
state_domains([F|T]) ←
  F ∈ 0..9 ◇
  state_domains(T).

state_constraints([S,E,N,D,M,O,R,Y]) ←
  S ≠ 0,
  M ≠ 0,
            1000 × S + 100 × E + 10 × N + D +
            1000 × M + 100 × O + 10 × R + E =
  10000 × M + 1000 × O + 100 × N + 10 × E + Y ◇
  alldifferent([S,E,N,D,M,O,R,Y]).

alldifferent([]).
alldifferent([F|T]) ←
  outof(F,T),
  alldifferent(T).

outof(X,[]).
outof(X,[F|T] ←
  X ≠ F ◇
  outof(X,T).

generate_values([]).
generate_values([F|T]) ←
  member(X,[1,2,3,4,5,6,7,8]),
  generate_values(T).

member(X,[X|Y]).
member(X,[F|T]) ←
  member(X,T).
```

Figure 1 "send + more = money"

	0	1	2	3	4	5	6	7	8	9
S										
E										
N										
D										
M										
O										
R										
Y										

Figure 2 The initial search space

addition constraint. The addition constraint multiplies the individual digits by the power of 10 corresponding to their positions.

Generating values is achieved in the traditional style of logic programming by Procedure `generate_values`. A recursive procedure is used that generates a value to each of the digits. This procedure is nondeterministic (the only nondeterministic procedure used in the above program) and may potentially generate all combinations of values for the digits. The potential search space is 10^8 but, as we will see in a moment, a fraction of that search space will be explored thanks to the use of constraints.

Indeed, consider the behaviour of the program. After execution of predicate `state_domains`, all variables are restricted to take only values between 0 and 9. The search space is depicted in Figure 2. A blank space denotes a possible value, a "−" an inconsistent value, and a "+" an assigned value.

The next step of the computation amounts to introducing the two disequations and the equations. The two disequations prevent S and M from being assigned to 0. The equation in conjunction with the domains enables the system to deduce that M must be smaller than 2 (and hence must be equal to 1 since it cannot be assigned to 0), and similarly that S must be equal to 9. The reason is as follows. The equation can be rewritten as

$$1000S + 91E + 10R + D = 9000M + 900O + 90N + Y.$$

M cannot be greater than 1 since the left-hand side of the equation will never be able to reach 18 000. Also, for the left-hand side to be greater or equal to 9000, it is necessary that S be greater than 8 since the maximal value of

$$91E + 10R + D$$

is obviously smaller than 1000. The search space is now depicted in Figure 3. Note also the possible values for variable O. O cannot be given a value greater than 1 without violating the equation.

	0	1	2	3	4	5	6	7	8	9
S	−	−	−	−	−	−	−	−	−	+
E										
N										
D										
M	−	+	−	−	−	−	−	−	−	−
O			−	−	−	−	−	−	−	−
R										
Y										

Figure 3 The search space after the equation

	0	1	2	3	4	5	6	7	8	9
S	–	–	–	–	–	–	–	–	–	+
E	–	–	–	–					–	–
N	–	–	–	–	–					–
D	–	–								
M	–	+	–	–	–	–	–	–	–	–
O	+	–	–	–	–	–	–	–	–	–
R	–	–								–
Y	–	–								–

Figure 4 The search space after the disequations

Now the system generates the remaining disequations. After this processing step, the search space is depicted in Figure 4. Let us explain briefly why. Since all variables must be different, it is clear that the variables E, R, D, O, N, Y cannot be equal to either 9 or 1. Variable O has only one possible value 0, and the equation at this point reduces to

$$91E + 10R + D = 90N + Y.$$

The system deduces that N must be greater than 3, since the left member is at least greater than 204 (only accounting for the domains). Pursuing similar reasoning entails that N must be greater than or equal to 3 and leads to the search space depicted in Figure 4.

The system then enters the generation phase and assigns first a value to E. The first non-inconsistent value (i.e., 4) is rejected immediately. The constraints are not satisfiable since N must then be equal to 5 and R to 8, leading to the equation

$$D = Y + 6,$$

which would require D to be 8. But this is impossible, since R has been assigned to 8 already, and all digits must be distinct. The second value leads to the unique solution depicted in Figure 5. To give the reader an idea, the time required to find such a solution using a CLP language over finite domains (Van Hentenryck, 1989b) is less than 20 milliseconds on a SUN-4 workstation.

The purpose of the example was to illustrate three points:

1 At each step of the computation, a CLP system tries to check the satisfiability of a set of constraints and to reduce the search space; choices are postponed until no information can be gained from the constraints. In the above example, before even starting the generation of values, the search space had been reduced from 10^8 to $7^3 4^2$.
2 Constraint solving can be naturally interleaved with nondeterminism leading to a sophisticated search procedure; as soon as a choice is made, it is propagated through the constraints,

	0	1	2	3	4	5	6	7	8	9
S	–	–	–	–	–	–	–	–	–	+
E	–	–	–	–	–	+	–	–	–	–
N	–	–	–	–	–	–	+	–	–	–
D	–	–	–	–	–	–	–	+	–	–
M	–	+	–	–	–	–	–	–	–	–
O	+	–	–	–	–	–	–	–	–	–
R	–	–	–	–	–	–	–	–	+	–
Y	–	–	+	–	–	–	–	–	–	–

Figure 5 The solution

producing new information and possibly discovering failures. In the above example, the choice $E = 4$ immediately leads to a failure when propagating the constraints. Also the choice $E = 5$ leads to the assignment of a value to all variables.

3 The programming style remains very close to "generate & test". The constraints are stated first and followed, if necessary, by nondeterministic choices. Hence CLP enables to preserve the declarative aspects of logic programming while providing the efficiency of special-purpose constraint solvers.

In the rest of the paper, we will describe more carefully how the above computation is performed and how constraint solving in general is carried out in CLP languages. We will also discuss applications of these ideas to "industrial" combinatorial problems.

3 The CLP scheme

In this section we present the CLP scheme, i.e., the class of languages whose primitive operation is constraint solving. The CLP scheme was first defined by Jaffar & Lassez (1987). Generalizations were proposed by Maher (1987) and Saraswat (1989), among others. For the purpose of this survey, the presentation will be necessarily informal. The reader interested in more mathematical rigor can refer to the above references, or to the technical report version of this paper. We will first describe the syntax of CLP programs and fix the notation used in the paper. We will then describe the declarative and operational semantics. See Saraswat et al. (1991) for a denotational semantics of CLP languages in terms of information systems and closure operators.

3.1 Syntax

A CLP program is a set of clauses of the form

$$H \leftarrow c_1, \ldots, c_m \diamond B_1, \ldots, B_n$$

or

$$H \leftarrow B_1, \ldots, B_n,$$

where $H, B_1, \ldots, B_n$ are atoms and $c_1, \ldots, c_m$ are constraints.

A CLP goal is a clause without head and constraint

$$\leftarrow B_1, \ldots, B_n.$$

An atom is an expression of the form $p(t_1, \ldots, t_n)$, where $t_1, \ldots, t_n$ are terms. A term is a variable (e.g., X) or a function symbol of arity n applied to n terms (e.g., $f(X, g(Y))$). In this paper, variables are denoted by uppercase letters, constraints by the letter c, conjunctions of constraints by the letter σ, terms by letters t, u, atoms by letters H, B and goals by the letter G, all of them possibly subscripted or superscripted.

To illustrate the declarative and operational semantics, we make use of a simple CLP program, depicted in Figure 6.

```
p(X,Y) ←
    X ≥ Z + 3, Y ≤ Z ◇
    q(X,Y,Z).
q(X,Y,Z) ←
    r(X,Y).
q(X,Y,Z) ←
    Z ≥ Y + 2 ◇.
r(X,Y) ←
    X ≤ Y + 2 ◇.
```

Figure 6 A simple CLP program over rational numbers

3.2 Declarative reading and semantics

CLP programs (and logic programs in general) can be read both declaratively and operationally. Declaratively, a clause

$$H \leftarrow c_1, \ldots, c_m \diamond B_1, \ldots, B_n$$

is an implication

$$H \text{ is true if } c_1 \,\&\, \ldots \,\&\, c_m \text{ and } B_1 \,\&\, \ldots \,\&\, B_n \text{ are true,}$$

where all variables are assumed to be universally quantified. A program is then simply a conjunction of clauses.

In CLP, an answer to a goal is a conjunction of constraints. This conjunction of constraints has the nice property that all its solutions (e.g., all assignments of values to its variables that satisfy the constraints) are also solutions to the goal. Sometimes the result is a constraint consisting only of assignments as in the "send + more = money" program but, in general, it can be any conjunction of constraints. For instance, the goal $\leftarrow p(X, Y)$ when executed using our simple program (see Figure 6) admits a unique solution

$$X \geqslant Y + 5.$$

Note that any assignment of values to X and Y satisfying the constraint (e.g., $X = 5 \,\&\, Y = 0$) is also a solution to the goal. This form of symbolic solution is one of the originalities of the CLP framework, and enables finite representation of infinitely many solutions.

In logical terms, a solution to a goal G in the context of some program P is a conjunction of constraints σ such that the universal closure of the implication $(\sigma \rightarrow G)$[1] is true in all models of the program that are extensions of the models of the constraint system, or in symbols

$$P, \mathfrak{C} \models (\forall)(\sigma \rightarrow G),$$

where $\mathfrak{C}$ is a structure formalizing the constraint system.

3.3 Operational semantics

The operational reading of CLP programs is a generalization of the operational reading of procedure calls in imperative languages. Operationally, a clause

$$H \leftarrow c_1, \ldots, c_m \diamond B_1, \ldots, B_n$$

can be read as

"To solve H,
solve $B_1 \,\&\, \ldots \,\&\, B_n$
provided that $c_1 \,\&\, \ldots\, c_m$ are satisfied."

A goal

$$\leftarrow B_1, \ldots, B_n$$

can be read as

Solve $B_1 \,\&\, \ldots \,\&\, B_n$.

Note that, since there may be several clauses with the same head, the computation model is essentially nondeterministic: any atom can be solved in various ways. We now turn to the

[1] The universal closure of a formula amounts to quantifying universally all its free variables.

operational semantics itself. We first present it in an informal way and then formalize it in terms of transition systems.

3.3.1 Informal presentation

The operational semantics is a simple (at least from a conceptual standpoint) generalization of the semantics of logic programming. It can be described as a goal-directed derivation procedure from the initial goal using the program clauses. A *computation state* is best described by

1 *a goal part*: the conjunction of goals that remains to be solved;
2 *a constraint store*: the set of constraints accumulated up to this point of the execution.

Initially the constraint store is empty and the goal part is the initial goal. In the following, we denote the computation state by pairs $\langle G, \sigma \rangle$, where G is the goal part and σ is the constraint store. We use ε to denote an empty goal part or constraint store. An example of computation state is

$$\langle q(X, Y, Z), X \geqslant Z + 3 \,\&\, Y \leqslant Z \rangle.$$

A *computation step* (i.e., the transition from a computation state to another computation state) amounts to

1 Selecting an atom in the goal part;
2 Finding a clause such that the clause constraints and the constraint store are satisfiable;
3 Defining the new computation state as the old one where the selected atom has been replaced by the atoms in the body of the clause and the clause constraints have been added to the constraint store.

For instance, given a computation state

$$\langle q(X, Y, Z), X \geqslant Z + 3 \,\&\, Y \leqslant Z \rangle,$$

a computation step can be performed using clause 2 of q to obtain a new computation state

$$\langle \varepsilon, X \geqslant Z + 3 \,\&\, Y \leqslant Z \,\&\, Z \geqslant Y + 2 \rangle$$

as the resulting constraint store is satisfiable.

As should be clear, the basic operation of the language is step 2 which amounts to deciding the satisfiability of a conjunction of constraints. Note also that each computation state has a satisfiable constraint store. This property is exploited inside CLP languages to avoid solving the satisfiability problem from scratch at each step. Instead, CLP languages keep a reduced (e.g., solved) form for the constraints and transform the existing solution into a solution including the new constraints. Hence the constraint solver is made incremental.

A computation state is *terminal* if

- the goal part is empty;
- no clause can be applied to the selected atom to produce a new computation state.

A *computation* is simply a sequence of computation steps ending in a terminal computation state or *diverging*. A finite computation is *successful* if the final computation state has an empty goal. It is *failed* otherwise.

To illustrate computations in a CLP language consider our simple program again. The program has only one successful computation depicted as follows:

$$\begin{array}{l} \langle p(X, Y), \varepsilon \rangle \\ \qquad \downarrow \qquad\qquad\qquad \text{(using clause 1 of } p\text{)} \\ \langle q(X, Y, Z), X \geqslant Z + 3 \,\&\, Y \leqslant Z \rangle \\ \qquad \downarrow \qquad\qquad\qquad \text{(using clause 2 of } q\text{)} \\ \langle \varepsilon, X \geqslant Z + 3 \,\&\, Y \leqslant Z \,\&\, Z \geqslant Y + 2 \rangle. \end{array}$$

The program has also one failed computation:

$$\begin{array}{ll} \langle p(X, Y), \varepsilon \rangle & \\ \quad\downarrow & \texttt{(using clause 1 of } p\texttt{)} \\ \langle q(X, Y, Z), X \geqslant Z + 3 \,\&\, Y \leqslant Z \rangle & \\ \quad\downarrow & \texttt{(using clause 1 of } q\texttt{)} \\ \langle r(X, Y), X \geqslant Z + 3 \,\&\, Y \leqslant Z \rangle. & \end{array}$$

The last computation state is terminal since the conjunction of constraints

$$X \geqslant Z + 3 \,\&\, Y \leqslant Z \,\&\, X \leqslant Y + 2$$

is not satisfiable.

A number of important remarks need to be stressed at this point. First the results of the computation are the constraint stores of the successful computations. In general, CLP languages try to present the constraint store in a way meaningful from the user standpoint, for instance by making solutions (or non-solutions) explicit. In particular, when the constraint system enables elimination of variables, the solution will be presented only in terms of the query variables. For instance, the result of the successful computation for our simple program will be

$$X \geqslant Y + 5,$$

which is the projection of the constraint store on variables X and Y.

Second, the operational semantics is sound and complete wrt the declarative semantics. In other words, all solutions produced by the operational semantics are correct and there exists a successful computation associated with each solution.

Finally, nothing has been said so far on the strategy used to explore the space of computations. Most CLP languages use a computation model similar to Prolog. Atoms are selected from left to right in the clauses, clauses are tried in textual ordering, and the search space is explored in a depth-first manner with chronological backtracking in case of failures.[2] For instance, on the simple program, a CLP language will typically use clause 1 for p, then clause 1 for q, and will finally encounter a failure when trying to solve r. Execution will then backtrack to clause 2 of q giving the successful computation.

3.3.2 Formalization

To describe precisely the operational semantics, we make use of a transition system (see Plotkin, 1981; Saraswat, 1989). Although this section is slightly more technical, it should be accessible to most readers.[3]

Definition 1 A transition system is a triple $\langle \Gamma, T, \mapsto \rangle$ where Γ is a set of configurations, $T \subseteq \Gamma$ is the set of terminal configurations, and $\mapsto \subseteq \Gamma \times \Gamma$ is the transition relation satisfying

$$\forall \gamma \in T, \quad \forall \gamma' \in \Gamma, \quad \gamma \not\mapsto \gamma'.$$

The configurations of the transition system are the computation states $\langle G, \sigma \rangle$. When the goal part is empty, we take the convention of representing the configuration by the constraint part only. Similarly, when the constraint part is empty, we take the convention of representing the configuration by the goal part only. Terminal configurations are simply successful computation states.

A transition $\gamma \mapsto \gamma'$ can be read as "configuration γ nondeterministically reduces to γ'". The transition rules in this paper are presented using the format

[2] More sophisticated search procedures are studied in the context of *ask* and *tell* languages.
[3] This section may be skipped in a first reading.

$$\frac{\begin{array}{l}< \text{condition 1} > \\ \cdots \\ < \text{condition n} >\end{array}}{\gamma \mapsto \gamma'}$$

expressing the fact that a transition from γ to γ' can take place if the conditions are fulfilled.

There is only one transition rule necessary to define the operational semantics:

$$\frac{\begin{array}{l}1.\ p(u_1, \ldots, u_n) \leftarrow \sigma' \diamond B_1, \ldots, B_m \in P \\ 2.\ \mathcal{C} \models (\exists)(\sigma \wedge \sigma' \wedge t_1 = u_1 \wedge \ldots \wedge t_n = u_n)\end{array}}{\langle p(t_1, \ldots, t_n)\ \&\ G, \sigma\rangle \mapsto \langle B_1\ \&\ \ldots\ \&\ B_m\ \&\ G, \sigma\ \&\ \sigma'\ \&\ t_1 = u_1\ \&\ \ldots\ \&\ t_n = u_n\rangle}$$

In the above transition rule, $p(t_1, \ldots, t_n)$ is the selected atom (it can be any atom in the goal since the order in a conjunction is irrelevant), P denotes the program, $(\exists)(\psi)$ represents the existential closure of ψ, and the program clause has been renamed properly not to share any variable with the goal.[4] The rule expresses formally what was presented informally in the last section. If there exists a clause in the program with the same predicate name as the selected atom (condition 1) and if the clause constraints are consistent with the constraint store (condition 2), then a computation step is possible. The new computation state is obtained from the old one by replacing the selected atom by the goals in the body and adding the constraints to the constraint store. Note also the equations between the arguments of the clause head and the selected atom.

The actual operational semantics of the language can be defined in terms of its success, divergence, and failure sets. We use the notation $P \vdash$ to denote the fact that the transition occurs in the context of program P. We denote by $\overset{*}{\mapsto}$ the transitive closure of $\mapsto$ and say that a configuration γ diverges in program P if there exists an infinite sequence of transitions $P \vdash \gamma \mapsto \gamma_1 \mapsto \cdots \mapsto \gamma_i \mapsto \cdots$.

The success and divergence sets are defined in the following way:

$$SS[P] = \{G | P \vdash G \overset{*}{\mapsto} \sigma\}$$
$$DS[P] = \{G | G \text{ diverges in } P\}.$$

The failure set can now be defined in terms of the above two sets:

$$FS[P] = \{G | G \notin SS[P] \cup DS[P]\}.$$

Another semantic definition can be given to capture the results of the computation:

$$RES[P, G] = \{\sigma | P \vdash G \overset{*}{\mapsto} \sigma\}.$$

In order to achieve the above semantics, the CLP language should be embedded with a complete constraint solver which means that, given a constraint σ, the constraint solver should return *true* if $\mathcal{C} \models (\exists)(\sigma)$, and *false* otherwise.

We now mention various results relating the declarative and operational semantics. The original versions of these theorems can be found in Jaffar & Lassez (1987), Maher (1987) and Saraswat (1989). The first result we mention is the soundness of the operational semantics.

Theorem 2 [Soundness] *Let P be a program, G a goal, and σ a constraint. We have*

$$\sigma \in RES[P, G] \quad \text{implies} \quad \mathcal{P}, \mathcal{C} \models (\forall)(\sigma \rightarrow G).$$
$$G \in FS[P] \quad \text{implies} \quad \mathcal{P}, \mathcal{C} \models \neg(\exists)(G).$$

The next theorem is the equivalent for CLP languages of the strong completeness theorem for logic programming.[5]

Theorem 3 [Strong Completeness] *Let P be a program, G be a goal, and σ a constraint. If $\mathcal{P}, \mathcal{C} \models (\forall)(\sigma \rightarrow G)$ and $\mathcal{C} \models (\exists)(\sigma)$, then there exist constraints $\sigma_1, \ldots, \sigma_n \in RES[P, G]$ such that*

[4] In recent work, Saraswat et al. (1991) give an operational semantics precluding the need for renaming.

[5] It is clear that a particular strategy (e.g., depth-first search) may lead to incompleteness.

$$\mathcal{C} \models (\forall)(\sigma \rightarrow (\exists y_1, \ldots, y_m)(\sigma_1 \vee \ldots \vee \sigma_n)),$$

where $y_1, \ldots, y_n$ *are variables appearing in* $\sigma_1, \ldots, \sigma_n$ *and not in* σ.

3.4 CLP languages

The above framework defines a class of programming languages parametrized by the constraint system. Several "real" CLP languages have been designed and implemented on various constraint systems. The most well-known of these languages are probably CHIP (Dincbas et al., 1988; Van Hentenryck, 1989b), CLP($\Re$) (Jaffar et al., 1990), Prolog III (Colmerauer, 1990), and Trilogy (Voda, 1988). Their constraint systems include Boolean Algebra (CHIP, Prolog III), linear rational arithmetic (CHIP, Prolog III), linear real arithmetic (CLP($\Re$)), linear integer arithmetic (Trilogy), and finite domains (CHIP). Prolog III also includes a restricted form of list concatenation. Note that some of the languages use different constraint solvers for the same constraint systems. Many other CLP languages have been, or are currently being, defined and implemented. They include BNR-Prolog (Older & Vellino, 1990) (approximation of real arithmetic), CAL (Aiba et al., 1988) (complex numbers), LOGIN (Ait-kaci & Nasr, 1986) and its successor LIFE (Ait-kaci & Podeloki, 1990) (equations between ψ-terms), CLP(Σ^*) (Walinsky, 1989) (strings), and Tangram (Parker & Muntz, 1988) (streams). Finally, Saraswat (1989) and Saraswat et al. (1990) show how various data-structures (e.g., arrays, hashtables, bags) can be seen as constraint systems.

4 Ask and tell languages

CLP languages have one primitive operation: constraint solving. These languages are substantial generalizations of logic programming, and open new application areas for which logic programming was not always best suited. However, CLP languages are sometimes not expressive enough from an operational standpoint to accommodate some reasoning techniques about constraints. Extending the CLP framework is thus an important area of research.

In this section we discuss one such generalization, *ask* and *tell* languages, i.e., the class of CLP languages with two primitive operations: constraint solving (i.e., *tell* a constraint) and constraint entailment (i.e., *ask* a constraint) (Saraswat, 1989). Constraint entailment amounts to finding out if a single constraint c is entailed by a conjunction of constraint σ, i.e.

$$\mathcal{C} \models (\forall)(\sigma \rightarrow c).$$

Constraint entailment algorithms can be derived from constraint-solving algorithms (provided that the negation of a basic constraint be a constraint, which is the case in the constraint systems we have studied so far). Hence we will not describe them in this paper. Note, however, that it is sometimes more efficient to devise specialized constraint-entailment algorithms instead of using constraint-solving algorithms.

Constraint entailment was introduced in the context of concurrent logic programming (e.g., Shapiro, 1990) by Maher (1987) to endow these languages with a logical semantics. It can be viewed as well as a generalization of languages allowing coroutining and delay mechanisms (e.g., Clark & McCabe, 1979; Colmerauer et al., 1983; Dincbas & Lepape, 1984; Gallaire & Lasserre, 1982; Naish, 1985). The concept of *ask* and *tell* languages was introduced by Saraswat (1989) in the context of concurrent constraint programming. In concurrent constraint programming, constraint entailment is used to synchronize concurrently executing agents. Saraswat (1989) provides a comprehensive account of the use of constraints for concurrency from the theory to its applications. Constraint entailment was also used in CHIP (see Dincbas et al., 1988; Graf et al., 1989, 1990) inside the `if_then_else` construct, and was instrumental in simulating hybrid circuits. Its interest for CLP languages lies in the opportunity to reason about the constraints and to use the gained information for achieving pruning. As we will see, it can be used to express non-primitive constraints following general principles from artificial intelligence and operations research.

Constraint entailment can be used in various ways. We present two examples of its use. The first example is the implication operator introduced in Saraswat (1989) in the context of concurrent logic programming. The second example is the cardinality operator proposed explicitly for CLP languages (Van Hentenryck & Deville, 1991).

4.1 The implication operator

There are very many ways of using Boolean constraints. One of them, which has turned out to be useful for many applications such as diagnostic and test generation of circuits, is local propagation (Sussman & Steele, 1980; Davis, 1984). The idea here is to deduce values for some variables given the values of other variables. For instance, an "and-gate" may be defined by rules of the form

> "If one input is 0 then the output is 0",
> "If the output is 1 then the inputs are both 1".

To implement a program achieving this form of propagation, it is necessary to introduce a form of data (or constraint-driven) computation where goals are suspended (when not enough information is available) and reactivated (when new information require to reconsider them). The purpose of the implication operator for CLP languages is precisely to achieve this form of behaviour and to generalize it to any constraint system.

4.1.1 Syntax and semantics

Syntax The implication operator has the form $c \Rightarrow A$ where c is a constraint and A is an atom.[6] We generalize the syntax of CLP languages by allowing the implication operator in addition to atoms in the body of clauses.

Declarative semantics The declarative semantics of the implication operator is given by logical implication $c \rightarrow A$.

Operational semantics: informal presentation The main originality of the implication operator lies in the operational semantics which is based on constraint entailment. The intuition is the following. The implication $c \Rightarrow A$ makes sure that A is executed only when (and as soon as) c is entailed by the constraint store. In other words, if c is entailed by the constraint store, $c \Rightarrow A$ reduces to A. If $\neg c$ is entailed by the constraint store, $c \Rightarrow A$ reduces to *true*. Otherwise, the computation flounders, waiting for more information.

Consider again the description of an "and-gate" using local propagation techniques:

```
and(X,Y,Z) ←
     X = 0 ⇒ Z = 0,
     Y = 0 ⇒ Z = 0,
     Z = 1 ⇒ (X = 1, Y = 1),
     X = 1 ⇒ Y = Z,
     Y = 1 ⇒ X = Z.
```

The above example originates from the demon declarations of CHIP. The above modelling is best for applications such as diagnostics (Simonis & Dincbas, 1987b) and test generation (Simonis, 1989) of digital circuits. The remainder of this section was inspired by Simonis & Dincbas (1987b).

The first rule expresses that, as soon as the constraint store entails `X = 0`, the constraint `Z = 0` must be added to the constraint store. Note also the last two rules which actually achieve more than local propagation of values; they also propagate symbolic equalities. Now the goal $\langle and(X, Y, Z), X = 0\rangle$ will produce a constraint store $X = 0$ & $Z = 0$, since the goal $\langle X = 0 \Rightarrow Z = 0, X = 0\rangle$ reduces to $\langle Z = 0, X = 0\rangle$ and hence to the constraint store $X = 0$ & $Z = 0$. However the goal

[6] In the example, we relax the syntax and allow A to be a constraint as well.

$\langle and(X, Y, Z), \varepsilon\rangle$ does not modify the constraint store, since none of the constraints in the implication constructs are entailed by the constraint store.

As mentioned previously, a goal which has floundered can be resumed when new information becomes available in the constraint store. Assume for instance the computation state

$$\langle X = 0 \Rightarrow Z = 0 \;\&\; T = 0 \Rightarrow X = 0,\ T = 0\rangle.$$

The first goal $X = 0 \Rightarrow Z = 0$ flounders, since $X = 0$ is not entailed by the constraint store. But the second goal can be executed leading eventually to the computation state

$$\langle X = 0 \Rightarrow Z = 0,\ X = 0 \;\&\; T = 0\rangle.$$

Now $X = 0$ is entailed by the constraint store and hence the first implication can be executed. The final constraint store will be $X = 0 \;\&\; T = 0 \;\&\; Z = 0$.

Now assume that we would like to build a full-adder using logical gates:

```
fa(X,Y,Cin,S,C) ←
   and(X,Y,C1),
   xor(X,Y,S1),
   and(Cin,S1,C2),
   xor(Cin,S1,S),
   or(C1,C2,C).
```

In the above circuit, `X`, `Y` are two input bits, `Cin` is the carry-in, `S` is the result bit and `C` is the carry-out. Now, using the implication operator to define all logical gates, the query `fa(X,Y,1,S,0)` will produce the constraint store

$$X = 0 \;\&\; Y = 0 \;\&\; S = 1.$$

The reason is the following. Since the result of the or-gate is 0, its two inputs `C1`, `C2` must be 0. Since the second and-gate has output `C2` equal to 0, and input `Cin` equal to 1, it follows that `S1` must be zero, which implies that `X` and `Y` must be equal because of the first xor-gate. Since `X,Y` appear both as inputs in the same and-gate, they must be equal to its output `C1` which is 0.

The implication operator thus introduces a notion of coroutining between goals in the language and the execution of goals can be interleaved in very complex ways. Note that the goals synchronize by "asking" if some constraints are entailed by the constraint store. Also, a suspended goal can be resumed by a modification of the constraint store by other goals. Note finally that the above example used very simple constraints. Nothing prevents us from asking more sophisticated constraints in more complex constraint systems as those presented later in this paper.

Operational semantics: formalization We now describe precisely the semantics of an *ask* and *tell* language including the implication operator.[7] First note that the configurations have to be generalized to include the terminal `flounder`. This terminal is necessary as not enough information might be available to decide entailment of all implications. In this case, the computation is said to flounder. Moreover, it is necessary to model conjunction explicitly in order to capture the interaction between the implication construct and constraint solving. We now turn to the transition rules which are based on those presented in Saraswat (1989).

Goal reduction: a goal can be reduced to the body of a clause if the constraint store is consistent with the new constraints.

$$\frac{p(u_1, \ldots, u_n) \leftarrow \sigma' \diamond B_1, \ldots, B_m \in P \qquad \mathcal{C} \models (\exists)(\sigma \wedge \sigma' \wedge t_1 = u_1 \wedge \ldots \wedge t_n = u_n)}{\langle p(t_1, \ldots, t_n), \sigma\rangle \mapsto \langle B_1 \;\&\; \ldots \;\&\; B_m,\ \sigma \;\&\; \sigma' \;\&\; t_1 = u_1 \;\&\; \ldots \;\&\; t_n = u_n\rangle}$$

Implication: an implication $c \Rightarrow A$ never fails. If c is entailed by the constraint store, it reduces to

[7] This paragraph may be skipped in a first reading.

the body A. If $\neg|c$ is entailed by the constraint store, the implication terminates successfully. Otherwise, the implication flounders:

$$\frac{\mathcal{C} \models (\forall)(\sigma \rightarrow c)}{\langle c \Rightarrow A, \sigma\rangle \mapsto \langle A, \sigma\rangle}$$

$$\frac{\mathcal{C} \models (\forall)(\sigma \rightarrow \neg|c)}{\langle c \Rightarrow A, \sigma\rangle \mapsto \sigma}$$

$$\frac{\mathcal{C} \models \neg|\,(\forall)(\sigma \rightarrow c) \quad \mathcal{C} \models \neg|(\forall)(\sigma \rightarrow \neg|c)}{\langle c \Rightarrow A, \sigma\rangle \mapsto \mathit{flounder}}$$

Conjunction: if any of the goals in a conjunction can make a transition, the whole conjunction can make a transition as well and the constraint store is updated accordingly. This is the traditional interleaving rule:

$$\frac{\langle G_1, \sigma\rangle \mapsto \langle G_1', \sigma'\rangle}{\langle G_1 \,\&\, G_2\, \sigma\rangle \mapsto \langle G_1' \,\&\, G_2, \sigma'\rangle} \qquad \langle G_2 \,\&\, G_1, \sigma\rangle \mapsto \langle G_2 \,\&\, G_1', \sigma'\rangle$$

$$\frac{\langle G_1, \sigma\rangle \mapsto \sigma'}{\langle G_1 \,\&\, G_2, \sigma\rangle \mapsto \langle G_2, \sigma'\rangle} \qquad \langle G_2 \,\&\, G_1, \sigma\rangle \mapsto \langle G_2, \sigma'\rangle$$

We are now ready to propose the new operational semantics. The operational semantics is now given in terms of three sets: the success, divergence, and floundering sets

$$SS[P] = \{G | P \vdash G \overset{*}{\mapsto} \sigma\}$$
$$DS[P] = \{G | G \text{ diverges in } P\}$$
$$FLS[P] = \{G | P \vdash G \overset{*}{\mapsto} \mathit{flounder}\}$$

The failure set can now be defined in terms of the above three sets

$$FS[P] = \{G | G \notin SS[P] \cup DS[P] \cup FLS[P]\}$$

Another semantic definition can be given to capture the results of the computation

$$RES[P, G] = \{\sigma | P \vdash G \overset{*}{\mapsto} \sigma\}$$

Note that the completeness theorem does not hold anymore for the above *ask* and *tell* language, the implication operator introducing a gap between the declarative and operational semantics. This is obviously an intentional decision. Since the operator does not add any theoretical expressive power, we could very well do without it. However, the language has more additional operational expressiveness which is useful for programming purposes.

4.2 The cardinality operator

In this section, we present a second operator using constraint entailment: the cardinality operator (Van Hentenryck & Deville, 1991). The cardinality operator is a declarative and relational operator, especially designed for CLP languages. It originates from an attempt to provide the user with a suitable and uniform way of expressing his own constraints (i.e., constraints that are not primitives of the language). It subsumes a number of constraints and control mechanisms that were previously introduced in a somewhat ad-hoc manner, yet preserving their efficiency. In the following, we will only present a restricted version of the operator. Before entering into the description of the operator, let us give an example to motivate the reader. Consider, for instance, a scheduling problem and assume that we face a disjunctive constraint between two tasks, i.e., the execution of the two tasks cannot overlap. Assume that $S1$, $S2$ represent the starting dates of the tasks and $D1$, $D2$ their durations, the constraint can be expressed as

```
disjunctive(S1,D1,S2,D2) ←
  S1 + D1 ≤ S2 ◇.
disjunctive(S1,D1,S2,D2) ←
  S2 + D2 ≤ S1 ◇.
```

Unfortunately, the above constraint is nondeterministic and will introduce choice points during the execution. The first alternative, i.e., task 2 is scheduled after task 1, will be selected and its constraint will be added to the constraint store. Subsequent execution may lead to a failure and require this choice to be reconsidered. The second alternative, i.e., task 1 is scheduled after task 2, will then be considered. In general, it is better to postpone choices as long as possible. The above constraint can be used in two (symmetric) ways to achieve pruning: (1) if the maximal start date of S_2 is smaller than the minimal start date of S_1 added to D_1, then task 2 cannot be scheduled after task 1 and (2) if the maximal start date of S_1 is smaller than the minimal start date of S_2 added to D_2, then task 1 cannot be scheduled after task 2. The cardinality operator will enable us to express this pruning in a very natural way.

4.2.1 Syntax and semantics

Syntax The cardinality operator has the form

$$\#(l, u, [c_1, \ldots, c_n])$$

where l, u are integers and $c_1, \ldots, c_n$ are constraints. The syntax of CLP programs is generalized to allow for the cardinality operator in the body of the clause.

Declarative semantics The declarative semantics requires a generalization of satisfiability. The truth value of $\#(l, u, [c_1, \ldots, c_n])$ is true if the number of constraints c_i $(1 \leq i \leq n)$ assigned to true is not less than l and not more than u. It is false otherwise.

Note that this operator is quite expressive. A conjunction $c_1 \wedge \ldots \wedge c_n$ can be expressed as $\#(n, *, [c_1, \ldots, c_n])$ where $*$ is a don't care value. A disjunction $c_1 \vee \ldots \vee c_n$ as $\#(1, *, [c_1, \ldots, c_n])$ and $\neg c$ as $\#(*, 0, [c])$. Other connectives such as equivalence $\Leftrightarrow$ can now be obtained easily. In the examples below, we feel free to use the logical operators instead of the cardinality operator when convenient.

Using the cardinality operator, the disjunctive constraint can now be implemented as follows:

```
disjunction(S1,D1,S2,D2) ←
  #(1,*,[S1 + D1 ≤ S2, S2 + D2 ≤ S1]).
```

Operational semantics: informal presentation Once again, the main interest of the cardinality operator lies in its operational semantics. The operator implements a principle well-known in operations research and artificial intelligence: "infer simple constraints from difficult ones". The intuitive idea is to make sure that the cardinality operator can be satisfied in some ways. Moreover, if there is only one way to satisfy it, then the constraints necessary to satisfy it are introduced in the constraint store. To check if there is a way to satisfy the constraint, constraint entailment is used. In the disjunctive example, the system makes sure that either the first task can be scheduled before the second one or the second task can be scheduled before the first one (or both). If the constraint store entails that the first task cannot be scheduled before the second one, then a constraint forcing the second task to be scheduled first will be added to the constraint store.

Let us take a simple example to clarify further the behaviour.

```
⟨#(1,2,[X = 4, Y = 10]) & X > 6, ε⟩
                ↓
⟨#(1,2,[X = 4, Y = 10]), X > 6⟩
                ↓
⟨#(1,2,[Y = 10]), X > 6⟩
                ↓
        X > 6 & Y = 10
```

In the above example, we have a cardinality operator requiring that $X = 4$ or $Y = 10$ be true. Initially, none of these two constraints are entailed by the constraint store. Also, none of their negations is entailed by the constraint store. So the execution of the cardinality operator flounders. The second goal $X > 6$ is selected which implies that $X \neq 4$ is entailed by the constraint store. There is only one way now to satisfy the cardinality operator, i.e., adding the constraint `Y = 10` to the constraint store.

Operational semantics: formalization The precise behaviour of the operator can be described by the following transition rules taken from Van Hentenryck & Deville (1991).[8]

Trivial satisfaction: if $l \leq 0$ and u is greater than or equal to the number of constraints $c_1, \ldots, c_n$, then $\#(l, u, [c_1, \ldots, c_n])$ is trivially satisfied:

$$\frac{l \leq 0 \wedge n \leq u}{\langle \#(l, u, [c_1, \ldots, c_n]), \sigma \rangle \mapsto \sigma}$$

Positive satisfaction: a formula $\#(n, u, [c_1, \ldots, c_n])$ with $n \leq u$ can be satisfied only if the conjunction $c_1 \wedge \ldots \wedge c_n$ is consistent with the constraint store:

$$\frac{l \leq u \wedge l = n \qquad \mathcal{C} \models (\exists)(\sigma \wedge c_1 \wedge \ldots \wedge c_n)}{\langle \#(l, u, [c_1, \ldots, c_n]), \sigma \rangle \mapsto \sigma \,\&\, c_1 \,\&\, \ldots \,\&\, c_n}$$

Negative satisfaction: a formula $\#(l, 0, [c_1, \ldots, c_n])$ with $l \leq 0$ can be satisfied only if the conjunction $\neg c_0 \wedge \ldots \wedge \neg c_n$ is consistent with the constraint store:

$$\frac{l \leq u \wedge u = 0 \qquad \mathcal{C} \models (\exists)(\sigma \wedge \neg c_1 \wedge \ldots \wedge \neg c_n)}{\langle \#(l, u, [c_1, \ldots, c_n]), \sigma \rangle \mapsto \sigma \,\&\, \neg c_1 \,\&\, \ldots \,\&\, \neg c_n}$$

The above three rules make up the basic cases for the cardinality operator. Two of them allow the interference of primitive constraints, and hence prune the search space with the help of the transition rules for conjunction.

Positive reduction: when a constraint c_i is entailed by the constraint store, the cardinality formula can be simplified by dropping the constraint and decrementing the bounds

$$\frac{\mathcal{C} \models (\forall)(\sigma \rightarrow c_i) \qquad 0 < l < n \wedge l \leq u \vee 0 < u < n \wedge l \leq 0}{\langle \#(l, u, [c_1, \ldots, c_i, \ldots, c_n]), \sigma \rangle \mapsto \langle \#(l - 1, u - 1, [c_1, \ldots, c_{i-1}, c_{i+1}, \ldots, c_n]), \sigma \rangle}$$

The condition on l and u forces the rule to be mutually exclusive with the three satisfaction rules.

Negative reduction: when the negation of a constraint c_i is entailed by the constraint store (i.e., c_i inconsistent with σ), the cardinality formula can be simplified by dropping the constraint:

$$\frac{\mathcal{C} \models (\forall)(\sigma \rightarrow \neg c_i) \qquad 0 < l < n \wedge l \leq u \vee 0 < u < n \wedge l \leq 0}{\langle \#(l, u, [c_1, \ldots, c_i, \ldots, c_n]), \sigma \rangle \mapsto \langle \#(l, u, [c_1, \ldots, c_{i-1}, c_{i+1}, \ldots, c_n]), \sigma \rangle}$$

The above two rules achieve progress towards the satisfaction rules by reducing the number of constraints and (possibly) the bounds. But the computation with the cardinality operator may now flounder as none of the constraints can be decided upon (for entailment) wrt the constraint store.

Floundering: the cardinality operator flounders if there is no constraint c_i such that either c_i or its negation $\neg c_i$ is entailed by the constraint store and none of the satisfaction rules apply:

[8]This paragraph may be skipped in a first reading.

$$\frac{\mathcal{C} \not\models (\forall)(\sigma \rightarrow c_j) \quad \text{for all } 1 \leq j \leq n \qquad \mathcal{C} \not\models (\forall)(\sigma \rightarrow \neg c_j) \quad \text{for all } 1 \leq j \leq n \qquad 0 < l < n \wedge l \leq u \vee 0 < u < n \wedge l \leq 0}{\langle \#(l, u, [c_1, \ldots, c_n]), \sigma \rangle \mapsto \textit{flounder}}$$

We reconsider now our simple example and indicate which transition rules were used in the derivation

$$\begin{array}{l} \langle \#(1, 2, [\mathtt{X} = 4, \mathtt{Y} = 10]) \,\&\, \mathtt{X} > 6, \varepsilon \rangle \\ \qquad\qquad \downarrow \quad \textit{conjunction} \\ \langle \#(1, 2, [\mathtt{X} = 4, \mathtt{Y} = 10]), \mathtt{X} > 6 \rangle \\ \qquad\qquad \downarrow \quad \textit{negative reduction} \\ \langle \#(1, 2, [\mathtt{Y} = 10]), \mathtt{X} > 6 \rangle \\ \qquad\qquad \downarrow \quad \textit{positive satisfaction} \\ \qquad \mathtt{X} > 6 \,\&\, \mathtt{Y} = 10 \end{array}$$

4.2.2 Examples

Let us now describe two important constraints that can be easily described by the cardinality operator. The first example is used to implement the `element` constraint which has been instrumental in the solving of a variety of combinatorial problems using finite domains. Declaratively, `element(X,[v1,...,vn],E)` holds if `E` is equal to v_X ($1 \leq X \leq n$). Operationally, $v_1, \ldots, v_n$ are assumed to be integers (non necessarily distinct), `X` and `E` are assumed to be variables. The predicate enforces a dependency between $X \in \{1, \ldots, n\}$ and $E \in \{v_1, \ldots, v_n\}$ so that each time the domain of `X` is modified, the domain of `E` is updated accordingly, and vice versa. The operational semantics can be characterized by a conjunction of constraints, with one constraint of the following form for each different value $v \in \{v_1, \ldots, v_n\}$

$$E = v \Leftrightarrow X \in \{i_1, \ldots, i_m\},$$

where $v_{i_1}, \ldots, v_{i_m}$ are all the elements equal to v. A simple logic program can be written to generate the above constraints.

The second example is the `atmost` constraint which will be used later on in the car-sequencing problem. Declaratively the constraint `atmost(Nb,[X1,...,Xn],Val)` which holds if at most `Nb` elements X_i are equal to `Val`. The expected operational behaviour of the constraint in the application (the one that the programmer would use in an imperative language) can be described in the following way. The constraint has to check that, at any time of the computation, the constraint store does not entail more than `Nb` elements X_i to be equal to `Val`. Moreover, if exactly `Nb` elements X_i are entailed to be equal to `Val`, then all other elements must be forced to be different from `Val`. The `atmost` constraint can be implemented as follows using the meta-level facility to build constraints dynamically

```
atmost(Nb,L,Val) ←
     collect(L,Val,Lcstrs),
     #(*,Nb,Lcstrs).

collect([],Val,[]).
collect([X|Xs],Val,[X = Val|Lcstrs]) ←
     collect(Xs,Val,Lcstrs).
```

Intuitively, the collect predicate constructs the list of constraints and the cardinality operator uses the list to make sure that at most `Nb` of them are true. Given a goal "`←atmost(1,[X1...,X3],4)`", the above program generates the cardinality formula

$$\#(*, 1, [X_1 = 4, X_2 = 4, X_3 = 4]).$$

Hence, as soon as one of the variables is fixed to be equal to 4 by the constraint store, the two other variables are constrained to be different from 4.

5 Constraint systems

The semantics described in the previous section assume an underlying constraint system as well as a constraint solver for deciding the satisfiability of constraints. The present section considers various constraint systems, describes their syntax, semantics and constraint-solving algorithms, and illustrates their use on simple examples. Three constraint systems are studied in detail: Boolean algebra, linear rational arithmetic, and finite domains. The three constraint systems were chosen because their constraint solvers are well-documented, and because they have numerous industrial applications. Other constraint systems were briefly mentioned in section 3. Note also that the most basic constraint system is the Herbrand system which contains only equations between first-order terms and is included in all CLP languages.

Before studying the various constraint systems, it is worth mentioning some of the requirements of CLP languages on the constraint solvers. First the constraint solver should be incremental, i.e., it should use the fact that the constraint store is consistent when adding a new constraint. Incrementality is closely related to the concept of solved form for the constraints. Also, since the constraint solver is to be used in a nondeterministic setting, provisions may be necessary to support backtracking. Second the constraint solver should be complete, i.e., it should provide a decision procedure for the constraints. Finally, the constraint solver should be efficient, as constraint solving is the primitive operation of the language. The above requirements, especially the last two, are somewhat conflicting. Although it is clear that a complete constraint solver is desirable, there are very few decision procedures and some of them are of very high complexity. Hence, in practice, they might turn out to be inappropriate. On the other hand, it may be possible to design efficient constraint-solving algorithms that are incomplete, yet useful in many industrial applications. Eventually the design of a CLP language is a tradeoff between expressiveness, completeness, and efficiency of the language. The language will be successful if it supports adequately the paradigms underlying a class of problems.

5.1 Boolean algebra

Constraints on Boolean variables play a fundamental role in computer science, both in theory and in practice. Boolean constraint solving has also been studied for a very long time, starting with Boole himself. (See Radeanu, 1974 for a comprehensive presentation of Boolean algebra.) Applications of Boolean constraint solving are numerous, especially in the field of hardware design, and include formal verification and synthesis of combinatorial circuits. Three CLP languages include this constraint system, CAL, CHIP and Prolog III.

5.1.1 Syntax and semantics

Boolean terms are constructed from variables, truth values (*true* and *false* or 1 and 0), Boolean constants, and the Boolean operations (and, or, not, xor, . . .). Simple Boolean terms are Boolean terms without Boolean constants. We use $\wedge$, $\vee$, $\neg$, $\oplus$ to denote the logical operators and, or, not and xor on Boolean terms. It is sometimes convenient to denote Boolean terms together with their variables. In the following, Boolean terms on variables $x_1, \ldots, x_n$ will be denoted by $f(x_1, \ldots, x_n)$, $g(x_1, \ldots, x_n)$, and $h(x_1, \ldots, x_n)$, possibly subscripted.

A Boolean constraint is a constraint of the form $t_1 = t_2$ where t_1 and t_2 are Boolean terms. If $t_1 = t_2$ does not contain Boolean constants, it is called a simple Boolean constraint. Both CAL and CHIP allow for Boolean constraints. Prolog III considers simple Boolean constraints.

The semantics of the constraint system is the usual semantics of Boolean algebra. The only delicate point is the handling of constants. Intuitively, constants can be seen as universally

quantified variables. Hence, if t_1 and t_2 are Boolean terms with variables $x_1, \ldots, x_n$ and constants $a_1, \ldots, a_m$, the equation $t_1 = t_2$ can be interpreted as the formula

$$\forall a_1, \ldots, a_m \exists x_1, \ldots, x_n \, t_1 = t_2$$

Finally, it is useful to note that $\oplus$ can be constructed easily from $\wedge$, $\vee$, and $\neg$ using the equality

$$x \oplus y = x \wedge \neg y \vee \neg x \wedge y.$$

Similarly, $\vee$ and $\neg$ can be reconstructed from $\oplus$ and $\wedge$ using the equalities

$$x \vee y = x \wedge y \oplus x \oplus y$$
$$\neg x = 1 \oplus x.$$

Working with $\oplus$ and $\wedge$ might often simplify computations, and the rest of this presentation is mainly expressed in their terms. We feel free, however, to use $\neg x$ instead of $1 \oplus x$. We also assume the usual precedence rules (e.g., $\oplus$ binds more than $\wedge$).

5.1.2 Example

We illustrate the constraint system by presenting the description of an xor-gate at the ideal switch level (Brzozowski & Yoeli, 1985). An xor-gate can be represented using ideal switches in the following way (Simonis & Dincbas, 1987a):

```
xor(A,B,X) ←
   p_switch(1,A,T1),
   n_switch(0,A,T1),
   p_switch(B,A,X),
   n_switch(B,T1,X),
   p_switch(A,B,X),
   n_switch(T1,B,X).

n_switch(Drain,Gate,Source) ←
   Drain ∧ Gate = Gate ∧ Source ◇.
p_switch(Drain,Gate,Source) ←
   Drain ∧ ¬ Gate = ¬ Gate ∧ Source ◇.
```

The switches enforce the Boolean constraints required by the modelling at this level. A `n_switch` requires that the drain and the source be equal when the gate is "on" (i.e., assigned to the truth value *true*). A `p_switch` requires that the drain and the source be equal when the gate is "off" (i.e., assigned to the truth value *true*). The query ← `xor(a,b,R)` gives

$$\mathtt{R} = \mathtt{a} \oplus \mathtt{b}$$

as an answer, showing that the circuit is correct when modelled at the switch level. The computation of this answer requires the solving of a Boolean equation for each switch. The partial solutions are depicted below with the convention that variables beginning with an underscore are free variables introduced during constraint solving

```
← xor(a,b,X).

1) T1 = 1 ⊕ a ⊕ _A ∧ a
2) T1 = 1 ⊕ a
3) X = b ⊕ _C ∧ a ⊕ a ∧ b
4) X = b ⊕ _C ∧ a ⊕ a ∧ b
5) X = a ⊕ b ⊕ _D ∧ a ∧ b
6) X = a ⊕ b.
```

5.1.3 Constraint solving

Complexity Complexity of constraint solving in this constraint system varies, depending on the presence of Boolean constants. Deciding the satisfiability of a set of simple Boolean equations is $\mathcal{NP}$-complete. Deciding the satisfiability of a set of Boolean equations is Π_2^p-complete (Kanellakis et al., 1990).

Algorithms Various algorithms can be used to decide the satisfiability of Boolean equations. Prolog III uses a variant of SL-resolution, known informally as SL-resolution with production. Little is known about the algorithm itself. CAL uses Groebner basis (Buckberger, 1985) specialized to Boolean equations. CHIP uses a Boolean unification method based on variable elimination (Boole's method) (Buttner & Simonis, 1987). Other methods are possible, and include other Boolean unification algorithms (e.g., Martin & Nipkow, 1986), or model enumeration procedures (e.g., Davis & Puttman, 1960). Moreover, no algorithm is likely to be adequate for all purposes, and it is an active research topic to classify the merit of each of them. In the following, we concentrate on Boole's method and the Boolean unification algorithm of CHIP. This algorithm has turned out to be instrumental in the formal verification of digital circuits. The presentation is based on Radeanu (1974).

Constraint solving with one variable We first consider the simpler problem of solving a Boolean equation with one variable. First note that an equation of the form $t_1 = t_2$ is equivalent to $t_1 \oplus t_2 = 0$. So, without loss of generality, we can restrict ourselves to the solving of equations of the form $t = 0$. The following theorem gives a way to solve equations in one variable.

Theorem 4 *Let $a \wedge x \oplus b = 0$ be an equation where a, b are Boolean terms without variables. The equation is consistent iff $\neg a \wedge b = 0$. Moreover, if it is consistent, then a most general solution is given by the assignment*

$$x \leftarrow b \oplus \neg a \wedge y$$

where y is a new Boolean variable.

It follows that solving a Boolean equation with one variable amounts to determining if $\neg a \wedge b$ can be reduced to 0 given the axioms of Boolean algebra. Since a variety of canonical forms exist for Boolean functions, this can be achieved by reducing $\neg a \wedge b$ to its canonical form and comparing it with the canonical form of 0.

Constraint solving with several variables Constraint solving with several variables is a generalization of the case of one variable. Each variable is eliminated, generating a new equation to solve. Assume that $f(x_1, \ldots, x_n)$ is a Boolean term with variables $x_1, \ldots, x_n$. Equation $f(x_1, \ldots, x_n) = 0$ can be rewritten into

$$f_1(x_1, \ldots, x_n) \stackrel{def}{=} g_1(x_1, \ldots, x_{n-1}) \wedge x_n \oplus h_1(x_1, \ldots, x_{n-1}) = 0$$

This equation is consistent if and only if

$$f_2(x_1, \ldots, x_{n-1}) \stackrel{def}{=} \neg g_1(x_1, \ldots, x_{n-1}) \wedge h_1(x_1, \ldots, x_{n-1}) = 0$$

is consistent, in which case the assignment

$$x_n \leftarrow h_1(x_1, \ldots, x_{n-1}) \oplus \neg g_1(x_1, \ldots, x_{n-1}) \wedge y_n$$

is a most general solution, with y_n being a new variable. Note that the new equation

$$f_2(x_1, \ldots, x_{n-1}) = 0$$

has one less variable. The above process allows us to remove one variable at a time. It is thus possible to apply it until only one variable is left. The general pattern is thus

$$f_p(x_1, \ldots, x_{n-p+1}) \stackrel{def}{=} \neg g_{p-1}(x_1, \ldots, x_{n-p+1}) \wedge h_{p-1}(x_1, \ldots, x_{n-p+1}) = 0,$$

```
procedure BoolUnify(t1,t2)
   return EqualToZero(t1 ⊕ t2);

procedure EqualToZero(t)
begin
   if (t can be reduced to 0) then
      return TRUE;
   else if (t contains a variable x) then
      let t = a∧x⊕b and y be a new variable in
         if (EqualToZero(¬a∧b)) then
           x := b⊕¬a∧y;
           return TRUE;
        else
           return FALSE;
   else
      return FALSE;
end;
```

Figure 7 Boolean unification based on Boole's method

which can be expressed as

$$g_p(x_1, \ldots, x_{n-p}) \wedge x_{n-p+1} \oplus h_p(x_1, \ldots, x_{n-p}) = 0$$

and is consistent iff

$$\neg g_p(x_1, \ldots, x_{n-p}) \wedge h_p(x_1, \ldots, x_{n-p}) = 0$$

is consistent, in which case the assignment

$$x_{n-p+1} \leftarrow h_p(x_1, \ldots, x_{n-p}) \oplus \neg g_p(x_1, \ldots, x_{n-p}) \wedge y_{n-p+1}$$

is a most general solution, with y_{n-p+1} being a new variable. This process of variable elimination ends up with

$$f_n(x_1) \stackrel{def}{=} a \wedge x_1 \oplus b = 0$$

which we have seen how to solve in the previous paragraph. The assignments to $x_1, \ldots, x_n$ make up a most general solution to the original equation.

Boolean unification The Boolean unification algorithm of Buttner & Simonis (1987) is precisely using the above variable elimination technique. The algorithm is shown in Fig. 7.

5.1.4 Applications

CLP languages over Boolean algebra have been applied to various areas, including hardware design and propositional logic. In Simonis et al. (1988), formal verification of circuits at the gate, switch and transistor level is presented. The computation times for these examples are comparable to specialized tools. Other applications in hardware design include synthesis and circuit simplification. In Colmerauer (1990), applications to diagnosis and propositional logic are also presented.

5.1.5 Discussion

Several points deserve to be mentioned on this constraint system. First, it has the important property that solutions (and hence the constraint store) can be represented by substitutions, as it is the case in logic programming. This allows for compact representations of the constraints. Second, as mentioned already, the Boolean unification algorithm is only one way to solve Boolean constraints. The algorithm presented here is clearly appropriate when all solutions (or a most general solution) are required, as is the case in hardware verification, synthesis and specialization. In cases where only one solution is required, it is not clear, however, whether the algorithm is appropriate, or whether enumeration algorithms would perform better.

```
mortgage(P,T,I,R,B) ←
    T = 1,
    B = P*(1 + I / 1200) - R◇.
mortgage(P,T,I,R,B) ←
    T > 1,
    T1 = T - 1,
    P ≥ 0,
    P1 = P*(1 + I / 1200) - R◇
    mortgage(P1,T1,I,R,B).
```

Figure 8 A simple mortgage program

5.2 Linear rational programming

One of the most successful algorithms ever designed is certainly the simplex algorithm of Dantzig et al. (1955). In this section, we study a constraint system that is a generalization of phase I of the simplex algorithm. Constraints are stated over rational numbers, and can be linear equations, inequalities and disequations. This constraint system is included in CHIP, Prolog III and CLP($\Re$).[9] All three systems are based on the simplex algorithm. (See Graf, 1987; Jaffar & Michaylov, 1987; Jaffar et al., 1990; Stuckey, 1990; Van Hentenryck & Graf, 1990, for discussions of this constraint system.)

5.2.1 Syntax and semantics

A rational term is constructed from rational numbers, variables, and the addition and multiplication operations provided that the resulting term be linear.

A rational constraint is an expression of the form $t_1 \, \delta \, t_2$ where t_1 and t_2 are rational terms, and $\delta \in \{>, \geqslant, =, \leqslant, <, \neq\}$.

The generalization over the first phase of simplex comes from the possibility of stating constraints of the form $t_1 \neq t_2$, $t_1 > t_2$ and $t_1 < t_2$.

The semantics of the constraint system is given by considering the rational numbers as an ordered additive group. Multiplication by a constant is just a notation shorthand.

5.2.2 Example

We use a simple mortgage program from Jorgensen & Marriot (1990) to illustrate this constraint system. In the program shown in Figure 8, P is the payment, T is the time in months, I is the interest rate per year, R is the monthly repayment, and B is the balance. The main feature of this program is that it can be queried in various ways, some of them requiring simple calculation, others requiring the simplex algorithm. For instance, the query

```
← mortgage(100000,360,12,1025,B)
```

has answer constraint `B = 12625.9`. The query

```
← mortgage(P,360,12,R,B)
```

has answer constraint `P = 0.0278 * B + 97.22 * R & P ≥ 0`. Finally, the query

```
← 0 ≤ B, B ≤ 1030 ◇ mortgage(100000,360,12,1025,B)
```

has answer constraint `T = 355 & B = 385.449`.

5.2.3 Constraint solving

Complexity Deciding the satisfiability of a set of rational constraints can be done in polynomial time. The main result is due to Khachian (1979) who was first to propose a polynomial time

[9] CLP($\Re$) actually works over the real numbers, but this is not significant for the rest of the presentation.

algorithm for linear programming. The generalization to include constraints of the form $t_1 \neq t_2$, $t_1 > t_2$ and $t_1 < t_2$ is described, for instance, in Lassez & McAloon (1988).

Algorithms There are various algorithms to decide the satisfiability of linear equations and inequalities. On the one hand, there is the simplex algorithm which is exponential in the worst-case but polynomial in the average. On the other hand, there are the polynomial algorithms of Kachian and especially Karmarkar (1984). It is not presently clear how the polynomial algorithm of Karmarkar (1984) can be made incremental, i.e., how it can use the fact that the constraint store is satisfiable. Hence CLP languages are based on generalizations of the simplex algorithm. To present their constraint solvers, we proceed in several steps, starting with equations and then introducing inequalities and disequations.[10]

Equations Gaussian elimination is a standard technique to solve sets of linear equations. This technique can be easily adapted to CLP languages. Whenever the constraint solver faces a linear equation

$$a + a_1x_1 + \cdots + a_nx_n = 0 \quad (n > 0, a_i \neq 0)$$

it isolates one variable, say x_1, and produces the binding

$$x_1 \leftarrow -(a + a_2x_2 + \cdots + a_nx_n)/a_1.$$

Once again, the solver has the nice property that solutions can be represented compactly (e.g., through a substitution).

Inequalities Inequalities of the form $t_1 \geqslant t_2$ and $t_1 \leqslant t_2$ are rewritten into equations by introducing a slack variable $t_1 - s^+ = t_2$ or $t_1 + s^+ = t_2$. The slack variable is constrained to take only nonnegative values, and we refer to variables so constrained as nonnegative variables. Other variables are referred to as arbitrary variables. Gaussian elimination can still be used for equations containing nonnegative variables provided that the equation contains at least one arbitrary variable. The arbitrary variable is then isolated and the solver produces the binding. However, when only nonnegative variables appeared in the constraints, Gaussian elimination is not sufficient. The reason is that the resulting binding might produce subsequently a negative value for the variable. Moreover, simply checking the absence of this problem is not sufficient, as there may be several such constraints and they have to be consistent with each other.

Equations over nonnegative variables The problem to be solved now is to decide the satisfiability of a set of equations over nonnegative variables (in an incremental way).[11] This is the purpose of Phase I of linear programming.

The basic idea behind the simplex algorithm is to maintain a solved form for the constraints. A set of equations is in solved form iff it is of the form

$$\begin{aligned} y_1 &= b_1 \times a_{11}x_1 + \cdots + a_{1m}x_m \\ &\cdots \\ y_n &= b_n + a_{n1}x_1 + \cdots + a_{nm}x_m, \end{aligned}$$

where $y_1, \ldots, y_n$ are called the basic variables, $x_1, \ldots, x_m$ the non-basic variables, and $b_1, \ldots, b_n$ are nonnegative.

Phase I of the simplex algorithm introduces artificial variables to obtain an initial solved form for the constraints, and uses the pivoting operation to obtain a basis minimizing the value of the artificial variables. The constraints are satisfiable iff the result of the minimization is zero.

In CLP languages, the constraint store (in this case equations over nonnegative variables) is maintained in solved form. The main operation is thus to add a new constraint to a set of equations in solved form. This operation may be carried out in three steps:

[10] The presentation that follows should not be understood as proposing a particular implementation. We discuss the basic principles and there are various ways of putting them into practice.

[11] In the following two paragraphs, all variables are assumed to take only nonnegative variables.

1 The basic variables in the new constraint are replaced by their right members;
2 An artificial variable is introduced to provide an initial basis;
3 The value of the artificial variable is minimized.

Once again, the constraints are satisfied iff the minimization result is zero. Also, it is not difficult to extract a solved form for the initial constraints from the solved form of the minimization problem. It follows that the simplex algorithm can be turned into an incremental constraint-solving algorithm for inequalities. We now reconsider disequations and strict equalities.

Disequations and strict inequalities The main problem here is to handle disequations and to find a suitable solved form for them. Strict inequalities (e.g., $t_1 > t_2$) can be rewritten as a conjunction of an inequality and a disequation (e.g., $t_1 \geqslant t_2$ and $t_1 \neq t_2$). (See Imbert & Van Hentenryck, 1991 for an efficient handling of disequations in this computation domain.) It is natural to consider that a disequation is in solved form if it is of the form

$$0 \neq b + a_1x_1 + \cdots + a_nx_n$$

with at least one of $\{b, a_1, \ldots, a_n\}$ different from zero and $x_1, \ldots, x_n$ being non-basic variables.

However, a system of equations and disequations can be put in solved form, even if it is not satisfiable as illustrated by the following system:

$$y_1 = x_1 - x_2$$
$$y_2 = x_2 - x_1$$
$$0 \neq x_1 - x_2$$

Adding the two equations results in $y_1 + y_2 = 0$ and thus in $x_1 = x_2$.

It turns out that the solved form is only valid if the system of equations has no hidden constant, i.e., no variable constrained to take a unique value. The three systems mentioned previously, CHIP, CLP($\mathfrak{R}$) and Prolog III, have different ways of ensuring the absence of hidden constants.

In CLP($\mathfrak{R}$) and Prolog III, the idea is to check at the same time satisfiability and the presence of hidden constants by modifying the algorithm to add a new constraint. They exploit the property that, if the system of equations contains hidden constants, then one of them is the slack variable of the new constraint. Hence, instead of checking only satisfiability, they check if the slack variable is constrained to be zero or can take another value. If the constraints are satisfiable and there is an hidden constant, it is necessary to search for other hidden constants. A precise description of the CLP($\mathfrak{R}$) solution can be found in Stuckey (1990).

CHIP has a very different approach, described in Van Hentenryck & Graf (1990). The motivation underlying the CHIP approach was the desire to obtain a syntactic solved form for the equations which precludes the presence of hidden constants.[12] It turns out that if the constraints are lexicographically positive (e.g., the first non-zero coefficient in the right member is positive) then there is no hidden constant. Moreover, this new solved form can be maintained through pivoting, provided that the underlying simplex algorithm used a lexicographic pivoting rule as that of Dantzig to prevent cycling (Dantzig et al., 1955). Hence, the addition of a new constraint can proceed normally as the pivoting operation maintains the solved form. When some hidden constants are discovered, only those constraints not in solved form need to be reconsidered.

Which of the above solutions is most efficient is still unclear. The potential benefit of the CHIP solution, beside its simplicity, lies in the information it provides for finding other hidden constants once some have been found. Only the constraints not in solved form need to be reconsidered. CLP($\mathfrak{R}$) and Prolog III need to reconsider all homogeneous constraints. CHIP entails a slightly more costly pivoting rule, while CLP($\mathfrak{R}$) and Prolog III have more complicated algorithms to add a constraint.

[12] By syntactic we mean that the solved form can be defined, for instance, by a grammar.

5.2.4 Applications

There have been various applications of this constraint system. They range from simulation and diagnostics of various circuits and devices (Graf et al., 1989, 1990; Heintze et al., 1987; Gorlick et al., 1990) to decision-support systems (Berthier, 1983; Lassez et al., 1987) and geometrical problems (Colmerauer, 1990).

5.2.5 Discussion

The constraint solver for the above constraint system is complete and, although the simplex algorithm is exponential in the worst case, it is reasonable to conclude that it is efficient. Note, however, that the constraint store can no longer be represented as substitutions, but the system is required to maintain sets of constraints in solved form. Fourier's algorithm could also be used to remove intermediary variables from the answer constraint. In theory, this process is inherently exponential. Whether this can be done efficiently in practical applications is still an open issue. Recent work in that area has focused on the elimination of redundant constraints generated by Fourier's algorithm or present in the initial answer constraint (Lassez et al., 1987; Imbert, 1990). Also, it is worth mentioning that many programs will contain constraints that are not linear. In general, the query will be such that the constraints are actually linear at runtime. If it turns out not to be the case, the system will either raise an error message or delay the constraints in the hope that more information will be made available. Finally, note that some of the languages also allow for the optimization of a linear function. The implementation is based on the simplex algorithm once again.

5.3 Finite domains

Many combinatorial problems such as graph colouring, scheduling, and warehouse location are discrete in nature: their variables can only take a finite set of values. A common technique to solve these problems is partial enumeration (Parker & Rardin, 1988), i.e., a combination of tree searching and constraint solving. The constraint system presented in this section aims at supporting the above paradigm. Contrary to the previous two constraint systems, its constraint solver is not complete to reflect a traditional way of solving these problems. This constraint system is included in CHIP, and a comprehensive overview of this system is given in Van Hentenryck (1989b). The presentation that follows is mostly based on Van Hentenryck & Deville (1990). Note that the initial example of this paper was based on this constraint system.

5.3.1 Syntax

Finite terms are constructed from variables, natural numbers, and the addition and multiplication operations. Constraints on finite domains are of the form

$$x \in \{a_1, \ldots, a_n\} \quad \text{or} \quad t_1 \, \delta \, t_2,$$

where t_1, t_2 are finite terms and $\delta \in \{>, \geqslant, =, \leqslant, <, \neq\}$.

Note that the constraints need not be linear. However, we impose the restriction that any variable appearing in an arithmetic constraint also appears in a domain constraint.

The above presentation is a subset of the CHIP constraints. CHIP allows a number of symbolic constraints as well as user-defined constraints. These extensions are subsumed by the cardinality operator to be presented later in this paper. CHIP also allows for finite domains of constants with equations and disequations. There is no difficulty in extending the discussion here to cover this case as well.

The semantics of the constraint system is given by considering the natural numbers as an ordered ring. The semantics of the domain constraint is given as a restricted form of disjunction

$$x = a_1 \vee \ldots \vee x = a_n.$$

Similar semantics (as logical formulas over natural numbers) can be given for any symbolic constraint available in CHIP.

5.3.2 Constraint solving

Complexity Deciding the satisfiability of constraints in this constraint system is $\mathcal{NP}$-complete. It is decidable since we assume that each variable has a finite domain. Note that the problem remains $\mathcal{NP}$-complete if the domain requirement is relaxed and only linear constraints are considered. Note, however, that the constraint-solving algorithms to be presented are not complete, but provide efficient pruning techniques that reduce the search space. As a result, the constraint-solving algorithms are efficient polynomial algorithms.

Basic constraints Constraints in this constraint system can be divided into two subsets: basic constraints and non-basic constraints. Basic constraints are those that can be decided upon by the constraint solver. They fit easily in the CLP framework. Non-basic constraints are only approximated in terms of the basic constraints, i.e., they are used to generate new basic constraints. We first study basic constraints.

Definition 5 The *basic* constraints are either *domain* constraints or *arithmetic* constraints:

- domain constraint: $x \in \{a_1, \ldots, a_n\}$;
- arithmetic constraints:
  - $ax \neq b$;
  - $ax = b$;
  - $ax = by + c\ (a \neq 0 \neq b)$;
  - $ax \geqslant by + c$;
  - $ax \leqslant by + c$.

Note that the variables appearing in arithmetic constraints are expected to appear in some domain constraints. Every variable thus has a domain. This justifies the following definition:

Definition 6 A *system of constraints* S is a pair $\langle AC, DC\rangle$ where AC is a set of arithmetic constraints and DC is a set of domain constraints such that any variable occurring in an arithmetic constraint also occurs in some domain constraint of S.

Definition 7 Let $S = \langle AC, DC\rangle$ be a system of constraints. The set D_x is the *domain* of x in S (or in DC) iff the domain constraints of x in DC are $x \in D_1, \ldots, x \in D_k$ and D_x is the intersection of the D_is $(1 \leqslant i \leqslant k)$.

It follows that, provided that each variable has a domain, a conjunction of basic constraints can be represented by a system of constraints, and vice versa. In the following, if $c(x)$ and $c(x, y)$ are constraints, we denote by $c(x/a)$ and $c(x/a, y/b)$ the Boolean value obtained from $c(x)$ and $c(x, y)$ by replacing x and y by the values a and b, respectively.

Constraint solving for finite domains constraints is based on consistency techniques, a paradigm emerging from artificial intelligence (Mackworth, 1977). The basic idea behind consistency techniques is to reduce the domain of variables by removing values that cannot appear in a solution. Various notions of consistency and algorithms to enforce them have been designed. We review the main definitions.

Definition 8 Let $c(x)$ be a unary constraint and D_x be the domain of x. Constraint $c(x)$ is said to be *node constant wrt* D_x if $c(x/a)$ holds for each value $a \in D_x$.

Definition 9 Let $c(x, y)$ be a binary constraint and D_x, D_y be the domains of x, y. Constraint $c(x, y)$ are said to be *arc-consistent wrt* D_x, D_y if the following conditions hold:

1 $\forall a \in D_x\ \exists b \in D_y\ c(x/a, y/b)$ holds;
2 $\forall b \in D_y\ \exists a \in D_x\ c(x/a, y/b)$ holds.

We are in position to define a solved form for the constraints.

Definition 10 Let S be a system of constraints. S is in *solved form* iff any unary constraint $c(x)$ in S is node-consistent wrt the domain of x in S, and any binary constraint $c(x, y)$ in S is arc-consistent wrt the domains of x, y in S.

The satisfiability of a system of constraints in solved form can be tested in a straightforward way (Van Hentenryck & Deville, 1990).

Theorem 11 *Let $S = \langle AC, DC\rangle$ be a system of constraints in solved form. S is satisfiable iff $\langle \emptyset, DC\rangle$ is satisfiable.*

It remains to show how to transform a system of constraints into an equivalent one in solved form. This is precisely the purpose of the node- and arc-consistency algorithms (Mackworth, 1977).

Algorithm 12 To transform the system of constraints S into a system in solving form S':

1 apply a node-consistency algorithm to the unary constraints of $S = \langle AC, DC\rangle$ to obtain $\langle AC, DC'\rangle$;
2 apply an arc-consistency algorithm to the binary constraints of $\langle AC, DC'\rangle$ to obtain $S' = \langle AC, DC''\rangle$.

We now give a complete constraint solver for the basic constraints. Given a system of constraints S, Algorithm 13 returns *true* if S is satisfiable, and *false* otherwise.

Algorithm 13 To check the satisfiability of a system of constraints S:

1 apply Algorithm 12 to S to obtain $S' = \langle AC, DC\rangle$;
2 if the domain of some variable is empty in DC, return *false*; otherwise return *true*.

The complexity of Algorithms 12 and 13 is the complexity of arc-consistency algorithms. In Mohr & Henderson (1986) an arc-consistency algorithm is proposed whose complexity is $O(cd^2)$, where c is the number of binary constraints, and d is the size of the largest domain. Moreover, in Deville & Van Hentenryck (1991) it is shown that, provided that the constraints satisfy some well-defined properties (functionality or monotonicity), an $O(cd)$ algorithm can be obtained. These properties being satisfied by the basic constraints, it is possible to implement Algorithm 13 to run in $O(cd)$.

Non-basic constraints Basic constraints are not expressive enough to state a large variety of discrete combinatorial problems. Moreover, simple generalizations of these constraints (e.g., allowing for two variables in disequations or three variables in inequalities) lead to $\mathcal{NP}$-complete problems to decide satisfiability. Hence the choice made in this constraint system was to allow for non-basic constraints but to provide an incomplete constraint solver.

Consistency techniques are also used to solve non-basic constraints. The idea here is that each individual non-basic constraint is tested for satisfiability wrt to the domains of the solved form of basic constraints. Moreover, they are used to generate new basic constraints (e.g., domain constraints) that can be added to the current set of basic constraints. However, since the constraint solver is not complete, the above operational semantics does not apply directly. Intuitively, the test for satisfiability in the definition of the operational semantics should be replaced by something weaker, say local satisfiability. However, the results of the computation should be interpreted with care. The reader is referred to Van Hentenryck & Deville (1990) for a precise definition of the operational semantics. The discussion that follows is informal, and is intended to provide the reader with a preliminary understanding of the operational semantics.

Non-basic constraints are approximated using a generalization of arc-consistency for non-binary variables.

Definition 14 Let $c(x_1, \ldots, x_n)$ be a constraint and D_i be the domain of x_i $(1 \leq i \leq n)$. Constraint $c(x_1, \ldots, x_n)$ is said to be *arc-consistent wrt* D_i $(1 \leq i \leq n)$ if the following condition holds for all i

- $\forall b_i \in D_i \exists b_1, \ldots, b_{i-1}, b_{i+1}, \ldots, b_n \in D_1, \ldots, D_{i-1}, D_{i+1}, \ldots, D_n$ such that $c(b_1, \ldots, b_n)$ holds.

A reduced form for the constraints is obtained when all constraints are arc-consistent. Hence the test for satisfiability given previously can be replaced by a test for arc-consistency of the constraints and each step amounts to enforcing an arc-consistency algorithm, reducing the domains of the variables. Arc-consistency algorithms can be rather inefficient in practice, but the semantics of the constraints can be exploited to produce good specialized algorithms. We illustrate this point by presenting how reasoning about variation intervals can lead to an efficient algorithm for linear inequalities. Suppose that a linear inequality has been normalized into an expression of the following form:

$$a_1x_1 + \cdots + a_nx_n + a \geq b_1y_1 + \cdots + b_my_m + b.$$

Assume that

$$a_1x_1 + \cdots + a_nx_n + a$$

ranges over $[min_1, max_1]$, and that

$$b_1y_1 + \cdots + b_my_m + b$$

ranges over $[min_2, max_2]$. To satisfy the constraint, min_2 should be smaller or equal to max_1. Hence new constraints can be derived:

$$a_1x_1 + \cdots + a_nx_n + a \geq min_2$$
$$b_1y_1 + \cdots + b_my_m + b \leq max_1$$

But these constraints directly imply basic constraints on the values of variables. Indeed, each variable x_i should satisfy

$$a_ix_i \geq min_2 - \left(\sum_{k=1,k\neq i}^{n} (a_k max(x_k)) + a \right),$$

and each variable y_i should satisfy

$$b_iy_i \leq max_1 - \left(\sum_{k=1,k\neq i}^{m} (b_k min(x_k)) + b \right),$$

where $min(x)$ and $max(x)$ are respectively the minimum and maximum values in the domain of x. These new constraints can then be used to reduce the domains of the variables.

The above behaviour was used in the initial example of the paper. The interested reader could reconsider the example with the knowledge of the constraint solver. Similar handling of equations and inequalities has been used in Alice (Lauriere, 1978) and REF-ARF (Fikes, 1968).

5.3.3 Applications

This constraint system has given rise to numerous applications including graph colouring, scheduling, cutting stock, microcode labelling, warehouse location, car sequencing, planning and assignment problems to name a few (see, for instance, Dincbas et al., 1988a,b,c, 1990; Van Hentenryck, 1989a,b). For most of these applications, the computation time is comparable in efficiency with special-purpose programs (based on the same approach).

5.3.4 Discussion

Finite domains are a constraint system endowed with an incomplete constraint solver. The design choice was motivated by the desire to support, in an adequate manner, partial enumeration, a paradigm subsuming backtracking, constraint satisfaction and branch and bound. In partial

enumeration, incomplete constraint-solving and enumeration are interleaved, and this is precisely what is achieved by combining the above constraint system and nondeterminism. The language then abstracts both the constraint solving and enumeration components and shortens the development time of programs substantially while preserving the efficiency of specialized tools for many applications. It can be viewed as a search procedure with an arc-consistency algorithm executed at each node of the tree, i.e., the constraint store is always arc-consistent.

An alternative design would have been to provide a complete constraint solver that would necessarily require exponential time. However, given the variety of discrete combinatorial problems, it is unlikely that this approach be practical. By leaving to the programmer the responsibility of the choice process, heuristics and properties such as symmetries can be exploited and the algorithms specialized to the problem at hand. We refer the reader to Van Hentenryck (1989b) for more information on this topic.

Finally, note that some higher-order predicates to find an optimal solution given an objective function are available for this computation domain. The implementation uses a depth-first branch and bound technique (Van Hentenryck, 1989b).

6 Applications

In the previous sections, we have presented a general framework for CLP languages and discussed various constraint systems that have been used to instantiate the framework. Small examples have been presented and applications have been mentioned. In the following, we turn to larger applications in order to illustrate the potential of CLP languages. We present an application for each of the constraint systems presented previously. The applications are not always described in detail, but should convey the nature of the problems and solutions. Hence they give the reader an idea of the applicability of CLP languages. More information can be found in the references.

6.1 Formal verification of digital circuits

The purpose of this application is to demonstrate the use of Boolean algebra for the formal verification of circuits. Formal verification of circuits is an important area as exhaustive testing is impossible for large circuits. A formal verification amounts to comparing an implementation (i.e., a circuit description) with a circuit specification. Formal verification can be done at various levels of abstraction such as the gate level, the switch level, or the transistor level. (See Simonis et al., 1988, for an overview of formal verification of CLP languages.) In this paper, we consider only verification at the gate level (an example at the switch level was presented earlier in the paper), and we mainly follow the above-mentioned paper.

6.1.1 Circuit representation

Representing hardware in logic programming has been studied in various papers (e.g., Clocksin, 1987). Logic programming provides a simple, yet convenient, hardware description language supporting, for instance, hierarchical descriptions and buses. Logical gates can be described easily in terms of constraints:

```
and(Id,X,Y,Z) ← Z = X ∧ Y ◇.
or(Id,X,Y,Z) ← Z = X ∨ Y ◇.
xor(Id,X,Y,Z) ← Z = X ⊕ Y ◇.
not(Id,X,Y) ← Y = ¬X ◇.
```

The first argument in the gates always represents the unique identifier assigned to each gate of the circuit. Given the above description, a full adder, depicted in Figure 9, can be easily described as follows:

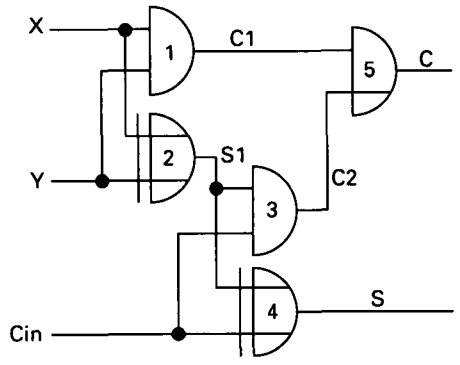

Figure 9 A full adder

```
fa(N,X,Y,Cin,S,C) ←
   and([1|N],X,Y,C1),
   xor([2|N],X,Y,S1),
   and([3|N],Cin,S1,C2),
   xor([4|N],Cin,S1,S),
   or([5|N],C1,C2,C).
```

In the full adder, X, Y, Cin, S, C are intended to represent single bits. Denoting by b_T the bit value of a Boolean variable T, the full adder is intended to perform the addition of b_X, b_Y, b_{Cin} to produce the result b_S and the carry b_C, and our modelling is intended to impose the constraint

$$b_X + b_Y + b_{Cin} = 2 * b_C + b_S$$

There are various points to mention here. First, wires are represented by shared logical variables. This is in no way necessary (wires can be represented explicitly if necessary) but provides a compact representation. Second, the description associates to each component a unique identifier by concatenating the unique identifier of the full adder with different integers. Now the query `fa([],x,y,z,S,C)` returns as an answer constraint:

$$S = x \oplus y \oplus z$$
$$C = x \wedge y \oplus x \wedge z \oplus y \wedge z.$$

In other words, S receives the value *true* (i.e., represents the bit 1) iff there is an odd number of bit 1 in x, y, z and C receives the value *true* (i.e., represents the bit 1) if there are at least two variables assigned to *true* (i.e., the result of the addition is 2 or 3). The intermediary bindings for the variables are extremely simple in this example:

1) $C1 = x \wedge y$
2) $S1 = x \oplus y$
3) $C2 = x \wedge z \oplus y \wedge z$
4) $S = x \oplus y \oplus z$
5) $C = x \wedge y \oplus x \wedge z \oplus y \wedge z.$

Suppose now that we would like to build a four-bit adder, as depicted in Figure 10. A four-bit adder adds two four-bit words and a carry-in to obtain a four-bit word and a carry-out:

```
              Cin
        X3 X2 X1 X0
        Y3 Y2 Y1 Y0
---------------
Cout S3 S2 S1 S0
```

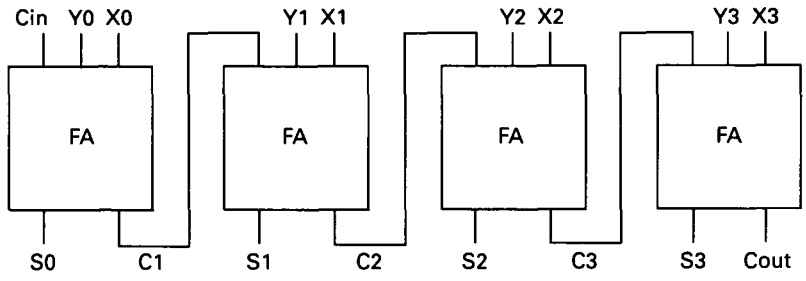

Figure 10 A four-bit adder

Four full-adders can be combined to obtain the desired circuit:

```
four_bit(N,Cin,X0,X1,X2,X3,Y0,Y1,Y2,Y3,S0,S1,S2,S3,Cout) ←
    fa([1|N],X0,Y0,Cin,S0,C1),
    fa([2|N],X1,Y1,C1,S0,C2),
    fa([3|N],X2,Y2,C2,S0,C3),
    fa([4|N],X3,Y3,C3,S0,Cout),
```

6.1.2 Formal verification

Formal verification, as mentioned, amounts to verifying whether a circuit is equivalent to its specification. A specification is usually described as a set of recursive equations. Assume, for instance, that the four-bit adder has to be verified formally. The following recursive specification can be used:

```
sp_n_bit_adder(C,[],[],[],C).
sp_n_bit_adder(Cin,[X|Xs],[Y|Ys],[S|Ss],Cout) ←
    S = X ⊕ Y ⊕ Cin,
    Cn = (X ∧ Y) ⊕ (X ∧ Cin) ⊕ (Y ∧ Cin) ◇
    sp_n_bit_adder(Cn,Xs,Ys,Ss,Cout).
```

In the above specification, the first argument represents the carry-in, the second argument the list of bits that are the X inputs, the third argument the list of bits that are the Y inputs, the fourth argument the list of bits that are the outputs, and the last argument is the carry-out.

Now comparing the specification and the circuit can be done through the following clause:

```
verify ←
    sp_n_bit_adder(cin,[x0,x1,x2,x3],[y0,y1,y2,y3],S,Cout),
    four_bit([],cin,x0,x1,x2,x3,y0,y1,y2,y3,S,Cout).
```

If the circuit is correct wrt the specification, the above clause will succeed. Otherwise, it will fail. Note that it is necessary to use Boolean constants. If variables were used instead, we would only be able to conclude that the circuit is compatible wrt the specification, i.e., the circuits can be specialized (i.e., instantiated) to achieve a similar behaviour.

6.1.3 Computation results

The above presentation should show the potential of CLP over Boolean algebra for formal verification of digital circuits. The approach has been applied to a number of circuits. For instance, the 74LS181 ALU given in Bryant (1986) has been verified for various wordsizes (from 8 to 64) with computation times varying from 10 to 280 seconds on BULL SPS-9 (about twice as fast as a VAX 11/780). These results are comparable with specialized tools for that problem. For large circuits described in an hierarchical way, the efficiency might be improved by verifying the subcomponents first and then using their specifications instead of their implementations. The approach also has limitations: multipliers cannot be verified efficiently but other techniques can be used.

Classes	1	2	3	4	5	6	*Capacity*
Option 1	y	–	–	–	y	y	1/2
Option 2	–	–	y	y	–	y	2/3
Option 3	y	–	–	–	y	–	1/3
Option 4	y	y	–	y	–	–	2/5
Option 5	–	–	y	–	–	–	1/5
Cars	1	1	2	2	2	2	

Figure 11 A car sequencing example

	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9	S_{10}
Class 1										
Class 2										
Class 3										
Class 4										
Class 5										
Class 6										

Figure 12 Car sequencing: the initial search space

6.2 The car sequencing problem

We now present a problem using finite domains. The problem was motivated by an article published in AI Expert (Parrello, 1988) reporting the failure of an expert system to solve the problem and concluding that fifth-generation tools were not appropriate to solve the problem. A CLP solution to the problem was reported first in Dincbas et al. (1988c). The presentation here essentially follows that paper, although it has been revised to accommodate refinements to the CLP framework.

6.2.1 Problem statement

The problem arises in the car industry where it is necessary to produce cars requiring different options. The cars are placed on an assembly line which moves through the various production units responsible to set up the options. The production units have limited capacity, implying that they may not be able to set the option on each car. More generally, their capacity constraints are of the form *r outof s* meaning that, on each sequence of *s* cars, the unit is able to produce at most *r* cars with the option. The problem can now be specified as follows: given a number of cars with their association options, find a sequencing of the cars satisfying the capacity constraints of the production units.

We illustrate the problem on a simple example. Since cars requiring the same set of options cannot be distinguished here for any useful purpose, we cluster them in classes. Figure 11 presents a problem with 5 options, 6 classes, and 10 cars. A "y" in the table means that a particular option is required by the class while a "_" means that the option is not requested. The capacity constraint *r/s* should be read as *r outof s*.

The search space in this problem is made up of the possible values for the slots of the assembly line, as depicted in Figure 12. We take the same convention as in the "send + more = money" example to describe the search space. The assembly line itself is best described by the options selected for each slot, as depicted in Figure 13. Figures 14 and 15 describe a solution to the problem.

	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9	S_{10}
Option 1										
Option 2										
Option 3										
Option 4										
Option 5										

Figure 13 Car sequencing: the initial assembly line

	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9	S_{10}
Class 1	+	-	-	-	-	-	-	-	-	-
Class 2	-	+	-	-	-	-	-	-	-	-
Class 3	-	-	-	+	-	-	-	-	+	-
Class 4	-	-	-	-	-	+	+	-	-	-
Class 5	-	-	-	-	+	-	-	+	-	-
Class 6	-	-	+	-	-	-	-	-	-	+

Figure 14 Car sequencing: a solution

	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9	S_{10}
Option 1	+	-	+	-	+	-	-	+	-	+
Option 2	-	-	+	+	-	+	+	-	+	+
Option 3	+	-	-	-	+	-	-	+	-	-
Option 4	+	+	-	-	-	+	+	-	-	-
Option 5	-	-	-	+	-	-	-	-	+	-

Figure 15 Car sequencing: the assembly line in a solution

6.2.2 Problem solution

The problem is clearly a discrete combinatorial problem and its solution uses constraints over finite domains. As is typical with finite domains programs, the program contains two parts: a constraint part that generates the problem constraints and a choice part that assigns values to (some of) the problem variables. The presentation that follows concentrates mainly on the constraints that have to be generated.

Conventions We assume that we are given n classes of cars. Each class i constrains n_i cars ($n_i \geq 0$) such that the total numbers of cars is $nc = \Sigma_{i=1}^{n} n_i$. We also assume m different options. For each class i and option j, we have a Boolean o_{ij} which is true if class i requires option j and false otherwise. For convenience, we will assume that *true* is represented by 1 and *false* by 0.

Problem variables The first step towards the solution is to identify the problem variables in terms of which the constraints are stated. To each slot i ($1 \leq i \leq nc$) is associated a variable S_i denoting the class of cars assigned to the slot. These variables are called the slot variables, and represent the main output of the program. Each slot i is also associated with m variables, one for each option denoted $O_i^1, O_i^2, \ldots, O_i^m$. $O_i^j (1 \leq i \leq nc$ and $1 \leq j \leq m)$ is equal to 1 if the class S_i (i.e., the class assigned to slot i) requires option j and 0 otherwise. These variables are called the option variables. There are $O(nc)$ slot variables and $O(nc \times m)$ option variables. In the above example, there are 10 slot variables and 50 option variables.

Domain constraints We now turn to the problem constraints. The first constraints are the domain constraints for the slot and option variables. Each slot variable S_i has a constraint

$S_i \in \{1, \ldots, n\}$ while each option variable O_i^j has a constraint $O_i^j \in \{0, 1\}$. The domain constraints generated for the example are as follows:

$$S_1 \in 1..6, \ldots, S_{10} \in 1..6,$$
$$O_1^1 \in 0..1, \ldots, O_1^5 \in 0..1,$$
$$\ldots$$
$$O_{10}^1 \in 0..1, \ldots, O_{10}^5 \in 0..1.$$

Capacity constraints The capacity constraints are stated in terms of the slot variables. If the capacity constraint for option j ($1 \leq j \leq m$) is of the form r *outof* s, constraints of the form

$$O_i^j + \cdots + O_{i+s-1}^j \leq r \; (1 \leq i \leq nc - s + 1)$$

have to be generated. In the above example, the capacity constraints are as follows:

$$O_1^1 + O_2^1 \leq 1,$$
$$\ldots$$
$$O_9^1 + O_{10}^1 \leq 1,$$
$$\ldots$$
$$O_1^5 + O_2^5 + \cdots + O_5^5 \leq 1,$$
$$\ldots$$
$$O_6^5 + O_7^5 + \cdots + O_{10}^5 \leq 1.$$

There are $O(nc \times m)$ capacity constraints.

Demand constraints It is also necessary to make sure that the cars actually requested are produced. For each class i ($1 \leq i \leq n$), a constraint

$$exactly(n_i, [S_1, \ldots, S_{nc}], i)$$

has to be generated, where $S_1, \ldots, S_{nc}$ are the slot variables and n_i is the number of cars in class i. Since they are nc slot variables, and each of them will be assigned to a class (and thus a car), it is only necessary to make sure that the assignment does not produce more cars from a class than is actually necessary. Hence the above constraints reduce to `atmost` constraints seen previously,

$$atmost(n_i, [S_1, \ldots, S_{nc}], i).$$

There are n demand constraints. In the example, the following constraints are generated:

```
atmost(1,[S1,...,S10],1),
...
atmost(2,[S1,...,S10],6).
```

Relation constraints So far, the option variables are not connected to the slot variables as required by the semantics specified previously (see Paragraph Problem Variables). The connection can be achieved through the `element` constraint that was introduced previously in the paper, and built using the cardinality operator. Each option j will be connected with slot i by the constraint

$$element(S_i, [o_{1j}, \ldots, o_{nj}], O_i^j).$$

where $o_{1j}, \ldots, o_{nj}$ are the zero-one values specifying which classes require option j. In the example, the connection between the slots and options is enforced by the constraints.

```
element(S1,[1,0,0,0,1,1],O1^1),
...
element(S1,[0,0,1,0,0,0],O1^5),
...
element(S10,[1,0,0,0,1,1],O10^1),
...
element(S10,[0,0,1,0,0,0],O10^5).
```

There are $O(nc \times m)$ relation constraints.

	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9	S_{10}
Class 1	+	–	–	–	–	–	–	–	–	–
Class 2	–									
Class 3	–									
Class 4	–									
Class 5	–	–	–							
Class 6	–	–								

Figure 16 Car sequencing: search space after 1 choice

	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9	S_{10}
Option 1	+	–								
Option 2	–									
Option 3	+	–	–							
Option 4	+									
Option 5										

Figure 17 Car sequencing: the assembly line after 1 choice

Basic program The basic program can now be presented. It amounts to generating the constraints (a process that has been illustrated in the "send + more = money" puzzle) and to generating values to the slots variables

```
sequencing(Line) ←
   generate_constraints(Line),
   generate_values(Line).
```

The argument of the predicates is the list of slots variables. The generation of constraints will be responsible for creating as many variables as there are slots in the assembly line, for creating the option variables, and for stating all the abovementioned constraints. Generating the constraints can be done by simple recursive programs and does not raise any particular difficulty. Assigning a value to the slot variables will produce a solution satisfying the constraints. The generation of values simply assigns to each of the slot variables a value between 1 and n, that is a class of cars.

Redundant constraints The program efficiency can be improved by generating redundant constraints. These constraints are not necessary to guarantee the correctness of the program, but improve the pruning by exploiting properties of the solutions. Generating redundant constraints is a common practice in operations research. In the car sequencing problem, the idea is the following. Assume that option j has a capacity constraint r *outof* s. We know that the last s slots can only contain r cars so that the other slots should contain all the remaining cars having that option. If p cars require option j, we may generate a constraint

$$O_1^j + \cdots + O_{nc-s}^j \geqslant p - r.$$

More generally, the last $k \times s$ ($k = 1, 2, \ldots, nc/s$) can only contain $k \times r$ cars, and hence the

$$O_1^j + \cdots O_{nc-k\times s}^j \geqslant p - k \times r$$

may be generated.

Illustrating the computation The above program will first state the constraints and then make choices. After the first choice (i.e., $S_1 = 1$), the search space and the assembly line are depicted in Figures 16 and 17.

	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9	S_{10}
Option 1	+	-								
Option 2	-	-								
Option 3	+	-	-							
Option 4	+	+	-	-	-					
Option 5		-								

Figure 18 Car sequencing: the assembly line at an intermediary state

Now, giving to S_2 its first possible value (i.e., 2) will lead directly to the solution presented at the beginning of the example. Indeed, this choice removes immediately the value 2 for all other slot variables and prevents variables S_3, S_4, S_5 from taking the value 4 because of option 4. This intermediary state is depicted in Figure 18. But now S_3 and S_4 can only take a class taking option 2 because of the redundant constraints. This fixes all the slots requiring option 2 and the position of the 2 cars of class 5. Pursuing the reasoning leads to the solution with two choices (i.e., S_1, S_2) and no backtracking.

6.2.3 Computation results

The resulting program for the car-sequencing is less than three pages long. It was developed in a rather short time (less than a week). To evaluate its efficiency, a number of experiments have been carried out with the program. They are fully reported in Dincbas et al. (1988c). The main result is that the program can sequence problems involving several hundred cars with a resource utilization over 90% in a couple of minutes on a SUN-3 workstation. It was observed that for these randomly generated problems, the average time of the program is quadratic in the size of the assembly line.

Note also that an integer programming solution would require a much larger set of variables and constraints due to the inability to state the relation and demand constraints in the above forms. It would require recasting the problem in terms of zero-one variables and hence losing much efficiency.

6.3 Modelling and simulation of electrical and hybrid circuits

The purpose of this section is to illustrate the use of linear rational (or real) arithmetic in the simulation of hybrid circuits (i.e., circuits containing electro-mechanical, hydraulic, and other types of components) such as machine tools and the landing gear of an aircraft. A complete presentation of the application is beyond the scope of this paper and the reader is referred to Graf et al. (1990) for more comprehensive presentation. In the following, we try to convey some of the ideas and techniques used in this application.

6.3.1 Problem description

The problem amounts to simulating hybrid circuits at a high level of abstraction, yet with a sufficient precision. The idea is to achieve a compromise between a qualitative approach (too imprecise for this application) and a quantitative approach (that provides too low-level information). The simulation should provide an early design check and detect anomalies such as blowing of components and oscillations. It should also provide the basis for a diagnostics tool able to detect design errors. The simulation might be nondeterministic (i.e., there may be several states succeeding a particular state), and hence the problem is combinatorial. The nondeterminism comes from various factors: for instance, the circuit may be ambiguous and hence its behaviour is ill-defined or some component may have been removed for the purpose of diagnostics and hence the circuit behaviour is underspecified. Moreover, because the circuit is described at a high level, it is difficult to identify the successor of a given state.

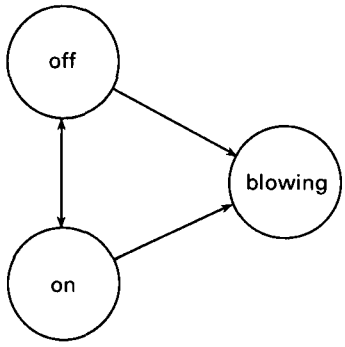

Figure 19 Transition graph for a light bulb

```
State = off iff
   U1 − U2 = Res × I1 &
   I1 + I2 = 0 &
   |I1| < Ithreshold.
State = on iff
   U1 − U2 = Res × I1 &
   I1 + I2 = 0 &
   |I1| ≥ Ithreshold &
   |I1| < Imax.
State = blowing iff
   U1 − U2 = Res × I1 &
   I1 + I2 = 0 &
   |I1| ≥ Imax.
```

Figure 20 State definitions for a light bulb

6.3.2 Device models

Any device model consists of

- *a static part* defining the possible states of the device.
- *a dynamic part* defining the temporal aspects of the device behaviour.

The behaviour of the device is described in terms of a finite set of states, each of which is defined using constraints. The constraints defining the states can be classified into constraints defining

- physical laws applying to the state (e.g., Ohm's laws);
- conditions on the device physical values proper to the state (e.g., the current in one port is higher than a threshold value).

The dynamic part defines *temporal aspects* of the device behaviour by defining

- the possible *transitions* between the device states;
- the *events* implied by the transitions; an event is a demand to fix up, or to restrict the possible values of, the state of a device.

We now illustrate these principles on a simple example: a light bulb. We make use of state diagrams where states are represented by circles and transitions by arrows. It is assumed that a state can be its own successor.

The behaviour of the light bulb is approximated through three different states: `on`, `off` and `blowing`. The transition graph and the state definitions are shown in Figure 19 and Figure 20. The

```
schedule(Circuit,Agenda,State) :-
   empty(Agenda).

schedule(Circuit,OldAgenda,OldState) :-
   notempty(Agenda),
   impose_trans_constraints(OldState,NewState),
   apply_first_event(OldAgenda,NewState,Agenda,Now),
   compute_newstate(Circuit,NewState),
   update_agenda(OldState,NewState,Agenda,NewAgenda,Now),
   print_and_report(NewState,Now),
   schedule(Circuit,NewAgenda,NewState).

compute_newstate(Circuit,Newstate) :-
   Call =..  [Circuit|NewState],
   Call.
```

Figure 21 A meta program for the scheduler

states share two constraints expressing Kirchhoff's current law and Ohm's law. They differ by their respective constraints on the current flowing through ports 1 and 2.

For each class of device, we define a predicate of the following form:

typeDevice(*Name*, *State*, *Lparams*, *Lvalues*)

where `Name` is the name of the particular device, `State` is the state of the device, `Lparams` is the list of parameters of the device (e.g., an internal resistance), and `Lvalues` is the list of physical values for the ports of the device.

A circuit is described by a clause whose body consists of goals representing its devices. The connections are achieved through shared logical variables representing the physical values at the connectors of the devices. We refer to a circuit `nameCircuit` by the predicate

```
nameCircuit(LStates)
```

where `LStates` is a list containing the states of the circuit devices.

6.3.3 The scheduler

The simulation is directed by a scheduler which makes use of an agenda containing the current set of events to carry out. Events are the smallest relevant units of the simulation. Each event in the agenda is characterized by a device identification, a set of possible states and the time at which the action takes place. Initially, the agenda contains the set of external events. During the simulation, state changes of some devices can introduce new internal events to be executed at a later step. The scheduler behaviour is defined by the successive application of the following steps:

- impose the constraints implied by the transition graph;
- remove the first event from the agenda and apply it;
- compute the new circuit state;
- update the agenda;
- report possible anomalies;

until the agenda is empty.

The scheduler is a meta-program whose definition is shown in Figure 21. The first clause defines the halting condition. The second clause is the core of the scheduler.

The predicate `impose_trans_constraints` constructs the new state skeleton and imposes the transition constraints. The state skeleton is a list of variables, one for each device. The variables will be assigned to the states of the devices. The transition constraints prevent the device from receiving a state that cannot be reached from the current state using the transition graph.

The predicate `apply_first_event` removes the first event in chronological order from the agenda and applies it to the state skeleton.

```
light_bulb(Name, off, [Res, Ithreshold, Imax],[I1, U1, I2, U2]) ←
   U1 − U2 = Res × I1,
   I1 + I2 = 0,
   absolute_value(I1, I1abs),
   I1abs < Ithreshold.
light_bulb(Name, on, [Res, Ithreshold, Imax],[I1, U1, I2, U2]) ←
   U1 − U2 = Res × I1,
   I1 + I2 = 0,
   absolute_value(I1, I1abs),
   I1abs ≥ Ithreshold.
   I1abs < Imax.
light_bulb(Name, blowing, [Res, Ithreshold, Imax],[I1, U1, I2, U2]) ←
   U1 − U2 = Res × I1,
   I1 + I2 = 0,
   absolute_value(I1, I1abs),
   I1abs ≥ Imax.
```

Figure 22 A CLP representation for the light bulb

The predicate `compute_newstate` is the core of the simulator. It computes a new consistent circuit state. It constructs the call to the circuit and executes it. The way this computation is achieved is the topic of the next section.

The predicate `update_agenda` adds to the agenda new events that might have been generated by some transitons.

The last goal in the body is a recursive call to the scheduler with the new agenda and the new state.

The above scheduler only finds one possible state at each simulation. In general, we are interested in finding all possible states. There is no difficulty in generalizing the above program for that purpose.

6.3.4 Implementation of device models

There are various approaches to implement the device model in a CLP language. A possible implementation amounts to associating a clause to each state of the device. The body of the clause contains the constraints defining the state. This is mainly the approach used in Heintze et al. (1987). Using this representation, the light bulb can be implemented as depicted in Figure 22.

The problem with this representation is that it leads to a rather inefficient program for finding a state of the circuit. The resulting program implements mainly a standard backtracking approach exhibiting pathological behaviours. When the circuit is called for finding a consistent state, the computation basically selects a state (a clause) for each device and checks if its constraints are compatible with the constraint store coming from the assignment of states to the previous devices. If they are consistent, it proceeds to the next device; otherwise, it backtracks and assigns another state to the device. If no state is compatible, it goes back to the previous device and tries another state for the device, and so on until a consistent assignment is found. The above process does not use the constraints to prune the search space, i.e., to reduce the possible states of the devices.

However, by changing the representation of the devices, it is possible to achieve a much better behaviour. The idea here is to use only one clause per device, to use the implication construct and constraints over finite domains (to implement the states), and to state three kinds of constraints:

1 The physical constraints;
2 Value/state constraints that restrict the state of a device from information on its physical values;
3 State/value constraints that restrict the physical values of a device from information on its state.

Moreover, components with only one topology (e.g., the light bulb) only need the first two types of constraints. The light bulb using this representation is depicted in Figure 23.

Note that the circuit cannot find a consistent state on its own with the above representation. It

```
light_bulb(Name, State, [R, Ithreshold, Imax], [I1, U1, I2, U2]) ←
   I1 + I2 = 0,
   U1 − U2 = R * I1,
   absolute_value(I1, I1abs),

   I1abs < Ithreshold ⇒ State = off,
   I1abs ≥ Ithreshold ⇒ State ≠ off,
   I1abs < Imax ∧ Iabs ≥ Ithreshold ⇒ State = on,
   I1abs ≥ Imax ⇒ State = blowing.
   I0abs < Imax ⇒ State ≠ blowing.
```

Figure 23 Another representation of the light bulb

should be executed in conjunction with a generator of values for the states. There is no difficulty in generalizing the scheduler for that purpose.

6.3.5 Computation results

A program based on the above ideas has been applied to the simulation of hybrid circuits containing hundreds of devices such as machine tools and landing gear of aircraft. A simulation step requires about 1 or 2 seconds on a SUN-3 workstation. Note that the development of a similar program in an imperative language would require a substantial development effort given the variety of techniques used in the application.

7 Conclusion

This survey has attempted to convey some of the most important results emerging from research CLP. We have reviewed the syntax and the declarative and operational semantics of the two main classes of CLP languages: CLP languages (constraint solving) and *ask* and *tell* languages (constraint solving + constraint entailment). Three constraint systems, included in the main CLP languages implemented at the time of writing, have been studied in detail: Boolean algebra, linear rational (resp. real) arithmetic, and finite domains. Significant applications of CLP languages over these constraint languages have been described and include formal verification of digital circuits, simulation of hybrid circuits and car-sequencing.

The paper has not tried to be comprehensive, but has tried to convey some of the main ideas, concepts and methods behind CLP research. Beside the areas explicitly mentioned in the paper, we should mention (without hope to be exhaustive)

- concurrent constraint programming (e.g., Saraswat & Rinard, 1990; Rossi, 1991);
- extensions to the framework: for instance, meta-constraint programming (Heintze et al., 1989; Lim & Stuckey, 1990), constraint hierarchies (Montanari & Rossi, 1986; Borning et al., 1989), dynamic constraint solving (Maher & Stuckey, 1989; Van Hentenryck, 1990), and forward rules (Graf, 1989);
- parallel implementation (Van Hentenryck, 1989c), transformation, and analysis of CLP programs (Smith & Hickey, 1990; Marriot & Sondergaard, 1990);
- complexity analysis of CLP programs (Kanellakis et al., 1990; Cox et al., 1990);
- relation to other LP languages such as Andorra (Haridi 1990).

8 Further reading

The reader interested in the theory of CLP should definitely consult the seminal paper of Jaffar and Lassez (1987) and the paper by Saraswat et al. (1991). This paper provides an elegant denotational semantics of *ask* and *tell* languages in terms of information systems and closure operators.

The reader interested in particular CLP languages should refer to the references given in section 3.4. See also the *CACM* issue of July 1990 and the special issue of *Byte* magazine on CLP (August 1987).

The reader interested in applications of CLP should refer to the applications mentioned in sections 5.1.4, 5.2.4 and 5.3.3, and to the special issue of the *Journal of Logic Programming* (Volume 1–2, 1990).

The reader interested in application of CLP ideas to other fields related to Logic Programming should consult Maher (1987) and Saraswat (1989) for concurrent logic programming, and Kanellakis et al. (1990) for constraint query languages.

Finally most papers on CLP can be found in the proceedings of logic programming conferences (ICLP, NACLP, ILPS) distributed by MIT Press, the Artificial Intelligence conferences (AAAI, IJCAI), and the programming languages conferences (POPL, PLDI).

Acknowledgements

This paper is based on a tutorial that I gave at North-American Conference on Logic Programming at Austin, Texas. I would like to thank the organizers for inviting me and G. Hillebrand, J. Jaffar, A. Mayer, V. Saraswat, P. Stuckey and P. Wegner for their careful comments. Thanks are also due to my former colleagues from ECRC, especially M. Dincbas, H. Gallaire, T. Graf, and H. Simonis, and to Y. Deville and B. Le Charlier. Last but not least, I would like to thank J. Fox whose interest is responsible for my writing this paper.

References

Aiba, A, Sakai, K, Sato, Y, Hawley, DJ and Hasegawa, R, 1988. "Constraint logic programming CAL" In: *Proceedings of the International Conference on Fifth Generation Computer Systems* Tokyo, Japan, December.

Ait-kaci, H and Nasr, R, 1986. "LOGIN: a logic programming language with built-in inheritance" *Journal of Logic Programming* **3**(3) 185–215, October.

Ait-kaci, H and Podelski, A, 1990. "Is there a meaning to LIFE" (submitted for publication).

Berthier, F, 1988. "Using CHIP to support decision making" In: *Actes du Seminaire 1988—Programmation en Logique* Tregastel, France, May.

Borning, A, Maher, M, Martingale, A and Wilson, M, 1989. "Constraint hierarchies and logic programming" In: *Sixth International Conference on Logic Programming* Lisbon, Portugal, June.

Bryant, RE, 1986. "Graph based algorithms for Boolean function manipulation" *IEEE Transactions on Computers* **35**(8) 677–691.

Brzozowski, JA and Yoeli, M, 1985. *Combinatorial Static CMOS Networks. Technical Report CS-85-42*, University of Waterloo.

Buchberger, B, 1985. *Groebner Bases: An Algorithmic Method in Polynomial Ideal Theory*, Reidel Publishing Co.

Buttner, W and Simonis, H, 1987. "Embedding Boolean expressions into logic programming" *Journal of Symbolic Computation* **4** 191–205, October.

Clark, KL and McCabe, F, 1979. "The control facilities of IC-PROLOG" In: D Mitchie (ed.), *Expert Systems in the Micro Electronic Age* Edinburgh University Press.

Clocksin, WF, 1987. "Logic programming and digital circuit analysis" *J. Logic Programming* **4**(1) 59–82.

Cohen, J, 1990. "Constraint logic programming languages" *Commun. ACM* **28**(4).

Colmerauer, A, Kanoui, H, Pasero, R and Roussel, P, 1973. *Un système de communication homme-machine en français. Rapport de recherche*, Groupe Intelligence Artificielle, Université d'Aix-Marseille II.

Colmerauer, A, Kanoui, H and Van Caneghem, M, 1983. "Prolog, bases theoriques et developpements actuels" *Techniques et Sciences Informatiques* **2**(4) 271–311.

Colmerauer, A, 1987. "Opening the Prolog-III universe" *BYTE Magazine* **12**(9).

Colmerauer, A, 1990. "An introduction to Prolog III" *Commun. ACM* **28**(4) 412–418.

Cox, J, McAloon, K and Tretkoff, C, 1990. "Computational complexity and constraint logic programming languages" In: *Proceedings of the North-American Conference on Logic Programming* (NACLP-90), Austin, TX, October.

Dantzig, GB, Orden, A and Wolfe, P, 1955. "The generalized simplex method for minimizing a linear form under linear inequality constraints" *Pacific J. Math.* **5**(2) 183–195.

Davis, M and Putman, H, 1960. "A computation procedure for quantification theory" *J. ACM* **7** 201–215.

Davis, R, 1984. "Diagnostic reasoning based on structure and behavior" *Artificial Intelligence* **24** 347–410.

Deville, Y and Van Hentenryck, P, 1991. "An efficient arc consistency algorithm for a class of CSP problems" In: *International Joint Conference on Artificial Intelligence* Sydney, Australia, August.

Dincbas, M, Simonis, H and Van Hentenryck, P, 1988a. "Solving a cutting-stock problem in constraint logic programming" In: *Fifth International Conference on Logic Programming* Seattle, WA, August.

Dincbas, M, Simonis, H and Van Hentenryck, P, 1988b. "Solving large scheduling problems in logic programming" In: *EURO-TIMS Joint International Conference on Operations Research and Management Science* Paris, France, July.

Dincbas, M, Simonis, H and Van Hentenryck, P, 1988c. "Solving the car sequencing problem in constraint logic programming" In: *European Conference on Artificial Intelligence (ECAI-88)* Munich, Germany, August.

Dincbas, M, Simonis, H and Van Hentenryck, P, 1990. "Solving large combinatorial problems in logic programming": *J. Logic Programming* **8**(1–2) 75–93.

Dincbas, M, Van Hentenryck, P, Simonis, H, Aggoun, A, Graf, T and Berthier, F, 1988. "The constraint logic programming language CHIP" in: *Proceedings of the International Conference on Fifth Generation Computer Systems* Tokyo, Japan, December.

Fikes, RE, 1968. *A Heuristic Program for Solving Problems Stated as Non-deterministic Procedures,* PhD thesis, Computer Science Department, Carnegie-Mellon University, Pittsburgh.

Gallaire, H, 1985. "Logic programming: further developments" In: *IEEE Symposium on Logic Programming*, pp 88–99, Boston, MA, July.

Gallaire, H and Lasserre, C, 1982. *Metalevel Control for Logic Programs* Academic Press.

Gorlick, MM, Kesselman, CVF, Marotta, DA and Parker, DS, 1990. "Mockingbird: a logical methodology for testing" *J. Logic Programming* **8**(1–2) 95–119.

Graf, T, 1987. *Extending Constraint Handling in Logic Programming to Rational Arithmetic. Internal Report*, ECRC, Munich, Germany, September.

Graf, T, 1989. *Raisonner sur les contraintes en programmation logique*, PhD thesis, Université de Nice, France.

Graf, T, Van Hentenryck, P, Pradelles, C and Zimmer, L, 1989, "Simulation of hybrid circuits in constraint logic programming" In: *International Joint Conference on Artificial Intelligence* Detroit, MI, August.

Graf, T, Van Hentenryck, P, Pradelles, C and Zimmer, L, 1990. "Simulation of hybrid circuits in constraint logic programming" *Computers and Mathematics with Applications* **20**(9/10) 45–56,

Haridi, S, 1990. "A logic programming language based on the Ando¡rra model" *New Generation Computation* (to appear).

Heintze, NC, Michaylov, S and Stuckey, PJ, 1987. "CLP($\Re$) and some electrical engineering problems" In: *Fourth International Conference on Logic Programming* Melbourne, Australia, May.

Heintze, N, Michaylov, S, Stuckey, P and Yap, R, 1989. "On meta-programming in CLP($\Re$)" In: *Proceedings of the North-American Conference on Logic Programming (NACLP-89)*, pp 52–68, Cleveland, OH, October.

Imbert, JL, 1990, "About redundant inequalities generated by Fourier's algorithm" In: *AIMSA '90, Fourth International Conference on Artificial Intelligence: Methodology, Systems, Applications* Albena-Varna, Bulgaria, September.

Imbert, JL and Van Hentenryck, P, 1991. *Efficient Handling of Disequations in CLP over Linear Rational Arithmetics. Technical Report CS-91-23*, CS Department, Brown University.

Jaffar, J and Lassez, J-L, 1987. "Constraint logic programming" In: *POPL-87* Munich, Germany, January.

Jaffar, J and Michaylov, S, 1987, "Methodology and implementation of a CLP system" In: *Fourth International Conference on Logic Programming* Melbourne, Australia, May.

Jaffar, J, Michaylov, S, Stuckey, PJ and Yap, R, 1990. "The CLP($\Re$) *Language and System. Research report*", IBM.

Jorgensen, N and Marriot, K, 1990. "Useful compile-time optimizations for CLP($\Re$)", (draft).

Kanellakis, PC, Kuper, GM and Revesz, PZ, 1990. "Constraint query languages" In: *PODS-90* Nashville, TE.

Karmarkar, N, 1984. "A new polynomial-time algorithm for linear programming" *Combinatorica* **4**(4) 373–395.

Khachian, LG, 1979. "A polynomial algorithm in linear programming" *Soviet Math. Dokl.* **20**(1) 191–194.

Kowalski, R, 1974. "Predicate logic as programming language" In: *Proceedings of the IFIP Congress 74*, pp 569–574, North-Holland.

Lassez, J-L, Huynh, T and McAloon, K, 1989. "Simplification and elimination of redundant arithmetic constraints" In: *Proceedings of the North-American Conference on Logic Programming (NACLP-89)* Cleveland, OH, October.

Lassez, J-L and McAloon, K, 1988. "Applications of a canonical form for generalized linear constraints" In: *Proceedings of the International Conference on Fifth Generation Computer Systems* Tokyo, Japan, December.

Lassez, C, McAloon, K and Yap, R, 1987. "Constraint logic programming and option trading" *IEEE Expert* **2**(3) 42–50.

Lauriere, J-L, 1978. "A language and a program for stating and solving combinatorial problems" *Artificial Intelligence* **10**(1) 29–127.

Lim, P and Stuckey, P, 1990. "Meta programming as constraint programming" In: *Proceedings of the North-American Conference on Logic Programming (NACLP-90)* Austin, TX, October.

Mackworth, AK, 1977. "Consistency in networks of relations" *AI Journal* **8**(1) 99–118.

Maher, MJ, 1987. "Logic semantics for a class of committed-choice programs" In: *Fourth International Conference on Logic Programming*, pp 858–876, Melbourne, Australia, May.

Maher, M and Stuckey, P, 1989. "Expanding query power in constraint logic programming languages" In: *Proceedings of the North-American Conference on Logic Programming (NACLP-89)* Cleveland, OH, October.

Marriot, K and Sondergaard, H, 1990. "Analysis of constraint logic programs": In: *Proceedings of the North-American Conference on Logic Programming (NACLP-90)* Austin, TX, October.

Martin, U and Nipkow, T, 1986. "Unification in Boolean rings" In: *Proceedings of the 8th Conference on Automated Deduction*, pp 506–513, Oxford, UK, July.

Mohr, R and Henderson, TC, 1986. "Arc and path consistency revisited" *Artificial Intelligence* **28** 225–233.

Montanari, U and Rossi, F, 1986. "An efficient algorithm for the solution of hierarchical networks of constraints": In: *Workshop on Graph Grammars and their Applications in Computer Science* Warrenton, DC, December.

Naish, L, 1985. *Negation and Control in Prolog*, PhD thesis, University of Melbourne, Australia.

Older, W and Vellino, A, 1990. "Extending Prolog with constraint arithmetics on real intervals" In: *Canadian Conference on Computer & Electrical Engineering* Ottawa.

Parker, DS and Muntz, RR. "A Theory of Directed Logic Programs and Streams" In: Fifth International Conference on Logic Programming, Seattle, WA, August.

Parker, RG and Rardin, RL, 1988. *Discrete Optimization* Academic Press.

Parrello, BD, 1988. "CAR WARS: the (almost) birth of an expert system" *AI Expert* **3**(1) 60–64.

Plotkin, GD, 1981. *A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19*, CS Department, University of Aarhus.

Radeanu, S, 1974. *Boolean Functions and Equations* North-Holland.

Rossi, F, 1991. *Towards an Ideal Notion of Constraint Programming*, PhD Thesis Proposal, Computer Science Department, University of Pisa, Italy.

Saraswat, VA, 1989. *Concurrent Constraint Programming Languages*, PhD thesis, Carnegie-Mellon University.

Saraswat, VA, Kahn, K and Levy, J, 1990. "Janus: a step towards distributed constraint programming" In: *Proceedings of the North-American Conference on Logic Programming (NACLP-90)* Austin, TX, October.

Saraswat, VA and Rinard, M, 1990. Concurrent constraint programming" In: *Proceedings of Seventeenth ACM Symposium on Principles of Programming Languages* San Francisco, CA, January.

Saraswat, VA, Rinard, M and Panangaden, P, 1991. "Semantic foundations of concurrent constraint programming" In: *Proceedings of Ninth ACM Symposium on Principles of Programming Languages* Orlando, FL, January.

Shapiro, E, 1990. "The family of concurrent logic programming languages" *Computing Surveys* **21**(3) 413–510.

Simonis, H, 1989. "Test generation using the constraint logic programming language CHIP" In: *Sixth International Conference on Logic Programming* Lisbon, Portugal, June.

Simonis, H and Dincbas, M, 1987a. "Using an extended Prolog for digital circuit design" In: *IEEE International Workshop on AI Applications to CAD Systems for Electronics*, pp 165–188, Munich, Germany, October.

Simonis, H and Dincbas, M, 1987b. "Using logic programming for fault diagnosis in digital circuits" In: *German Workshop on Artificial Intelligence (GWAI-87)*, pp 139–148, Geseke, Germany, September.

Smith, DA and Hickey, TJ, 1990. "Partial evaluation of a CLP language" In: *Proceedings of the North-American Conference on Logic Programming (NACLP-90)* Austin, TX, October.

Stuckey, PJ, 1990. "Incremental linear arithmetic constraint solving and detection of implicit equalities" (submitted for publication).

Sussman, GJ and Steele, GL, 1980. "CONSTRAINTS—A language for expressing almost-hierarchical descriptions" *AI Journal* **14**(1).

Van Hentenryck, P, 1989a. "A logic language for combinatorial optimization" *Annals of Operations Research* **21** 247–274.

Van Hentenryck, P, 1989b. *Constraint Satisfaction in Logic Programming* Logic Programming Series, The MIT Press.

Van Hentenryck, P, 1989c. "Parallel constraint satisfaction in logic programming: preliminary results of CHIP within PEPSys" In: *Sixth International Conference on Logic Programming* Lisbon, Portugal, June.

Van Hentenryck, P, 1990. "Incremental constraint satisfaction in logic programming" In: *Seventh International Conference on Logic Programming* Jerusalem, Israel, June.

Van Hentenryck, P and Deville, Y, 1990. *Operational Semantics of Constraint Logic Programming over Finite Domains. Technical Report CSA-90-23*, CS Department, Brown University.

Van Hentenryck, P and Deville, Y, 1991, "The cardinality operator: a new logical connective and its application to constraint logic programming" In: *Eighth International Conference on Logic Programming (ICLP-91)* Paris, France, June.

Van Hentenryck, P and Graf, T, 1990. "Standard forms for rational linear arithmetics in constraint logic programming" In: *International Symposium on Artificial Intelligence and Mathematics* Fort Lauderdale, FL, January. (also *ECRC internal Report IR-LP-2217.*)

Voda, P, 1988. *The Constraint Language Trilogy: Semantics and Computations. Technical report*, Complete Logic Systems, North Vancouver, BC, Canada.

Walinsky, C, 1989. "CLP(Σ^*)" In: *Sixth International Conference on Logic Programming* Lisbon, Portugal, June.