Constraint Logic Programming

Pascal Van Hentenryck

Department of Computer Science
Brown University
Providence, Rhode Island 02912

Technical Report No. CS-91-05
January 1991

Constraint Logic Programming

Pascal Van Hentenryck
Brown University, Box 1910,
Providence, RI 02912
USA
Email: pvh@cs.brown.edu

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

Logic programming [CKPR73, Kow74] is based on the idea that predicate logic (restricted to Horn clauses) can be given a procedural interpretation. It originates from research in automatic theorem-proving, especially on the resolution principle [Rob65]. Logic programming languages, including Prolog, are nondeterministic programming languages whose primitive operation is unification.

Constraint logic programming (CLP) is a simple, yet fundamental, generalization of logic programming. It amounts to replacing unification by constraint solving. The move from unification to constraint solving is appealing both from a theoretical and practical point of view.

From a theoretical standpoint, the move 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.

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.

CLP has received increased attention in recent years, originating from the initial work on Prolog II [CKVC83] and Prolog III [Col90], the CLP scheme [JL87], and the development and implementation of the first generation of CLP languages such as CHIP [DVHS+88], CLP($\Re$) [JM87], and Trilogy [Vod88]. Various constraint systems have been investigated for inclusion in CLP languages, such as Boolean algebra [ASS+88, BS87, Col90], linear rational arithmetic [Col90, Gra87, JM87, Stu90, VHG90], linear integer programming [Vod88], and finite domains [VH89b]. CLP languages have also been applied to numerous combinatorial problems in operations research (e.g. [DSVH90, VH89a]), hardware design (e.g. [SND88]), and decision-support systems (e.g. [LMY87]). The application of CLP ideas to other areas of logic programming has been fruitful as well. In the case of concurrent logic programming [Sha90], it led to the notion of constraint entailment [Mah87] (i.e. deciding whether a constraint is implied by a conjunction of constraints) and the definition of the *ask* and *tell* family of programming languages [Sar89]. Constraint entailment should now be considered as an additional primitive operation for CLP languages.

The purpose of this paper is to survey this new class of programming languages from the underlying theory, to the design and implementation of CLP languages, and to their applications to combinatorial problems. The paper is divided into four parts:[1]

1. *tell* languages;
2. constraint systems;
3. *ask* and *tell* languages;
4. applications.

The first part, *tell* languages, describes the class of CLP languages whose main operation is constraint solving. This class, initially called the CLP scheme [JL87], is parametrized on the constraint system (in which constraint solving) takes place. The syntax, and declarative and operational semantics of *tell* languages is studied in some detail and contrasted with those of logic programming.

The second part investigates various constraint systems which can be used to instantiate the framework of *tell* languages. Three of them are studied in detail: Boolean algebra, linear rational (resp. real) arithmetic, and finite domains. For each of them, we describe the syntax and semantics of constraints, discuss the constraint solver and the applications, and present a simple example.

The third part investigates *ask* and *tell* languages, i.e. the class of languages with two primitive operations: constraint solving and constraint entailment. We investigate various ways to introduce constraint entailment in CLP languages, describe their syntax, declarative and operational semantics, and some of their applications.

[1] The terminology for classifying CLP languages is borrowed from [Sar89].

The last part presents three significant applications of CLP languages to combinatorial problems: one for each of the constraint systems studied in the second part. The applications include formal verification of digital circuits, simulation of hybrid circuits, and car sequencing.

This survey does not attempt to be comprehensive and some areas of research have been left out: they will be mentioned in the conclusion. In particular, the focus here is on the use of constraints for combinatorial problems. We do not consider the class of concurrent constraint programming languages [Sar89] where constraints are used for synchronization and communication of concurrently executing agents.

2 Tell Languages

In this section, we present *tell* languages, i.e. the class of languages whose primitive operation is constraint solving. The design and implementation of *tell* languages and the development of the underlying theory have progressed in parallel. *Tell* languages can be traced back to Prolog II [CKVC83] and the first generation of these languages include CAL [ASS+88], BNR-Prolog [OV90], CHIP, CLP(R), Prolog III and Trilogy. The semantics of *tell* languages was first investigated by Jaffar and Lassez in the CLP scheme [JL87] and subsequently generalized by Maher [Mah87] and Saraswat [Sar89]. In the following, after some preliminaries, we describe the syntax and the declarative and operational semantics of *tell* languages, and we study various equivalence results. The presentation is consistent with the above papers and follows closely [Sar89]. We also overview the main *tell* languages.

2.1 Preliminaries

We consider a many-sorted first-order language Σ with an associated set S of sorts and sets T of signatures. Σ is the union of the function symbols $\mathcal{F}$ and predicate symbols $\mathcal{P}$. We assume that $\mathcal{P}$ contains an equality symbol $=_s$ for each sort $s \in S$. When there is no ambiguity, we will use $=$ instead of $=_s$. A signature for an n-ary function or predicate symbol is a sequence of $n+1$ or n sorts respectively. The sort of a function symbol is the last element in the sequence. We also assume a set Var of variables $\{V_s\}_{s \in S}$, where V_s is the set of variables of sort s. We denote by $term(\mathcal{F})$ and $term(\mathcal{F}, Var)$ the set of ground and (possibly) non-ground terms. In the following, function symbols will be denoted by the letter f, predicate symbols by the letter c, variables by the letters x, y, z, terms by the letters t, u, and formulas by the letter ψ, all of them possibly subscripted. Also constants (i.e. function symbols of arity 0) are denoted by constants a, b, c, possibly subscripted.

Definition 1 A structure $\mathcal{A}$ for a language Σ consists of

1. a collection D of non-empty sets $\{D_s\}_{s \in S}$, where D_s is called the carrier of sort s.

2. an interpretation I that assigns

(a) to each function symbol f of arity n, a function

$$I(f) : D_{s_1} \times \ldots \times D_{s_n} \rightarrow D_s$$

where $\langle s_1, \ldots, s_n, s \rangle$ is the signature of f.

(b) to each predicate symbol c of arity n, a relation

$$I(c) \subseteq D_{s_1} \times \ldots \times D_{s_n}$$

where $\langle s_1, \ldots, s_n \rangle$ is the signature of p. We also assume that each equality symbol is interpreted as the underlying equality of the carrier.

We now turn to the concept of valuation.

Definition 2 Let $V \subseteq Var$. An $(\mathcal{A}, V)$-valuation ϑ is an assignment of a value in D_s to each variable of sort s in V. We denote by $\vartheta(x)$ the value assigned to x in ϑ. The domain of a valuation can be extended to terms by the following inductive rule:

- if $f(t_1, \ldots, t_n)$ is a term, then $\vartheta(f(t_1, \ldots, t_n))$ is $I(f)(\vartheta(t_1), \ldots, \vartheta(t_n))$.

The notion of satisfiability is defined in the standard way.

Definition 3 Let $\mathcal{A}$ be a structure, ψ, ψ_1, ψ_2 formulas, and ϑ a $(\mathcal{A}, V)$-valuation. The truth of ψ in $\mathcal{A}$ wrt ϑ, denoted $\mathcal{A} \models_\vartheta \psi$, can be defined inductively as follows:

- if $\psi \equiv \mathit{true}$ then $\mathcal{A} \models_\vartheta \psi$;
- if $\psi \equiv \mathit{false}$ then not $\mathcal{A} \models_\vartheta \psi$;
- if $\psi \equiv c(t_1, \ldots, t_n)$ then $\mathcal{A} \models_\vartheta \psi$ iff $\langle \vartheta(t_1), \ldots, \vartheta(t_n) \rangle \in I(c)$;
- if $\psi \equiv \neg\psi_1$, then $\mathcal{A} \models_\vartheta \psi$ iff not $\mathcal{A} \models_\vartheta \psi_1$;
- if $\psi \equiv \psi_1 \wedge \psi_2$, then $\mathcal{A} \models_\vartheta \psi$ iff $\mathcal{A} \models_\vartheta \psi_1$ and $\mathcal{A} \models_\vartheta \psi_2$;
- if $\psi \equiv \forall x\, \psi_1$, then $\mathcal{A} \models_\vartheta \psi$ iff $\mathcal{A} \models_{\vartheta[x/d]} \psi_1$ for all $d \in D_s$ where s is the sort of x and $\vartheta[x/d]$ is the same valuation as ϑ except that x is assigned to d.

The other logical connectives can be given their semantics by the standard rewriting rules. We are now in position to define the constraint systems we consider.

Definition 4 [Sar89] A constraint system is a quadruple $(\Sigma, \Delta, V, \Phi)$, where Σ is a many-sorted first-order language with associated sorts S and signatures T, Δ is a class of Σ-structures, V is an S-sorted set of variables and Φ, the set of basic constraints, is a non-empty set of (Σ, V)-formulas. Constraints are simply conjunctions of basic constraints.

Note that it is often the case that Δ contains a single structure. We will see later that, in such a case, the declarative semantics of *tell* languages is given by a unique model. In the following, we assume an underlying constraint system $C = (\Sigma_c, \Delta_c, V_c, \Phi_c)$. Constraints are denoted by σ, σ', possibly subscripted.

2.2 Syntax

CLP programs are constructed from the constraint system C and a set Σ_p of predicate symbols, disjoint from Σ_c. Predicate symbols in Σ_p are denoted by the letters p, q, possibly subscripted. An atom is an expression $p(t_1, \ldots, t_n)$ where $t_1, \ldots, t_n \in term(\mathcal{F}, Var)$. Atoms are denoted by the letters H, A, B possibly subscripted.

A CLP program (in a *tell* language) is a set of clauses of the form

$$H \leftarrow \sigma \diamond B_1 \wedge \ldots \wedge B_n.$$

A CLP goal is a clause without head and constraint

$$\leftarrow B_1 \wedge \ldots \wedge B_n.$$

In the following, goals will be denoted by the letter G, possibly subscripted.

Note that $\leftarrow, \wedge$ have their standard logical reading. $\diamond$ also denotes conjunction and is used to separate the constraint part from the body. Also a clause is a $(\Sigma_p \cup \Sigma_c)$-formula which is (implicitly) universally quantified. Moreover, logic programming is a particular case of CLP where only equations between first-order terms are allowed as constraints.

The syntax described above is abstract and was chosen to simplify the definition of the semantics. There is no difficulty in translating any actual concrete syntax into the abstract syntax. In the examples presented in this paper, we will slightly depart from the above syntax by replacing $\wedge$ by commas, using uppercase letters for variables, and allowing for constraints in the query.

2.3 Declarative Semantics

In logic programming, the declarative semantics is defined in terms of first-order logical consequences (e.g. [Llo84]). An answer to a goal G in the context of some program P is a substitution θ such that $(\forall)(G\theta)$ is logically implied by the program P, denoted

$$P \models (\forall)(G\theta).$$

where $(\forall)(\psi)$ represents the universal closure of ψ.

In CLP, an answer is no longer a substitution but rather a constraint σ. Moreover, in order to define the models of CLP programs, we consider $(\Sigma_p \cup \Sigma_c)$-structures extending those of the constraint system by assigning to each predicate symbol p in Σ_p of signature $\langle s_1, \ldots, s_n \rangle$ a subset of $D_{s_1} \times \ldots \times D_{s_n}$. A $(\Sigma_p \cup \Sigma_c)$-structure $\mathcal{A}$ is a model of a program P iff all its clauses are assigned to true in $\mathcal{A}$. In the following, we denote by Δ_P all models of a program P.

An answer to a goal G wrt CLP program P is a constraint σ such that $(\forall)(\sigma \rightarrow G)$ is true in all models of P, denoted

$$\Delta_P \models (\forall)(\sigma \rightarrow G).$$

There is an important case to stress here. If Δ_c contains a unique structure, then there is a minimal model M for program P, which is given by the intersection of all models. The declarative semantics is then given by this unique model. More generally, each structure $\mathcal{S}$ in Δ_c can be extended in a minimal model, which is the intersection of all models extending $\mathcal{S}$ and only those minimal models have to be considered when giving the declarative semantics.

2.4 Operational Semantics

We now turn to the description of the operational semantics. We use a simple structural operational semantics [Plo81].

The operational semantics of CLP is a simple generalization of the semantics of logic programming where unification is replaced by constraint solving. It might be defined by considering configurations of the form $\langle G, \sigma \rangle$ where G is a conjunction of atoms (the goal part) and σ is a constraint (the constraint part). 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. Now the only transition rule that needs to be defined between configurations is the following:

$$\frac{\begin{array}{l} p(u_1, \ldots, u_n) \leftarrow \sigma' \diamond B_1 \wedge \ldots \wedge B_m \in P \\ \Delta_c \models (\exists)(\sigma \wedge \sigma' \wedge t_1 = u_1 \wedge \ldots \wedge t_n = u_n) \end{array}}{\langle p(t_1, \ldots, t_n) \wedge G \ , \ \sigma \rangle \longmapsto \langle B_1 \wedge \ldots \wedge B_m \wedge G \ , \ \sigma \wedge \sigma' \wedge t_1 = u_1 \wedge \ldots \wedge t_n = u_n \rangle}$$

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), $(\exists)(\psi)$ represents the existential closure of ψ, and the program clause has been renamed properly not to share any variable with the goal.[2]

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 $\stackrel{*}{\longmapsto}$ the transitive closure of $\longmapsto$ and say that a configuration γ diverges in program P if there exists an infinite sequence of transitions $P \vdash \gamma \longmapsto \gamma_1 \longmapsto \ldots \longmapsto \gamma_i \longmapsto \ldots$
The success and divergence sets are defined in the following way:

$$\begin{array}{lcl} SS[P] & = & \{G \mid P \vdash G \stackrel{*}{\longmapsto} \sigma\} \\ DS[P] & = & \{G \mid G \text{ diverges in } P\}. \end{array}$$

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

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

[2] In recent work [SR90a], Saraswat gives an operational semantics precluding the need for renaming.

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

$$RES[P,G] \quad = \quad \{\sigma \mid P \vdash G \stackrel{*}{\longmapsto} \sigma\}.$$

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 $\Delta_c \models (\exists)(\sigma)$ and *false* otherwise.

Particular care is devoted in CLP languages to present the answer constraint in a way that makes one or all solutions apparent. It is difficult however to formalize this concept independently of the constraint solver. In the same way, it is not always possible to present the answers only in terms of the query variables.

Logic programming is once again a particular *tell* language where the satisfiability of constraints is checked by unification and the accumulated constraints are represented by substitutions.

2.5 Equivalences Between the Semantics

We now mention various results relating the declarative and operational semantics. The original versions of these theorems can be found in [JL87, Mah87, Sar89]. The first result we mention is the soundness of the operational semantics.

Theorem 5 [Soundness] Let P be a program, G a goal, and σ a constraint. We have

$$\begin{array}{lll} \sigma \in RES[P,G] & \text{implies} & \Delta_P \models (\forall)(\sigma \rightarrow G). \\ G \in FS[P] & \text{implies} & \Delta_P \models \neg(\exists)(G). \end{array}$$

The next theorem is the equivalent for *tell* languages of the strong completeness theorem for logic programming.

Theorem 6 [Strong Completeness] Let P be a program, G be a goal, and σ a constraint. If $\Delta_P \models (\forall)(\sigma \rightarrow G)$ and $\Delta_c \models (\exists)(\sigma)$, then there exist constraints $\sigma_1, \ldots, \sigma_n \in RES[P,G]$ such that

$$\Delta_c \models (\forall)(\sigma \rightarrow (\exists y_1, \ldots, y_m)(\sigma_1 \vee \ldots \vee \sigma_n))$$

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

As it is the case for logic programming (when restricting ourselves to Herbrand interpretations), there is no counterpart to the above theorem for negation.

2.6 Tell Languages

The above framework defines a class of programming languages parametrized by the constraint system. Several "real" *tell* languages have been designed and implemented on various constraint systems. The most well-known of these languages are probably CHIP, CLP($\Re$),

Prolog III, and Trilogy. 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 *tell* languages have been, or are currently being, defined and implemented. They include BNR-Prolog [OV90] (approximation of real arithmetic), LOGIN [AkN86] and its successor LIFE [AkP90] (equations between ψ-terms), CLP(Σ^*) [Wal89] (strings), and Tangram [PR88] (streams). Finally, Saraswat in [Sar89, SKL90] shows how various data-structures (e.g. arrays, hastables, bags) can be seen as constraint systems.

3 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 the previous section. Note also that the most basic constraint system is the Herbrand system which contains only equations (and possibly disequations) between first-order terms and is included in all CLP languages (but not always with disequations).

Before studying the various constraint systems, it is worth mentioning some of the requirements of *tell* languages on the constraint solvers. First the constraint solver should be incremental, i.e. it should use the fact that the accumulated constraints are 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 (recall that a constraint is a conjunction of basic 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 *tell* 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.

3.1 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 [Rad74] 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.

3.1.1 Syntax

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.

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 allows for Boolean constraints. Prolog-III considers simple Boolean constraints.

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.

3.1.2 Semantics

The semantics of the constraint system is given in terms of Boolean algebra. Recall that a Boolean algebra is a system $\langle B, \cup, \wedge, ', 0, 1\rangle$ where B is a set called the carrier, $\cup$ and . are operations from $B \times B$ into B, $'$ is an operation from B to B, and $0, 1$ are distinguished constants ($0 \neq 1$), such that for $x, y, z \in B$, the following equations hold[3]

$$
\begin{aligned}
& x \cup y = y \cup x, \\
& xy = yx, \\
& (x \cup y) \cup z = x \cup (y \cup z), \\
& (xy)z = x(yz), \\
& x \cup xy = x, \\
& x(x \cup y) = x, \\
& x \cup yz = (x \cup y)(x \cup z), \\
& x(y \cup z) = xy \cup xz, \\
& x \cup 1 = x, \\
& x0 = 0, \\
& x \cup x' = 1, \\
& xx' = 0.
\end{aligned}
$$

[3] In the following, we denote $x \wedge y$ by xy.

The only delicate point about the semantics is the meaning assigned to constants. There are various possibilities corresponding to various Boolean algebras. In general, it is convenient to work with the free Boolean algebra generated from the constants. With this interpretation, a Boolean constant has a value in B that is fixed but unknown. Intuitively, it amounts to seeing the constants 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

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

Before considering constraint solving over Boolean algebra, it is useful to introduce the concept of Boolean ring. A Boolean ring is a commutative ring of characteristic 2 (i.e. $x + x = 0$) whose elements are idempotent (e.g. $xx = x$). It turns out that any Boolean algebra $\langle B, \cup, \wedge, ', 0, 1\rangle$ can be transformed into a Boolean ring with unit $\langle B, +, \wedge, 0, 1\rangle$ and vice-versa by fixing

$$x + y = xy' \cup x'y$$

and

$$x \cup y = xy + x + y$$
$$x' = 1 + x$$

Working in the Boolean ring might often simplify computations. Also, when working in the Boolean ring, we feel free to use x' instead of $x + 1$.

3.1.3 Constraint Solving

We now investigate constraint solving in Boolean algebra.

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 [KKR90].

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 [Buc85] specialized to Boolean equations. CHIP uses a Boolean unification based on variable elimination (Boole's method) [BS87]. Other methods are possible and include other Boolean unification algorithms (e.g. [MN86]) or model enumeration procedures (e.g. [DP60]). 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.

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 $t_1 + t_2 = 0$. 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 7 Let $ax + b = 0$ be an equation where a, b are Boolean terms without variables. The equation is consistent iff $a'b = 0$. Moreover, if it is consistent, then a most general solution is given by the assignment

$$x \leftarrow b + a'y$$

where y is a new Boolean variable.

It follows that solving a Boolean equation with one variable amounts to determining if $a'b$ 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 reduce $a'b$ 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

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

This equation is consistent if and only if

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

is consistent in which case the assignment

$$x_n \leftarrow h_1(x_1, \ldots, x_{n-1}) + g_1'(x_1, \ldots, x_{n-1})y_n$$

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

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

which can be expressed as

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

```
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 = ax + 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 1: Boolean Unification based on Boole's Method

and is consistent iff

$$g'_p(x_1, \ldots, x_{n-p})\ h_p(x_1, \ldots, x_{n-p}) = 0$$

is consistent in which case the assignment

$$x_{n-p+1} \leftarrow h_p(x_1, \ldots, x_{n-p}) + g'_p(x_1, \ldots, x_{n-p}) y_{n-p+1}$$

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

$$f_n(x_1) \stackrel{\text{def}}{=} ax_1 + b = 0$$

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 [BS87] is precisely using the above variable elimination technique. The algorithm is shown in Figure 1.

3.1.4 Example

We illustrate the constraint system by presenting the description of an xor-gate at the ideal switch level [BY85]. An xor-gate can be represented using ideal switches in the following way [SD87a].

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

```
R = a + b
```

as an answer. 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 by Boolean unification.

```
← 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
```

3.1.5 Applications

Tell languages over Boolean algebra have been applied to various areas including hardware design and propositional logic. In [SND88], 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 [Col90], applications to diagnosis and propositional logic are also presented.

3.1.6 Discussion

Several points deserve to be mentioned on this constraint system. First it has the important property that solutions (and hence accumulated constraints) 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 case where only one solution is required, it is not clear however whether the algorithm is appropriate or whether enumeration algorithms would perform better.

3.2 Linear Rational Programming

One of the most successful algorithms ever designed is certainly the simplex algorithm of G. Dantzig [DOW55]. 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($\Re$)[4]. All three systems are based on the simplex algorithm. See [Gra87, JM87, Stu90, VHG90] for discussions of this constraint system.

3.2.1 Syntax

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 $\delta \in \{>, \geq, =, \leq, <, \neq\}$.

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 $t_1 < t_2$.

3.2.2 Semantics

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.

3.2.3 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 [Kha79], who was first to propose a polynomial time 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 [LMA88].

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 [Kar84]. It is not clear presently how the polynomial algorithm of [Kar84] can be made incremental, i.e. how it can use the fact that the

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

accumulated constraints are 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.[5]

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

$$a + a_1x_1 + \ldots + a_nx_n \;=\; 0 \;\; (n > 0, a_i \neq 0)$$

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

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

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

Inequalities Inequalities of the form $t_1 \geq t_2$ and $t_1 \leq t_2$ 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 they 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).[6] 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

$$y_1 = b_1 + a_{11}x_1 + \ldots + a_{1m}x_m$$
$$\ldots$$
$$y_n = b_n + a_{n1}x_1 + \ldots + a_{nm}x_m$$

[5]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.

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

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 accumulated constraints (in this case equations over nonnegative variables) are 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:

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 consider 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. $t_1 \geq t_2$ and $t_1 \neq t_2$). It is natural to consider that a disequation is in solved form if it is of the form

$$0 \neq b + a_1 x_1 + \ldots + a_n x_n$$

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($\Re$), and Prolog III, have different ways of ensuring the absence of hidden constants.

In CLP($\Re$) 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

```
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 2: A Simple Mortgage Program

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($\Re$) solution can be found in [Stu90].

CHIP has a very different approach described in [VHG90]. 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.[7] 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 [DOW55]. 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($\Re$) and Prolog III need to reconsider all homogeneous constraints. CHIP entails a slightly more costly pivoting rule while CLP($\Re$) and Prolog III have more complicated algorithms to add a constraint.

3.2.4 Example

We use a simple mortgage program from [JM90] to illustrate this constraint system. In the program shown in Figure 2, 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

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

$\leftarrow$ `mortgage(100000,360,12,1025,B)`

has answer constraint `B = 12625.9`. The query

$\leftarrow$ `mortgage(P,360,12,R,B)`

has answer constraint `P = 0.0278 * B + 97.22 * R` $\wedge$ `P` $\geq$ `0`. Finally the query

$\leftarrow$ `0` $\leq$ `B, B` $\leq$ `1030` $\diamond$ `mortgage(100000,360,12,1025,B)`

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

3.2.5 Applications

There have been various applications of this constraint system. They range from simulation and diagnostics of various circuits and devises [GVHPZ89, GVHPZ90, HMS87, GKMP90] to decision-support systems [Ber88, LMY87] and geometrical problems [Col90].

3.2.6 Discussion

The constraint solver for the above constraint system is complete and, although the simplex algorithm is exponential is the worst case, it is reasonable to conclude that it is efficient. Note however that the accumulated constraints 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 [LHM89, Imb90]. 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 allow as well for the optimization of a linear function. The implementation is based on the simplex algorithm once again.

3.3 Finite Domains

Many combinatorial problems such as graph coloring, 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 [PR88], 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 [VH89b]. The presentation that follows is mostly based on [VHD90b].

3.3.1 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 $\delta \in \{>, \geq, =, \leq, <, \neq\}$.

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.

3.3.2 Semantics

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 number constraints) can be given for any symbolic constraint available in CHIP.

3.3.3 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 8 The *basic* constraints are either *domain* constraints or *arithmetic* constraints.

- domain constraint: $x \in \{a_1, \ldots, a_n\}$;

- arithmetic constraints:
  - $ax \neq b$;
  - $ax = b$;
  - $ax = by + c$ $(a \neq 0 \neq b)$;
  - $ax \geq by + c$;
  - $ax \leq by + c$;

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 9 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 10 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_i's $(1 \leq \mathrm{i} \leq \mathrm{k})$.

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 research [Mac77]. 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 11 Let $c(x)$ be a unary constraint and D_x be the domain of x. Constraint $c(x)$ is said to be *node-consistent wrt* D_x if $c(x/a)$ holds for each value $a \in D_x$.

Definition 12 Let $c(x, y)$ be a binary constraint and D_x, D_y be the domains of x, y. Constraint $c(x, y)$ is 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 13 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 [VHD90b].

Theorem 14 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 [Mac77].

Algorithm 15 To transform the system of constraints S into a system in solved 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 16 returns *true* if S is satisfiable and *false* otherwise.

Algorithm 16 To check the satisfiability of a system of constraints S:

1. apply Algorithm 15 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 15 and 16 is the complexity of arc-consistency algorithms. In [MH86], 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 [DVH90], 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 16 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 [VHD90b] 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 specialized versions of k-consistency.

Definition 17 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 *k-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 k-consistent (k being determined by the number of variables in the constraint). Hence the test for satisfiability given previously can be replaced by a test for k-consistency of the constraints and each step amounts to enforcing a k-consistency algorithm, reducing the domains of the variables. k-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 a 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:

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

Assume that

$$a_1x_1 + \ldots + a_nx_n + a$$

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

$$b_1y_1 + \ldots + b_my_m + b$$

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:

$$a_1x_1 + \ldots + a_nx_n + a \geq min_2$$

$$b_1y_1 + \ldots + b_my_m + b \leq max_1$$

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

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

and each variable y_i should satisfy

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

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.

Similar handling of equations and inequalities has been used in Alice [Lau78] and REF-ARF [Fik68].

3.3.4 Example

The well-known SEND+MORE=MONEY puzzle consists of giving to each letter a different digit from {0,...,9} so that the addition is satisfied. A program to solve this problem is depicted in Figure 3. The program can be simplified by removing the carries and having only one equation. However, the present version will illustrate best the way computations are achieved in this constraint system.

Finite domain programs are generally made up of two parts: a first part that generates the constraints and a second part that generates values for some of the problem variables. The second part is necessary because the constraint solver alone might not be able to find a solution. The resulting effect is to obtain a partial enumeration algorithm where constraint solving is interleaved with nondeterministic choices.

In the SEND+MORE=MONEY problem, the constraint part contains various constraints. First, the domain constraints created by the `allin` predicate; second, the disequations stated by the `alldifferent` predicate; third, the equality constraints (and two disequations) enforcing the addition constraints. The `alldifferent` predicate illustrates how constraints can be generated dynamically. This is of primordial importance to write generic programs. The choice part here is given by the two `labelling` goals (whose definition is not shown) that generate values for the variables.

Consider the first steps of the program. Since M $\neq$ 0 and R1 = M, the program immediately concludes that M = R1 = 1. The reason is that M is now ranging over {1,...,9} because of the disequation and that R1 ranges over {0,...,1}. Thus, the only way to satisfy the equation is to assign 1 to both variables. All disequations involving M are then considered removing the value 1 from the domains of all the other letters. The constraint

$$R2 + S + M = O + 10 \times R1$$

is then considered. Since M = R1 = 1, this constraint is now

$$R2 + S + 1 = O + 10.$$

R2 + S + 1 ranges over [3,11] while O + 10 ranges over [10,19]. The program concludes that R2 + S + 1 must be greater than 9 reducing the range of S to [8,9]. Also, it concludes that O + 10 must be smaller than 12 implying that O must be smaller than 2. However, it was

```
sendmory([S,E,N,D,M,O,R,Y],[R1,R2,R3,R4]) ←
   generate_constraints([S,E,N,D,M,O,R,Y],[R1,R2,R3,R4]),
   labeling([R1,R2,R3,R4]),
   labeling([S,E,N,D,M,O,R,Y]).

generate_constraints([S,E,N,D,M,O,R,Y],[R1,R2,R3,R4]) ←
   allin([S,E,N,D,M,O,R,Y],0,9),
   allin([R1,R2,R3,R4],0,1),
   alldifferent([S,E,N,D,M,O,R,Y]),
   addition([S,E,N,D,M,O,R,Y],[R1,R2,R3,R4]).

addition([S,E,N,D,M,O,R,Y],[R1,R2,R3,R4]) ←
   S ≠ 0,
   M ≠ 0,
   R1 = M,
   R2 + S + M = O + 10×R1,
   R3 + E + O = N + 10×R2,
   R4 + N + R = E + 10×R3,
   D + E = Y + 10×R4 ◇.

allin([],L,U).
allin([F|T],L,U) ←
   F ∈ L..U ◇
   allin(T,L,U).

alldifferent([]).
alldifferent([X|Xs]) ←
   outof(X,Xs),
   alldifferent(Xs).

outof(X,[]).
outof(X,[Y|Ys]) ←
   X ≠ Y ◇
   outof(X,Ys).
```

Figure 3: The SEND+MORE=MONEY Example

already found that O must be different from 1 and this leads directly to the conclusion that O must be equal to 0. Note at this point that the constraint is not solved. It is possible to give a value to R2 and S such that it is not satisfied. The second equation is now considered

$$R3 + E = N + 10 \times R2,$$

leading to R2 = 0. Indeed, since R3 + E ranges over [2,10] and N + 10×R2 over [2,19], the program concludes that N + 10×R2 must be smaller than 11 which implies R2 = 0. The previous constraint is next reconsidered and leads to the conclusion that S=9. Pursuing the same kind of reasoning, the program finds a solution without any backtracking in a couple of milliseconds.

Note that the above result is independent from the ordering of the constraints. The algorithm simply enforces k-consistency on the constraints at each step. Hence exactly the same pruning (with roughly the same efficiency) is achieved at each choice point whatever the constraint ordering.

3.3.5 Applications

This constraint system has given rise to numerous applications including graph coloring, scheduling, cutting stock, microcode labelling, warehouse location, car sequencing, planning and assignment problems to name a few (see for instance [DSVH88b, DSVH88c, DSVH88a, DSVH90, VH89b, VH89a]). For most of these applications, the computation time is comparable in efficiency with special-purpose programs (based on the same approach).

3.3.6 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 shorten the development time of programs substantially while preserving the efficiency of specialized tools for many applications.

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 [VH89b] 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 [VH89b].

4 Ask and Tell Languages

Tell 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 *tell* 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 and constraint entailment [Sar89]. Constraint entailment amounts to finding out if a basic constraint c is entailed by a constraint σ, i.e.

$$\Delta_c \models (\forall)(\sigma \rightarrow c).$$

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 device specialized constraint-entailment algorithms instead of using constraint-solving algorithms.

Constraint entailment was introduced in the context of concurrent logic programming by Maher [Mah87] 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. [CMC79, CKVC83, DL84, GL82, Nai85]). The concept of *ask* and *tell* languages was introduced by Saraswat [Sar89], in the context of concurrent constraint programming. In concurrent constraint programming, constraint entailment is used to synchronize concurrently executing agents. Constraint entailment was also used in CHIP (see [DVHS+88, GVHPZ89, GVHPZ90]) 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 [Sar89] in the context of concurrent logic programming. The second example is the cardinality operator proposed explicitly for CLP languages [VHD90a].

4.1 The Implication Operator

4.1.1 Syntax and Semantics

Syntax The implication operator has the form $c \Rightarrow A$ where c is a basic constraint and A is an atom[8]. We generalize the syntax of *tell* languages by allowing the implication operator in addition to atoms in the body of clauses.

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

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

Operational Semantics 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 accumulated constraints. In other words, if c is entailed by the accumulated constraints, $c \Rightarrow A$ reduces to A. If $\neg c$ is entailed by the accumulated constraints, $c \Rightarrow A$ reduces to *true*. Otherwise, the computation flounders, waiting for more information.

We now describe the semantics of an *ask* and *tell* language including the implication operator. 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.

Goal Reduction: A goal can be reduced to the body of a clause if the accumulated constraints are consistent with the new constraints.

$$\frac{p(u_1,\ldots,u_n) \leftarrow \sigma' \diamond B_1 \wedge \ldots \wedge B_m \in P \quad \Delta_c \models (\exists)(\sigma \wedge \sigma' \wedge t_1 = u_1 \wedge \ldots \wedge t_n = u_n)}{\langle p(t_1,\ldots,t_n)\ ,\ \sigma\rangle \longmapsto \langle B_1 \wedge \ldots \wedge B_m\ ,\ \sigma \wedge \sigma' \wedge t_1 = u_1 \wedge \ldots \wedge t_n = u_n\rangle}$$

Implication: An implication $c \Rightarrow A$ never fails. If c is entailed by the accumulated constraints, it reduces to the body A. If $\neg c$ is entailed by the accumulated constraints, the implication terminates successfully. Otherwise, the implication flounders.

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

$$\frac{\mathcal{D} \models (\forall)(\sigma \rightarrow \neg c)}{\langle c \Rightarrow A\ ,\ \sigma\rangle \longmapsto \sigma}$$

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

Conjunction: If any of the goals in a conjunction can make a transition, the whole conjunction can make a transition as well and the accumulated constraints are updated ac-

cordingly. This is the traditional interleaving rule.

$$\frac{\langle G_1 , \sigma\rangle \longmapsto \langle G_1' , \sigma'\rangle}{\begin{array}{l}\langle G_1 \wedge G_2 , \sigma\rangle \longmapsto \langle G_1' \wedge G_2 , \sigma'\rangle \\ \langle G_2 \wedge G_1 , \sigma\rangle \longmapsto \langle G_2 \wedge G_1' , \sigma'\rangle\end{array}}$$

$$\frac{\langle G_1 , \sigma\rangle \longmapsto \sigma'}{\begin{array}{l}\langle G_1 \wedge G_2 , \sigma\rangle \longmapsto \langle G_2 , \sigma'\rangle \\ \langle G_2 \wedge G_1 , \sigma\rangle \longmapsto \langle G_2 , \sigma'\rangle\end{array}}$$

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.

$$\begin{array}{lcl} SS[P] & = & \{G \mid P \vdash G \stackrel{*}{\longmapsto} \sigma\} \\ DS[P] & = & \{G \mid G \text{ diverges in } P\} \\ FLS[P] & = & \{G \mid P \vdash G \stackrel{*}{\longmapsto} \mathit{flounder}\} \end{array}$$

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

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

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

$$RES[P,G] \quad = \quad \{\sigma \mid P \vdash G \stackrel{*}{\longmapsto} \sigma\}$$

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. See [SR90b] for a denotational semantics of *ask* and *tell* languages.

4.1.2 Example

We illustrate the use of the implication operator to model an and-gate with four signals [Sim89]. The underlying constraint system includes equations and disequations over first-order terms.

```
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,
X = dnot ⇒ (Z = X , Y = 1),
X = d ⇒ (Z = X , Y = 1),
Y = dnot ⇒ (Z = Y , X = 1),
Y = d ⇒ (Z = Y , X = 1).
```

The effect of the above modelling is to achieve local propagation [SS80, Dav84], i.e. inferring values for the unknown variables from the known variables. For instance, as soon as Z is 1, X and Y are also forced to be equal to 1.

The above example originates from the demon declarations of CHIP. The above modelling is best for applications such as diagnostics [SD87b] and test generation [Sim89] of digital circuits.

4.2 The Cardinality Operator

In this section, we present a second operator using constraint entailment: the cardinality operator [VHD90a]. 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 its own constraints (i.e. constraints that are not primitive 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.

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 basic constraints. The syntax of *tell* 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 $\#(l, u, [c_1, \ldots, c_n])$ in a structure $\mathcal{A}$ wrt valuation ϑ is true if the number of constraints c_i $(1 \leq i \leq n)$ assigned to true in $\mathcal{A}$ wrt ϑ 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.

Operational Semantics 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". It can be defined by the following transition rules.

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:

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

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 accumulated constraints:

$$\frac{\begin{array}{l} l \leq u \ \wedge \ l = n \\ \mathcal{D} \models (\exists) \, (\sigma \wedge c_1 \wedge \ldots \wedge c_n) \end{array}}{\langle \#(l, u, [c_1, \ldots, c_n]) \, , \, \sigma \rangle \longmapsto \sigma \wedge c_1 \wedge \ldots \wedge c_n}$$

Negative Satisfaction: A formula $\#(l, 0, [c_1, \ldots, c_n])$ with $l \leq 0$ can be satisfied only if the conjunction $\neg c_1 \wedge \ldots \wedge \neg c_n$ is consistent with the accumulated constraints:

$$\frac{\begin{array}{l} l \leq u \ \wedge \ u = 0 \\ \mathcal{D} \models (\exists) \, (\sigma \wedge \neg c_1 \wedge \ldots \wedge \neg c_n) \end{array}}{\langle \#(l, u, [c_1, \ldots, c_n]) \, , \, \sigma \rangle \longmapsto \sigma \wedge \neg c_1 \wedge \ldots \wedge \neg c_n}$$

The above three rules make up the basic cases for the cardinality operator. Two of them allow the inference 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 accumulated constraints, the cardinality formula can be simplified by dropping the constraint and decrementing the bounds.

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

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 accumulated constraints (i.e. c_i inconsistent with σ), the cardinality formula can be simplified by dropping the constraint:

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

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 accumulated constraints.

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

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

4.2.2 Example

We give two examples of the use of the cardinality operator. The first example is used to implement the **element** constraint which has been instrumental in the solving 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 accumulated constraints do 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,[`X_1`,...,`X_3`],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 accumulated constraints, the two other variables are constrained to be different from 4.

5 Applications

In the previous sections, we have presented a general framework for CLP languages and discuss 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.

5.1 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 [SND88] 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 abovementioned paper.

5.1.1 Circuit Representation

Representing hardware in logic programming has been studied in various papers (e.g. [Clo87]). Logic programming provides a simple, yet convenient, hardware description language sup-

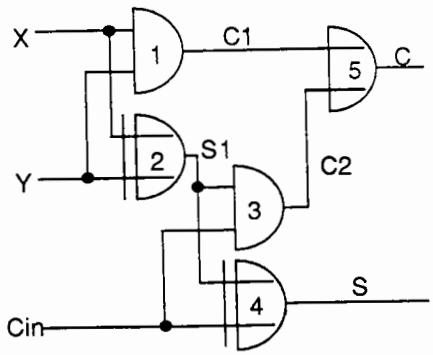

Figure 4: A Full Adder

porting, 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 4, can be easily described as follows.

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

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 makes sure to associate 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 + y + z
C = xy + xz + yz.
```

Suppose now that we would like to build a four-bit adder as depicted in Figure 5. Four full-adders can be combined to obtain the desired circuit:

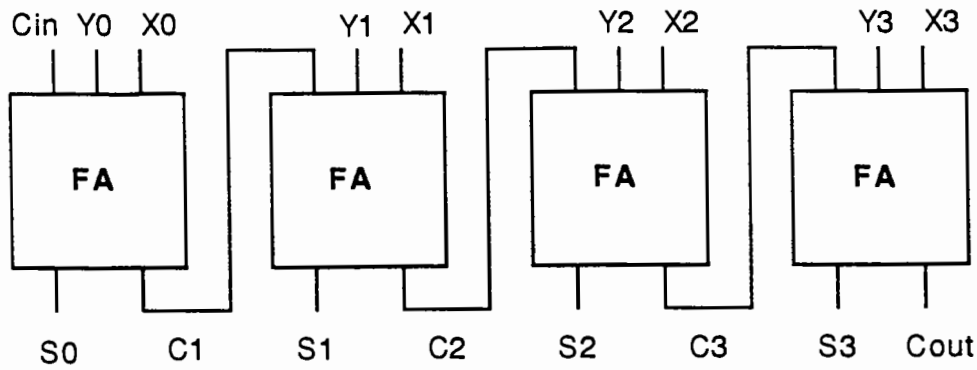

Figure 5: A Four-bit Adder

```
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).
```

5.1.2 Formal Verification

Formal verification, as mentioned, amounts to verify 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,[xo,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.

5.1.3 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 [Bry86] 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 has also limitations: multipliers cannot be verified efficiently but other techniques can be used.

5.2 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 [GVHPZ90] for more comprehensive presentation. In the following, we try to convey some of the ideas and techniques used in this application.

5.2.1 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 blowings 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.

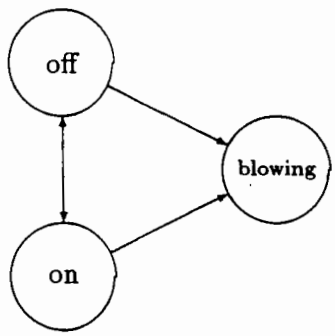

Figure 6: Transition graph for a light bulb

5.2.2 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 Figures 6 and 7. The 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:

```
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 7: State definitions for a light bulb

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.

5.2.3 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;

```
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 8: A meta program for the scheduler

- 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 8. 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.

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 transitions.

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 the possible states. There is no difficulty in generalizing the above

```
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 9: A CLP Representation for the Light Bulb

program for that purpose.

5.2.4 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 [HMS87]. Using this representation, the light bulb can be implemented as depicted in Figure 9.

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 accumulated constraints 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

```
light_bulb(Name, State, [R, Ith, Im], [I1, U1, I2, U2]) ←
   I1 + I2 = 0,
   U1 - U2 = R * I1,
   absolute_value(I1, I1abs),

   I1abs < Ith  ⇒  state = off,
   I1abs ≥ Ith  ⇒  state ≠ off,
   I1abs < Im ∧ I1abs ≥ Ith  ⇒  state = on,
   I1abs ≥ Im ∧ I1abs ≥ Ith  ⇒  state = blowing.
```

Figure 10: Another Representation of the Light Bulb

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 10.

Note that the circuit cannot find a consistent state by its own with the above representation. It should be executed in conjunction with a generator of values for the states. There is no difficulty in generalizing the scheduler for that purpose.

5.2.5 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.

5.3 The Car Sequencing Problem

We now present a problem using finite domains. The problem was motivated by an article published in AI Expert [Par88] 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 [DSVH88c]. The presentation here essentially

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

follows that paper although it has been revised to accommodate refinments to the CLP framework.

5.3.1 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 associated 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. Table 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*.

A solution to the above problem is the sequencing

<1,2,6,3,5,4,4,5,3,6>.

A sequencing which is not a solution is given by

<1,2,6,3,3,5,4,4,5,6>.

because the subsequence <6,3,3> does not satisfy the capacity constraint for option 2.

5.3.2 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 $nc = \sum_{i=1}^{n} n_i$. 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 represents 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 $O_i^j \in \{0, 1\}$.

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

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

have to be generated. In the above example, the capacity constraints for option 3 are as follows:

$O_1^3 + O_2^3 + O_3^3 \leq 1,$
$O_2^3 + O_3^3 + O_4^3 \leq 1,$
$\ldots$
$O_8^3 + O_9^3 + O_{10}^3 \leq 1.$

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 constraint

$$atmost(2, [S_1, \ldots, S_{10}], 3).$$

is generated for class 3.

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 first slot and option 5 is enforced by the constraint

$$element(S_1, [0, 0, 1, 0, 0, 0], O_1^5).$$

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

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 contains r cars so that the other slots should contain all the remaining cars having that option. If p cars requires option j, we may generate a constraint

$$O_1^j + \ldots + O_{nc-s}^j \geq p - r.$$

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

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

may be generated.

5.3.3 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 [DSVH88c]. The main result is that the program can sequence problems involving several hundreds 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 Conclusion

This survey has attempted to convey some of the most important results emerging from research in CLP. We have reviewed the syntax and the declarative and operational semantics of the two main classes of CLP languages: *tell* 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. Several areas of CLP research have been left out and include

- extension to the framework: for instance, meta-constraint programming [HMSY89, LS90], constraint hierarchies [BMMW89], dynamic constraint solving [MS89, VH90], and forward rules [Gra89];
- implementation [VH89c], transformation, and analysis of CLP programs [SH90, MS90];

- complexity analysis of CLP programs [KKR90, CMT90];
- relation to other LP languages such as [Har90].

Acknowledgments

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, and P. Stuckey 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

[AkN86] H. Ait-kaci and R. Nasr. LOGIN: A Logic Programming Language with Built-in Inheritance. *Journal of Logic Programming*, 3(3):185–215, October 1986.

[AkP90] H. Ait-kaci and A. Podelski. Is There a Meaning to LIFE. 1990. Submitted for publication.

[ASS+88] A. Aiba, K. Sakai, Y. Sato, D.J. Hawley, and R. Hasegawa. Constraint Logic Programming CAL. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, Tokyo, Japan, December 1988.

[Ber88] F. Berthier. Using CHIP to Support Decision Making. In *Actes du Seminaire 1988 - Programmation en Logique*, Tregastel, France, May 1988. CNET.

[BMMW89] A. Borning, M. Maher, A. Martingale, and M. Wilson. Constraint Hierarchies and Logic Programming. In *Sixth International Conference on Logic Programming*, Lisbon, Portugal, June 1989.

[Bry86] R.E. Bryant. Graph Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[BS87] W. Buttner and H. Simonis. Embedding Boolean Expressions into Logic Programming. *Journal of Symbolic Computation*, 4:191–205, October 1987.

[Buc85] B. Buchberger. *Groebner Bases: An Algorithmic Method in Polynomial Ideal Theory*, pages 184–232. N.K. Bose Ed., D. Reidel Publishing Co., 1985.

[BY85] J.A. Brzozowski and M. Yoeli. Combinational Static CMOS Networks. Technical Report CS-85-42, University of Waterloo, December 1985.

[CKPR73] A. Colmerauer, H. Kanoui, R. Pasero, and P. Roussel. Un système de communication homme-machine en français. Rapport de recherche, Groupe Intelligence Artificielle, Université d'Aix-Marseille II, 1973.

[CKVC83] A. Colmerauer, H. Kanoui, and M. Van Caneghem. Prolog, Bases Theoriques et Developpements Actuels. *T.S.I. (Techniques et Sciences Informatiques)*, 2(4):271–311, 1983.

[Clo87] W.F. Clocksin. Logic Programming and Digital Circuit Analysis. *Journal of Logic Programming*, 4(1):59–82, 1987.

[CMC79] K.L. Clark and F. Mc Cabe. The Control Facilities of IC-PROLOG. In D. Mitchie, editor, *Expert systems in the micro electronic Age*, pages 122–149. Edinburgh University Press, 1979.

[CMT90] J Cox, K. McAloon, and C. Tretkoff. Computational Complexity and Constraint Logic Programming Languages. In *Proceedings of the North-American Conference on Logic Programming (NACLP-90)*, Austin, Tx, October 1990.

[Col90] A. Colmerauer. An Introduction to Prolog III. *CACM*, 28(4):412–418, 1990.

[Dav84] R. Davis. Diagnostic Reasoning Based on Structure and Behavior. *Artificial Intelligence*, 24:347–410, 1984.

[DL84] M. Dincbas and J-P. Lepape. Metacontrol of Logic Programs in METALOG. In *Proceedings of the International Conference on Fifth Generation Computer Systems (FGCS'84)*, pages 361–370, Tokyo, Japan, November 1984. ICOT.

[DOW55] G.B. Dantzig, A. Orden, and P. Wolfe. The Generalized Simplex Method for Minimizing a Linear Form under Linear Inequality Constraints. *Pacific J. Math.*, 5(2):183–195, 1955.

[DP60] M. Davis and H. Putman. A Computation Procedure for Quantification Theory. *Journal of ACM*, 7:201–215, 1960.

[DSVH88a] M. Dincbas, H. Simonis, and P. Van Hentenryck. Solving a Cutting-Stock Problem in Constraint Logic Programming. In *Fifth International Conference on Logic Programming*, Seattle, WA, August 1988.

[DSVH88b] M. Dincbas, H. Simonis, and P. Van Hentenryck. Solving Large Scheduling Problems in Logic Programming. In *EURO-TIMS Joint International Conference on Operations Research and Management Science*, Paris, France, July 1988.

[DSVH88c] M. Dincbas, H. Simonis, and P. Van Hentenryck. Solving the Car Sequencing Problem in Constraint Logic Programming. In *European Conference on Artificial Intelligence (ECAI-88)*, Munich, W. Germany, August 1988.

[DSVH90] M. Dincbas, H. Simonis, and P. Van Hentenryck. Solving Large Combinatorial Problems in Logic Programming. *Journal of Logic Programming*, 8(1-2):75–93, 1990.

[DVH90] Y. Deville and P. Van Hentenryck. An Efficient Arc-Consistency Algorithm for a Class of CSP Problems. Technical Report CS-90-36, CS Department, Brown University, 1990.

[DVHS+88] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The Constraint Logic Programming Language CHIP. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, Tokyo, Japan, December 1988.

[Fik68] R.E. Fikes. *A Heuristic Program for Solving Problems Stated as Nondeterministic Procedures.* PhD thesis, Comput. Sci. Dept., Carnegie-Mellon Univ. Pittsburgh, PA, 1968.

[GKMP90] M.M. Gorlick, C.F. Kesselman, D.A. Marotta, and D.S. Parker. Mockingbird: a Logical Methodology for Testing. *Journal of Logic Programming*, 8(1-2):95–119, 1990.

[GL82] H. Gallaire and C. Lasserre. *Metalevel Control for Logic Programs*, volume Logic Programming, pages 173–185. Academic Press, 1982.

[Gra87] T. Graf. Extending Constraint Handling in Logic Programming to Rational Arithmetic. Internal Report, ECRC, Munich, Septembre 1987.

[Gra89] T. Graf. *Raisonner sur les contraintes en programmation logique.* PhD thesis, Universite de Nice (France), 1989.

[GVHPZ89] T. Graf, P. Van Hentenryck, C. Pradelles, and L. Zimmer. Simulation of Hybrid Circuits in Constraint Logic Programming. In *International Joint Conference on Artificial Intelligence*, Detroit, Michigan, August 1989.

[GVHPZ90] T. Graf, P. Van Hentenryck, C. Pradelles, and L. Zimmer. Simulation of Hybrid Circuits in Constraint Logic Programming. *Computers and Mathematics with Applications*, 20(9/10):45–56, 1990.

[Har90] S. Haridi. A Logic Programming Language based on the Andorra Model. *New Generation Computation*, 1990. To Appear.

[HMS87] N.C. Heintze, S. Michaylov, and P.J. Stuckey. CLP($\Re$) and Some Electrical Engineering Problems. In *Fourth International Conference on Logic Programming*, Melbourne, Australia, May 1987.

[HMSY89] N. Heintze, S. Michaylov, P. Stuckey, and R. Yap. On Meta-Programming in CLP($\Re$). In *Proceedings of the North-American Conference on Logic Programming (NACLP-89)*, pages 52–68, Cleveland, Ohio, October 1989.

[Imb90] J.L. Imbert. About Redundant Inequalities Generated by Fourier's Algorithm. In *AIMSA'90, Fourth International Conference on Artificial Intelligence: Methodology, Systems, Applications*, Albena-Varna, Bulgaria, September 1990.

[JL87] J. Jaffar and J-L. Lassez. Constraint Logic Programming. In *POPL-87*, Munich, FRG, January 1987.

[JM87] J. Jaffar and S. Michaylov. Methodology and Implementation of a CLP System. In *Fourth International Conference on Logic Programming*, Melbourne, Australia, May 1987.

[JM90] N. Jorgensen and K. Marriot. Useful Compile-Time Optimizations for CLP($\Re$), 1990.

[Kar84] N. Karmarkar. A New Polynomial-Time Algorithm for Linear Programming. *Combinatorica*, 4(4):373–395, 1984.

[Kha79] L.G. Khachian. A Polynomial Algorithm in Linear Programming. *Soviet Math. Dokl.*, 20(1):191–194, 1979.

[KKR90] P.C. Kanellakis, G.M. Kuper, and P.Z. Revesz. Constraint Query Languages. In *PODS-90*, Nashville, Te, 1990.

[Kow74] R. Kowalski. Predicate Logic as Programming Language. In North Holland, editor, *Proceedings of the IFIP Congress 74*, pages 569–574, 1974.

[Lau78] J-L. Lauriere. A Language and a Program for Stating and Solving Combinatorial Problems. *Artificial Intelligence*, 10(1):29–127, 1978.

[LHM89] J-L. Lassez, T. Huynh, and K. McAloon. Simplification and Elimination of Redundant Arithmetic Constraints. In *Proceedings of the North-American Conference on Logic Programming (NACLP-89)*, Cleveland, Ohio, October 1989.

[Llo84] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, New York, 1984.

[LMA88] J.L. Lassez and K. Mc Aloon. Applications of a Canonical Form for Generalized Linear Constraints. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, Tokyo, Japon, December 1988.

[LMY87] C. Lassez, K. McAloon, and R. Yap. Constraint Logic Programming and Option Trading. *IEEE Expert*, 2(3):42–50, Fall 1987.

[LS90] P. Lim and P. Stuckey. Meta Programming as Constraint Programming. In *Proceedings of the North-American Conference on Logic Programming (NACLP-90)*, Austin, Tx, October 1990.

[Mac77] A.K. Mackworth. Consistency in Networks of Relations. *AI Journal*, 8(1):99–118, 1977.

[Mah87] M.J. Maher. Logic Semantics for a Class of Committed-Choice Programs. In *Fourth International Conference on Logic Programming*, pages 858–876, Melbourne, Australia, May 1987.

[MH86] R. Mohr and T.C. Henderson. Arc and Path Consistency Revisited. *Artificial Intelligence*, 28:225–233, 1986.

[MN86] U. Martin and T. Nipkow. Unification in Boolean Rings. In *Proceedings of the 8th Conference on Automated Deduction*, pages 506–513, Oxford, England, July 1986.

[MS89] M. Maher and P. Stuckey. Expanding Query Power in Constraint Logic Programming Languages. In *Proceedings of the North-American Conference on Logic Programming (NACLP-89)*, Cleveland, Ohio, October 1989.

[MS90] K. Marriot and H. Sondergaard. Analysis of Constraint Logic Programs. In *Proceedings of the North-American Conference on Logic Programming (NACLP-90)*, Austin, Tx, October 1990.

[Nai85] L. Naish. *Negation and Control in Prolog.* PhD thesis, University of Melbourne, Australia, 1985.

[OV90] W. Older and A. Vellino. Extending Prolog with Constraint Arithmetics on Real Intervals. In *Canadian Conference on Computer & Electrical Engineering*, Ottawa, 1990.

[Par88] B.D. Parrello. CAR WARS: The (Almost) Birth of an Expert System. *AI Expert*, 3(1):60–64, January 1988.

[Plo81] G.D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, CS Department, University of Aarhus, 1981.

[PR88] R.G. Parker and R.L. Rardin. *Discrete Optimization.* Academic Press, London (England), 1988.

[Rad74] S. Radeanu. *Boolean Functions and Equations.* North-Holland, Amsterdam, 1974.

[Rob65] J.A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *Journal of ACM*, 12(1):23–41, January 1965.

[Sar89] V.A. Saraswat. *Concurrent Constraint Programming Languages.* PhD thesis, Carnegie-Mellon University, 1989.

[SD87a] H. Simonis and M. Dincbas. Using an Extended Prolog for Digital Circuit Design. In *IEEE International Workshop on AI Applications to CAD Systems for Electronics*, pages 165–188, Munich, RFA, October 1987.

[SD87b] H. Simonis and M. Dincbas. Using Logic Programming for Fault Diagnosis in Digital Circuits. In *German Workshop on Artificial Intelligence (GWAI-87)*, pages 139–148, Geseke, RFA, Septembre 1987.

[SH90] D.A. Smith and T.J. Hickey. Partial Evaluation of a CLP language. In *Proceedings of the North-American Conference on Logic Programming (NACLP-90)*, Austin, Tx, October 1990.

[Sha90] E. Shapiro. The Family of Concurrent Logic Programming Languages. *Computing Surveys*, 21(3):413–510, 1990.

[Sim89] H. Simonis. Test Generation using the Constraint Logic Programming Language CHIP. In *Sixth International Conference on Logic Programming*, Lisbon, Portugal, June 1989.

[SKL90] V.A. Saraswat, K. Kahn, and J. Levy. Janus: A Step Towards Distributed Constraint Programming. In *Proceedings of the North-American Conference on Logic Programming (NACLP-90)*, Austin, Tx, October 1990.

[SND88] H. Simonis, H.N. Nguyen, and M. Dincbas. Verification of Digital Circuits Using CHIP. In *Proceedings of the IFIP WG 10.2 International Working Conference on the Fusion of Hardware Design and Verification*, Glasgow, Scotland, July 1988.

[SR90a] V.A. Saraswat and M. Rinard. Concurrent Constraint Programming. In *Proceedings of Seventeenth ACM Symposium on Principles of Programming Languages*, San Francisco, CA, January 1990.

[SR90b] V.A. Saraswat and M. Rinard. Concurrent Constraint Programming. In *Proceedings of Eighth ACM Symposium on Principles of Programming Languages*, San Francisco, CA, January 1990.

[SS80] G.J. Sussman and G.L. Steele. CONSTRAINTS-A Language for Expressing Almost-Hierarchical Descriptions. *AI Journal*, 14(1), 1980.

[Stu90] P.J. Stuckey. Incremental Linear Arithmetic Constraint Solving and Detection of Implicit Equalities. Submitted for publication, 1990.

[VH89a] P. Van Hentenryck. A Logic Language for Combinatorial Optimization. *Annals of Operations Research*, 21:247–274, 1989.

[VH89b] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* Logic Programming Series, The MIT Press, Cambridge, MA, 1989.

[VH89c] P. Van Hentenryck. Parallel Constraint Satisfaction in Logic Programming: Preliminary Results of CHIP within PEPSys. In *Sixth International Conference on Logic Programming*, Lisbon, Portugal, June 1989.

[VH90] P. Van Hentenryck. Incremental Constraint Satisfaction in Logic Programming. In *Seventh International Conference on Logic Programming*, Jerusalem, Israel, June 1990.

[VHD90a] P. Van Hentenryck and Y. Deville. A new Logical Connective and its Application to Constraint Logic Programming. Technical Report CS-90-24, CS Department, Brown University, 1990.

[VHD90b] P. Van Hentenryck and Y. Deville. Operational Semantics of Constraint Logic Programming over Finite Domains. Technical Report CS-90-23, CS Department, Brown University, 1990.

[VHG90] P. Van Hentenryck and T. Graf. Standard Forms for Rational Linear Arithmetics in Constraint Logic Programming. In *International Symposium on Artificial Intelligence and Mathematics*, Fort Lauderdale, Florida, January 1990. Also ECRC internal Report IR-LP-2217.

[Vod88] P. Voda. The Constraint Language Trilogy: Semantics and Computations. Technical report, Complete Logic Systems, North Vancouver, BC, Canada, 1988.

[Wal89] C. Walinsky. CLP($\sum^*$). In *Sixth International Conference on Logic Programming*, Lisbon, Portugal, June 1989.

Contents