Conjunto: Constraint Logic Programming with Finite Set Domains

Carmen Gervet

Conjunto: Constraint Logic Programming with Finite Set Domains

Carmen Gervet
ECRC
Arabellastraße 17, D-81925 Munich, Germany
carmen@ecrc.de

Abstract

Combinatorial problems involving sets and relations are currently tackled by integer programming and expressed with vectors or matrices of 0-1 variables. This is efficient but not flexible and unnatural in problem formulation. Toward a natural programming of combinatorial problems based on sets, graphs or relations, we define a new CLP language with set constraints. This language Conjunto[1] aims at combining the declarative aspect of Prolog with the efficiency of constraint solving techniques. We propose to constrain a set variable to range over *finite set domains* specified by lower and upper bounds for set inclusion. Conjunto is based on the inclusion and disjointness constraints applied to set expressions which comprise the union, intersection and difference symbols. The main contribution herein is the constraint handler which performs constraint propagation by applying consistency techniques over set constraints.

1 Introduction

Various systems of set constraints have been defined for purposes such as axiomatizing a set theory in {log} [4], prototyping combinatorial problems with sets, multisets and sequences in CLPS[13], manipulating strings in CLP(Σ^*) [16], analyzing programs [7] [1] [2] among others. According to the objectives aimed at, each of these languages proposes a constraint solver for a class of set constraints over a computation domain.

The motivation for Conjunto originates from a desire to combine the efficient constraint satisfaction techniques with the declarative aspect of Prolog in order to solve combinatorial problems based on sets, relations or graphs (minimum node cover, minimum set cover, set partitioning, set packing,...). These NP-complete problems correspond to real life problems such as warehouse location, resources allocation, graph colouring, etc. They belong to the class of Operations Research (OR) problems and are currently solved using Integer Linear Programming (ILP) or constraint satisfaction techniques over 0-1 domain variables. A problem formulated in ILP is tightly coupled with the resolution technique and modifying some constraints or adding new

[1] Conjunto: [konrʔʊnto], Spanish for set.

ones would require a revision of the whole system. In Conjunto, like in any Constraint Satisfaction Problem language, the constraint solving is incremental and independent of the various constraints of the problem at hand. The current 0-1 formulation of combinatorial problems on sets or graphs is not natural and forces one to handle very large matrices with respect to the size of the problem. For example partitioning the set $\{a, b, c, d\}$ into a set of 3 subsets $P^* = \{S_1, S_2, S_3\}$ (without optimization criterion for simplicity reasons) would be defined in ILP by stating the following system of arithmetic constraints:

$x_{a1} + x_{a2} + x_{a3} = 1,\ \ x_{b1} + x_{b2} + x_{b3} = 1,$
$x_{c1} + x_{c2} + x_{c3} = 1,\ \ x_{d1} + x_{d2} + x_{d3} = 1,$
where $x_{ij} = 0..1$ (1 if $i \in S_j$, 0 otherwise).

Solving combinatorial problems aims at computing a feasible or optimized solution from a given finite search space. In particular, the set covering and set partitioning problems consist of computing a subset or a set of subsets from a given one. No order is required in the solution, and often the resolution would gain if the symmetries were avoided. Toward these aims, the use of set variables and set constraints looks ideal. To initialize the search space, we propose to attach a finite set domain to each set variable. A more natural and concise statement of the above problem is:

$S_1 \cap S_2 = \{\}$, $S_1 \cap S_3 = \{\}$, $S_2 \cap S_3 = \{\}$, $S_1 \cup S_2 \cup S_3 = \{a, b, c, d\}$,
$S_1, S_2, S_3 :: \{\}..\{a, b, c, d\}$. (i)
These domain constraints (i) initialize the set variable domains. They have the same semantics as : $\{\} \subseteq S_i \subseteq \{a, b, c, d\}$.

Two general questions arise from this comparison: can set constraints applied to set variables be expressive enough to describe combinatorial problems on sets and graphs ? Can consistency techniques be powerful enough to solve such problems ? In this paper we aim at answering a more specific question derived from the two previous ones: can we find a good tradeoff between the expressive power of set constraints and the efficiency of consistency techniques ?

In Conjunto, we propose to compute extensional finite sets using set constraints over finite *set domain* variables. A set domain attached to a set variable is specified by its greatest lower bound (glb) and least upper bound (lub). The inclusion and disjointness constraints over these domain variables are solved by applying consistency techniques which allow to perform deterministic computations until we reach a fixed point. This approach can be seen as an adaptation of finite domains [3][9] to finite set domains where the number of elements of the domain is no longer linear but exponential in the size of the upper bound and where the order relation is not total ($<$) but partial ($\subseteq$). As a consequence, checking the consistency of arithmetic constraints over each value of a domain would take a polynomial time in the domain size whereas for set constraints, this would lead to an exponential number of tests in the size of the largest domain upper bound. To avoid such inefficient computations, the consistency algorithms in Conjunto perform reasoning on the bounds of the set domains: the consistency is checked

over the lower and upper bounds of a set domain. This reasoning on bounds removes inconsistent values from a set domain by increasing the lower bound and/or reducing the upper one.

This article is organized as follows. Its first part presents the related work. The second part informally introduces the approach upon which Conjunto uses active constraint propagation on finite set domains. The third part is devoted to the language description. The operational semantics is presented in the fourth part and comparisons are made with finite integer domains languages. Some areas of applications using finite set domains are presented in the fifth part in particular a bin packing example is given. Directions for future works are mentioned in the conclusion.

2 Related Work

2.1 Sets in CLP

Several approaches have been tackled in the recent years to embed sets in the CLP framework.

CLP(Σ^*) [16] (string handling) represents an instance of the CLP scheme in the computation domain of regular sets which are finite sets composed from finite strings. CLP(Σ^*) constraints are of the form *A in (X."ab". Y)* which states that any string attached to variable A must contain the substring ab. A scheduling strategy for selecting constraints ensures termination of the satisfiability procedure.

{log} [4] is based on an axiomatized set theory where set terms are constructed using the interpreted functor *with*, e.g. $\emptyset$ *with* x *with* ($\emptyset$ *with* y *with* *z*) = {{z,y},x}. The satisfaction procedure of the complete solver is based on a non-deterministic selection of constraints by taking into account all the possible substitutions between the elements of two sets. This non-determinism leads, in the worst case, to a hidden exponential growth in the search tree (if $s_1 = s_2$ and $\#s_1 = n$, there are 2^n computable solutions). Nevertheless, {log} allows to define a very large class of sets such as hereditarily finite sets (sets of finite depth).

Example 2.1 For ease of presentation, we use $\{x, y\}$ for $\emptyset$ *with* x *with* y. Let a system of constraints be $s \subseteq \{3, 2, a\}$. To solve it in $\{log\}$, the set term s has been previously constructed. It could be the set $s = \{x_1, x_2, ..., x_k\}$. The variables x_i are not necessarily distinct. The constraint handler infers (using a built-in universal quantifier) the constraints: $\forall\, t \in \{x_1, x_2, ..., x_k\}$, $t \in \{3, 2, a\}$. An atomic equality between t and one element of $\{3, 2, a\}$ is derived in a non-deterministic way.

The CLPS language is founded on the set notion of sets of finite depth over Herbrand terms (simple sets are sets of depth one). The satisfaction of constraints is performed using consistency checking techniques [8] over

set elements defined as domain variables. Completeness is guaranteed at an exponential cost in the number of set elements by computing a tensorial product which builds the set of all the possible combinations linking elements of the various given domains: $a_1 \otimes ... \otimes a_n = \{\{x_1, ..., x_n\} \mid x_1 \in a_1, ...x_n \in a_n\}$. This language is already used for prototyping several problems.

Example 2.2 Let us consider the above mentioned example $s \subseteq \{3, 2, a\}$. s can be specified by $s = \{x_1, x_2, ..., x_k\}$ in CLPS, where the variables x_i are not necessarily distinct. The constraint handler generates an equivalent system of constraints : $depth(s) \leq depth(\{3, 2, a\})$, $\#s \leq \#\{3, 2, a\}$, $x_1 \in \{3, 2, a\}, x_2 \in \{3, 2, a\}$, ..., $x_k \in \{3, 2, a\}$, $s \in \{\{3\}, \{3, 2\}, \{3, a\}, \{2, a\}, \{3, 2, a\}\}$. The exhaustive set of possible instances of s is computed. As soon as the domain of one x_i is modified, the domain of s is recomputed.

2.2 Systems of set constraints

A related line of work is program analysis systems [7] [1] [2] among others. They handle a larger class of sets (infinite sets) than Conjunto, $\{log\}$ or CLPS. The set variables are introduced to model a program. The different resolution algorithms are based on transformation algorithms. These transformations preserve consistency either by computing a least model [7] which does not preserve all solutions or by computing a finite set of systems in solved form [1]. [2] demonstrated that the latter algorithm is solvable in non-deterministic exponential time.

3 Representing sets by lower and upper bounds

The previous section reflects that to embed sets in a CLP language, there are two alternatives: (i) to define a set constructor which allows to build a set term , (ii) to apply set constraints over set expressions. The first alternative founds expression in "lists" of variables $\{x_1, x_2, ..., x_n\}$ for the representation of a set term ($\{log\}$, CLPS). In this case, the non determinism of the set unification leads to compute a number of combinations which may grow exponentially with the largest number of set elements. The second alternative has been chosen by program analysis systems and CLP(Σ^*). In Conjunto, we propose a specific case of this alternative, by constraining a set variable to belong to a *finite set domain*. The notion of finite domain has first been used and defined in the constraint logic programming language CHIP [3]. Such a computation domain has proved its efficiency in the CLP framework by a powerful use of consistency checking techniques.

In Conjunto, a set is an extensional set which contains only Herbrand terms (no variables and no sets). We use the word *ground* to define it. A *finite set domain* (for set inclusion) is defined by a finite set of ground sets and specified by its greatest lower bound (glb)- intersection of the ground

sets - and its least upper bound (lub)- union of the ground sets. Elements of the lower bound are elements of the set variable whereas elements of the upper bound are possible elements of the set variable. For example $\{\{3,a\}..\{3,a,g(1),5\}\}$ defines a set domain with glb $= \{3,a\}$ and lub $= \{3,a,g(1),5\}$. The values in this domain are the sets $\{3,a\}$, $\{3,a,g(1)\}$, $\{3,a,5\}$, $\{3,a,g(1),5\}$.

Example 3.1 The example introduced in section 2 is handled in Conjunto as a single constraint $S \subseteq \{3,2,a\}$. The domain of S has been previously initialized and is reduced so as to satisfy this constraint.

Our motivation behind the set domain concept in a CLP framework is to combine a natural and flexible statement of combinatorial problems on sets (set partitioning, bin packing) and graphs with the efficiency of constraint satisfaction techniques. But from our experiment set constraints are not expressive enough to tackle the problems on graphs. In fact, our objectives are not limited to the definition of set domains but also aim at describing relations in the same way. Laurière first addressed this issue in his seminal language ALICE [12]. In Conjunto, a finite *relation domain* constrains a relation variable $\mathcal{R} \subseteq S_1 \times S_2$ where S_1 and S_2 are respectively the domain and the range of $\mathcal{R}$ (ground sets). A graph is a specific relation where the domain and range coincide. A forthcoming report will describe this extension to relations and graphs in more detail.

4 The Conjunto language

4.1 A set domain for set variables

Definition 4.1 A finite set domain D attached to a set variable S is the discrete lattice or powerset $\{S \in 2^{lub_s} \mid glb_s \subseteq S\}$ under inclusion specified by the notation $glb_s..lub_s$. glb_s and lub_s represent respectively the intersection and union of elements of D.

The indexical domain is defined using the functions (i) $d(X)$ for the domain of a set expression X, (ii) $lub(X)$ for the least upper bound of X, (iii) $glb(X)$ for the greatest lower bound of X. By definition $d(X) = glb(X)\,..\,lub(X)$.

4.2 The constraint domain

Domain of discourse The domain of discourse is $\mathcal{D} = \mathcal{FP}(HU) \cup HU$ where $\mathcal{FP}(HU)$ is the set of finite sets in 2^{HU} and HU the finite Herbrand Universe. Thus, values of $\mathcal{D}$ could be either simple Herbrand terms or ground sets. For example $s = \{a, b, f(a), V\}$ where $a, b, f(a)$ are Herbrand terms and V is a variable, is not an element of $\mathcal{D}$ because V is a variable and thus s is not ground.

Definition 4.2 A Conjunto language $\mathcal{L}$ consists of:

1. Binary predicate symbols: arithmetic constraint symbols ($=, \geq, \leq, \neq$) and set constraint symbols ($\subseteq, \in, \notin, \neq^0, ::$). The two latter predicates $\neq^0$ and :: are respectively interpreted as the disjointness constraint and the domain constraint.

2. Unary function symbols introduced previously: d, lub, glb of arity one.

3. Binary operator symbols: arithmetic ones ($+,-$) and set operators ($\cup$, $\cap$, $\setminus$, $\#$). The two latter are interpreted as a complementary difference ($S \setminus S' = \{x \mid x \in S, x \notin S'\}$) and the usual set cardinality operator.

4. Constants: C_s (ground sets), C_h (Herbrand terms) belonging to $\mathcal{D}$, the empty set $\emptyset$[2] and its complementary Top in $\mathcal{FP}(HU)$.

5. Variables and domain variables D_v taking their value in $\mathcal{FP}(HU)$ or HU.

Set expressions A set expression S of $\mathcal{L}$ where S_1 and S_2 are set expressions is given by the following abstract syntax: $S ::= C_s \mid D_v \mid S_1 \cup S_2 \mid S_1 \cap S_2 \mid S_1 \setminus S_2$.

A set expression is composed of set domain variables together with set operator symbols. The domain of a set expression is also a finite lattice under inclusion [6]. It could be represented by computing its exact bounds at an exponential cost in the size of the largest upper bound invoked. But for efficiency reasons, it is represented in Conjunto by approximating its bounds in terms of the domain bounds of the set variables. The following properties give the equivalences and/or implications which exist between the upper and lower bounds of a set expression domain and the upper and lower ones of the set variable domains invoked.

Properties 4.3

1. $glb(Y) \subseteq Y \subseteq lub(Y)$
2. $lub(Y \cup Z) = lub(Y) \cup lub(Z)$
3. $glb(Y \cap Z) = glb(Y) \cap glb(Z)$
4. $lub(Y \cap Z) \subseteq lub(Y) \cap lub(Z)$
5. $glb(Y \cup Z) \supseteq glb(Y) \cup glb(Z)$
6. $lub(Y \setminus Z) = lub(Y) \setminus glb(Z)$
7. $glb(Y \setminus Z) = glb(Y) \setminus glb(Z)$

Proof The proof of the first five properties is given in [15]. To demonstrate 6. and 7., assume X and Y are set variables whose domains are respectively specified by $glb(X)..lub(X)$, $glb(Y)..lub(Y)$.

6. $x \in lub(X \setminus Y)$ iff $\{x\} \cap X \setminus Y \neq \{\}$ iff $\{x\} \cap X \neq \{\} \wedge \{x\} \cap Y = \{\}$ iff $x \in lub(X) \wedge x \notin glb(Y)$. Thus $lub(X \setminus Y) = lub(X) \setminus glb(Y)$.

7. $x \in glb(X \setminus Y)$ iff $\{x\} \subseteq X \setminus Y$ iff $\{x\} \subseteq X \wedge \{x\} \not\subseteq glb(Y)$ iff $\{x\} \subseteq glb(X) \wedge \{x\} \not\subseteq glb(Y)$ iff $x \in glb(X) \setminus glb(Y)$. □

[2] $\emptyset$ is represented by the syntax $\{\}$ and interpreted as the empty set.

In [15] they use this notion of bounds as knowledge approximation on behalf of the whole knowledge base. In Conjunto, these properties constitute a very important issue on a constraint propagation viewpoint. It means that constraints over set expressions can be approximated in terms of constraints over set variables (see section 5) with a limited loss of the approximated set expression domain. The loss in the computation of the lower bound of the domain of a set expression $s \cup s_1$ is not very surprising for it is very close to the problem of handling disjunctions in Prolog.

4.2.1 Basic constraints

$$\sum = \sum_1 + \sum_2$$

$\sum$ denotes the *system of basic constraints* composed of set constraints and arithmetic constraints. $\sum_2$ is the set of basic arithmetic constraints defined in [9] ($\{ax = by + c, ax \neq c, ax \geq by + c, ax \leq by + c, x \in \{a_1, .., a_n\}\}$ where the a,b,c,a_1, a_n are positive integers and x,y are domain variables). We include $\sum_2$ in the system as the solver handles finite (integer) domains when dealing with the cardinality operator #. For reasons of space, its handling will not be presented here.

$\sum_1$ comprises set relation constraints and set domain constraints as defined below.

$$\sum_1 = (\sum_r, \sum_d)$$

In the following, let us denote a set domain variable (not a set expression) by S or S' and a ground set by s. $a, a_1, ..., a_m$ denote elements of the finite HU and X any variable taking its value in the finite Herbrand Universe. The semantics of set equality, membership, inclusion and disjointness is the usual one.

Set relation constraints Let $\sum_r = \{\ S \subseteq S', S \neq^0 S'\ \}$. The equality constraint is defined using the following rewriting rule:

$$S\, is^3\, S_1 \rightarrow S \subseteq S_1, S_1 \subseteq S$$

Set domain constraints Elements of $\sum_d$ are set domain constraints *i.e.*, of the form $b(S) :: glb(S)..lub(S)$. In other terms if $glb(S) = \{a_1, ..., a_l\}$ and $lub(S) = \{a_1, ..., a_q\}$ this constraint corresponds to $\{a_1, ..., a_l\} \subseteq S \subseteq \{a_1, ..., a_q\}$. Nevertheless the set domain constraint can not be replaced by the latter as the special handling of domains is fundamental to Conjunto's consistency techniques.

[3] avoids confusions with the arithmetic equality and fits the Conjunto implementation syntax.

4.2.2 n-ary constraints

Set expressions together with binary set predicate symbols are n-ary constraints. In other terms, if *cons* is any constraint predicate symbol in $\{\subseteq, \supseteq, \neq^0\}$, n-ary constraints are of the form: $S_{exp1}\ cons\ S_{exp2}$.

Remark The disjointness constraint ($\neq^0$) is equivalent from a semantical viewpoint to $S \cap S_1 = \{\}$, a specific case of the n-ary constraint $S \cap S_1 = S_2$ where $S_2 = \{\}$. As $S \neq^0 S_1$ is of much use in partitioning problems, it has been embedded in Conjunto as a basic set relation constraint. A local arc consistency algorithm has been implemented to solve it. Doing so, we convert a n-ary constraint into a set relation one and avoid the more general approximation process which is useless here and looses information about the domain bound computation (see properties 4.3).

4.2.3 Mixed computation domain constraints

This last class of constraints establishes links between variables from two computation domains: (i) the finite HU universe (ii) the finite $\mathcal{FP}$(HU) universe. It concerns the set of $\{X \in S, X \notin S\}$[4] constraints.

4.2.4 Admissible system of constraints

As an adaptation of [9], an *admissible system* of constraints is a system of constraints where every set constrained variable occurs in some set domain constraints. Set constraints are only considered in a given context (where domains are attached to the variables).

Having defined the foundations of Conjunto we need to define its operational semantics comprising the consistency algorithms.

5 Operational semantics for set constraints

Conjunto does not fit to the standard CLP scheme [11] as the operational semantics is based on the notions of postponing some constraints and propagating other constraints whose satisfiability is not always provable. The solver schedules in a data-driven way, the set constraints checked through consistency techniques.

5.1 Preliminary definitions

First let us consider a constraint graph G to represent a constraint satisfaction problem. The approach is the usual one, that is each node s_i of the

[4] In {log} [4] they do not need to distinguish these constraints from $\subseteq$ as they can write $x \in S \leftrightarrow \{x\} \subseteq S$. In Conjunto $\{x\}$ is not a term (if x is a variable) so we do need to define $\subseteq$ as a primitive constraint.

graph corresponds to a set domain variable; each directed arc (s_i, s_j) linking the variables s_i and s_j corresponds to a single constraint C_{ij}. Thus we assume for simplicity reasons that there is at most one constraint linking two variables in a given order of variables ($s_1 \subseteq s_2$ and $s_2 \subseteq s_1$ are two distinct constraints). This assumption simplifies the algorithm description but no restriction is actually imposed on real Conjunto programs.

The definitions of node, arc consistency [14] and solved form [9] are kept and recalled hereafter.

Definition 5.1 Let $c(S)$ be a unary constraint (*i.e*, with one set variable) such that a set domain $D_s = glb_s..lub_s$ is attached to S. $c(S)$ is *node consistent* iff $\forall v \in D_s$, $c(v)$ is true.

Definition 5.2 Let $c(S, S')$ be a binary constraint such that set domains $D_s = glb_s..lub_s$ and $D_{s'} = glb_{s'}..lub_{s'}$ are respectively attached to S and S'. $c(S, S')$ is *arc consistent* iff for all $v \in D_s$, there exists $w \in D_{s'}$ such that $c(v, w)$ is true.

Definition 5.3 A system of basic set constraints is in *solved form* iff every unary constraint is node consistent and every binary constraint is arc consistent.

5.2 Solved form computation

5.2.1 The internal set representation

Unlike for finite integer domains, the time complexity for operations on ground sets ($+$, $-$ versus $\cup$, $\cap$, $\setminus$) can not be considered as constant as it closely depends on the internal set representation. We made the choice to represent each domain bound with a sorted list where the time complexity for any set operation ($\cup$, $\cap$, $\setminus$) is upper bounded by $\mathcal{O}(2d)$ where d is $\#lub(s) + \#glb(s)$ and s the set with the largest domain. $\#$ is the cardinality operator. We have experimented another approach which consists of representing a set domain as a vector of 0-1 variables. This reduces the time complexity of the $\cup$ and $\cap$ operations to $\mathcal{O}(\#lub(s))$ where $lub(s)$ is the largest domain upper bound. But it leads to a much larger occupation of the memory space. In the following d will always stand for $\#lub(s) + \#glb(s)$.

5.2.2 Node consistency for basic set relation constraints

The node consistency checking for unary set constraints (set constraints with only one set variable) is quite straightforward. Let the algorithm be noted NC_{sets}. It performs the following tests: for each unary constraint C_i on the set variable s_i with domain D_i remove all the inconsistent sets from the domain D_i, by reducing its upper bound and/or by increasing its lower bound. More detail concerning the computations over the bounds are given in the general case of arc consistency (`arc_cons-`$\{\subseteq, \neq^0\}$).

For example, the system of constraints: $S :: \{a,3\}..\{a,3,7,f\}, S \subseteq \{a,f,3\}$ is node consistent iff the domain of S is reduced to $S :: \{a,3\}..\{a,3,f\}$.

Complexity issues Let n be the number of variables and d the sum of the largest upper bound and lower bound in the set domains. The time complexity for NC_{sets} is in the worst case $\mathcal{O}(2nd)$.

5.2.3 Arc consistency for basic set relation constraints

Recall the set of basic set relation constraints $\sum_b = \{ S \subseteq S', S \neq^0 S'\}$ where S and S' are set domain variables. The existing arc consistency algorithms can not be simply adapted to check consistency of set relation constraints over set domain variables. The reason is that these algorithms are based on a domain reasoning (except for AC-5 over arithmetic constraints [10]). That is, an arc (i,j) is consistent for each element of an integer domain D_i. This reasoning takes polynomial time in the length of the largest domain. In the case of set domains, this reasoning would lead to an exponential number of tests in the largest upper bound length. For efficiency reasons, this reasoning is replaced in Conjunto by a reasoning on the domain bounds.

As formally introduced in [10], the existing arc consistency algorithms manipulate a list or queue of elements to reconsider. Once a variable domain j has been modified, some constraints need to be checked again. In terms of arcs, this means that some arcs (i,j) need to be reconsidered.

In Conjunto we consider two queues: one noted Q_{glb} contains the arcs (s_i, s_j) for which the glb of s_j has been increased and requires to reconsider arcs (s_i, s_j). The definition of the second queue noted Q_{lub} is then straightforward. It contains the arcs (s_i, s_j) for which the lub of s_j has been modified and requires to reconsider the arcs (s_i, s_j). As a logical consequence, the first queue might contain constraints of the form $s_j \subseteq s_i$, $s_i \neq^0 s_j$, $s_j \neq^0 s_i$ and the second queue might contain constraints of the form $s_i \subseteq s_j$. Handling two queues avoids checking again arcs where a modification of the domain of s_j does not justify a need to reconsider arcs (s_i, s_j). This optimization goes in the same line as the one given in AC-5 where the objective is to take into account the semantics of constraints and thus to check again only the constraints for which a need is justified.

Now that the different queues have been defined, we need to give the specification of the consistency algorithm for the inclusion and the disjointness constraints `arc_cons-`$\{\subseteq, \neq^0\}$. This algorithm reconsiders an arc (s_i, s_j) according to the constraint predicate it represents. This arc comes from a given queue due to a reduction of the domain of s_j. It computes and returns the new domain D_i of s_i and a boolean value indicating a possible modification of D_i (1 if D_i has changed, 0 otherwise). In the following, the domain D_i will stand for the two sets $glb(s_i)$, $lub(s_i)$ and the arc (s_i, s_j) will be replaced

by its associated constraint predicate C_{ij}. As usual $C_{ij}(s, s')$ represents the constraint C_{ij} between the set values s and s' and denotes a boolean value.

procedure arc_cons-$\{\subseteq, \neq^0\}$(**in:**(C_{ij}),**inout:** D_i, **out:** CHANGED)
begin
  case C_{ij}**of**
  1. $s_i \subseteq s_j$
    if $\neg\, lub(s_i) \subseteq lub(s_j)$ **then**
      begin
      $lub(s_i) \leftarrow lub(s_i) \cap lub(s_j)$;
      CHANGED $= 1$;
      end
    else CHANGED $= 0$;
  2. $s_i \supseteq s_j$
    if $\neg\, glb(s_i) \supseteq glb(s_j)$ **then**
      begin
      $glb(s_i) \leftarrow glb(s_i) \cup glb(s_j)$;
      CHANGED $= 1$;
      end
    else CHANGED $= 0$;
  3. $s_i \neq^0 s_j$ or $s_j \neq^0 s_i$
    if $\neg\, lub(s_i) \neq^0 lub(s_j)$ **then**
      begin
      $lub(s_i) \leftarrow lub(s_i) \setminus glb(s_j)$;
      CHANGED $= 1$;
      end
    else CHANGED $= 0$;
end

Fig. 2 : local arc consistency for the inclusion and disjointness constraints

The two first cases differentiate the constraints where s_j appears in the right of left hand side of $\subseteq$.

Complexity issues. Like for the node consistency algorithm, the time complexity for this algorithm is closely linked to the set operation cost and is in $\mathcal{O}(2d)$.

Queue issues. Note first that the lower bound $glb(s_i)$ can only get modified in the $\supseteq$ case and second that the $lub(s_i)$ might get reconsidered either due to a $glb(s_j)$ modification in the $\neq^0$ case, or due to a $lub(s_j)$ modification in the $\subseteq$ case. Thus the constraint $s_i \subseteq s_j$ comes from the Q_{lub} queue and might imply (if D_i gets modified) to add constraints of the same kind in the same queue. The constraint $s_i \supseteq s_j$ comes from the Q_{glb} queue and could also require to add constraints of the same kind or disjointness ones to the same queue. On the other hand, the $s_i \neq^0 s_j$ constraint comes from the Q_{glb} queue and modifications of $lub(s_i)$ could lead to add $s_j \subseteq s_i$ constraints in the Q_{lub} queue.

In Conjunto the constraints linked with a variable s_j are stored in a list L_{s_j} initialized in the beginning of the resolution. Like in arc consistency algorithms for arithmetic constraints, the access to these constraints will not be taken into account in the time complexity results. So, to add constraints to one queue, requires to select from a given list specified by the string BOUND, the constraints for which the need to check them again is justified. The previous paragraph gave us some indications about the constraints to be selected given a modified domain bound. The following algorithm performs this selection. Q stands either for the Q_{glb} or Q_{lub} queue which is given as an input. Note that the $s_k \subseteq s_j$ constraints are the only ones which need to be reconsidered due to a modification of $lub(s_j)$.

procedure `addto_queue` (**in:** D_j, BOUND **inout:** Q)
begin
  if BOUND = "lub" **then** $Q = Q \cup \{\forall k,\, s_k \subseteq s_j \in L_{s_j}\}$
  else if BOUND = "glb" **then** $Q = Q \cup \{\forall k,\, C_{kj} \in L_{s_j} \mid C_{kj} \neq s_k \subseteq s_j\}$
end

The next step consists of initializing these queues. To do so, we need to perform once the above defined local arc consistency algorithm upon each arc (s_i, s_j) of the graph G. The algorithm `arc_cons` considers once each arc and stores it (in case of a domain modification) in the right queue according to the constraint considered. It returns the two queues Q_{glb} and Q_{lub}. We keep the previous notation for $(s_i,\, s_j)$ that is C_{ij}.

procedure `arc_cons`(**in:** G **out:** Q_{glb}, Q_{lub})
begin
  Q_{glb}, $Q_{lub} \leftarrow \{\}$;
  for each $C_{ij} \in$ G **do**
    begin
      `arc_cons-`$\{\subseteq, \neq^0\}$ (C_{ij} , D_i, CHANGED);
      if CHANGED **then**
        if $C_{ij} = s_i \supseteq s_j$ **then** `addto_queue` (D_i, "glb", Q_{glb})
        else `addto_queue` (D_i, "lub", Q_{lub})
    end
end

Let e be the number of directed arcs or constraints and d the number introduced above. The time complexity of `arc_cons` is in $O(2ed)$.

We now give the complete algorithm for arc consistency checking of set relation constraints over finite set domains. It performs deterministic computations to reach a consistent system. The algorithm first applies the node consistency algorithm to check the unary constraints. Then it performs the `arc_cons` algorithm once to initialize the queues by performing a first propagation on each arc of the graph G. The two next interwoven loops aim at reaching the solved form. One iteration of the largest loop consists of emptying the Q_{lub} loop and then of checking one arc C_{ij} from the Q_{glb} queue. This checking might lead to add arcs to any of the two queues (to the Q_{lub} one in case of $C_{ij} = s_i \neq^0 s_j$, to Q_{glb} otherwise).

```
begin AC_sets
    for i ← until n do
        NC_sets(i);
    arc_cons(G, Q_glb, Q_lub) ;

    while Q_glb not empty do
        begin
            while Q_lub not empty do
                begin
                    select and remove C_ij from Q_lub;
                    arc_cons-{⊆, ≠^0} (C_ij, D_i, CHANGED) ;
                    if CHANGED then addto_queue (D_i, "lub", Q_lub) ;
                end

            select and remove C_ij from Q_glb;
            arc_cons-{⊆, ≠^0} (C_ij, D_i, CHANGED);
            if CHANGED then
                if C_ij = s_i ⊇ s_j then addto_queue (D_i, "glb", Q_glb)
                else addto_queue (D_i, "lub", Q_lub) ;
        end
end
```

Fig. 5 AC_{sets}: the new arc consistency algorithm

Theorem 5.4 AC_{sets} is correct and terminates.

Proof (correctness) First, each set s removed from a domain D_i can not belong to any arc consistent solution: a set is removed if it does not satisfy a local consistency (cf. `arc_cons-`$\{\subseteq, \neq^0\}$). Furthermore it can never be added further on. So by a continuous reduction of D_i all the sets removed can never belong to any solution, so s in particular can not belong to any solution. Second, AC_{sets} is totally arc consistent, that is for all arcs (s_i, s_j), for all sets s in D_i there is one set s' in D_j such that $C_{ij}(s, s')$: the continuity of the set inclusion and disjointness predicate symbols assure that if the domain bounds are sound values then any set value in the domain is also sound. The two points demonstrated guarantee that AC_{sets} builds the largest arc consistent solution: all the reduced domains do not contain a set which might lead to an inconsistent solution and thus should be removed. □

Proof (termination and complexity) The size of the set domains can only get continuously reduced (see operations on the domains in `arc_cons-`$\{\subseteq, \neq^0\}$). Once a variable domain is reduced to one single set no constraint containing this variable is added to any queue. So if d' is the largest value of $\#lub(s) - \#glb(s)$ a constraint could be checked at most d' times. Termination is thus guaranteed for each loop.

Now, let l be the size of Q_{lub} and $e - l$ the one of Q_{glb}. The cost of `arc_cons-`$\{\subseteq, \neq^0\}$ is d for one constraint (d being the largest $\#lub(s) +$

$\#glb(s)$). So for one constraint AC_{sets} could be iterated d' times till the constraint is solved. If only one queue was handled, e constraints would be reconsidered in the worst case. So the time complexity would be $\mathcal{O}(edd')$. In the case of Conjunto all the constraints are not reconsidered each time a modification occurs. If the constraints to be checked again only belong to the Q_{lub} queue, the time complexity would be $\mathcal{O}((e-l)d + ldd')$. If they would only belong to the Q_{glb} queue, the time complexity would be $\mathcal{O}(ld + (e-l)dd')$. Assuming that $max(l,\, e-l) = e-l$, the upper bound time complexity of AC_{sets} is $\mathcal{O}(ld + (e-l)dd')$. The gain versus a single handled list is $ld - ldd'$.□

5.2.4 Partial lookahead for n-ary constraints

Recall the class of n-ary constraints: S_{exp1} *cons* S_{exp2} where S_{expi} are set expressions and *cons* is any constraint predicate symbol in $\{\subseteq, \supseteq, \neq^0\}$.

These constraints are handled efficiently by a reasoning about variation lattice bounds just like in AC_{sets} using the properties 4.3 of the functions $glb(s)$, $lub(s)$ for set expressions s. Thus constraints over set expressions are approximated in terms of constraints over set variables. Let a n-ary constraint be $S_{exp1} = S_{exp2}$ for example. Let the set expression S_{exp1} range over $glb_1 .. lub_1$ and S_{exp2} over $glb_2 .. lub_2$. For the equation to be satisfied, the two terms must range over $glb .. lub$ where glb is the maximum of glb_1 and glb_2 and lub is the minimum of lub_1 and lub_2. In the following, we use the term *PLH-solved form* for this class of constraints upon which a new partial lookahead algorithm is applied.

Note that for efficiency reasons, very nested constraints such as $S_1 \cap ... \cap S_n = \{\}$ are split into more simple ones like $S_1 \cap S_2 = S'$, $S' \cap S_3 = S''$, ..., $S^k \cap S_n = \{\}$. This process might avoid awakening the initial constraint which involves a large amount of set domain variables (if $lub(S_1)$ gets modified but the bounds of S' remain the same). This approach prefers efficiency gain over memory loss. It has been used in the bin packing application.

5.2.5 Forward checking for mixed computation domain constraints

The constraints $\{X \in S,\, X \notin S\}$ are currently handled in Conjunto using forward checking over the integer variables X, that is the constraints are postponed until X is ground. If X becomes a ground term a, the constraints are rewritten into:

$$a \in S \rightarrow \{a\} \subseteq S \qquad a \notin S \rightarrow \{a\} \neq^0 S$$

Now that the consistency algorithms for set relation constraints and n-ary constraints have be defined, we need to define the solver which uses these algorithms to transform a system of set constraints into a consistent system.

5.3 The constraint solver

The constraint solver of Conjunto transforms a system of set relation constraints into a system in solved form and a system of n-ary constraints into a system in PLH-solved form. The membership and nonmembership constraints are delayed (flounder notion in [9]) until they become unary constraints.

Algorithm The solver acts in a data driven way using a relation between *states*. A state of the program is the system of constraints or pair $S =< SC, DC >$ where SC is a set of set relation constraints and n-ary constraints and where DC is a set of set domain constraints. First apply the node consistency algorithm NC_{sets} to the unary constraints to obtain the state $< SC, DC' >$, then depending on the constraints apply the arc consistency algorithm AC_{sets} or the partial lookahead algorithm to obtain the state $S' =< SC, DC'' >$.

Theorem 5.5 A system $< SC, DC >$ in solved form (*i.e.*, containing only set relation constraints resulting from AC_{sets}) is satisfiable if the set of constraints DC is satisfiable.

Proof First, if any set domain is unsound ($glb(s) \geq lub(s)$) then the system is clearly unsatisfiable. Second, if the set of set relation constraints is not empty, it is always possible to find a solution by computing the least model. In fact, the continuity of the inclusion and disjointness predicates guarantees that any set value within the respective domains leads to a solution. So assigning to each of the domain variable the respective lower bound of their domain leads to a solution. □

Expressive power It is worth noting that in a system in PLH-solved form which contains some union together with some intersection operators, the set domain bounds of a set variable can be locally consistent but not globally.

The reason is that the union operator does not preserve the lower bound computation (see property 4.3.5) of a set expression domain whereas the intersection does not preserve the upper one (property 4.3.4). So global consistency is not provable for systems of set constraints comprising union and intersection operators unless the solver performs exhaustive computations at an exponential cost in the largest upper bound among the set domain.

Example 5.6
The following set of constraints:
$S_1 :: \{\}..\{1, 2, a, b\},\ S_2 :: \{\}..\{1, 2, a, b\},\ S_3 :: \{\}..\{1, 2, a, b\},$
$S_1 \cup S_2 \cup S_3 = \{1, 2, a, b\},\ S_1 \cap S_2 \cap S_3 = \{\}.$
is in PLH-solved form but not globally consistent. Assigning respectively to each set variable the lower bound of its domain (or the upper one) does not lead to a solution.

6 Application

The following example of bin packing illustrates how constraint propagation acts actively over set constraints and is sufficient to solve such problems, and how set domains bring expressiveness and conciseness to the program.

Problem description Bin packing problems belong to the class of set partitioning problems [5]. A multiset of n integers is given $\{w_1, ..., w_n\}$ and specifies the weight elements to partition. Another integer W_{max} is given and represents the weight capacity. The aim is to find a partition of the n integers into a minimal number of m bins (or sets) $\{s_1, .., s_k\}$ such that in each bin the sum of all integers does not exceed W_{max}. This problem is usually stated in terms of arithmetic constraints over 0-1 variables and solved using various OR techniques or constraint satisfaction ones. It requires one matrix (a_{ij}) to represent the elements of each set, one vector x_j to represent the selected subsets s_k and one vector w_i to represent the weights of the elements a_{ij}. Hereafter is the abstract formulation of the bin packing problem in Integer Programming (IP) and in Conjunto.

IP abstract formulation	Conjunto abstract formulation
$\sum_{j=1}^{m} a_{ij}\, x_j = 1 \ \forall i \in \{1, .., n\}$	$s_1 \cap s_2 = \{\}$, $s_1 \cap s_3 = \{\}$, .., $s_{n-1} \cap s_m = \{\}$ $s_1 \cup s_2 \cup ... \cup s_m = \{(1, w_1), .., (n, w_n)\}$
where: $x_j = 0..1$ (1 if $s_j \in \{s_1, .., s_k\}$) $a_{ij} = 0..1$ (1 if $i \in s_j$)	$s_j :: \{\}..\{(1, w_1), .., (n, w_n)\}$
$\sum_{i=1}^{n} a_{ij}\, w_i \leq W_{max} \ \forall j \in \{1, ..., m\}$	$weight(i, w_i) = w_i$; $\sum_{i=1}^{\#glb(s_j)} weight(i, w_i) \leq W_{max} \ \forall s_j$
Under these assumptions, the program to solve is to minimize the number of bins:	
$min\, x_0 = \sum_{j=1}^{m} x_j$	$minx_0 = \#\{s_j \mid s_j \neq \{\}\}$

Problem statement Let $P = \{\ item(1, w_1), ..., item(i, w_i), ..., item(n, w_n)\}$ be a non empty set of items i with a weight w_i. The aim is to partition P into a minimal number of N subsets such that the sum of the w_i in a computed subset of P does not exceed a limited weight $Wmax$. The heuristic used is the first fit descending which first sorts the objects in decreasing order of their weight. Bins are then filled one after another. The program uses the union operators as a constraint predicate (cf. previous remark on efficient handling of nested set expressions) and exploits the set representation with finite domains. We evoke a simple Conjunto program partly shown on figure 6 which solves large instances (80 items partitioned into 30 sets) and finds the optimal solution in about 22 seconds on a SUN 4/40. The part of the program given only shows the partitioning for a given N. The optimization predicate is the classical one which initializes N to

the value $weight(P)/Wmax$ and extends N at each call of the top level predicate until a failure is encountered. The solution is the last successful partition.

```
solve(N,Sets) :-
     pieces(P),
     make_sets(N,P,Sets),
     state_constraints(Sets,P),
     labeling(Sets).

make_sets(0,Plub,[]).
make_sets(N,Plub,[Set|Sets]):-
     $Set ::{}..Plub,
     N1 is N - 1,
     make_sets(N1, Plub,Sets).

state_constraints(Sets, P) :-
     restrict_weight(Sets),
     $all_disjoints(Sets),
     $all_union(Sets,P).

labeling([]).
labeling([S1|Sets]) :-
     $refined(S1),!,
     labeling(Sets).
labeling([S1|Sets]) :-
     $refine(S1),
     labeling([S1|Sets]).
```

Figure 6: *A partitioning program with Conjunto built-ins*

Problem solving The predicates preceded by $ are Conjunto built-ins. The main idea behind the program consists of retrieving the initial set (`pieces()`), creating a number N of sets whose initial domains are $\{\}..P$ (`make_sets()`), stating the constraints (`restrict_weight()` , `all_disjoint()` , `all_union()`) and adding (or removing in case of failure) elements to the sets by choosing first the element with the greatest weight (`labeling()`). The weight constraint (`restrict_weight()`) constrains the total weight of the elements of each set not to be greater than the limited weight $Wmax$. The top-level predicate is `solve(N,S)` if the partition represents N sets.

Problem data The problem receives as data the maximum weight allowed in each computable set `weight_max(50)` and a finite set of items `pieces({item(it1,W_1),...,item(it80,W_80)})`.

Conjunto built-ins The refine procedure tries to add elements to the glb of each set and in case of failure (the weight of the items is strictly greater than `weight_max(50)`) removes them from the lub. The `all_disjoint()` predicate constrains the upper bound of each S_j not to contain any lower bound of the remaining set domains D_i with $(i \neq j)$. It has the semantics of the following set of constraints $S_1 \cap S_2 = \{\}$, $S_1 \cap S_3 = \{\}, \ldots, S_{n-1} \cap S_n = \{\}$. The `all_union()` predicate constrains the union of all the upper bounds to be equal to the set P. It is encoded by splitting the $\bigcup S_i = P$ into simple constraints $S_1 \cup S_2 = S', \ldots, S^k \cup S_n = P$.

Experimental results and comparisons We made a complete comparative study with a 0-1 Finite Domain (FD) formulation. For the encoding of sets and set constraints, we used respectively lists of 0-1 variables and arithmetic constraints on the variables as described previously. The arithmetic constraint predicates were handled using the ECLiPSe solver[5] for arithmetic constraints over finite domains. The FD program was encoded so as to use the same first fit descending heuristics and the same labeling procedure as the Conjunto program. The following array gives the time together with space consumption results. The number of backtracks in the two program executions is the same.

Criterion	Conjunto program	FD program
global stack peak (bytes)	847 872	2 334 720
trail stack peak (bytes)	126 968	987 136
garb. collection number	27	77
cpu time (sec.)	21.6	31.5
garb. collection time (sec.)	1.21	6.28

The two programs differ in the data structure used and thus in the constraints applied to these data. The first point to note is that this difference has an impact both on the space consumption (stack peaks[6]) and on the cpu time. The space consumption comprises among other stacks, the global stack and the trail stack. The data structure is largely responsible for the growth of the global stack peak. The difference of space consumption (stack sizes) in the two approaches comes from the set-like representation as list of 0-1 domain variables versus two sorted lists in Conjunto: (i) The lists of 0-1 variables are never reduced because retrieving an element from a set corresponds to setting a variable domain to zero. This is not the case with the set domain representation. (ii) The trail stack is used to record information (set domains or lists of zero-one variables) that is needed on backtracking. The number of backtracks in the two program executions is the same, so the difference comes from the amount of information needed to be recorded.

The difference in the garbage collection number comes also from the space consumption as this number is the number of stack garbage collection.

The difference of cpu time is due first to the time needed for garbage collection which is a direct consequence of the size of the stacks which are garbage collected; and second to the time needed for performing operations on the data. A profile on the cpu time consumption indicated that half of the consumption in the FD program resolution is a time needed for performing arithmetic operations on the 0-1 variables. The weight constraint applied to each set is one of the costly computations. The weight constraint

[5] based on consistency techniques which perform a reasoning about variation domain bounds or about variation domains depending on the constraint predicate.

[6] the peak value indicates what was the maximum allocated amount during the session.

$a_{i1} \times W_1 + a_{i2} \times W_2 + ... a_{in} \times W_n \leq W_{max}$ which is awakened each time an a_{ij} is set to 1, consists of a cartesian product of two lists. In the Conjunto program, it consists of constraining the sum of weights W_i directly available from the elements (i, W_i) of a domain upper bound. Another costly computation in the FD formulation, is the one of the largest weight not already considered for one set. This requires to check the 0-1 variable in link with one weight. A weight is not yet considered if the corresponding domain variable is not instantiated. In the Conjunto program, this computation corresponds to the difference of the two bounds of a set domain. The resulting set constains the elements (i, W_i) which have not been considered yet. This difference operation is in fact the most time consuming in the Conjunto program resolution, for it is also performed to compute disjoint sets. But it represents half of the cpu time consumption of arithmetic operations.

Thus, it arises from this application based on the computation of a minimal set of bins, that set constraints together with set domains are expressive enough to embed the problem semantics and allow to avoid encoding the information as lists of 0-1 variables or handling additional data (the list of weights), and also that consistency techniques for set constraints are efficient to solve combinatorial problems on sets.

7 Conclusion and future work

A new CLP language embedding sets, called Conjunto is presented. In Conjunto, set variables range over finite set domains. This representation has several strengths: it is rather natural; it is powerful enough to express the set semantics; it leads to an efficient use of consistency techniques. New consistency algorithms for reasoning on the set domain bounds are the basis of the constraint solver. An $\mathcal{O}(ld + (e - l)dd')$ arc consistency algorithm is presented for the class of set relation and set domain constraints where l is the number of inclusion constraints, $e-l$ the number of remaining ones, d the sum of the cardinalities of the largest domain bounds and d' their difference. The operational semantics is described and guarantees satisfiability for a large class of constraints. An application of this computation domain to a bin packing problem is presented. It illustrates how efficiency can be combined with expressive power.

We are currently investigating the applicability of relation constraints to set covering problems. In this extended domain of computation based on the same notions of bounds, some complexity issues are currently being investigated for the consistency algorithms handling relation and graph constraints. Some work is still to be done though, both to complete the experimental work on set constraints and to evaluate the expressive power and practical interest of relations.

Acknowledgements

Special thanks to Pascal Van Hentenryck for his comments and worthwhile suggestions on a previous version of the paper. Many thanks also to Alexander Herold for his support, to Mark Wallace and Christophe Bonnet for their proofreading and useful comments. This work was supported in part by the ESPRIT Project 5291 CHIC.

References

[1] Alexander Aiken and Edward L. Wimmers. Solving Systems of Set Constraints. In *IEEE Symposium on Logic in Computer Science*, June 1992.

[2] L. Bachmair, H. Ganzinger, and U. Waldmann. Set Constraints are the Monadic Class. In *Proceedings of the LICS'93*, 1993.

[3] M. Dincbas, H. Simonis, and P. Van Hentenryck et al. The Constraint Logic Programming Language CHIP. In *FGCS*, Japan, Aug. 1988.

[4] A. Dovier and G. Rossi. Embedding Extensional Finite Sets in CLP. In *ILPS'93*, 1993.

[5] M.R. Garey and D. S. Johnson. *Computers and intractability, A guide to the theory of NP-completeness.* Victor Klee, 1979. 124-130.

[6] G. Gierz and K.H. Hofman et al. *A Compendium of Continuous Lattices.* Springer Verlag, Berlin Heidelberg New York, 1980. Chapter 0.

[7] N. Heintze and J. Jaffar. A Decision Procedure for a Class of Set Constraints. In *Proceedings of the Sixth Annual IEEE Symposium on Logic in CS*, pages 300–309, July 1991.

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

[9] P. Van Hentenryck and Y. Deville. Operational Semantics of Constraint Logic Programming over Finite Domains. In *Proceedings of PLILP'91*, pages 396–406, Passau, Germany, Aug. 1991.

[10] P. Van Hentenryck, Y. Deville, and C.-M. Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57:291–321, 1992.

[11] J. Jaffar and J.-L. Lassez. Constraint Logic Programming. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages*, pages 111–119, Munich, Germany, 1987.

[12] J. L. Laurière. A Language and a Program for Stating and Solving Combinatorial Problems. *Artificial Intelligence*, 10:29–127, 1978.

[13] B. Legeard and E. Legros. Short overview of the CLPS System. In *Proceedings of PLILP'91*, Passau, Germany, Aug. 1991. 3rd International Symposium on Programming Language Implementation and Logic Programming.

[14] A. K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 1977.

[15] Z. Pawlak. *Rough Sets: Theoretical Aspects of Reasoning about Data*. D: System theory, Knowledge engineering and Problem solving. Kluwer Academic Publishers, 1991.

[16] C. Walinsky. CLP(Σ^*): Constraint Logic Programming with Regular Sets. In *ICLP'89*, pages 181–190, 1989.