Constraints, An International Journal, 1, 191–246 (1997)

Interval Propagation to Reason about Sets: Definition and Implementation of a Practical Language

CARMEN GERVET c.gervet@doc.ic.ac.uk
IC-Parc, Imperial College, William Penney Laboratory, London SW7 2AZ, U.K.

Abstract. Local consistency techniques have been introduced in logic programming in order to extend the application domain of logic programming languages. The existing languages based on these techniques consider arithmetic constraints applied to variables ranging over finite integer domains. This makes difficult a natural and concise modelling as well as an efficient solving of a class of $\mathcal{NP}$-complete combinatorial search problems dealing with sets. To overcome these problems, we propose a solution which consists in extending the notion of integer domains to that of set domains (sets of sets). We specify a set domain by an interval whose lower and upper bounds are known sets, ordered by set inclusion. We define the formal and practical framework of a new constraint logic programming language over set domains, called Conjunto. Conjunto comprises the usual set operation symbols ($\cup, \cap, \setminus$), and the set inclusion relation ($\subseteq$). Set expressions built using the operation symbols are interpreted as relations ($s \cup s_1 = s_2$,...). In addition, Conjunto provides us with a set of constraints called graduated constraints (e.g. the set cardinality) which map sets onto arithmetic terms. This allows us to handle optimization problems by applying a cost function to the quantifiable, *i.e.*, arithmetic, terms which are associated to set terms. The constraint solving in Conjunto is based on local consistency techniques using interval reasoning which are extended to handle set constraints. The main contribution of this paper concerns the formal definition of the language and its design and implementation as a practical language.

Keywords: CSP, finite set domains, relational set constraints, consistency techniques, interval propagation, CLP, constraint programming language.

1. Introduction and motivation

This paper presents a means to tackle set based combinatorial search problems in a Constraint Logic Programming (CLP) framework (Jaffar and Lassez, 1987, Colmerauer, 1987, Jaffar and Maher, 1994). The main contribution of the work is a new language allowing set based constraint satisfaction problems to be modelled and solved in an elegant way using constraint logic programming. We introduce the notion of set domain following the concept of finite integer domain (Fikes, 1970). The elements of a set domain are known sets containing arbitrary values, and the set domain itself represents a powerset. It is defined as a set interval specified by its lower and upper bounds. The constraints of the language are built-in relations applied to variables ranging over set domains. The solver is based on an extension of consistency techniques (Mackworth, 1977, Mackworth and Freuder, 1985) —originating in artificial intelligence— to deal with set constraints. Closely related to our work are the notions of finite domains, sets and intervals embedded in a

constraint logic programming framework. These notions presented hereafter come from various backgrounds and were originally meant for different purposes.

1.1. Constraint satisfaction using CLP

Logic programming (Kowalski, 1974, Colmerauer et al., 1983, Lloyd, 1987) is a powerful programming framework which enables the user to state nondeterministic programs in relational form. Some ten years ago, the concept of finite domain variables (Van Hentenryck and Dincbas, 1986) i.e., variables ranging over a set of natural numbers, has been embedded into logic programming to allow for efficient tackling of combinatorial search problems modelled as Constraint Satisfaction Problems (CSPs)(Mackworth, 1977). A CSP is commonly described by a set of variables ranging over a set of possible values (the domains) and a set of constraints applied to the variables. It is well known that combinatorial search problems are $\mathcal{NP}$-complete (Papadimitriou and Steiglitz, 1982). The solving of a CSP utilizes local consistency techniques. These are constraint propagation techniques aiming at pruning the search space, associated to a CSP, by removing values that can never be part of any feasible solution. One use of these techniques in logic programming has aimed at extending a logic-based language with consistency techniques at the language level (Van Hentenryck and Dincbas, 1986). This has led to the first development of a Constraint Logic Programming (CLP) language on finite domains, CHIP (Dincbas et al., 1988) (Constraint Handling In Prolog).

CHIP extends the application domain of logic programming to the efficient solving of combinatorial search problems. Typical examples are scheduling applications, warehouse location problems, disjunctive scheduling and cutting stock (Dincbas et al., 1988) which are artificial intelligence or operations research problems. The success of CHIP prompted the development of new finite domain CLP languages, classified as CLP(FD) languages (e.g. (Carlson et al., 1994)), but also raised the question of its limitations. Some of the limitations are concerned with the difficulties CLP(FD) languages have to model and solve a class of combinatorial problems based on the search for sets or mapping objects. Set partitioning, set covering, matching problems are such combinatorial search problems. The main motivation of our work is to provide a solution to this problem. So far, a finite domain CSP approach models a set either as a list of variables taking their value from a finite set of integers ($[x_1, ..., x_m], x_i \in \{1, 2, ..., n\}$ if $m \leq n$ and the cardinality of the set is known to be m), or as a list of 0-1 variables ($[y_1, ..., y_m], y_i \in \{0, 1\}$). The first approach requires the removal of order and multiplicities among the elements of the list, which is achieved by adding ordering constraints ($x_1 < x_2 < ... < x_m$). Constraints over sets are modelled using arithmetic constraints. This is not natural, costly in variables, and this often makes the program non-generic. The second approach, based on the use of 0-1 variables, originates from 0-1 Integer Linear Programming (ILP) (Schrijver, 1986). It makes use of the one-to-one correspondence which exists between a subset s of a known set S and a boolean algebra. This correspondence is defined by the characteristic function:

$f : y_i \longrightarrow \{0,1\}$ where $f(y_i) = 1$ iff $i \in s$

In other words, a 0-1 variable is associated with each element in S and takes the value 1 if and only if the element belongs to the set s. This approach requires a lot of variables. In addition it does not ease the statement of set constraints such as the set inclusion, because the inclusion of one list into another requires considering a large amount of linear constraints over the 0-1 variables. This is not very natural, nor concise. To cope with this problem, two solutions have been proposed. One consists in defining a class of built-in predicates, referred to as global constraints (Beldiceanu, 1990, Beadiceanu and Contejean, 1994), which allow for the concise statement and global solving of a collection of constraints. One way to achieve such a global reasoning is to use operations research techniques in a CLP setting. This approach aims to achieve a better pruning of the variable domains by taking into account several constraints at a time. It also extends the programming facilities of CLP(FD) languages to handle efficiently specific problems such as disjunctive scheduling, computation of circuits in a graph, etc. *The second solution, presented in this paper, aims at extending the expressiveness of the language by embedding sets and providing set and mapping constraints for general purposes.* This requires an investigation of how CLP languages based on sets tackle the set satisfiability problem and how well expressiveness can be combined with efficiency.

1.2. Set data structures in logic-based programming languages

A set is a collection of distinct elements commonly described by $\{x_1, ..., x_n\}$. The first application to embed sets as a high level programming abstraction was in rapid software prototyping and problem specification (Oxford 1986, Schwartz et al., 1986, Turner, 1986) More recent proposals in database query languages, assume a logic-based language as the underlying framework. These proposals aimed at strengthening typical existing set facilities of languages like Prolog (e.g. setof, bagof) to handle sets of terms and complex data structures. In this line of work sets have been embedded in (Beeri et al., 1991, Kuper, 1990, Shmueli et al., 1992, Dovier et al., 1991). All these languages converge on one aspect: representing a set variable by a set constructor so as to nest objects in a natural manner. This constructor is specified either by an extensional representation $\{x_1, ..., x_n\}$ ((Beeri et al., 1991, Kuper, 1990)) or by an iterative one $\{x\} \cup E$ where E can be unified with a set of terms containing possibly set variables (concept of sets of finite depth in (Dovier et al., 1991, Legeard and Legros, 1991, Stolzenburg, 1996)). The equality relation over constructed sets is a particular case of Associative, Commutative and Idempotent (ACI) relation (Livesey and Siekmann, 1976). Each property is usually modelled by a set of axioms. Ensuring the satisfiability of these properties, i.e. solving the satisfiability problem of constructed sets, is $\mathcal{NP}$-complete or even $\mathcal{NP}$-hard (Livesey and Siekmann, 1976, Perry et al., 1986, Kapur and Narendran, 1986, Hibti, 1995), depending on the class of axioms and operations considered (e.g. $\cup, \cap, \setminus$). The main reason is the absence of a unique most general

unifier when unifying constructed sets. This is clear from the following example: the equality $\{X, Y\} = \{3, 4\}$ derives two solution sets: $\{X = 3, Y = 4\}$ and $\{X = 4, Y = 3\}$ neither of which is more general than the other. Thus, in practice the unification procedure of constructed sets is achieved by computing a minimal collection of set unifiers, that is a set of substitutions. This means that the satisfaction of the ACI axioms introduces nondeterminism in the unification procedure by deriving disjunctions of a finite number of equalities. In (Beeri et al., 1991, Jayaraman and Plaisted, 1989) a term-matching procedure is considered (unification of two sets when one of them contains no variables). This approach reduces significantly the set of unifiers. But term-matching for constructed sets remains an NP-complete problem (Perry et al., 1986, Kapur and Narendran, 1986). Indeed, if $\{x_1, ..., x_n\} = \{1, ..., m\}$ $(m < n)$ there are at most 2^{n-m} computable solutions.

These approaches allow for a high level of abstraction when representing collections of terms. Unfortunately they are very inefficient in time complexity results. Recently, some alternative approaches have focused on embedding constructed sets in constraint logic programming. CLP languages dealing with sets, CLP(Sets), are defined as instances of the CLP scheme (Jaffar and Lassez, 1987) over a specific computation domain describing the class of allowed set expressions and set constructors. These CLP(Sets) languages provide a sound and complete solver. Hereafter, we put a particular attention into the description of CLP(Σ^*) which deals with regular sets, the revisited language {log} which axiomatizes a set theory, and CLPS which aims at prototyping combinatorial problems using sets, multisets and sequences.

1.3. Set data structures in constraint logic programming languages

Constraint Logic Programming (CLP) combines the positive features of logic programming with constraint solving techniques. The concept of constraint solving replaces the unification procedure in logic programming and provides, among others, a uniform framework for handling set constraints (eg. $x \in s, s \subseteq s_1, s = s_2$).

CLP(Σ^*) (Walinsky, 1989) represents an instance of the CLP scheme over the computation domain of regular sets. A regular set is a finite set composed of strings which are generated from a finite alphabet Σ. This language incorporates strings into logic programming to strengthen the standard string-handling features (eg. concat, substring). CLP(Σ^*) does not deal with sets in the general sense but nevertheless, it constitutes a first attempt to compute regular sets by means of constraints like the membership relation. The complexity of the satisfiability procedure is not given, but infinite computations are avoided thanks to the use of floundering.

{log} (Dovier and Rossi, 1993, Bruscoli et al., 1994) has been revisited from a LP to a CLP framework in order to provide a uniform framework for the handling of set constraints $(\in, =, \neq, \notin)$. The author does not know of any application developed using this language but its design and implementation have settled the theoretical foundations for embedding constructed sets of the form $\{x\} \cup S$ into logic program-

ming and constraint logic programming. The soundness and completeness of its solver allow us to use it for theorem proving and problem specification. In {log}, the nondeterministic satisfaction procedure of constructed sets reduces a given constraint to a collection of constraints in a suitable form by introducing choice points in the constraint graph itself. This leads to a hidden exponential growth in the search tree. In this approach, completeness of the solver is required if one aims at performing theorem proving. Thus, there is no possible compromise here between completeness and efficiency.

CLPS (Legeard and Legros, 1991, Legeard and Legros, 1992) aims at prototyping combinatorial problems using sets, multisets and sequences. It proposes a couple of interesting methods to handle extensional sets $\{x_1, ..., x_n\}$ of finite depth (e.g. $s = \{\{\{e, a\}\}, c\}$ is a set of depth three). Unlike {log}, CLPS comprises the set cardinality operation which in this prototyping context is of a great practical use. One of the distinctive features of CLPS is to allow set elements to range over integer domains. When set elements are finite domain varaibles, the satisfiability problem of constructed sets is tackled by an arc-consistency algorithm of type AC-3 (Mackworth, 1977) combined with a local search procedure (forward checking). A system of set constraints where each set element ranges over a finite domain is consistent if each of the set constraints it contains is locally consistent. For example, the system $x \in \{1, 2\}, y \in \{1, 3, 4\}, [z, t] \in \{1, 2, 4, 5\}, \{x, y\} = \{z, t\}$ is consistent if $x \in \{1, 2\}, y \in \{1, 4\}$ and $[z, t] \in \{1, 2, 4\}$. Note that the set equality relation should be associative, commutative and idempotent. It might happen that due to the satisfaction of the ACI axioms, distinct selected values for the elements will generate identical instances of the sets (e.g. the two sets of selected values $\{x = 1, y = 4, z = 1, t = 4\}$ and $\{x = 1, y = 4, z = 4, t = 1\}$ generate a unique instance $\{1, 4\}$ for both sets). While some a priori pruning can be achieved, the search procedure which unifies the constructed sets remains exponential. This is a main drawback of this language when solving set-based combinatorial search problems (e.g. bin packing, set partitioning). However, their later work on constructed terms for multisets and sequences proved to be appropriate for modelling and solving scheduling problems with a reasonable efficiency (Baptiste et al., 1994, Boucher and Legeard, 1996).

To achieve a better efficiency in the area of combinatorial search problem solving, a set should be represented by a variable as opposed to a constructed term; this allows us to have a deterministic set unification procedure which consists of testing in polynomial time the equality between set variables and ground sets (e.g. $S = \{1, 2\}$). In addition, sets should range over domains so as to make use of powerful constraint propagation techniques. To achieve this, *we propose a language which enables us to model a set-based problem as a set domain CSP —where set variables range over set domains—, and which tackles set constraints by using consistency techniques.* A set domain can be a collection of known sets of arbitrary elements like $\{\{a, b\}, \{c, d\}, \{e\}\}$. It might happen that the elements of the domain are not ordered at all, and thus if large domains are considered, it is not possible to approximate the domain reasoning by an interval reasoning as in some CLP(FD)

systems. To cope with this, *we propose to approximate a set domain by a set interval specified by its upper and lower bounds, thus guaranteeing that a partial ordering exists. This allows us to make use of consistency techniques by reasoning in terms of interval variations, when dealing with a system of set constraints.* The set interval $[\{\}, \{a, b, c, d, e\}]$ represents the convex closure of the set domain above.

The strengths of handling intervals in CLP have recently been proved when dealing, in particular, with integers and reals. On the one hand, interval reasoning does not guarantee that all the values from a domain are locally consistent, versus domain reasoning. On the other hand, it removes at a minimal cost some values that can never be part of any feasible solution. This is achieved by pruning the domain bounds instead of considering each domain element one by one. Interval reasoning is particularly suitable to handle monotonic binary constraints (e.g. $x \leq y, s \subseteq s_1$), where it guarantees the correctness properties of domain reasoning while being more efficient in terms of time complexity.

1.4. Interval reasoning using CLP

The introduction of real intervals into CLP aims at avoiding the errors resulting from finite precision of reals in computers. A real interval is an approximation of a real and is specified by its lower and upper bounds. It does not denote the set of possible values a variable could take but a variation of an infinite number of values. Cleary (Cleary, 1987) introduced a relational arithmetic of real intervals into logic programming based on the interpretation of arithmetic expressions as relations. Such relations are handled by making use of projection functions and closure operations, which correspond to the definition of transformation rules expressing each real interval in terms of the other intervals involved in the relation. These transformation rules approximate the usual consistency notions. The handling of these rules is done by a relaxation algorithm which resembles the arc-consistency algorithm AC-3 (Mackworth, 1977). This approach prompted the development of the class of CLP(Intervals). A formalization of this approach is given in (Benhamou, 1995).

While CLP(Intervals) languages make use of consistency techniques, they do not model CSPs because the solving of a problem modelled in a CLP(Intervals) language searches for the smallest real intervals such that the computations are correct. It guarantees that the values which have been removed are irrelevant, but does not bind the real variables to a value. Set intervals in constraint logic programming resemble the real interval arithmetic approach in terms of interpreting set expressions (e.g. $s \cup s_1, s \cap s_1$) as relations and using interval reasoning to perform set interval calculus when handling the constraints. However, *set intervals in constraint logic programming contribute to the definition of a language which allows one to model and solve discrete CSPs.* In practice, this corresponds to providing a labelling procedure in order to reach a complete solution. This requirement differs from that of CLP(Intervals) languages where the completeness issue is still an open problem because of the infinite size of real intervals.

1.5. Contribution

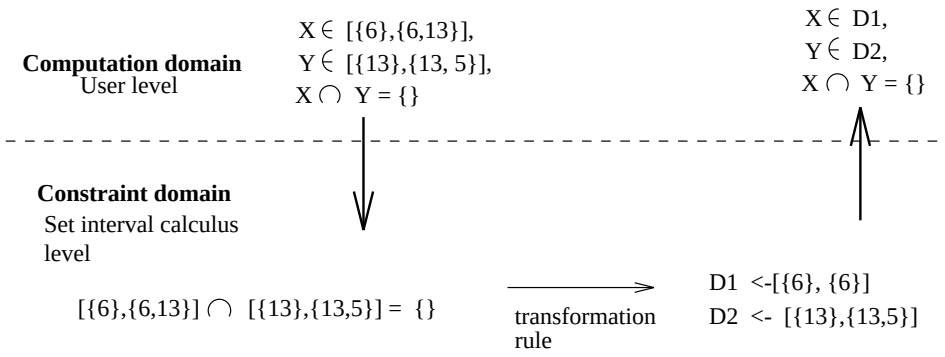
This paper contains the following contributions:

- A formal framework for solving a system of set constraints over set domains. This framework defines the algebraic structure of the constraint domain over set intervals. It is generic and can be adapted to formalize the class of languages which make use of consistency techniques as main constraint solving tool.
- A practical framework describing the Conjunto language which we have designed and implemented using the constraint logic programming platform ECL^{i}PS^e (ECRC, 1994).
- Applications developed in Conjunto. They illustrate the modelling facilities of the language and its ability to solve in an efficient way large search problems (Gervet, 1994, Gervet, 1995).

I Formal Framework

This part describes a constraint logic programming system dealing with sets which range over a finite domain —*i.e.*, sets which belong to a powerset— and whose solver is based on consistency techniques.

A CLP system is parameterized by its computation domain and more generally by its constraint domain (Jaffar and Maher, 1994). The computation domain is the algebraic structure over which constraints are applied to set variables and the constraint domain is the algebraic structure over which consistency techniques are performed in terms of set interval reasoning. A clear distinction should be made between them. On the one hand, the user manipulates sets in a logic-based language and on the other hand set interval calculus is performed to search for set values as illustrated on the following figure.



A constraint logic programming language with sets, set operations and relations is not expressive enough to tackle set-based search problems. In particular optimization problems require the statement of a cost function which necessarily deal

with quantifiable, *i.e.*, arithmetic, terms. To cope with this, an extension of the language is presented and consists in adding to the language syntax and to the constraint domain of the system a class of functions which map sets to integers (e.g. the set cardinality #, the set weight, etc.). These functions are called graded functions when they map elements from a lattice (e.g. a powerset equipped with the operations $\cup, \cap$ and the partial ordering $\subseteq$) to the set of integers.

2. Basics of powerset lattices

Some definitions, properties and results on lattices are necessary to understand the main features of the formal description of the system. These can be found in (Birkhoff, 1967, Graetzer, 1971, Gierz and Hoffman, 1980). The particular lattice we deal with is the powerset lattice. To give an intuitive idea of the subsequent use of these definitions, some examples relating to powerset lattices are given. Readers familiar with these notions can skip this subsection.

2.1. Lattices

DEFINITION 1 *A* poset *(also known as* partially ordered set*) is a set S equipped with a binary relation $\preceq$ (formally a subset of $S \times S$) that satisfies the following laws:*

P1. Reflexivity $\quad \forall x, x \preceq x$
P2. Antisymmetry $\quad (x \preceq y \text{ and } y \preceq x) \Rightarrow (x = y)$
P3. Transitivity $\quad (x \preceq y \text{ and } y \preceq z) \Rightarrow x \preceq z$

Example: Let S be a finite set and $\mathcal{P}(S)$ the set of all subsets of S or powerset of S. Then the set inclusion $\subseteq$ is easily seen to be a partial order on $\mathcal{P}(S)$. $\mathcal{P}(S)$ is a poset. □

DEFINITION 2 *Let S be a poset, X a subset of S and y an element of S. Then y is a* meet *or* greatest lower bound *or* glb *for X iff:*
y is a lower bound for X, i.e., if $x \in X$ then $y \preceq x$ and,
if z is any other lower bound for X then $z \preceq y$
The notation we use is $y = \bigwedge (X)$.

DEFINITION 3 *Let S be a poset, $X \subseteq S$ and $y \in S$. Then y is a* join *or* least upper bound *or* lub *for X iff:*
y is an upper bound for X, i.e., if $x \in X$ then $y \succeq x$ and,
if z is any other upper bound for X then $z \succeq y$
The notation we use is $y = \bigvee (X)$.

PROPOSITION 1 *Let S be a poset and X a subset of S. Then X can have at most one* meet *and at most one* join.

Proof: By *P2*, meet and join are clearly unique whenever they exist. If a and b are two meets then we have on the one hand $a \preceq b$ and on the other hand $b \preceq a$. This infers $a = b$. ■

The following property establishes a link between $\preceq$ and the pair $(\bigwedge, \bigvee)$ as actual meet and join.

Property 1 (Consistency property) *Let S be a poset. Then for all $x, y \in S$,*
$x \preceq y \Leftrightarrow x = \bigwedge(\{x, y\})$
$x \preceq y \Leftrightarrow y = \bigvee(\{x, y\})$

PROPOSITION 2 *(Graetzer, 1971) The following definitions are equivalent:*
(i) A poset is a lattice *iff every finite subset has a meet and a join.*
(ii) A poset S is a lattice iff every two elements have a meet and a join.

Example: The powerset $\mathcal{P}(X)$ is a lattice where the meet operator is the intersection $\cap$ and the join operator is the union $\cup$. Every two elements x, y of $\mathcal{P}(X)$ have a meet $x \cap y$ and a join $x \cup y$.

The partial order as set inclusion $\subseteq$ satisfies the consistency property:

$$x \cup y = y \Leftrightarrow x \subseteq y \Leftrightarrow x \cap y = x$$

This equivalence defines the correspondence between the relational definition of the structure $[\mathcal{P}(X), \subseteq, \cap, \cup]$ in terms of properties of the partial order $\subseteq$ (existence of a glb and a lub) with its algebraic definition in terms of properties of the operations $\cup, \cap$. □

2.2. Intervals in powerset lattices

Reasoning with and about intervals within a powerset lattice constitutes the core of our system. The following definitions and properties give the basic properties of set intervals in powerset lattices. A set interval delimited by two sets x and y is specified by the syntax $[x, y]$ such that $x \subseteq y$. In case $x = y$ this interval is reduced to a singleton. One important task in set interval reasoning is the computation of set intervals which describe the smallest convex powerset containing a collection of sets. This subsection focuses on the definitions and properties of these convex set intervals.

DEFINITION 4 *An* interval *of two arbitrary sets x, y in a powerset lattice is the set $[x \cap y, x \cup y]$.*

DEFINITION 5 *A subset S of a powerset lattice L is* convex *if $x, y \in S$ imply*

$$[x \cap y, x \cup y] \subseteq S$$

Property 2 *The meet and join operators in a powerset lattice are isotone (preserve the order):*

$x \subseteq y \Rightarrow x \cap z \subseteq y \cap z$

$x \subseteq y \Rightarrow x \cup z \subseteq y \cup z$

Example: This property is extremely useful when reasoning about set intervals in a powerset lattice $\mathcal{P}(X)$. Consider the following inclusion relations between elements of $\mathcal{P}(X)$:

$$a \subseteq x \subseteq b \text{ and } c \subseteq y \subseteq d$$

x and y belong to the respective intervals $[a, b]$ and $[c, d]$. From property 2, we infer $a \cap c \subseteq x \cap y \subseteq b \cap d$ and dually for the union operation. So if x and y are only defined from the intervals they belong to, their union and intersection can be approximated by the new intervals $[a \cup c, b \cup d]$ and $[a \cap c, b \cap d]$. □

PROPOSITION 3 *A closed set interval $[x \cap y, x \cup y]$ is convex.*

Proof: Let $I = [x \cap y, x \cup y]$ be a set interval. If $z, t \in I$ then $z \cap t \in I$ and $z \cup t \in I$, and by Property 2: $[z \cap t, z \cup t] \subseteq I$. ■

3. Set intervals in CLP

Consider an arbitrary collection of sets. Take the smallest convex set which contains this collection of sets. This convex part denotes a set interval. This concept of set interval is the means we use to reason with and about sets in a CLP system. On the one hand the user manipulates sets in a logic-based language and on the other hand set interval calculus is performed to search for set values. This section describes the algebraic structure of the system called the constraint domain. This is the structure over which set interval calculus is performed.

3.1. Preliminaries

Let Σ_S be the set of predefined function and predicate symbols necessary to reason with and about sets in the language:

$$\Sigma_S = \{\emptyset, \cup, \cap, \setminus, \subseteq, \in_{[a,b]}\}$$

The predicate symbol $\in_{[a,b]}$ applied to a variable s will be interpreted as the double ordering $a \subseteq s \subseteq b$.

The set of constants defines the domain of discourse of the language. It extends the Herbrand universe to provide the concept of set constant.

DEFINITION 6 *The domain of discourse is the powerset*

$$D_S = \mathcal{P}(H_u) \text{ where } H_u \text{ refers to the Herbrand universe}$$

A set constant is any element from $\mathcal{P}(H_u)$ represented by the abstract syntax $\{e_1, ..., e_n\}$ where the e_i belong to H_u.

DEFINITION 7 *A* set variable *is any variable taking its value in* $\mathcal{P}(H_u)$.

DEFINITION 8 *A* set expression S *of* $\mathcal{D}_S$ *where* s_1, s_2 *are set constant or variables is inductively defined by:* $s_1 \cup s_2 \mid s_1 \cap s_2 \mid s_1 \setminus s_2$

Notations. Set variables will be represented by the letters x, y, z, s. Set constants will be represented by the letters a, b, c, d. Natural numbers will be represented by the letters m, n and integer variables by v, w. All these symbols can be subscripted.

3.2. Computation domain

The computation domain of the system is the powerset algebra $\mathcal{D}_S$ which interprets (over the domain of discourse D_S) the function symbols $\cup, \cap, \setminus$ belonging to Σ_S in their usual set theoretical sense (*i.e.*, $\emptyset$ is the empty set, $\setminus$ the set difference, etc.).

The interpreted set union and intersection symbols have the following algebraic properties:

C.	$x \cap y = y \cap x$	$x \cup y = y \cup x$	*commutativity*
As.	$(x \cap y) \cap z = x \cap (y \cap z)$	$(x \cup y) \cup z = x \cup (y \cup z)$	*associativity*
I.	$x \cap x = x$	$x \cup x = x$	*idempotence*
Ab.	$x \cap (x \cup y) = x$	$x \cup (x \cap y) = x$	*absorption*

3.3. Constraint domain

The constraint domain represents the structure of the system over which set interval calculus is performed. This structure is built from the computation domain equipped with the predicate symbols $\subseteq, \in_{[a,b]}$ belonging to Σ_S and interpreted as constraint relations. The predicate symbol $\subseteq$ is interpreted as the set inclusion and the predicate $\in_{[a,b]}$ is interpreted as the set domain constraint. This relation constrains a set variable to take its value in a specific domain. Since the main idea of the system is to perform set interval calculus, we must guarantee that the domain of any set variable is an interval.

The structure denoted by $[\mathcal{D}_S, \subseteq]$ describes a powerset lattice with the partial order $\subseteq$. Any two of its elements c, d have a unique least upper bound $c \cup d$ and a unique greatest lower bound $c \cap d$ (cf. section 2.1.). The existence of limit elements for any set $\{c, d\}$ belonging to D_S allows us to define a notion of set domain as a convex subset of D_S, that is a set interval $[c \cap d, c \cup d]$.

DEFINITION 9 *A* set interval domain *or* set domain *is a convex subset of* D_S *specified by* $[a, b]$ *such that* $a \subseteq b$ *and* $a, b \in \mathcal{P}(H_u)$.

DEFINITION 10 *A set variable* s *is said to* range *over a set domain* $[a, b]$ *if and only if* $s \in [a, b]$.

DEFINITION 11 *The* definite *elements of a set s such that $s \in [a, b]$ are the elements contained in the greatest lower bound a.*

DEFINITION 12 *The* possible *elements of a set s such that $s \in [a, b]$ are the elements contained in the least upper bound b which are not in a.*

Example: The constraint $s \in [\{3,1\}, \{3,1,5,6\}]$ means that the elements $3, 1$ are definite elements of s (they belong to s) and that 5 and 6 are possible elements of s. □

Set intervals have been used so far to specify the domain of a set variable. Regarding set expressions, the domain of a union or intersection of sets is not a set interval because it is not a convex subset of D_S (e.g. $I = [\{1\}, \{1,3\}] \cup [\{\}, \{2,6\}]$, $\{1,3\}, \{6\} \in I$ but $[\{\}, \{1,3,6\}] \not\subseteq I$). It is possible to maintain such disjunctions of domains during the computation, but this leads to a combinatorial explosion. This handling of "holes" can be avoided by considering the convex closure of a set expression domain. Consequently, the constraint domain of the system is defined as the powerset lattice over the convex parts of $\mathcal{P}(D_S)$ (convex subsets of D_S), equipped with a convex closure operation.

DEFINITION 13 *The set of all convex parts of $\mathcal{P}(D_S)$ is a subset of $\mathcal{P}(D_S)$ ordered by set inclusion and designated by ΩD_S.*

DEFINITION 14 *The* constraint domain *$\mathcal{CD}$ is the algebraic structure of the lattice ΩD_S of set intervals ordered by set inclusion such that:*
$\mathcal{CD} = [\Omega D_S, \mathcal{D}_S, \subseteq, \in_{[a,b]}]$

Convex closure operation To ensure that any set domain is a set interval, we define a convex closure operation which associates to any element of $\mathcal{P}(D_S)$ its convex closure as being a set interval, element of ΩD_S.

DEFINITION 15 *The* convex closure *operation $\vec{conv} : \mathcal{P}(D_S) \to \Omega D_S$ is such that $\vec{conv} : x \to \overline{x}$ satisfies:*

$$x = \{a_1, ..., a_n\} \to \overline{x} = [\bigcap_{a_i \in x} a_i, \bigcup_{a_i \in x} a_i]$$

For example, the convex closure of the set $\{\{3,2\}, \{3,4,1\}, \{3\}\}$ belonging to $\mathcal{P}(D_S)$ is the set interval $[\{3\}, \{1,2,3,4\}]$.

Property 3 *An element x of $\mathcal{P}(D_S)$ is convex under the above convex closure operation when x is equal to its "closure" $\overline{x}$.*

COROLLARY 1 *All singleton sets are convex.*

In the following, the operations $\bigcap_{a_i \in x} a_i$ and $\bigcup_{a_i \in x} a_i$ will be respectively written $glb(x)$ and $lub(x)$ which stand for greatest lower bound and least upper bound of x, respectively.

Property 4 *The operation* $\vec{conv}(x) = \overline{x} = [glb(x), lub(x)]$ *has the following* properties*:*

C1.	$x \subseteq \overline{x}$	*Extension*
C2.	$\overline{x} = \overline{\overline{x}}$	*Idempotence*
C3.	*If* $x \subseteq y$*, then* $\overline{x} \subseteq \overline{y}$	*Monotonicity*

If we consider the $\subseteq$ relation as a logical implication, the extension property *C1* can be interpreted by "any element of x belongs to $\overline{x}$ (thus to $glb(x)$) and any element definitely not in $\overline{x}$ (not in $lub(x)$) does not belong to x". This allows the set calculus to be performed in ΩD_S while ensuring that the computed solutions are valid in D_S. Property *C3* guarantees that the partial order $\subseteq$ is preserved in ΩD_S.

ΩD_S equipped with the operation $\vec{conv}$ allows us to define the constraint domain from an algebraic point of view, *i.e.*, from the properties of the union and intersection operations in ΩD_S.

DEFINITION 16 *The constraint domain* $\mathcal{CD}$ *is a powerset lattice* $[\mathcal{D}_S, \subseteq, \in_{[a,b]}]$ *with the family* ΩD_S *of set intervals that satisfies:*

P1.	*Each* union *of elements of* ΩD_S *is also an element of* ΩD_S
P2.	*Each* finite intersection *of elements of* ΩD_S *is also an element of* ΩD_S
P3.	$\mathcal{P}(D_S)$ *and the empty set* $\{\}$ *are elements of* ΩD_S.

Properties *P1* and *P2* define the distributivity of $\cup$ and $\cap$ in ΩD_S. It follows from *P2* and the first statement of *P3* ($\mathcal{P}(D_S) \in \Omega D_S$) that a convex closure operation satisfying *C1-C3* is defined in $\mathcal{CD}$. This operation is $\vec{conv}$. Because of *P1* and *P2* this operation satisfies:

$$\overline{x \cup y} = \overline{x} \cup \overline{y} \text{ and } \overline{x \cap y} = \overline{x} \cap \overline{y}$$

Finally *P3* implies that $\overline{\emptyset} = \emptyset$.

3.4. Set interval calculus

In order to satisfy the properties *P1*, *P2* and *P3*, we define a set interval calculus within ΩD_S. This consists in deriving equality relations from the following ordering relations:

$$[a,b] \cup [c,d] \subseteq [a \cup c, b \cup d] \text{ and } [a,b] \cap [c,d] \subseteq [a \cap c, b \cap d]$$

This is achieved by making use of the convex closure operation. The resulting set interval calculus is described as follows:

$$\overline{[a,b] \cup [c,d]} = [a \cup c, b \cup d]$$
$$\overline{[a,b] \cap [c,d]} = [a \cap c, b \cap d]$$
$$\overline{\mathcal{P}(D_s)} = \mathcal{P}(D_s) \text{ and } \overline{\emptyset} = \emptyset$$

With regard to the set difference operation $[a,b]\setminus[c,d]$, its set theoretical definition is $x \setminus y = x \cap y'$ where y' is the complement of y. The complement of a set interval $[c,d]$ is the set interval $[D_s \setminus d, D_s \setminus c]$ which is characterized by the fact that it does not contain the elements in c and that the elements of d should not a priori be definite elements of this interval. So the convex closure of a set interval difference is:

$$\overline{[a,b] \setminus [c,d]} = [a \setminus d, b \setminus c]$$

The consistency property $x \subseteq y \Leftrightarrow y = x \cup y$ and $x \subseteq y \Leftrightarrow x = x \cap y$ (cf. 2.1. property 1) characterizes $\subseteq$ by the set operations of a powerset lattice (in fact by either of them). This embeds the notions of right inclusion ($y = x \cup y$), which defines the least upper bound (join operator) for x and y to be y, and the left inclusion ($x = x \cap y$), which defines the greatest lower bound (meet operator) for x and y to be x. From an operational point of view, obtaining such a characterization is essential. However since computations are performed in $\mathcal{CD}$, this property needs to be defined for set intervals using te set interval calculus within ΩD_S.

Consider two set intervals $[a,b]$ and $[c,d]$. They denote powersets and thus sets. Consequently we have: $[a,b] \subseteq [c,d] \Leftrightarrow [a,b] = [a,b] \cap [c,d] \Leftrightarrow [c,d] = [c,d] \cup [a,b]$. Using the set interval calculus, this is equivalent to:

$$\begin{aligned}[a,b] \subseteq [c,d] &\Leftrightarrow [a,b] = [a \cap c, b \cap d] \\ &\Leftrightarrow [c,d] = [c \cup a, d \cup b] \\ &\Leftrightarrow a \subseteq c, b \subseteq d\end{aligned}$$

DEFINITION 17 *Assuming that $[a,b],[c,d]$ specify set domains, the consistency property in $\mathcal{CD}$ is defined by:*

$$[a,b] \subseteq [c,d] \Leftrightarrow a \subseteq c,\ b \subseteq d$$

This definition of consistency gives us the necessary conditions to be satisfied when checking and/or inferring consistency of the set inclusion constraint over set domains.

3.5. Graded functions

The expressivity of the system can be increased if some graded functions are applied to sets. A graded function maps a non quantifiable term to an integer value denoting a measure of the term. The set cardinality is one example of such a function. They allow the user to deal with optimization functions in a set-based language (e.g. minimizing the cardinality of a set). The constraint domain presented so far does not contain any such graded functions. In this subsection, we extend the language alphabet and the constraint domain of the system to deal with such functions. In order not to limit the extension of the language to the set cardinality function, the general case of an arbitrary graded function f is studied.

DEFINITION 18 *A graded function f is a function from $[\mathcal{D}_S, \subseteq]$ to $\mathcal{N}$ (set of positive integers) which maps each element $x \in D_S$ to a unique m such that $f(x) = m$ and which satisfies:*

$s_1 \subset s_2 \Rightarrow f(s_1) < f(s_2)$ *($\subset$ is the strict inclusion and $<$ the arithmetic inequality)*

The convex closure of a graded function f is required to deal with elements from ΩD_S. The closure function, written $\overline{f}$, maps elements from ΩD_S to a subset of the powerset $\mathcal{P}(\mathcal{N})$ containing intervals of positive integers. This subset is designated by $\Omega\mathcal{N}$.

Example: Let s be a set and $\#s$ its cardinality (a positive integer). Consider the constraint $s \in [\{\}, \{1,2\}]$. The cardinality function $\#$ is approximated by $\overline{\#}$. Intuitively we have $\overline{\#}(s) = [0,2]$. □

DEFINITION 19 *Let $f : D_S \to \mathcal{N}$. The function $\overline{f} : \Omega D_S \to \Omega\mathcal{N}$ is derived from f as follows:*

$\overline{f}([a,b]) = [f(a), f(b)]$

Property 5 *If $s \in [a,b]$ then $f(s) \in \overline{f}([a,b])$.*

Proof: By definition f is a graded function. So if $a \subset s \subset b$ then we have $f(a) < f(s) < f(b)$. Consequently we have $f(x) \in [f(a), f(b)]$ which means $f(s) \in \overline{f}([a,b])$. ■

This property guarantees that the output of the function $\overline{f}$ applied to a set domain contains the actual graduation value of the concerned set variable.

3.6. Extended constraint domain

Graded functions add expressive power to the language. They can be embedded as predefined symbols in the language, if the constraint domain is extended to deal with integer intervals and integer variables. The constraint domain associated with integer intervals is that of integer interval domains (subset of the standard constraint domain over finite integer domains). It is defined by the structure:

$$\mathcal{FD} = [\Omega\mathcal{N}, (\mathcal{N}, +), =, \neq, \geq, \in_{[m,n]}]$$

where the relation $\in_{[m,n]}$ is interpreted in $\Omega\mathcal{N}$ as the integer domain constraint such that: $x \in_{[m,n]} [m,n]$ is equivalent to $m \leq x \leq n$. The other symbols are interpreted in their usual arithmetic sense. The extended constraint domain of our system should contain $\mathcal{FD}$.

The extended constraint domain $\mathcal{CD}_e$ with graded functions, is the structure:

$$[\Omega D_S, \mathcal{D}_S, f, \subseteq, \in_{[a,b]}] \cup \mathcal{FD}$$

$\mathcal{CD}_e$ interprets graded function symbols as unary set operations with respect to their intended meaning. For example the symbol # is interpreted as the set cardinality operation.

4. Execution model

The execution model is based on constraint solving in $\mathcal{CD}_e$. It is a top-down execution model which defines the operational semantics of the system. The model describes how the constraints are processed over $\mathcal{CD}_e$ and what they lead to. The idea consists in (1) constraining each set variable to range over a set domain, and (2) removing some values of the set domains that can never be part of any feasible solution. This is achieved by making use of local consistency techniques adapted to the handling of constraints defined in $\mathcal{CD}_e$. A transformed system is commonly called a locally consistent system. One necessary condition for dealing with local consistency techniques is that each set variable ranges over a set domain.

4.1. Definition of an admissible system of constraints

The set of predefined constraints in $\mathcal{CD}_e$ can contain any of the following:

- set domain constraints $s \in [a, b]$ where s is a set variable.
- set constraints $S \subseteq S_1$ where S, S_1 are set expressions (comprising constants, variables and possibly set operation symbols in $\{\cup, \cap, \setminus\}$).
- graduated constraints $f(S) \in [m, n]$ where f is any predefined graded function and $[m, n]$ any element in ΩN (*i.e.*, an integer if $m = n$ or an integer domain).

DEFINITION 20 *An* admissible system *of constraints in $\mathcal{CD}_e$ is a system of constraints such that every set variable s ranges over a set domain.*

4.2. From n-ary constraints to primitive ones

The predefined constraints might denote n-ary constraints like $s_1 \cup s_2 \subseteq s_3 \cap s_4$. Ensuring the local consistency of these constraints via interval refinement methods requires us to express each set variable in terms of the others. Since there is no inverse operation for $\cup, \cap, \setminus$ there is no way to move all the operation symbols on one side of the constraint predicate. So it is necessary to decompose n-ary constraints into primitive ones.

Consider the following set of basic set expressions $\{s \cap s_1, s \cup s_1, s \setminus s_1\}$. The proposed method consists in approximating each basic set expression by a new set variable with its appropriate domain. The resulting constraints are binary or unary ones called primitive constraints.

DEFINITION 21 *A* primitive *constraint is (1) a predefined set constraint containing at most two set variables or, (2) a graduated constraint containing at most one set variable.*

In the former example the n-ary constraint is approximated by the system of constraints:

$$s_1 \cup s_2 = s_{12},\ s_3 \cap s_4 = s_{34},\ s_{12} \subseteq s_{34}$$

This approach is similar to the relational form of arithmetic constraints over real intervals introduced by Cleary (Cleary, 1987).

A relation denoting a basic set expression represents a subset of the Cartesian product of the set domains attached to each set variable. In order to deal with the consistency of these relations, we define projection functions which allow each set domain to be expressed in terms of the others. Consider a relation $r \subseteq [a_1, b_1] \times [a_2, b_2] \times [a_3, b_3]$. The set it denotes must belong to the domain ΩD_S over which the computations are performed. Since ΩD_S contains convex sets, each value of a projection function must be a convex set, that is a set interval. Consequently, to each projection function designated by ρ_i we associate its closure $\overline{\rho_i}$. The closure is derived from ρ_i by making use of the closure operator defined above which satisfies:

$$\overline{\rho_i} = \vec{conv}(\rho_i)$$

$\overline{\rho_i}$ represents the approximation of this projection of the relational form r on the s_i-axis.

DEFINITION 22 *The i-th projection function $\overline{\rho_i}$ of a relation r denoting a set expression is the mapping :*
$\overline{\rho_i} = \vec{conv}\{s_i \in [a_i, b_i] \mid \exists (s_j, s_k) \in [a_j, b_j] \times [a_k, b_k]$ *such that* $j, k \neq i :\ (s_i, s_j, s_k) \in r\}$

These relational forms of set expressions are not visible to the user but they are necessary to define the local consistency of an n-ary constraint.

4.3. Consistency notions

The standard notions of consistency (Mackworth, 1977) applied to integer domains state conditions that must be satisfied by each element belonging to a variable domain. For example, arc-consistency states conditions that must be satisfied by each value belonging to a variable domain:

DEFINITION 23 *A binary constraint $c(x, y)$ such that $x \in D_x$ and $y \in D_y$ is* arc consistent *if and only if (1) for any value $i \in D_x$, there is a value $j \in D_y$ such that $c(i, j)$ is true, and (2) for any value $j \in D_y$, there is a value $i \in d_x$ such that $c(i, j)$ is true.*

This domain reasoning approach is not useful for set variables since set domains specified by set intervals can contain an exponential number of elements (e.g. the set interval $[\{\}, \{1, ..., n\}]$ contains 2^n elements). Instead, we derive conditions that must be satisfied by the set domain bounds. These conditions guarantee that a

constraint relation which does not hold for the bounds of the variable domains does not hold for any set between these bounds. For this purpose we define here the local consistency notions for each constraint appearing in an admissible system.

Consider a set variable s. The lower and upper bounds of the domain of s will be respectively defined by the functions $glb(s)$ and $lub(s)$.

DEFINITION 24 *Let $s_1 \subseteq s_2$ be a primitive set constraint. We say that this constraint is* locally consistent *if and only if:*

SC1. $glb(s_1) \subseteq glb(s_2)$ *and*

SC2. $lub(s_1) \subseteq lub(s_2)$.

Property 6 *A primitive set constraint is locally consistent if an only if it is arc-consistent.*

Proof: This property holds because the operations $\cup$ and $\cap$ are isotone. The domain constraint $s \in [a, b]$ is equivalent to $\forall e_s \in [a, b]$ we might have $s = a \cup e_s$. The isotony of $\cup$ means that $a \subseteq e_s \subseteq b \Rightarrow a \subseteq e_s \cup a \subseteq b$ (since $a \subseteq b$).

Assume the domain constraints $s \in [a, b]$, $s_1 \in [c, d]$. The set constraint $s \subseteq s_1$ is interval consistent iff:

$$\begin{aligned} a \subseteq c \text{ and } b \subseteq d \quad &\Leftrightarrow \quad \forall e_s \in [a, b]\ a \cup e_s \subseteq c \cup e_s \text{ and } b \cup e_s \subseteq d \cup e_s \\ &\Leftrightarrow \quad \forall e_s \in [a, b],\ \exists e_{s1} \in [c, d], e_{s1} = c \cup e_s \text{ such that } e_s \subseteq e_{s_1} \\ &\Leftrightarrow \quad s \subseteq s_1 \text{ is arc-consistent.} \end{aligned}$$

■

DEFINITION 25 *A primitive graduated constraint $f(s) \in [m, n]$ is* locally consistent *iff:*

SC3. $f(glb(s)) \leq m$ *and* $f(lub(s)) \geq n$

The local consistency of the relational forms of basic set expressions is defined through the consistency of the projection functions. Since the set domain of a basic set expression is approximated it is clear that we cannot get the equivalent of arc-consistency. Some elements in the resulting set interval are meant to fulfill "holes" and are not expected to be part of any feasible solution.

THEOREM 1 *A* relation *r denoting the relational form of a basic set expression is* locally consistent *if and only if each of the projection functions $\overline{\rho_i}$ describing r is locally consistent.*

DEFINITION 26 *A projection function $\overline{\rho_i}$ associated to the relation $r \subseteq \Pi_{j \in \{1,\ldots,3\}}[a_j, b_j]$ is locally consistent if and only if:*

SC4. $glb(\overline{\rho_i}) \subseteq a_i$ *and* $b_i \subseteq lub(\overline{\rho_i})$

4.4. Inference rules

The consistency notions define conditions to be satisfied by set domain bounds so that a set constraint is locally consistent. If such conditions are not satisfied

this means that elements in the domain are irrelevant. Local consistency can be inferred by moving such elements "out of the boundaries of the domain" which means pruning the bounds of the domain. The essential point is that a refinement of both bounds allows us to prune a domain. Reducing the set of possible values a set could take can be achieved either by extending the collection of *definite* elements of a set *i.e.*, adding elements to the glb of a set domain, or by reducing the collection of *possible* elements *i.e.*, removing elements from the lub of a set domain. Both computations are deterministic.

4.4.1. For set constraints

Consider the constraint $s \subseteq s_1$ such that $s \in [a,b], s_1 \in [c,d]$. Inferring its local consistency amounts to possibly extending the lower bound of the domain of s_2 and to possibly reducing the upper bound of the domain of s_1. This is depicted by the following inference rule:

$$\text{I1.} \quad \frac{b' = b \cap d \ , \qquad c' = c \cup a}{\{s \in [a,b],\ s_1 \in [c,d],\ s \subseteq s_1\} \longmapsto \{s \in [a,b'] \ , s_1 \in [c',d],\ s \subseteq s_1\}}$$

When s, s_1 denote set expressions, the relational forms are created and the following additional inference rule is necessary to deal with the projection functions. For each projection function $\overline{\rho_i}$ describing the domain of an s_i appearing in a set expression, we have:

$$\text{I2.} \quad \frac{a'_i = a_i \cup c \ , \qquad b'_i = b_i \cap d}{\{\ s_i \in [a_i,b_i],\ \overline{\rho_i} = [c,d]\ \} \ \longmapsto \{\ s_i \in [a'_i,b'_i]\}}$$

4.4.2. For primitive graduated constraints.

The constraint $f(s) \in [m,n]$ such that $s \in [a,b]$ describes a mapping from an element belonging to a partially ordered set to an element belonging to a totally ordered set. Consequently, it might occur that two distinct elements in $[a,b]$ have the same valuation in $[m,n]$. This implies that inferring the local consistency of this constraint might require refining $[a,b]$ only if a single element in $[a,b]$ satisfies the constraint. If this element exists, it corresponds necessarily to one of the domain bounds since they are uniquely defined and are strict subset (or superset), of any element in the domain. Thus, the value of the graded function mapped onto them cannot be shared. The inference mechanism is depicted by the following rules. $min()$ and $max()$ are functions which take as input a collection of integers and return respectively the minimal and maximal integer value of this collection.

$$\text{I3.} \quad \frac{[m',n'] = [max(m,f(a)), min(n,f(b))]}{\{\ s \in [a,b]\ , f(s) \in [m,n]\} \ \longmapsto \{s \in [a,b]\ , f(s) \in [m',n']\ \}}$$

$$\text{I4.}\quad \frac{n = f(a)}{\{\ s \in [a,b]\ , f(s) \in [m,n]\} \ \longmapsto \{s = a\ \}}$$

$$\text{I5.}\quad \frac{m = f(b)}{\{\ s \in [a,b]\ , f(s) \in [m,n]\} \ \longmapsto \{s = b\ \}}$$

4.4.3. For domain constraints

The inference rules defined here, describe the cases when two distinct set domains are applied to a single set variable, or when the set domain of a set variable is reduced to one value or is inconsistent.

$$\text{I6.}\quad \frac{a = b}{\{\ s_i \in [a,b]\ \} \ \longmapsto \{\ s = a\}} \qquad \text{I7.}\quad \frac{a \supset b}{\{\ s_i \in [a,b]\ \} \ \longmapsto fail}$$

$$\text{I8.}\quad \frac{a' = a \cup c\ , \quad b' = d \cap b}{\{s \in [a,b],\ s \in [c,d]\} \longmapsto \{s \in [a',b']\}}$$

Three similar inference rules exist for the integer domain constraints. They are not specific to our system but are recalled hereafter since we also deal with integer domains. The integer variable is specified by v.

$$\text{I9.}\quad \frac{m = n}{\{\ v \in [m,n]\ \} \ \longmapsto \{\ v = m\}} \qquad \text{I10.}\quad \frac{m > n}{\{v \in [m,n]\} \ \longmapsto fail}$$

$$\text{I11.}\quad \frac{m' = max(m, m_1)\ , \quad n' = min(n, n_1)}{\{v \in [m,n],\ v \in [m_1,n_1]\} \longmapsto \{v \in [m',n']\}}$$

4.4.4. Properties of the inference rules

The behaviour of all the inference rules I1 to I11 is captured by the following scheme.

Let us denote a set/graduated constraint relation by c and its arity by k. Let us represent an inference rule as a mapping from a Cartesian product of set/integer domains, onto another Cartesian product.

Let $\Pi_{j\in\{1,..,k\}}\ [a_j,b_j], \Pi_{j\in\{1,..,k\}}\ [a'_j,b'_j]$ be two distinct Cartesian products of the domains of the variables appearing in c. These Cartesian products can be made into ordered sets by imposing the strict set inclusion ordering defined by:

$$\Pi_{j\in\{1,..,k\}}[a_j,b_j] \subset \Pi_{j\in\{1,..,k\}}[a'_j,b'_j] \Leftrightarrow \forall j \in \{1,..,k\},\quad [a_j,b_j] \subset [a'_j,b'_j]$$

i.e., all the elements in $[a_j,b_j]$ are in $[a'_j,b'_j]$.

An inference rule $\overline{\rho}$ applied to a constraint relation c maps a Cartesian product $\Pi_{j\in\{1,..,k\}}[a_j, b_j]$ onto a newly computed Cartesian product of domains. Each new domain is the output of a projection $\overline{\rho_i}$ of $\overline{\rho}$ onto the i-axis (cf. Definition 22). A projection function $\overline{\rho_i}$ derives a new domain by intersecting c with $\Pi_{j\in\{1,..,k\}}[a_j, b_j]$, projecting the result back onto the i-axis, and computing the convex closure of this projection. Thus, an inference rule is defined in algebraic terms by:

$$\overline{\rho}(\ \Pi_{j\in\{1,..,k\}}\ [a_j, b_j]) = \quad \Pi_{i\in\{1,..,k\}}\ \overline{\rho_i}\ (\Pi_{j\in\{1,..,k\}}[a_j, b_j])$$

One can easily see that this generic procedure (and thus each inference rule) is: (i) *correct* (all possible solutions are kept) since only irrelevant values are removed from the domains, (ii) *contracting* (final domains are subset of the initial domains), since the domains can only get refined, and (iii) *idempotent* (the smallest domains have been computed the first time), since every element that can be removed has been removed the first time.

Moreover, an inference rule $\overline{\rho}$ applicable to c is *inclusion monotone* if:

$$\Pi_{j\in\{1,..,k\}}[a_j, b_j] \subset \Pi_{j\in\{1,..,k\}}[a'_j, b'_j] \Rightarrow \overline{\rho}(\Pi_{j\in\{1,..,k\}}[a_j, b_j]) \subset \overline{\rho}(\Pi_{j\in\{1,..,k\}}[a'_j, b'_j])$$

This means that smaller initial domains yield smaller final domains.

LEMMA 1 *The inference rules are inclusion monotone.*

Proof: The monotonicity property of the inference rules follows from that of the projection functions.

Assume that $\forall j \in \{1,..,k\} : [a_j, b_j] \subset [a'_j, b'_j]$. Each projection function $\overline{\rho_i}$ ($i \in \{1,..,k\}$) is monotone since the set intersection is isotone (Property 2) and the convex closure operation is monotone (Property 4). This implies that:
$\forall i \in \{1,..,k\} :\ \vec{conv}\ ((\ c \cap \Pi_{j\in\{1,..,k\}}[a_j, b_j])_i) \subset \vec{conv}\ ((\ c \cap \Pi_{j\in\{1,..,k\}}[a'_j, b'_j])_i)$
which is equivalent to:
$\forall i \in \{1,..,k\} :\ \overline{\rho_i}\ (\Pi_{j\in\{1,..,k\}}[a_j, b_j]) \subset \overline{\rho_i}\ (\Pi_{j\in\{1,..,k\}}[a'_j, b'_j])$, and consequently to:
$\overline{\rho}(\Pi_{j\in\{1,..,k\}}\ [a_j, b_j])\ \subset\ \overline{\rho}(\ \Pi_{j\in\{1,..,k\}}\ [a'_j, b'_j])$. Thus $\overline{\rho}$ is monotone. ■

4.5. Operational semantics

The inference rules described so far can be applied to individual constraints. The operational semantics shows how to check and infer the consistency of a system of constraints. This system should correspond to an admissible system of constraints. The consistency of such a system results from the consistency of each constraint appearing in it. The operational semantics is described by the following algorithm.

Let a tuple $(c, \vec{s}, \vec{d_s})$ denote a constraint c over a set of variables designated by $\vec{s}$ where each variable s_i is constrained by a domain constraint d_{s_i}. The set of relevant domain constraints with respect to $\vec{s}$ is designated by $\vec{d_s}$. The initial set of constraints to be considered is designated by G. The set of domain constraints is designated by A. A set C which represents the constraint store contains the

constraints whose consistency has been checked. The operational semantics is based on one non deterministic transition rule which takes as input one constraint c in G and applies to it the adequate local inference rule using a depth first search strategy. Each constraint c is determined to be locally consistent if the inference rule infers consistent domains. This might require some domain refinements and consequently a need to reconsider some constraints in C whose variables intersect with those in c. Such constraints are moved from C to G. The constraint c is then added to the constraint store C and another constraint is selected in G. The last state of the resolution is reached once no goal remains in G, or when a failure is encountered (*i.e.*, at least one set domain $[a, b]$ or integer interval $[m, n]$ is such that $a \not\subseteq b$ or $m \not\leq n$). The general schema of the algorithm is depicted in the following figure.

begin
  Initialize G to the set of all the constraints in the admissible system
  Initialize C to the empty set
  Initialize A to the empty set
  while G is not empty **do**
    begin
      select and remove a constraint $(c, \vec{s})$ from G
      select and remove the relevant domain constraints $\vec{d_s}$ in $G \cup A$
      apply the adequate inference rule on $(c, \vec{s}, \vec{d_s})$ which returns $(c, \vec{s}, \vec{d_s}')$
      if $\vec{d_s}'$ is inconsistent **then**
        exit with failure
        else if $\vec{d_s} \neq \vec{d_s}'$ **then**
          begin
            $\vec{d_s} \leftarrow \vec{d_s}'$
            for each $(p, \vec{v})$ in C **do**
              if $\vec{s} \cap \vec{v} \neq \emptyset$ **then**
                remove $(p, \vec{v})$ from C and add it to G
          end
        if $\vec{d_s} \cap G \neq \emptyset$ **then** remove the domain constraints in $\vec{d_s} \cap G$ from G
        and add them to A
        add $(c, \vec{s})$ to C.
    end
end

This whole process amounts to considering a transition system on states where each state contains the constraints as yet unconsidered and the constraints which have already been checked out. One state i is specified by the tuple $\langle G_i, A_i, C_i \rangle$. The initial state of the transition system is specified by the tuple $\langle G_0, \emptyset, \emptyset \rangle$ where all the constraints need to be checked. The final state is either *fail* or $\langle \emptyset, A', C' \rangle$.

THEOREM 2 *A transformed system of constraints* $\langle \emptyset, A, C \rangle$ *is locally consistent if and only if each domain constraint in* A *is locally consistent.*

Proof: This follows simply from the various inference rules. Inferring the consistency of a system amounts to considering the consistency of each constraint in conjunction with the already consistent ones. An inconsistency is detected if one of the inference rules I7 or I10 is successfully applied which means a failure is encountered in one (integer, set) domain. ■

This algorithm resembles the relaxation algorithm used by CLP(Intervals) systems (Lee and van Emden, 1993) also referred to as fixed point algorithm in (Benhamou et al., 1994, Benhamou, 1995) All of those can be seen as an adaptation of the AC-3 algorithm (Mackworth, 1977) where domains are specified by intervals. The only difference between the algorithms lies in the inference rules applied. The generic algorithm satisfies the following properties of fixed point algorithms : termination, existence of a unique fixed point independent of the constraint ordering, and correctness.

THEOREM 3 *The algorithm always terminates.*

Proof: This comes from the fact that the domains are finite and can only get refined (contractance property of the inference rules). Also, if a failure is detected, the algorithm terminates with $fail$. ■

THEOREM 4 *The algorithm has a unique fixed point independent of the ordering of the inference rules.*

Proof: (Older and Vellino, 1993) proved that propagation methods based on the AC-3 algorithm compute a unique fixed point independent of the ordering of the inference rules, if the states of the iteration process can be ordered within a lattice and if the inference rules applied are contracting, idempotent and inclusion monotone. They show that the contractance and idempotence properties guarantee the existence of a fixed point. In addition, due to the monotonicity of the inference rules, the fixed point is unique and independent of the ordering of the inference rules.

In our case, the only things that change during our iteration process are the bounds of the domains. Thus the states can be characterized by the set of domains. The domains are partially ordered by the set inclusion within the lattice of set and integer domains $\Omega D_S \cup \Omega N$. Additionally, we have shown (section 4.4.4) that the contractance, idempotence and inclusion monotone properties are satisfied by our inference rules. Thus, the generic algorithm has a unique fixed point independent of the ordering of the inference rules. ■

THEOREM 5 *If a solution exists, it can be derived from the consistent system of constraints.*

Proof: This follows directly from the monotonicity of the convex closure operation and the correctness of the inferences rules applied. Monotonicity guarantees that the actual value of a set or integer lies in the approximated domains. Moreover, the inference rules are correct, so all possible solution values are kept. ■

4.6. Satisfiability issue

Ensuring the satisfiability of a consistent system requires guaranteeing that a solution exists. This is not possible when both symbols $\cup$ and $\cap$ belong to some n-ary constraints since we work on domain approximations. However satisfiability can be guaranteed in some particular cases which are of practical interest (eg. for constraints of the form $s_1 \cap s_2 = \emptyset$). The following properties give the equivalences and/or implications which exist between the lower and upper bounds of a set expression domain and the lower and upper bounds of the set variables invoked.

Properties 7 *(Pawlak, 1991)*

1. $glb(s_1) \subseteq s_1 \subseteq lub(s_1)$
2. $lub(s_1 \cup s_2) = lub(s_1) \cup lub(s_2)$
3. $glb(s_1 \cap s_2) = glb(s_1) \cap glb(s_2)$
4. $lub(s_1 \cap s_2) \subseteq lub(s_1) \cap lub(s_2)$
5. $glb(s_1 \cup s_2) \supseteq glb(s_1) \cup glb(s_2)$

Properties 7-2 and 7-5 show respectively that the union operation preserves the upper bounds but not the lower bounds. By duality, properties 7-3 and 7-4 show respectively that the intersection preserves the lower bounds but not the upper bounds. This means that inferring the local consistency of an n-ary constraint containing only the set union symbol is achieved by computing the exact upper bounds of each set variable and by approximating the lower bounds of the set variables using the set interval calculus. The dual case is considered for a n-ary constraint containing the set intersection symbol. Consequently, we have the following properties:

Property 8 *Let $s_1 \cap s_2 = s_{12}$ be the relational constraint associated to the set expression $s_1 \cap s_2$. If this relational constraint is locally consistent then we have $glb(s_1) \cap glb(s_2) = glb(s_{12})$.*

Property 9 *Let $s_1 \cup s_2 = s_{12}$ be the relational constraint associated to the set expression $s_1 \cup s_2$. If this constraint is locally consistent then we have $lub(s_1) \cup lub(s_2) = lub(s_{12})$.*

With respect to the primitive set inclusion constraint $s_1 \subseteq s_2$, we have proved at an earlier stage that if this constraint is locally consistent then it is arc-consistent (cf. property 6). In other words, the domain bounds are possible values for the set variables as well as any set value between the bounds.

THEOREM 6 *A locally consistent system built from set domain constraints, primitive set inclusion constraints and relational constraints containing either the union or intersection symbol is satisfiable if the domain constraints embedded in the system are satisfiable.*

Proof: Clearly, if some set domain constraints are not locally consistent, the system is not consistent and a fortiori not satisfiable. Otherwise, it is always possible to construct a solution to this system. By property 8, all the relational constraints of the form $s_1 \cap s_2 = s_{12}$ are true if we assign to each set variable the lower bound of their domain. These assignments also hold for the primitive set inclusion constraints. By property 9, all the constraints of the form $s_1 \cup s_2 = s_{12}$ are true if we assign to each set variable the upper bound of their domain. These assignments also hold for the primitive set inclusion constraints. Thus in either of the two consistent systems of constraints we guarantee that a solution exists. ■

Note that a system containing both forms of relational constraints can be locally consistent 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. With respect to graduated constraints, consistency does not guarantee satisfiability since a consistent graduated constraint $f(s) = m$ does not guarantee that some elements of the domain of s might satisfy the constraint. The satisfiability for systems containing such constraints is not provable unless the solver performs exhaustive computations at an exponential cost in the largest upper bound among the set domains.

Example: Consider the system of constraints:
$s_1, s_2, s_3 \in [\{\}, \{1,2,4,5\}]$, $s_1 \cup s_2 = s_{12}$, $s_{12} \cup s_3 = \{1,2,4,5\}$,
$s_1 \cap s_2 = s_{21}$, $s_{21} \cap s_3 = \{\}$
It is locally consistent but not satisfiable. No possible value for each set variable leads to a solution. □

II Practical Framework

The formal framework has given us the structure of a set-based system whose solver is based on consistency techniques. It constitutes the basis of the design of a practical language called Conjunto (Conjunto means "set" in Spanish). Conjunto is a constraint logic programming language designed and implemented to reason with and about sets ranging over a set domain. Its functionalities (apart from those of a logic-based language like Prolog (Colmerauer et al., 1983)) are set operations and relations from set theory together with some graded functions which provide set

measures like cardinality, weight, etc. The graded functions map set domains to subsets of the natural numbers (finite domains). This requires from an implementation point of view to establish a cooperation between two solvers (the set constraint solver of Conjunto and a finite domain solver). In this part, we describe the implementation of Conjunto which raises among others the issues of (1) this cooperation between two solvers, (2) the dynamic handling of a system of constraints by means of delay mechanisms, (3) the specific set data structure which is required to attach all the relevant information related to a set variable, (4) the way set calculus is achieved in algorithmic terms. Since Conjunto aims at solving set-based combinatorial search problems, the local consistency ensured by the solver via some local transformation rules should be enriched by a labelling procedure in order to reach a complete solution. This procedure is described together with some programming facilities which enhance the expressive power of the language.

5. Design of Conjunto

We describe the functionalities of the Conjunto language and omit a detailed description of the traditional predicates and functions on Prolog terms (Colmerauer et al., 1983).

5.1. Syntax

The Conjunto language is a logic-based programming language with the alphabet of a Prolog language (constants, predicates, functions, connectives, etc). It is characterized by a signature Σ which contains the following set of predefined function and predicate symbols in their concrete syntax:

- the constant `{}`.
- the binary set predicate symbols {`` `< ``, `` `<> ``, `` `:: ``, `#`, `weight`} and arithmetic predicate symbols $\{=, \geq, \neq\}$.
- the binary set function symbols {`\/`, `/\`, `\`} and the arithmetic sum symbol `+`.

DEFINITION 27 *(Lloyd, 1987) An* atomic formula *(or atom) is defined as follows: If p is an n-ary predicate and $t_1, ..., t_n$ are terms, then $p(t_1, ..., t_n)$ is an atom.*

The atoms which are built from set terms and predefined predicate symbols in Σ are called constraints. They are subject to a specific interpretation in Conjunto.

A program built from the language is based on definite clauses of the form:

$$(1) \quad a : -b_1, ..., b_n \quad \text{and} \quad (2) \quad : -g_1, ..., g_n$$

where a is an atom and the b_i, g_i are atoms or constraints. (1) is called a program clause and (2) a program goal. The constraints constitute the core functionalities of the language and are characterized by a specific terminology and semantics.

5.2. Terminology and semantics

The main objective of Conjunto is to perform set calculus over sets defined as elements from a powerset domain. Some constraints like set cardinality or set weight require us to deal also with finite domains, that is integers and arithmetic constraints.

DEFINITION 28 *The computation domain is the set* $\mathcal{D} = \mathcal{P}(H_u) \cup H_u$ *where* $\mathcal{P}(H_u)$ *is the powerset of the Herbrand universe.*

5.2.1. Terminology

The terminology gives names to the predicate and function symbols in Σ and defines the notions of set domains and set terms necessary to reason with and about sets in $\mathcal{D}$.

The symbols in {`<, `<>, `::, #, weight} refer respectively to the set inclusion constraint predicate, the set disjointness constraint predicate, the set domain constraint, the set cardinality constraint predicate and the weight constraint predicate. The symbols in {\/, /\, \} represent the concrete syntax of the set operations $\cup, \cap, \setminus$. They will be interpreted in their usual set theoretical sense; the set difference is a complementary difference (e.g. $s \setminus s_1 = \{x \in s \mid x \notin s_1\}$). The other symbols in Σ refer simply to the arithmetic operations they denote.

DEFINITION 29 *A* ground set *is an element of* $\mathcal{P}(H_u)$ *which represents a finite set of Herbrand terms delimited by the characters* { *and* }.

Example: `{2,3,f(f(u,o))}` *is a ground set.* □

DEFINITION 30 *A set variable is any variable taking its value in* $\mathcal{P}H_u$.

DEFINITION 31 *A* set term *is defined by:*
(1) *any* set constant *a is a set term*
(2) *any* set variable *s is a set term*

The concepts of set domain and set expressions are those from the formal description.

The syntax of a set variable is $s = s$`{[a,b]}` where s`{[a,b]}` denotes the domain attached to a variable s. We introduce a new concept which is that of weighted set domain.

DEFINITION 32 *A* weighted set domain *is a specific set domain where each element of the set domain bounds has the syntax* (e, m) *such that* e *is a Herbrand term and* m *is a positive integer.*

Example: `S= S{[{(a,1)},{(a,1),(c,2),(d,2)}]}` *is a set variable whose weighted set domain is the set interval* `[{(a,1)},{(a,1),(c,2),(d,2)}]`. □

Similarly, variables denoting integers will take their value in a finite set of integers (finite domain). In Conjunto these domains are approximated by integer interval domains. An *integer interval domain* is the convex closure of a finite set of integers and will be simply referred to as an integer interval.

DEFINITION 33 *An* integer variable *is a logical variable whose value lies in an integer interval.*

Notation. Conjunto's predicate and function symbols are written in a bold font. Set variables are denoted by s, v, w, set expressions t, integer variables are denoted by x, y, z, ground sets a, b, c, d, integers m, n. These symbols may be subscripted.

5.2.2. Semantics

The interpretation of the elements of Σ in $\mathcal{D}$ is given by distinguishing set constraints from graduated constraints.

A *primitive set constraint* is one of the following constraints:

- s `'::' $[a, b]$` is semantically equivalent to $a \subseteq s \subseteq b$ (cf. the $\in_{[a,b]_{a \subseteq b}}$ predicate in the formal part).
- s `'<' $s_1$` is equivalent to the set inclusion relation $s \subseteq s_1$.
- s `'<>' $s_1$` is equivalent to the empty intersection of the two sets s, s_1.

Note that the set equality can be derived from the double inclusion:
s `'=' $s_1$` $\Leftrightarrow$ s `'<' $s_1$` and s_1 `'<' $s$`.

Remark The set disjointness constraint `'<>'` which was not included in the formal part has been embedded as a primitive constraint in Conjunto mainly for practical reasons. Since the disjointness of two sets appears in almost all set based problems, it is simpler to use a specific syntax and more efficient to handle it as a primitive constraint.

A *primitive graduated constraint* is one of the following:

- `#`(s, x) is equivalent to the arithmetic equality $\#s = x$ where $\#s$ is the standard cardinality function of set theory.
- `weight`(s, x) is semantically equivalent to the arithmetic operation $\sum_i m_i = x$ such that $(e_i, m_i) \in s$.

DEFINITION 34 *The constraint system of a* Conjunto program *is an admissible system (cf. definition 20) of set constraints and graduated constraints where every set variable is constrained by a set domain constraint.*

In this admissible system of constraints the searched objects are the sets. The integer variables are not part of the initialization of the search space which is attached to the system. They constitute essentially a means to get to the final solution. This is described in the following definition.

DEFINITION 35 *A* set domain constraint satisfaction problem *is an admissible system of set and graduated constraints,* i.e. *a constraint satisfaction problem where the initial search space is defined by the set domains attached to the set variables.*

5.3. Constraint solving

The constraint solving in Conjunto focuses on efficiency rather than on completeness. The Conjunto solver based on the fixed point algorithm presented earlier aims at checking and inferring the consistency of an admissible system of constraints. This is achieved by:

- applying some local transformation rules, which allow for the consistency of one constraint to be checked/inferred, using a top-down search strategy,
- delaying consistent constraints which are not completely solved.

The Conjunto solver considers one constraint at a time and checks/infers its consistency in conjunction with the set of delayed constraints (constraint store). This process might require the local consistency of some delayed constraints to be reconsidered. These constraints are woken using a data driven mechanism based on suspension handling mechanisms. Each newly consistent constraint is added to the constraint store. The final state of the program is achieved when all atoms appearing in a goal clause have been checked and when no further domain refinement is required. This state is either denoted by "fail" when some constraints have been marked inconsistent or it contains a set of delayed constraints together with the set variables and their associated domains.

Example: The goal:

```
:- S `:: [{1},{1,2,3,4}], S1 `:: [{3},{1, 2, 3}], S `< S1.
```

produces the refined domains:

```
S = S{[{1},{1,2,3}]}  S1 = S1{[{1,3},{1,2,3}]}
```

and the delayed goal: `S `< S1` □

Example: The goal:

```
:- S `:: [{1},{1,2,3,4}], #(S,1).
```

produces the instantiation `S = {1}` and no delayed goal since the initial goal is completely solved. □

5.4. Programming facilities

One of the application domains we have investigated using Conjunto is the modelling and solving of set based combinatorial problems (e.g. set partitioning, bin packing, hypergraph computations). To allow the user to state short and concise programs, some programming facilities have been added to the initial set of primitive constraints. They consist of a collection of constraints defined from the primitive ones, some predicates necessary to access information related to the variable domains, and a built-in set labelling procedure. The most important ones are presented below, others are described in (Gervet, 1995).

5.4.1. Set domain access

Set domains are represented as abstract data types, and the users are not supposed to access them directly. So two predicates are provided to allow operations on set domains : `glb`(s, s_{glb}) and `lub`(s, s_{lub}). If s denotes a set variable, each term is respectively assigned the value of the domain's lower and upper bound. Otherwise it fails. Similar predicates are defined to access integer domain bounds: `min`(x, x_{min}) and `max`(x, x_{max}).

5.4.2. Set labelling

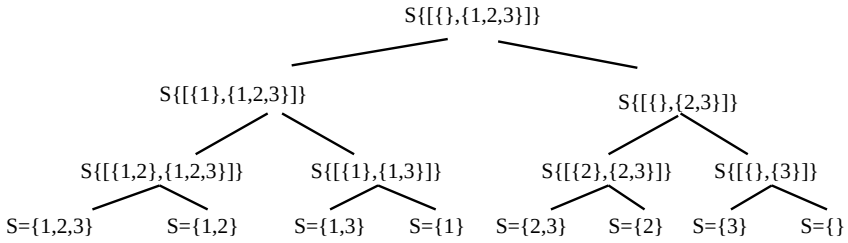
Assigning a value to a set variable is a nondeterministic problem which can be tackled by different labelling strategies. Since the Conjunto solver uses local consistency techniques, an adequate strategy should aim at making an active use of the constraints in the constraint store. On the one hand, a procedure which would consist in instantiating a set by directly selecting an element from the set domain makes a passive use of the constraints whose consistency is only local. In the worst case this process might require considering all the elements belonging to a set domain even if some of them are irrelevant. On the other hand refining a set domain by adding one by one elements to the lower bound of the domain is more likely to minimize the possible choices to be made. The `refine` predicate embedded in Conjunto behaves as follows:
`refine`(s) labels s, if s is a set variable. If there are several instances of s, it creates choice points. If s is a ground set, nothing happens. If not, the following actions are performed recursively until the set gets instantiated: (1) select an element e from the ground set $lub(s) \setminus glb(s)$, (2) add the membership constraint e `in` s to the program. This added constraint is handled by the solver which checks its consistency in conjunction with the actual constraint store. In case of failure the program backtracks and (3) the nonmembership constraint is added (successfully) to the program so as to remove the irrelevant value e from the domain. The points (2) and (3) correspond to the disjunctive set of constraints:

```
( e in S ; e notin S)
```

Example: Consider the goal:

```
:- S ‘:: [{},{1,2,3}], refine(S).
```

The search tree generated during the labelling procedure and covered using a depth first search strategy is described in the following figure.



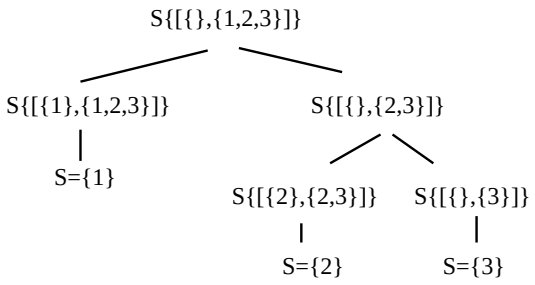
□

The strategy, which consists in adding membership constraints to the program, aims in particular at making an active use of those graduated constraints whose consistency is only local.

Example: Consider the goal:

```
:- S ‘:: [{},{1,2,3}], #(S,1), refine(S).
```

The irrelevant branches of the search tree are cut in an a priori way *i.e.*, no useless choice point is created. The search tree generated during the solving of this goal is depicted in the following figure.



□

5.4.3. Optimization predicates

The notion of optimization is common in problem solving. It aims at minimizing or maximizing a cost function which denotes a specific arithmetic expression. The

notion of cost defines a kind of measure or quantification applied to some terms. A set can not denote a quantity and is not measurable. Only its possible graded functions are. Thus there are no specific optimization predicates for sets. Existing predicates embedded in a finite domains solver (e.g. for a branch and bound search) can be directly applied to expressions over integer intervals occurring in graduated constraints. For example, minimizing a set cardinality acts over a set through the link existing between a set variable and its cardinality.

5.4.4. Relations and constraints

When dealing with sets, it sounds quite natural to deal with relations and functions as well. Functions are more restrictive than relations since they constrain each element from its DS-domain (DS-domain stands here for departure set) to have exactly one image. Providing relations at the language level extends the expressive power of the language when dealing for example with circuit problems and matching problems originating from Operations research. In relation theory (Fraissé, 1986), a relation $\mathcal{R}$ is represented as a set of ordered pairs (x_i, y_j) such that x_i belongs to the DS-domain d of $\mathcal{R}$ and y_j to its AS-range (AS-range stands here for arrival set) a. In other words, a relation $\mathcal{R}$ on two ground sets d and a is a subset of the Cartesian product $d \times a$. Keeping this representation to deal with relations as specific set terms containing pairs of elements can be very costly in memory. Indeed, the statement of the Cartesian product referring to a relation requires us to consider explicitly a huge set of pairs. This is very inconvenient. Instead, a relation in Conjunto is represented as a specific data structure which is characterized by two ground sets (DS-domain and AS-range) and a list containing the successor sets attached to each element of DS-domain (Gervet, 1993, Gervet, 1993a). Considering one successor set per element splits the domain of a relation into a collection of set domains. The resulting value of a relation is clearly the union of the successor sets. This approach is close to the one introduced in the seminal work ALICE (Laurière, 1978) which dealt essentially with functions. However in ALICE there is no explicit notion of set domain.

DEFINITION 36 *Let a relation be $r \subseteq d \times a$. The successor set s of an element $x \in d$ is the set $s = \{y \in a \mid (x, y) \in r\}$.*

DEFINITION 37 *A relation variable r is a logical variable whose value is a compound term* `birel`(l, d, a) *such that* `birel` *is a functor of arity three, l is a list of $\#d$ set variables s_i such that s_i* `':: [{}, a]` *and d, a are two ground sets.*

This compound term is associated to a free variable by means of the predicate r `bin_r` d `-->` a.

Example: The goal:

```
:- R bin_r {1,2} --> {a,b,c}.
```

creates the term:

```
R = birel([Set1{[{},{a,b,c}]}, Set2{[{},{a,b,c}]}], {1,2}, {a,b,c})
```

□

The definition of constraints applied to relation variables abstracts from stating directly constraints over the set DS-domain and AS-range or over the successor sets. The following constraints have been embedded in Conjunto:

- (i, j) `in_r` r, (i, j) `notin_r` r which adds or retrieves pairs to the relation
- `funct`(r) which constrains a relation to be a function,
- `inj`(r) which constrains a relation to be an injective function,
- `surj`(r) which constrains a relation to be a surjective function,
- `bij`(r) which constrains a relation to be a bijective function.

The schema of these constraints is directly derived from their usual interpretation issued from relation theory (Fraissé, 1986). They are represented below using the mathematical cardinality operation #, the usual set operation symbols ($\cup, \cap$) and the arithmetic inequality ($\geq$).

Constraints	Interpretation
r `bin_r` d `-->` a	$r =$ `birel`(l, d, a) where $l = \{s_i \mid \forall i \in d,\ s_i\ \in \{\}..a\}$
(i, j) `in_r` r	if $i \in d, j \in a$ then $j \in s_i$
(i, j) `notin_r` r	if $i \in d, j \in a$ then $j \notin s_i$

Constraints	Interpretation
`funct`(r)	$\forall i \in d,\ \#s_i = 1$
`inj`(r)	$\#d \leq \#a,\ \#d = n$ $s_1 \cap s_2 = \emptyset,\ s_1 \cap s_3 = \emptyset,\ ...,\ s_{n-1} \cap s_n = \emptyset$ $\forall i \in d,\ \#s_i = 1$
`surj`(r)	$\#d \geq \#a,\ \#d = n$ $s_1 \cup s_2 ... \cup s_n = a$ $\forall i \in d,\ \#s_i = 1$
`bij`(r)	$\#d = n,\ \#a = n$ $s_1 \cap s_2 = \emptyset,\ s_1 \cap s_3 = \emptyset,\ ...,\ s_{n-1} \cap s_n = \emptyset$ $\forall i \in d,\ \#s_i = 1$

These schema tell us how each constraint over a relation is described and implemented in Conjunto by means of set and graduated constraints. These constraints over relations do not require any specific solver since the reasoning is based on

the successor set variables. The Conjunto solver is simply used. The expressivity of these relation variables and constraints is illustrated in the set partitioning application presented subsequently.

Example: The goal:

```
:- R bin_r {1, 2} --> {a, b, c}, funct(R).
```

creates the term:
`R = birel([Set1{[{},{a,b,c}]}, Set2{[{},{a,b,c}]}], {1,2}, {a,b,c})`
and the list of delayed goals:
`#(Set1{[{},{a,b,c}]}, 1), #(Set2{[{},{a,b,c}]}, 1)` □

Since the created compound term is not visible to the user, a collection of predicate relations allows him/her to access the properties of the relation:

- `succs`(r, l) instantiates l to the list of successor sets of r.
- `dom`(r, s) instantiates s to the DS-domain of r.
- `ran`(r, s) instantiates s to the AS-range of r.
- `succ`(r, e, s) instantiates s to the successor set of the element e belonging to DS-domain, such that $s = \{x \mid (e, x) \in r\}$.

6. Implementation of Conjunto

The implementation of Conjunto was done in the ECL^{i}PS^e (ECRC, 1994) system which extends the plain Prolog language with features dedicated to the implementation of specific constraint solvers. The main features provided at the language level comprise the attributed variable data structure and the suspension handling predicates. An attributed variable is a special data type (Le Huitouze, 1990, Holzbaur, 1992) which consists of a variable with a set of attributes attached and whose behaviour on unification can be explicitly defined by the user in a way that differs from Prolog unification. Attributed variables aim at dealing with specific computation domains distinct from the Herbrand universe. The suspension handling predicates provide means to (1) delay a goal or constraint, (2) store it in a specific list with respect to one or several variables, (3) awake a list of delayed goals when some given conditions are satisfied. The suspension handling predicates allowed us to implement the data driven constraint handling in Conjunto. In addition, the Conjunto solver makes use of the finite domain library of ECL^{i}PS^e to deal with integer interval terms (as well implemented as attributed variables).

6.1. Set data structure

A set variable is not represented as a standard Prolog variable, but as an attributed variable which is subject to a dedicated unification algorithm. The internal representation of ground sets is also given since it influences the time complexity of the

transformation rules. Both the data structure and the internal representation of ground sets are not visible to the user and will be ignored in the description of the transformation rules.

6.1.1. Set variable representation

A set variable is an attributed variable comprising the following list of attributes. This structure stores for each set variable all the necessary information regarding its domain, cardinality, and weight (null if undefined) together with three suspension lists. The attribute arguments have the following meaning:

- **setdom:** [**Glb,Lub**] represents the set domain. The user can access it using the built-in predicates `glb`, `lub`.
- **card:** **C** represents the set cardinality. This attribute **C** is initialized as soon as a set domain is attached to a variable. It is either an integer interval or an integer. It can be accessed and modified using specific built-in predicates from a finite domain library.
- **weight:** **W** represents the set weight. **W** is intialized to zero if the domain is not a weighted set domain, otherwise it is computed as soon as a weighted set domain is attached to a set variable. It can be accessed and modified using specific built-in predicates from a finite domain library.
- **del_glb:** **Dglb** is a suspension list that should be woken when the lower bound of the set domain is updated.
- **del_lub:** **Dlub** is a suspension list that should be woken when the upper bound of the set domain is updated.
- **del_any:** **Dany** is a suspension list that should be woken when any set domain refinement is performed.

6.1.2. Ground set representation

The choice for the internal representation of sets is independent of the algorithms, and not visible to the user. However, it plays a role in the time complexity of the different set operations. In contrast to integer intervals, the time complexity for operations on ground sets ($+$, $-$ versus $\cup$, $\cap$, $\setminus$) can not be considered as constant for it closely depends on the internal representation of a set. In Conjunto each ground set is represented by a sorted list where the time complexity for any set operation ($\cup$, $\cap$, $\setminus$) is bounded from above by $\mathcal{O}(d)$ where d is $\#lub(s) + \#glb(s)$ and s the set with the largest domain.

Since we work essentially on set domains, another approach has been tried out which consists in representing a set domain as a boolean vector mapped onto a list

containing the actual value of the elements. The upper bound is specified by the set of elements whose corresponding 0-1 variable has the value 1 or 0-1 (undetermined). The lower bound is specified by the set of elements whose corresponding 0-1 variable has the value 1. This approach 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 this leads to much larger memory usage due to the size of the domains used in practice and to the handling of two lists (the list of 0-1 variables and the list of actual values).

From now on, the value of d in the complexity results will always stand for $\#lub(s) + \#glb(s)$.

6.2. Set unification procedure

A Conjunto program attaches a specific semantics to set terms. This semantics requires to extend the Prolog unification to the one of set terms. The behaviour of the set unification procedure comprises the following tests and inferences:

- the unification of a logical variable and a set variable. The logical variable is bound to the set variable.
- the unification of a ground set and a set variable. The set variable is instantiated to the ground set if it belongs to its domain.
- the unification of two set variables. The two variables are bound to a new variable whose domain is the convex intersection of the two domains (cf. set interval calculus). If this domain is empty the unification fails.
- the unification of a set variable with any other term fails.

6.3. Local transformation rules

Consistency notions and inference rules have been defined in the formal part for primitive set constraints and for the general case of projection functions and graduated constraints respectively. Here, we make use of these definitions to define the transformation rules which check and infer the local consistency of each primitive constraints implemented in Conjunto. The basic idea consists in pruning the set domains attached to the set variables by removing set values which can never be part of any feasible solution. Set values are removed by adding elements to the lower bound of the domain and/or by removing elements from the upper bound.

6.3.1. Transformation rules for primitive set constraints

Primitive set constraints are s `'<` s_1 and s `'<>` s_1 where s and s_1 denote set variables ranging over a set domain.

Consider the set inclusion constraint s_1 `'<` s_2 such that $s_1 \in d_1$, $s_2 \in d_2$. The transformation rule makes use of the lower and upper ordering of the set inclusion. Making this constraint consistent might require adding elements to the lower bound of the domain d_2 and removing elements from the upper bound of d_1. The refinements lead to the new domain bounds:

T1.	$\mathtt{glb}(d_1')$	$\leftarrow \mathtt{glb}(d_1)$	$\mathtt{lub}(d_1')$	$\leftarrow \mathtt{lub}(d_1) \cap \mathtt{lub}(d_2)$
T2.	$\mathtt{glb}(d_2')$	$\leftarrow \mathtt{glb}(d_2) \cup \mathtt{glb}(d_1)$	$\mathtt{lub}(d_2')$	$\leftarrow \mathtt{lub}(d_2)$

Consider the disjointness constraint s_1 `'<>` s_2 such that $s_1 \in d_1$, $s_2 \in d_2$. The only possible refinement aims at removing elements from each upper bound of a set domain which are definite elements of the other set. This constraint is locally consistent if the refined domains for the variables are:

T3.	$\mathtt{glb}(d_1')$	$\leftarrow \mathtt{glb}(d_1)$	$\mathtt{lub}(d_1')$	$\leftarrow \mathtt{lub}(d_1) \setminus \mathtt{glb}(d_2)$
T4.	$\mathtt{glb}(d_2')$	$\leftarrow \mathtt{glb}(d_2)$	$\mathtt{lub}(d_2')$	$\leftarrow \mathtt{lub}(d_2) \setminus \mathtt{glb}(d_1)$

Complexity issues. The time complexity for each transformation is bounded by $\mathcal{O}(d)$ since only one set operation is applied each time.

6.3.2. Projection functions for n-ary constraints

Constraints over set expressions require a special handling mechanism if we want to express each set variable in terms of the others involved in a constraint. This point requires us to tackle these n-ary constraints as "mini-programs". The approach implemented in Conjunto consists in approximating an n-ary constraint by (1) associating each basic set expression (s_1 `\/` s_2, s_1`/\` s_2, s_1`\`s_2) with its relational form, (2) applying inductively this process until the n-ary constraint can be expressed as a binary one. The relational forms of set expressions are derived by creating a new set variable whose domain is approximated by using the set interval calculus. The relational forms correspond to the following constraints:

`union` (s_1, s_2, s)	$\leftrightarrow$	s_1 `\/` s_2 `'=` s
`inter` (s_1, s_2, s)	$\leftrightarrow$	s_1 `/\` s_2 `'=` s
`diff` (s_1, s_2, s)	$\leftrightarrow$	s_1 `\` s_2 `'=` s

The local consistency of these 3-ary constraints ensures that no triples satisfying the constraint are excluded. The inference is performed using transformation rules that make use of the projection functions. Each projection function allows each set domain to be expressed in terms of the others (with respect to one constraint). Each such projection uniquely defines a smallest set domain which contains the possible solution values. Three projection functions are required per relational constraint. They are depicted in the following figures.

Projection functions associated to the constraint $\texttt{union}(s_1, s_2, s)$ *such that* $s_1 \in d_1, s_2 \in d_2, s \in d$. *T5 holds also for* s_2 *and a similar rule exists for* d_2.

T5. $\texttt{glb}(d'_1) \leftarrow \texttt{glb}(d_1) \cup \texttt{glb}(d) \setminus \texttt{lub}(d_2)$
$\texttt{lub}(d'_1) \leftarrow \texttt{lub}(d_1) \cap \texttt{lub}(d)$
T6. $\texttt{glb}(d') \leftarrow \texttt{glb}(d) \cup \texttt{glb}(d_1) \cup \texttt{glb}(d_2)$
$\texttt{lub}(d') \leftarrow \texttt{lub}(d) \cap (\texttt{lub}(d_1) \cup \texttt{lub}(d_2))$

The union of two sets represents a logical disjunction. So it is very unlikely that the addition of new elements to $\texttt{glb}(d)$ requires modifying the lower bound of the domains of s_1 or s_2. The one case which requires such a refinement occurs if some elements belong to the lower bound of d and can never belong to one of the two sets (cf. T5). Consequently they should be added to the other one.

Projection functions associated to the constraint $\texttt{inter}(s_1, s_2, s)$ *such that* $s_1 \in d_1, s_2 \in d_2, s \in d$. T7. *holds also for* d_2.

T7. $\texttt{glb}(d'_1) \leftarrow \texttt{glb}(d_1) \cup \texttt{glb}(d)$
$\texttt{lub}(d'_1) \leftarrow \texttt{lub}(d_1) \setminus ((\texttt{lub}(d_1) \cap \texttt{glb}(d_2)) \setminus \texttt{lub}(d))$
T8. $\texttt{glb}(d') \leftarrow \texttt{glb}(d) \cup \texttt{glb}(d_1) \cap \texttt{glb}(d_2)$
$\texttt{lub}(d') \leftarrow \texttt{lub}(d) \cap \texttt{lub}(d_1) \cap \texttt{lub}(d_2)$

The intersection of two sets represents a logical conjunction. So any addition of elements to one of the three domains requires modifying at least one of the lower bounds of the domains. A pruning of the upper bound of these domains is less frequent. However, it might occur in the case depicted in T7 which corresponds to the following configuration: some elements are definite ones of s_2 (or s_1) and possible ones of s_1 (or s_2). If they cannot belong to s then they should be removed from the upper bound of the domain of s_1 (respectively s_2).

Projection functions associated to the constraint $\texttt{diff}(s_1, s_2, s)$ *such that* $s_1 \in d_1, s_2 \in d_2, s \in d$:

T9. $\texttt{glb}(d'_1) \leftarrow \texttt{glb}(d_1) \cup \texttt{glb}(d)$
$\texttt{lub}(d'_1) \leftarrow \texttt{lub}(d_1) \setminus (\texttt{lub}(d_1) \setminus (\texttt{lub}(d) \cup \texttt{lub}(d_2)))$
T10. $\texttt{glb}(d'_2) \leftarrow \texttt{glb}(d_2)$
$\texttt{lub}(d'_2) \leftarrow \texttt{lub}(d_2) \setminus \texttt{glb}(d)$
T11. $\texttt{glb}(d') \leftarrow \texttt{glb}(d) \cup (\texttt{glb}(d_1) \setminus \texttt{lub}(d_2))$
$\texttt{lub}(d') \leftarrow \texttt{lub}(d) \cap (\texttt{lub}(d_1) \setminus \texttt{glb}(d_2))$

The second part of the rule T9 considers a particular case where the upper bound of d_1 should be pruned. If $\texttt{lub}(d_1)$ contains elements which do not belong both to

the upper bound of d *and* to the upper bound of d_2, then these elements cannot belong to s_1. Both conditions must be satisfied to prune $\texttt{lub}(d_1)$.

Complexity issues. Time complexity for each transformation rule is bounded by $\mathcal{O}(d)$ times the number of basic set operations, which is bounded by 4 for the rules T7 and T9.
Remark. The relational constraints are transparent to the user at the programming level. However, any temporary state of a program is given in terms of these newly created constraints.

Example: A locally consistent constraint of the form: `S1 \/ S2 '< S2 /\ S3` is stored using the set of delayed goals:

```
union(S1, S2, S12),
inter(S2, S3, S23),
S12 '< S23.
```

□

6.3.3. Graduated constraints: cardinality and weight constraints

Graduated constraints deal with set variables and integer variables. Inferring the local consistency of these constraints might require refining the integer domains or assigning a value to a set. Since graded functions are not bijective functions, a modification of the integer domains is not a sufficient condition to require a set domain refinement. The pruning for the set cardinality and the weight constraints achieved by the following transformation rules. It guarantees that (1) the values removed from the domains cannot be part of any feasible solution, (2) if a solution exists, its value lies in the remaining set and integer domains.

Consider the set cardinality constraint $\#(s, x)$ *where* $s \in d$ *and* $x \in [m, n]$. x *is an integer variable. We have:*

T12. $[m', n'] \leftarrow$ [*max(m,*$\#\texttt{glb}(d)$*), min(n,*$\#\texttt{lub}(d)$*)*]
T13. $d' \leftarrow \texttt{glb}(d)$ if $\#\texttt{glb}(d) = n$
T14. $d' \leftarrow \texttt{lub}(d)$ if $\#\texttt{lub}(d) = m$

The transformation rules for the weight constraint are similar. The only difference lies in the initial computation of the integer intervals.

Consider the weight constraint $\texttt{weight}(s, y)$ *where* $s \in d, y \in [m, n]$ *and* $\sum_{(e_k, m_k) \in\ glb(d)} m_k = w_{glb}$ *and* $\sum_{(e_k, m_k) \in\ lub(d)} m_k = w_{lub}$. *We have:*

T12'. $[m', n'] \leftarrow [max(m, w_{glb}), min(n, w_{lub})]$
T13'. $d' \leftarrow \texttt{glb}(d)$ if $m = w_{lub}$
T14'. $d' \leftarrow \texttt{lub}(d)$ if $n = w_{glb}$

6.4. Complexity of the constraint solver

The constraint solver is based on the generic fixed point algorithm described in the formal part. It applies these rules to check/infer the consistency of an admissible system of constraints in an incremental way. Incrementality refers to the nature of the Conjunto solver which stores each newly locally consistent constraint and handles the consistency of each constraint in conjunction with the constraint store.

Complexity issues Let G be the set of all the constraints to be considered and l its size. The cost of one transformation rule is bounded by $\mathcal{O}(d)$ (d being the largest $\#lub(s) + \#glb(s)$). For one constraint the algorithm can be iterated at worst d' times if $d' = \#lub(s) - \#glb(s)$. If these iterations are necessary for all the constraints the worst time complexity is then $\mathcal{O}(ldd')$.□

This time complexity does not occur in practice since a constraint is not systematically reconsidered if some of its variable domains get modified. Indeed, the constraints are stored in various suspension lists so as to avoid reconsidering them when there is no need to do so. These lists are described below.

Suspension lists Three different lists are attached to each set variable. They are meant to improve the time complexity and thus the efficiency of the solver by splitting the list C so that only those constraints concerned with the specific domain refinement are woken. Corresponding to each set variable s_i with domain d_i, each of the three lists could contain the following goals:

- Q_{glb} contains the primitive constraints for which a modification of the lower bound of d_i might require reconsidering the constraints. It contains only constraints of the form s_i `'<` s_j.

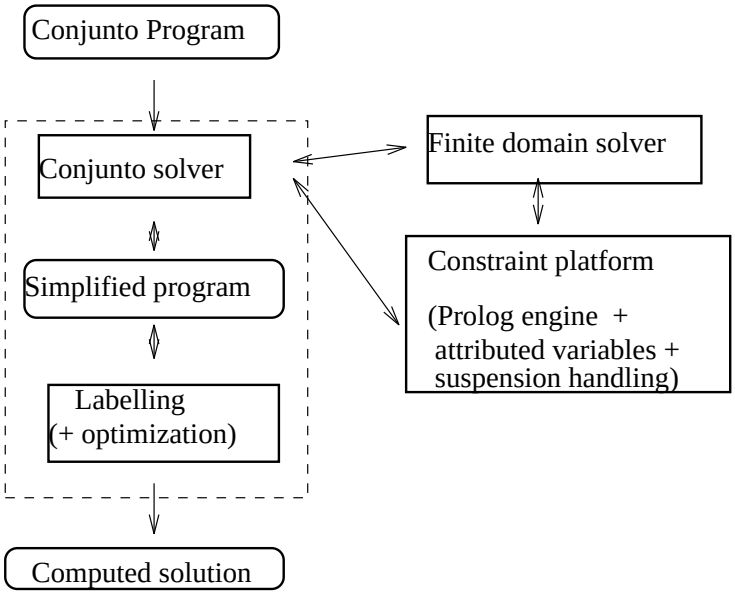
- Q_{lub} contains the primitive constraints for which a modification of the upper bound of d_i might require reconsidering the constraints. It contains the constraints of the form: s_j `'<` s_i, s_i `'<>` s_j, (and its symmetrical s_j `'<>` s_i).

- Q_{any} contains the remaining constraints for which any set domain modification might require reconsidering them. In other words it contains the relational constraints (relational forms of the set union, intersection and difference operations) and the graduated constraints in which the variable s_i appears.

In addition, the graduated constraints are also stored in the list of delayed goals attached to the integer variables appearing in it. While graduated constraints are delayed only once, they are attached to two lists and thus might be reactivated with respect to two different conditions. This process establishes the dynamic cooperation between the Conjunto solver and the finite domain solver. It guarantees that the local consistency of a graduated constraint is always maintained within a constraint system.

6.5. Solver modularity

The Conjunto solver can be embedded in any logic-based language provided a set of constraint solving facilities is given or can be defined. These facilities comprise (1) attributed variables or a similar structure which links a set variable to its domain and the required lists of delayed goals, (2) suspension handling mechanisms to deal with delayed goals, (3) possibly a finite domain library to tackle set based optimization problems. The following figure shows the modules and functionalities required during the execution of a Conjunto program.



III Applications

We show the applicability of the Conjunto language to the modelling and solving of set based search problems. The focus is on the expressiveness and the efficiency of the language when dealing with search problems and optimization problems arising from operations research and combinatorial mathematics.

7. Set domain CSPs

The modelling and solving of a set domain CSP follows the usual procedure for CSPs which consists of the problem statement, the labelling procedure and possibly the search for an optimal solution.

The labelling can be achieved by using the pre-defined labelling procedure `refine` described in the practical framework or by defining a new labelling procedure based

on specific labelling strategies. An efficient set labelling procedure should not try to directly instantiate a set to one of its domain elements. The reason is that by doing so, the satisfaction of those constraints for which only a local consistency is guaranteed is reached in a passive way.

The concept of optimality is related to the notion of minimizing or maximizing a cost function. This function necessarily denotes a measure, takes as input an arithmetic expression and returns an integer value. Possible cost functions associated with a set domain CSP are the sum of the set cardinality values, the sum of the weights, etc. Such a function constrains the sets via their associated measure and consequently no specific optimization predicate is required to deal with sets. The user can make use of existing predicates developed for integer domain CSPs with an optimization criterion. One of these predicates used in a subsequent application (set partitioning), performs the branch and bound search.

The predicate `min_max(Goal, Cost)` searches for a solution to the goal `Goal` that minimizes the value of the linear term `Cost` using the branch and bound method from operations research (Papadimitriou and Steiglitz, 1982). As soon as a partial solution to `Goal` is found whose cost is worse than the previous solution the search is not explored any further and a new solution is searched for.

Another predicate is often used to minimize the cost of a solution within a fixed range: `min_max(Goal, Cost, Min, Max, Percent)`. This predicate also makes use of the branch and bound method with some restrictions. It starts with the assumption that the value `Cost` to be minimized is less than or equal to `Max`. As soon as a solution is found whose minimized value is less than `Min`, this solution is returned. When one partial solution is found, the search for the next better solution starts with a minimized value `Percent` % less than the previous one.

The use of these predicates in a set domain CSP requires the definition of `Goal` as a set labelling procedure call, plus a graduated constraint whose integer value is `Cost`. The solving of `min_max/2/5` will execute the labelling procedure and incrementally refine the integer domain involved in the graduated constraint. Once all the sets are labelled the integer domain becomes one value (the cost) which can be evaluated. The optimization process will then constrain the integer variable appearing in the graduated constraint to have its value in a new domain whose upper bound is lower than the cost previously computed.

7.1. Modelling facilities

The two problems presented hereafter come from the areas of combinatorial mathematics (Lueneburg, 1989) and operations research. The first one —the ternary Steiner problem— is to find a specific hypergraph whose nodes are integer variables. Our approach illustrates how an hypergraph whose nodes are integer variables can be modelled as a simple graph whose nodes are set variables. The second problem is a set partitioning problem usually represented by mathematical models and solved using integer linear programming techniques. Here it is modelled as a set domain CSP.

7.1.1. Ternary Steiner problem

The ternary Steiner problem has its origins in combinatorial mathematics. It belongs to the class of block theory problems which deal with the computation of hypergraphs. A hypergraph is a graph with the property that some arcs connect collections of nodes. This problem has only recently been addressed in computer science. (Beldiceanu, 1990a) addresses this problem for the first time. The approach consists in representing the problem as an integer domain CSP in a constraint logic programming (CHIP (Dincbas et al., 1988)), using the new concept of global constraints. The integer domain CSP modelling corresponds to the hypergraph representation: the integer variables represent the nodes and the global constraints represent the hyperarcs.

Problem statement The statement is taken from (Beldiceanu, 1990a). A ternary Steiner system of order n is a set of $T = n(n-1)/6$ triples of distinct elements in $\{1, ..., n\}$ such that any two triples have at most one element in common. The mathematical properties of this problem prove that n modulo 6 has to be equal to 1 or 3 (Lindner and Rosa, 1980). One solution of Steiner 7 is for example:

$$\{1,2,6\},\{1,3,5\},\{2,3,4\},\{3,6,7\},\{2,5,7\},\{1,4,7\},\{4,5,6\}$$

The integer domain CSP modelling or hypergraph representation uses three nodes, or variables, ranging over $\{1, ..., n\}$ to represent a triple $\{X, Y, Z\}$. The constraints are (1) ordering constraints between the three nodes ($X < Y < Z$) so as to remove equivalent triples under permutations of the elements; (2), any triple must have at most one element in common with the other triples of nodes. This amounts to constraining each pair of a triple to be pairwise distinct from any other pair appearing in another triple. This requires constraining all the $n(n-1)$ possible pairs (6 per triple [X, Y, Z]: [X,Y], [Y,X], [X,Z], [Z,X], [Y,Z], [Z,Y]) to be pairwise distinct. This approach is sound but far too costly in variables and constraints. A global constraint `all_pair_diff` has been defined in (Beldiceanu, 1990, Beldiceanu, 1990a) to free the user from specifying all the pairwise distinct pairs.

If each set of three nodes, describing a triple, can be represented as one variable, then the modelling is simpler and requires less variables. Such a modelling corresponds to a set domain CSP approach. Also, the constraints applied between each set of three nodes become one constraint between two triples (set variables). Thus, the set domain CSP models a hypergraph with respect to the integer domain CSP modelling.

Problem modelling Modelling the problem as a set domain CSP involves representing each triple as *one* set variable. Let $S_i, 1 < i < T$ denote the T set variables which represent the triples. Their domains are initialized to the set domain `[{},{1,...,n}]`.

The constraint "any two triples have at most one element in common" is simply represented by: `#(` S_i `/\` S_j`) =< 1`. The constraint generation is summed up in the short program:

```
constraints(Lsets) :-
     card_all(Lsets, 3),
     intersect_atmost1(Lsets).

card_all([], N).
card_all([Set1|LSets], N) :-
     #(Set1, N),
     card_all(LSets, N).
```

```
intersect_atmost1([]).
intersect_atmost1([S1 |L]) :-
     distinctsfrom(S1, L),
     intersect_atmost1(L).

distinctsfrom(_S, []).
distinctsfrom(S, [S1 | L]) :-
     #(S /\ S1, C), C =< 1,
     distinctsfrom(S, L).
```

`card_all` constrains the cardinality of each set variable in the list `Lsets` to be equal to 3. The predicate `intersect_atmost1` generates the main constraint to be satisfied by each pair of triples.

Problem solving The resolution makes use of the labelling procedure `refine(S)` for each triple `S`. If $n = 7$, the first set is instantiated to $\{1, 2, 3\}$. Then the system tries to instantiate the second set by first adding the element 1 to its lower bound. This domain refinement requires reconsidering the constraint `#(S1 /\ S2, C)`. This results in a refinement of the domain of `S2` by a removal of the values 2 and 3 from the upper bound of its domain. At this stage in the resolution, the refined domains are:

```
S1 = {1,2,3}, S2 ':: [{1},{1,4,5,6,7}],
[S3,S4,S5,S6,S7] ':: [{},{1,...,7}].
```

Computation results The problem was solved in 0.8 sec on a Sun4/40 for $n = 7$. Six choice points were created during the solution step. (Beldiceanu, 1990a) says that 21 choice points were generated and 0.08 sec were sufficient to solve the problem. This difference in choice points and time was surprising. Unfortunately the global constraint and the program developed were not available and so, in order to make a sound comparison, we developed the same program as described in the paper using the ECL^{i}PS^e integer domain library. The choice points and the time required were then similar to the Conjunto approach, but the program was much less natural.

The complexity of this problem grows exponentially with n. In (Beldiceanu, 1990a) the problem has not been tackled for larger values than 7. Indeed, it turned out that using the same program to solve the problem when $n = 9$ leads to a combinatorial explosion. We defined a labelling strategy which consists in constraining each element to belong to at most $(n-1)/2$ triples. Indeed, there are at most $n-1$ distinct pairs containing one element i and a triple containing i must contain 2 of these pairs. In practice this labelling strategy corresponds to a simple occur check before adding one element to a set domain. This does not help when $n = 7$ but for $n = 9$ it reduced the number of choice points from 7180 to 116 and consequently the computation time from 501 sec. to 18 sec.

Remark. For one value of n there exists more than one solution. The search for all the possible solutions requires us to take into account the symmetries inherent

to the problem *i.e.*, those which do not depend on the modelling. A permutation of two sets does not change the actual solution but corresponds, from a computational point of view, to new instances of the set variables. In fact, the modelling of a search problem as a set domain CSP removes the symmetries that come from an integer domain CSP approach. Consequently, set constraints resemble some global constraints in terms of problem solving and pruning ability, but to cope with this actual symmetries of the problem a global reasoning on sets is necessary.

7.1.2. The set partitioning problem

The set partitioning problem (Gondran and Minoux, 1984) is an optimization problem that comes from operations research. Consider a mapping from a set of elements to a collection of equivalence classes each of which contains a subset of these elements, and has a specific cost. The objective is to find a subset of the classes such that they are all pairwise disjoint, each element is mapped onto exactly one class and the total cost of the selected classes is minimal.

This problem is currently tackled as a 0-1 integer linear programming problem using the following mathematical model:

$$\text{minimize } (c * x), \quad (a_{ij}) \ \times x = e_m$$

where c is a cost vector $1 * n$, (a_{ij}) is an $m * n$ known matrix comprising 0 and 1 values, x is an $n * 1$ vector of 0-1 variables and e_m is a vector of m entries equal to 1. We have:

$$\forall i \in Dom, \forall j \in \{1, ..., n\}, \ a_{ij} = \begin{cases} 1 & \text{if } i \in S_j, \\ 0 & \text{otherwise} \end{cases}$$

Each equivalence class is denoted by a set S_j.

Example: A 0-1 modelling corresponds to the following system of constraints: min $c_1x_1 + c_2x_2 + c_3x_3 + c_4x_4 + c_5x_5 + c_6x_6$

$$\begin{array}{cccccccc} x_1+ & & x_3+ & & x_5 & & = 1 \\ x_1+ & x_2+ & x_3+ & & & & = 1 \\ x_1+ & & x_3+ & & & x_6 & = 1 \\ & & & x_4+ & x_5+ & x_6 & = 1 \\ & & & x_4+ & & x_6 & = 1 \end{array}$$

Each column represents an equivalence class. Each line refers to one element in $\{1, .., 5\}$. The equality constraints specify that an element can belong to exactly one equivalence class. □

Problem statement The mathematical statement of the problem is depicted here in terms of relations and set constraints. Consider a mapping R from Dom to Ran which is constrained to be an application. Let the DS-domain be $Dom =$

$\{1, 2, ..., m\}$ and the AS-range be a family Ran of n subsets of Dom such that $Ran = \{S_1, ..., S_n\}$ where each S_j is an equivalence class (a ground set) and:

$$\bigcup_{j \in \{1,2,...,n\}} S_j = Dom$$

A subset P_0 of Ran is a partition of Dom if and only if:

$$\bigcup_{j \in \{1,2,...,n\}} S_j = Dom \bigwedge \forall S_j, S_k \in P_0, S_j \cap S_k = \emptyset$$

A cost set S_c is associated to the elements S_i of Ran by considering a weighted set composed of elements (S_i, w_i). The final problem is to determine a partition P^* such that:

$$\sum_i w_i \text{ is minimal}$$

This statement corresponds to the approach used with the Conjunto language.

Problem modelling Let a relation R on the ground sets Dom and Ran be constrained to be an applicative mapping. Each successor set is constrained to be a subset of the proposed sets. These constraints are not sufficient to solve the problem. Two other requirements are necessary:

- the final set P^* of equivalence classes should contain only disjoint sets.
- an instantiated successor set should also represent the successor set of all its predecessors.

This corresponds to adding two constraints which will be checked using the forward checking inference rule (*i.e.*, once a successor set becomes ground). Informally, as soon as one successor set $\texttt{succ}(R, i, \{s_k\})$ becomes ground we must have:

$$\forall j \in Dom, succ(R, j, s_j) \quad \begin{cases} \text{if } j \in s_k, \ s_j = \{s_k\} \\ \text{if } j \notin s_k, \ s_j \cap \{s_k\} = \emptyset \end{cases} \quad (1)$$

Example: The statement of the previous example corresponds to the following set of constraints Conjunto constraints:

```
R bin_r {1,2,3,4,5} --> {{1,2,3},{2},{1,2,3}, {4,5},{1,4},{3,4,5}},
appl(R),
succ(R, 1, S1), S1 '< {{1,2,3},{1,4}},
succ(R, 2, S2), S2 '< {{1,2,3},{2}},
succ(R, 3, S3), S3 '< {{1,2,3},{3,4,5}},
succ(R, 4, S4), S4 '< {{4,5},{3,4,5}},
succ(R, 5, S5), S5 '< {{4,5},{3,4,5}}.
```

□

Each element $i \in \{1, ..., 5\}$ is mapped to a set Si whose domain contains the possible equivalence classes (ie. those which contain i). Note that columns 1 and 3 in the ILP modelling correspond here to one equivalence class $\{1, 2, 3\}$.

The search space associated to these problems is usually very large and simplification rules are applied in order to reduce the initial problem size (e.g. in (Hoffman and Padberg, 1992, Padberg, 1979)). They consist in removing rows and columns in the adjacency matrix formulation. This corresponds to removing, in a deterministic manner, redundant sets from the successor set domains, and to bounding some successor sets to the same variable. The main operations amount to checking disjointness and/or inclusion of sets and to computing cliques over the successor set domains. This is achieved in a very natural manner using Conjunto (for a full description of the modelling see (Gervet, 1995)).

Problem solving One important strength of solvers based on constraint propagation techniques is their dynamic behaviour thanks to the delay mechanism. In particular, once the simplification rules have been applied, their ripple effects on the set of constraints allows to dynamically reduce the problem size. Linear programming solvers require the whole problem to be considered once again.

A large application has been developed, in which it is necessary to look for an optimal solution using the predicate `min_max/5` and to consider a specific labelling strategy. The strategy aims at selecting a set among the remaining ones whose cost is the lowest.

The labelling procedure considers each successor set S_i in order. The set `E` which belongs to the upper domain bound of S_i and which has the lowest cost is selected, and added to S_i. A choice point is created and in case of failure the program backtracks. The previous state is restored and the set `E` is removed from the domain of S_i.

```
labelling([], _).
labelling([S1 | LSuccs], S) :- set(S1), !,
     labelling(LSuccs, S).
labelling([S1 | LSuccs], S) :-
     lub(S, Lub),
     select_cheapest(S1, E, Lub),
     (E in S1
       ;
     E notin S1),
     labelling([S1 | LSuccs], S).
```

The optimization predicate for the set partitioning problem is:

```
min_max((labelling(LSuccs, S), take_min(C)), C, Min, Max, %).
```

`take_min(C)` is an integer domain predicate which binds an integer term C to its minimal value. `C` is the weight of the set variable `S`.

To solve the goal `labelling(LSuccs, S), take_min(C)`, we first label all the sets, instantiate the weight of the set domain of `S` to its minimal value and then search for a better solution according to the criteria given.

Computation results A set partitioning problem describing a 0-1 matrix of size 17x197 was implemented using the approach presented here. The complete program takes 200 lines of Conjunto code. The problem was taken from the (Hoffman and Padberg, 1992) library. The heuristics led to a simplified problem within 7 seconds and the optimal solution was found within 13 seconds on a Sun4/40. The proof of optimality required 31 additional seconds. (Hoffman and Padberg, 1992) make use of the simplex method combined with a tailored branch and cut search to tackle set partitioning problems (crew scheduling problems). The optimum solution to the 17x197 problem is found in 0.06 seconds on a VAX 8800.

On the one hand, the flexibility and conciseness of the Conjunto approach is a strength compared with existing mathematical models. On the other hand, constraint propagation techniques are not competitive when compared with global methods like the simplex (e.g. in (Hoffmann and Padberg, 1992, Guerinik and Van Caneghem, 1995)). While completing this work, it appeared to us that the set domain CSP approach is promising when investigating feasibility issues that are problematic with the simplex method. The simplex stops when the model is detected to be inconsistent but it cannot detect the reasons for failure. The inherent incremental solving of local consistency techniques can be of a great help. In addition, the partitioning problem appears as a sub_problem in numerous real life applications (eg. timetables, bus line balancing), which are currently solved using integer domain solvers. While integer domain CSP are well suited to the scheduling constraints of these problems, a set domain CSP can provide an easy way to tackle the partitioning constraints. The cooperation between the solvers is not a problem, provided that the constraints which involve set and integer variables can be attached to both. A real life application is worth considering.

7.2. Efficiency issues: A case study

The previous section illustrated the applicability of the system for dealing with a large class of search problems involving sets, relations, graded functions and optimization criteria. The question is: "can a gain in expressiveness be combined with a gain in efficiency ?". From a pruning point of view, the one-to-one correspondence between a set variable ranging over a set domain and a vector of 0-1 variables guarantees that if both sorts of variables are handled using the same labelling procedure (cf. `refine`), the pruning will be exactly the same. If there is a gain, it might therefore come from the saving in memory utilization and consequently from the garbage collection time. This point is illustrated through an integer linear programming optimization problem: the bin packing problem.

Problem description Bin packing problems belong to the class of set partitioning problems (Garey and Johnson, 1979). A multiset of n integers $\{w_1, ..., w_n\}$ is given

that specifies the weight elements to partition. Another integer W_{max} is given that 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 weights does not exceed W_{max}. This problem is usually stated in terms of arithmetic constraints over binary variables and solved using various operations research or constraint satisfaction techniques over binary finite domains. 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}. The cost function to be optimized is the total number of bins.

The mathematical formulation in 0-1 CSP and set domain CSP is described in the following figure.

0-1 CSP abstract formulation	set domain CSP abstract formulation
$\sum_{j=1}^{m} a_{ij}\, x_j = 1$ for all $i \in \{1, .., n\}$	$s_1 \cap s_2 = \{\},\ \ldots, s_{n-1} \cap s_m = \{\}$ $s_1 \cup \ldots \cup s_m = \{(1, w_1), .., (n, w_n)\}$
where:	
$x_j = 0..1 \begin{cases} 1 & \text{if } s_j \in \{s_1, .., s_k\} \\ 0 & \text{otherwise} \end{cases}$	$s_j \in [\{\}, \{(1, w_1), .., (n, w_n)\}]$
$a_{ij} = 0..1 \begin{cases} 1 & \text{if } i \in s_j \\ 0 & \text{otherwise} \end{cases}$	
$\sum_{i=1}^{n} a_{ij}\, w_i \leq W_{max}\ \forall j \in \{1, ..., m\}$	$weight(i, w_i) = w_i$ $\sum_{i=1}^{\#glb(s_j)} weight(i, w_i) \leq W_{max}\ \forall s_j$

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 $P = \{\ (1, w_1), ..., (i, w_i), ..., (n, w_n)\}$ 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 bins such that the sum of the w_i in a computed subset of P does not exceed a limited weight $Wmax$. A bin is represented by a set variable with initial domain $[\{\}, P]$. The union of all bins should be equal to P (represented using the `all_union` predicate). All the bins should be pairwise disjoint (`all_disjoint` predicate).

```
pb_statement(N,Max,Sets) :-
    pieces(P),
    make_sets(N,P,Sets),
    state_constraints(Sets,Max,P),

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(Max,Sets),
    all_disjoint(Sets),
    all_union(Sets,P).

restrict_weight(_M,[]).
restrict_weight(Max,[S| Sets]) :-
    weight(S,W),
    W =< Max,
    restrict_weight(Max,Sets).
```

Problem solving The labelling procedure makes use of the first fit descending heuristic. This heuristic sorts the elements (i, W_i) in decreasing order of their weight. Bins are then filled one after another, which is more efficient than filling all the bins in parallel. The optimization predicate is the classical one for packing problems which initializes the number of bins N to the value $weight(P)/Wmax$ and increases it at each call of goal predicate in case of failure. The solution is the first successful partition. This program was used to solve a large instance of 80 items partitioned into 30 sets. The optimal solution was found in about 22 seconds on a SUN 4/40.

Experimental results and comparisons A comparative study was made with a integer domain (0-1) formulation implemented using the finite domain library of ECL^{i}PS^e. For the encoding of sets and set constraints, we used respectively lists of binary variables and arithmetic constraints on the variables described previously. The arithmetic constraint predicates were handled using the ECLiPSe solver of arithmetic constraints over finite domains. It is based on consistency techniques which perform a reasoning about variation domain bounds or about variation domains, depending on the constraint predicate. The 0-1 integer domain program was encoded so as to use the same first fit descending heuristics and the same labelling procedure as the set domain CSP program. The following array gives the results regarding time consumption together with space utilization.

Criterion	Conjunto program	FD program
global stack peak (bytes) trail stack peak (bytes)	847 872 126 968	2 334 720 987 136
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 usage (stack peaks where the peak value indicates what the maximum amount allocated was during the session) and on the cpu time. The space utilization 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 in space utilization (stack sizes) between the two approaches comes from the set-like representation as a list of zero-one domain variables versus two sorted lists in Conjunto. The lists of zero-one 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.

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 execution is the same, so the difference comes from the amount of information recorded.

The garbage collection number is the times garbage collections are performed which is closely linked to the global and trail stack because the garbage collection on both at the same time. Thus, the difference in the garbage collection number comes again from the space utilization.

The difference between the cpu times is due first to the time needed for garbage collection which is a direct consequence of the size of the global and trail stacks; and secondly to the time needed for performing operations on the data.

Profiling the cpu time consumption indicates that half of time spent in the FD program resolution is the time needed for performing arithmetic operations on the zero-one variables. The weight constraint applied to each set is one of the most expensive computations. The weight constraint $a_{i1} \times w_1 + a_{i2} \times w_2 + ... a_{in} \times w_n \leq w_{max}$ which is woken each time an a_{ij} is set to 1, consists of a Cartesian product of two lists. In the Conjunto program, it consists in 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 computation of the largest weight not already considered for one set. This requires checking the value of the zero-one variable, and if this value is one, considering the weight associated to this variable. A weight is not 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, and the resulting set contains the elements (i, w_i) which have not yet been considered. Computing this difference is in fact the most time consuming step in the Conjunto program resolution, because it is also performed when computing disjoint sets, but it represents half of the cpu time consumption of arithmetic operations.

This application shows that set constraints together with set domains are expressive enough to embed the problem semantics, and to avoid encoding the information as lists of binary variables or handling additional data (the list of weights). It also shows that consistency techniques for set constraints are efficient enough to solve such combinatorial problems on sets.

7.3. General remarks

These applications have illustrated how the solving of set-based optimization problems is possible thanks to the graduated constraints (set cardinality and weight constraints).

With regard to an integer domain CSP, a set domain CSP approach contributes transparency with respect to the mathematical definition of set problems, and allows the user to go from a hypergraph to a graph representation, thus reducing the number of variables and simplifying the constraint statement phase. As far as efficiency is concerned, the first application (ternary Steiner problem) showed that the solving of set constraint achieves a pruning identical to that of global constraints. The cpu were also similar.

The second application (set partitioning) makes us of the one-to-one correspondence between a set variable ranging over a set domain and a 0-1 vector which allows us to model 0-1 Integer Linear Programming (ILP) problems as set domain CSPs. The modelling of 0-1 ILP problems as set domain CSPs in a constraint logic programming language shows the programming facilities of logic programming and enhances the class of CSPs. In particular, a CSP view of 0-1 ILPs brings flexibility to the modelling and can be useful when (1) unpure 0-1 ILP problems are to be tackled, (2) when their feasibility is problematical with ILP tools, (3) and when small 0-1 ILP problems are involved in some real CSP applications (eg. timetables, bus line balancing, etc).

The last application (bin packing) showed how a 0-1 CSP can be modelled more concisely as a set domain CSP using Conjunto with a possible gain in efficiency. The gain comes essentially from the time needed for garbage collection which is more important in the 0-1 CSP approach which uses a larger amount of variables.

Discussion and related works

Today, the Conjunto solver is available as a library in the ECL^{i}PS^e platform, developed at ECRC. Independently of our work, the concept of set domains was briefly introduced in (Puget, 1992) and several set constraints are implemented in the ILOG solver (Caseau and Puget, 1994, Puget, 1996). Detailed comparisons with the ILOG approach are difficult since ILOG solver is an industrial implementation not fully described in the public domain. However, personal communications with Jean-François Puget indicate that the two approaches are similar but differ on one main point: the generic algorithm used to handle set constraints. ILOG solver uses AC-5 (Van Hentenryck et al., 1992) whereas we make use of propagation methods based on the AC-3 algorithm (Mackworth, 1977).

A related line of work concerns the class of CLP(Sets) languages, that we have presented in the introduction (Walinski, 1989, Dovier and Rossi, 1993, Bruscoli et al., 1994). None of them is directly motivated by the class of applications we are dealing with; these approaches aim mainly at exploiting the expressiveness of constructed sets. Our study of set-based logic programming and CLP(Sets) languages came to the conclusion that complete solvers have severe efficiency problems due to the nondeterministic nature of the constructed set unification and its exponential complexity. Indeed, recent attempts have been made to tackle the bin packing problem using set constraints over constructed sets; the exponential unification procedure of constructed sets led to a combinatorial explosion. Our approach —even though it adds a lower level of abstraction than the LP or CLP approaches based on constructed sets— is more realistic and efficient when one aims at solving set-based search problems. The main difference is that we use variables with set domains and hence have a trivial unification procedure.

While our work has essentially aimed at defining a practical language towards the solving of applications, it has provided us with a matter for a formal definition of the language. The formal framework distinguishes between the computation domain of the constraint logic programming language, and the constraint

domain over which the computations are actually performed. These two levels of discourse are linked together by approximations and closure operations. Up to now, the class of CLP(FD) languages are defined as constraint logic programming languages, but their formal definition is still based on the formal framework defined by Van Hentenryck that is, embedding consistency techniques in logic programming (Van Hentenryck, 1989). The formal description of the Conjunto language can be used to give a formal definition of the class of CLP languages which embed consistency techniques as main constraint solving techniques.

We believe that some further research on applications and algorithms is needed. The concept of graduated constraints helps us with tackling set-based optimization problems, and studying the cooperation between two solvers (Conjunto and integer domain solvers), but the search space of such problems is defined with set domains essentially. The Conjunto language has not been used so far to tackle real life applications defined over a search space containing also integer domains. Applications involving scheduling constraints and set constraints are still to be developed. In particular, they would allow us to figure out whether it is possible or not to work on a mixed-search space. Time tables, bus line balancing, are some of the applications.

Regarding the class of consistency methods we have been using, we have essentially considered node and arc consistency techniques applied to set and graduated constraints. It sounds interesting to go beyond this, to use path consistency algorithms, and to take into account the latest researchs on the topology of constraint graphs. Some issues might be different from those already established with respect to integer domain CSPs. In this respect, the study of the ratio complexity/pruning is very important.

It would also be interesting to extend the set domain concept to that of lattice domains in order to cope with symmetry problems. For example, considering the lattice domains $\{\{1,3\},\{1,2\}\}$ and $\{\{1,2,3\}\}$, we have $\{\{1,3\},\{1,2\}\} \sqsubseteq \{\{1,2,3\}\}$. A set of constraints applied to variables ranging over lattice domains would ease the modelling and solving of set based problems dealing with the search for equivalence classes (partitioning, covering). They would remove the symmetries which come from permutations of instances of set variables. A solution to a set-based problem would not be a list of instantiated set variables but the one value of a lattice variable. Thus the order of the sets which define the lattice value would be irrelevant. Constraints over lattices would model a set domain CSP as a lattice domain CSP, and thus add a higher level of expressiveness with respect to set domains. On the other hand, the practical framework corresponding to embedding lattice intervals in CLP requires further works describing the algorithms and studying the trade-off between expressiveness and efficiency.

Acknowledgments

This paper is a revised version of my dissertation thesis carried out at ECRC under the supervision of Bruno Legeard and Mark Wallace. I am most grateful to Bruno Legeard, Gabriel Kuper and Alexander Herold who proofread the prelim-

inary versions of the thesis and to Mark Wallace for his valuable comments and corrections which substantially improved my work. I would also like to thank Hani El Sakkout, Robert Rodošek and the reviewers who have suggested a number of general comments and simplifications in the theoretical framework.

References

P. Baptiste, B. Legeard, and H. Lombardi. Sequence Constraints for Solving Scheduling Problems. In *3rd IFIP working conference*, 1994.

C. Beeri, S. Naqvi, O. Shmueli, and S. Tsur. Set constructors in a logic database language. In *Journal of Logic Programming*, pages 181–232, 1991.

N. Beldiceanu. Definition of Global Constraints. Internal Report IR-LP-22-30, ECRC, Dec 1990.

N. Beldiceanu. An example of introduction of global constraints in CHIP: Application to block theory problems. Technical Report TR-LP-49, ECRC, May 1990.

N. Beldiceanu and E. Contejean. Introducing Global Constraints in CHIP. In Elsevier Science, editor, *Mathematical Computation Modelling*, volume 20(12), pages 97–123, 1994.

F. Benhamou. Interval Constraint Logic Programming. In A. Podelski, editor, *Constraint Programming: Basics and Trends*. LNCS 910, Springer Verlag, 1995.

F. Benhamou, D. MacAllester, and P. Van Hentenryck. CLP (Intervals) revisited. In *ILPS'94*, pages 124–138, 1994.

G. Birkhoff. *Lattice Theory*, volume 25 of *Colloquium Publications*. American Mathematical Society, Providence, RI, 1967.

E. Boucher and B. Legeard. Fonctions de voisinage progressives et admissibles pour des méthodes d'optimisation en PLC sur séquences. In *actes du Congrès JFPLC'96*, pages 255–270, 1996.

P. Bruscoli, A. Dovier, E. Pontelli, and G. Rossi. Compiling Intensional Sets in CLP. In P. Van Hentenryck, editor, *ICLP'94*, pages 647–664, 1994.

B. Carlson, S. Haridi, and S. Janson. AKL(FD) A Concurrent Language for FD Programming. In M. Bruynooghe, editor, *ILPS'94*, pages 521–538, 1994.

Y. Caseau and Jean-F. Puget. Constraints on Order-Sorted Domains. In *Workshop on constraint processing, in conjunction with ECAI'94*, 1994.

J.G. Cleary. Logical arithmetic. In *Future Generation Computing Systems*, chapter 2(2), pages 125–149. 1987.

A. Colmerauer. Opening the prolog III Universe. In *BYTE magazine*. 1987.

A. Colmerauer, H. Kanoui, and M. Van Caneghem. Prolog, bases théoriques et développements actuels. *T.S.I. (Techniques et Sciences Informatiques)*, 2(4):271–311, 1983.

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

M. Dincbas, H. Simonis, and P. Van Hentenryck. Solving Large Combinatorial Problems in Logic Programming. *Journal of Logic Programming*, 1988.

A. Dovier, E. G. Omodeo, E. Pontelli, and G. Rossi. {log}: A Logic Programming Language with Finite Sets. In *ICLP'91*, pages 111–124, 1991.

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

ECRC. ECLiPSe (a) user manual, (b) extensions of the user manual. Technical report, ECRC, 1994.

R. E. Fikes. Ref-arf: A system for solving problems stated as procedures. *Artificial Intelligence*, 1:27–120, 1970.

R. Fraissé. *Theory of Relations*, volume 118 of *Studies in logic and the foundations of mathematics*. Elsevier Science, 1986.

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

C. Gervet. New structures of symbolic constraint objects: sets and graphs. In *WCLP'93*, 1993.

C. Gervet. Sets and binary relation variables viewed as constrained objects. In *Workshop on Logic Programming with Sets*, June 1993. In conjunction with ICLP'93.

C. Gervet. Conjunto : Constraint Logic Programming with Finite Set Domains. In M. Bruynooghe, editor, *ILPS'94*, pages 339–358, 1994.

C. Gervet. *Set Intervals in Constraint Logic Programming: Definition and Implementation of a Language*. PhD thesis, Université de Franche-Comté, France, September 1995. European thesis, in English.

G. Gierz and K.H. Hoffman et al. *A Compendium of Continuous Lattices*. Springer Verlag, Berlin Heidelberg New York, 1980.

M. Gondran and M. Minoux. *Graphs and algorithms*. Series in Discrete Mathematics. Wiley-interscience, 1984.

G. Graetzer. *LATTICE THEORY: first concepts and distributive lattices*. W.H. Freeman and company, 1971.

N. Guerinik and M. Van Caneghem. Solving Crew Scheduling Problems by Constraint Programming. In *CP'95*, Lecture notes in Computer Science, 1995.

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

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

P. Van Hentenryck and M. Dincbas. Domains in Logic Programming. In *AAAI-86*, 1986.

M. Hibti. *Décidabilité et complexité de systèmes de constraintes ensemblistes*. PhD thesis, Université de Franche-Comté, Besançon, 1995. In French.

K. L. Hoffman and M. Padberg. Solving Airline Crew-Scheduling Problems by Branch-and-Cut. Technical Report 376, George Mason and New York University, 1992.

C. Holzbaur. Metastructures vs. Attributed Variables in the Context of Extensible Unification. In *PLILP'92*, pages 260–268, 1992.

S. Le Huitouze. A New Datastructure for Implementing Extensions to Prolog. In *2nd Int. Work. Programming Languages Implementation and Logic Programming, LNCS 456*, pages 136–150, 1990.

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

J. Jaffar and M. J. Maher. Constraint Logic Programming: a Survey. In *Journal of Logic Programming*, chapter 19(20), pages 503–581. 1994.

B. Jayaraman and D.A. Plaisted. Programming with Equations, Subsets, and Relations. In Lusk and Overbeek, editors, *Proceedings of the North American Conference*, pages 1051–1068. Logic Programming, 1989.

D. Kapur and P. Narendran. Np-completeness of the set unification and matching problems. In *CADE*, pages 489–495, 1986.

R.A. Kowalski. Predicate Logic as a Programming Language. *IFIP*, pages 569–574, 1974.

G. Kuper. *Logic Programming with Sets*, volume 41 of *1*, pages 44–64. Academic Press, 1990.

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

J.H.M. Lee and H. van Emden. Interval Computation as Deduction in CHIP. In *Journal of Logic Programming*, chapter vol 16. numb. 3-4, pages 255–276. Elsevier, 1993.

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

B. Legeard and E. Legros. Test de satisfaisabilité dans le langage de programmation en logique avec contraintes ensemblistes: CLPS. In *Actes des JFPL*, 1992.

C.C. Lindner and A. Rosa. *Topics on Steiner Systems*, volume 7 of *Annals of Discrete Mathematics*. North Holland, 1980.

M. Livesey and J. Siekmann. Unification of Sets and Multisets. Memo seki-76-ii, University of St. Andrews (Scotland) and Universitât Karlsruhe (Germany) Department of Computer Science, 1976.

J.W. Lloyd. *Foundations of logic programming*. Springer-Verlag, 1987.

H. Lueneburg. *Tools and fundamental Constructions of Combinatorial Mathematics*. Wissenschaftverlag, 1989.

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

A. K. Mackworth and E. C. Freuder. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Artificial Intelligence*, 25, 1985.

W. Older and A. Vellino. Constraint Arithmetic on Real Intervals. In F. Benhamou and A. Colmerauer, editors, *Constraint Logic Programming: Selected Papers*. MIT Press, 1993.

M. W. Padberg. Covering, Packing and Knapsack Problems. In *Annals of Discrete Mathematics*, chapter volume 4, pages 265–287. North-Holland Publishing company, 1979.

C. H. Papadimitriou and K. Steiglitz. *COMBINATORIAL OPTIMIZATION: Algorithms and Complexity*. Prentice-Hall, 1982.

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

K. J. Perry, K. V. Palem, K. MacAloon, and G. M. Kuper. The Complexity of Logic Programming with Sets. *Computer Science*, 1986.

J-F. Puget. Programmation par contraintes orientée objet. In *Proceedings of Avignon*, pages 129–138, 1992.

J.F. Puget. Finite set intervals. In *Workshop on set constraints in conjunction with CP'96*, 1996.

A. Schrijver. *Theory of Linear and Integer Programming*. Discrete Mathematics. Wiley-interscience, 1986.

J. T. Schwartz, R. B. Dewar, E. Dubinsky, and E. Schonberg. *Programming with sets - An introduction to SETL*. Springer-Verlag Ed., 1986.

O. Shmueli, S. Tsur, and C. Zaniolo. Compilation of set terms in the logic data language (LDL). *The Journal of Logic Programming*, 12(12):89–119, 1992.

Frieder Stolzenburg. Membership-constraints and complexity in logic programming with sets. In Franz Baader and Klaus U. Schulz, editors, *Frontiers in Combining Systems*, pages 196–212. Kluwer Academic, 1996.

D. Turner. *An overview of Miranda*, volume 21 n. 12. SIGPLAN Notices, 1986.

Oxford University Computing Laboratory. *Z handbook*. 1986.

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