*Fundamenta Informaticae 96 (2009) 227-252*
*DOI 10.3233/FI-2009-177*
*IOS Press*



# Integrating Finite Domain and Set Constraints into a Set-based Constraint Language

**Federico Bergenti, Alessandro Dal Palù, Gianfranco Rossi***
*Dipartimento di Matematica*
*Università degli Studi di Parma*
*Viale G. P. Usberti 53, 43100 Parma, Italy*
*gianfranco.rossi@unipr.it*

**Abstract.** This paper summarizes a constraint solving technique that is used to reason effectively in the scope of a set-based constraint language that supersedes existing finite domain languages. The first part of this paper motivates the presented work and introduces the constraint language, namely the language of *Hereditarily Finite Sets* (*HFS*). Then, the proposed constraint solver is detailed in terms of a set of rewrite rules that exploit finite domain reasoning within the HFS language. The proposed solution improves previous work on CLP($\mathcal{SET}$) [11] by integrating intervals into the constraint system and by providing a new layered architecture for the solver that supports more effective constraint solving strategies. On the other hand, the proposed approach provides enhanced expressivity and flexibility of domain representation than those usually found in existing finite domain constraint solvers.

**Keywords:** Finite Domain constraints, Set Constraints, Hereditarily Finite Sets.

## 1. Introduction and Motivation

*Finite Domain* (*FD*) [6] constraint solvers have been effectively applied to a great variety of problems in different application domains. However, FD constraint solving typically suffers from inherent weaknesses.

*Address for correspondence: Dipartimento di Matematica, Università degli Studi di Parma, Viale G. P. Usberti 53, 43100 Parma, Italy

From the point of view of expressive power, earlier FD constraint languages forced the domains of variables to *bounded intervals* (subset of a discrete universe, often $\mathbb{Z}$), leading to potential limitations in real-world applications where domains of variables are often sparse sets, possibly of structured entities [19]. Recent FD constraint solvers address this issue by providing means to deal with sparse sets domains. For example, FDSET terms [29] of SICStus allow modeling domains in terms of finite unions of intervals, while GNU Prolog transparently switches between sparse and interval domain representations [6].

Furthermore, forcing domains in $\mathbb{Z}$ inhibits a natural representation of structured knowledge. The recent introduction of *Finite Set* (*FS*) constraints [18] aims at overcoming such a limitation because FS constraints model domains as collections of known sets. Domains are normally expressed in terms of *set intervals* $[l..u]$ where $l$ and $u$ are known sets, usually of integers. A set interval $[l..u]$ represents the set of all subsets of $u$ that contain $l$, i.e., $[l..u] = \{x \ : \ l \subseteq x \wedge x \subseteq u\}$. Many constraint solvers (including many CLP systems) now offer FS constraints, e.g., ECLiPSe [22], Oz [28] and B-Prolog [32]; see [19] for a very general and complete presentation of FS constraints and constraints over structured domains in general.

FS constraints are explicitly derived from FD constraints and therefore they retain the appreciated efficiency of the latter, while inheriting many of their limitations. In particular, available FS constraint solvers, e.g., Conjunto [18] and Cardinal [3], treat efficiently only interval domains that are defined as convex closures of collections of elements. For example, given a variable whose domain is $\{\{a, b\}, \{a, c\}, \{d\}\}$, the actual interval domain is the set interval $[\{\}..\{a, b, c, d\}]$. At our knowledge no FS constraint solver deals also with domains represented as sparse set of sets.

Besides the mentioned limitations, it is also worth discussing another important limitation of FD-like constraint languages. The practice of using FD-like reasoning in knowledge representation has shown that domains are often unknown prior to reasoning and an important side effect of reasoning is revealing the shape of domains. In many real-world cases, domains are not available prior to computation, and they have to be acquired and/or computed; anyway, constraints over them are often known in advance. Mentioned situations are not really tractable by FD-like languages because their solvers typically attach a possibly implicit and redundant domain to each variable before processing. The difficulties connected with required a priori knowledge in FD-like reasoning are explicitly tackled in [7, 17], which extends CLP($\mathcal{FD}$) to deal with incomplete knowledge on domains.

The mentioned limitations do not reduce the notable importance of FD-like constraint languages and solvers for their proved effectiveness in handling a great variety of problems. Nonetheless, the urge for an improved expressive power to support modeling of complex domains, that real-world applications require, motivates the definition of constraint languages and solvers that trade-off efficiency with expressive power and completeness, e.g., CLP($\mathcal{SET}$) [11] and CLPS [5].

In particular, CLP($\mathcal{SET}$) provides a constraint language that subsumes most of the mentioned FD-like languages while delivering a correct and complete constraint solver, at the cost of notable inefficiency. More in details, CLP($\mathcal{SET}$), and its Java porting JSetL [27][1], support modeling domains in terms of generic extensional sets, usually called *Hereditarily Finite Sets* (*HFS*), that contain any kind of object (and nested finite sets in particular). Moreover, such sets can be constructed dynamically by means of common set operations and constraint solving takes place even over partially or totally unspecified sets.

[1] JSetL is a Java library which delivers most of CLP($\mathcal{SET}$) facilities for set solving in an object-oriented framework.

Unfortunately, the constraint solver of CLP($\mathcal{SET}$) does not associate any domain information with the uninitialized variables that occur in a constraint. Rather it always non-deterministically assigns all possible values to constraint variables as soon as such values are available. For instance, the CLP($\mathcal{SET}$)-constraints $x \in \{1, 2, 3, 4, 5\}$ and $s \subseteq \{1, 2, 3\}$, with $x$ and $s$ uninitialized variables, are always solved by explicitly enumerating all possible values of $x$ and $s$. This prevents the CLP($\mathcal{SET}$) solver to take advantage of the efficient constraint propagation techniques provided by constraint solving over finite domains and it forces to use a *generate & test* approach in most cases. Moreover, as noted in [19], the nondeterminism embedded in many of the CLP($\mathcal{SET}$) constraint solving procedures, e.g., set unification, may lead to an exponential growth in the complexity of its satisfaction procedure.

While the expressive power and the computational features of CLP($\mathcal{SET}$) make it a good candidate to deliver a constraint language capable of addressing the mentioned issues of FD-like languages and solvers, the inefficiencies of available implementations of CLP($\mathcal{SET}$) prohibit its instant use in many problems where FD-like solvers proved their relevance. A step towards a more effective use of the CLP($\mathcal{SET}$) language is represented by CLP($\mathcal{SET}, \mathcal{FD}$) [8] which integrates FD constraints into CLP($\mathcal{SET}$) thus allowing efficient processing through the use of an embedded FD solver. Advantages, however, are limited to FD constraints only: CLP($\mathcal{SET}, \mathcal{FD}$) still retains the overall constraint solving technique of CLP($\mathcal{SET}$), with no use of domain information except for the handling of FD constraints.

The overall goal of this paper is to combine the flexibility and expressive power of CLP($\mathcal{SET}$)-like languages with the efficiency of FD-like languages in a general manner. In order to achieve this goal, we propose to integrate FD, FS and CLP($\mathcal{SET}$) constraints into a single and coherent constraint language and to extend the domain-driven constraint solving techniques of FD and FS to the whole solver. For this reason, we develop a new constraint solver that replaces the standard CLP($\mathcal{SET}$) solver and that benefits from the information on variable domains to obtain improved efficiency.

The contribution of this work is twofold. On the one hand, it generalizes the languages of FD and FS constraints by:

- allowing domains to be sparse, possibly partially specified collections of elements of any kind, including intervals, sets and nested sets;
- allowing sets and intervals (of any kind) to be mixed freely and to be manipulated through the same set of primitive constraints.

On the other hand, the proposed solution improves the work in [8, 11] by:

- taking into account also FS constraints;
- introducing the notion of *extended set terms* (see Def. 3.2) as a way to support a tighter integration between sets and intervals;
- providing a layered architecture for the solver that $(i)$ allows some inconsistencies in the input constraints to be detected at earlier stages and $(ii)$ allows the user to specify the depth of consistency check he/she wants to achieve.

The paper is organized as follows. Section 2 contains essential concepts and definitions necessary to discuss (finite domain) constraint languages and solvers. Section 3 presents the language of HFS-constraints, focusing on the notion of variable domain. Section 4 presents the architecture of the proposed

constraint solver, and Section 5 details the solver in terms of a set of rewrite rules that make use of FD-like reasoning within the HFS language. Section 6 provides some theoretical results about soundness and completeness. Section 7 discusses some efficiency issues and Section 8 offers an overview about related work. Finally, some conclusions and future lines of work are drawn in Section 9.

## 2. Background and Definitions

In this section, we review the basic definitions and notation underlying the notion of constraint language, with particular emphasis on the languages of Finite Domain constraints and Set Constraints (mostly in the style of [1]).

**Definition 2.1. (Constraint)**
Let $x$ be a variable. The *domain* of $x$, $D(x)$, is a collection of values from a universe $\mathcal{U}$ that can be assigned to $x$. Let $X$ be a finite sequence of variables, $x_1,\ldots,x_k$, $k > 0$, ranging over the domains $D_1,\ldots,D_k$, respectively (i.e., $D(x_i) = D_i$). A *constraint* $C$ on $X$ is a $k$-ary relation on $D_1,\ldots,D_k$, i.e. a subset of $D_1 \times \cdots \times D_k$.

**Definition 2.2. (CSP)**
A *Constraint Satisfaction Problem* (or CSP) is a triple $\langle \mathcal{C}; \mathcal{V}; \mathcal{D} \rangle$, where $\mathcal{D}$ is a finite sequence of domains, $D_1,\ldots,D_n$, $\mathcal{V}$ is a finite sequence of variables, $x_1,\ldots,x_n$, with $D_i = D(x_i)$, and $\mathcal{C}$ is a finite collection of constraints, each on a possibly different subsequence of $\mathcal{V}$.

**Definition 2.3. (Solution)**
A $k$-uple $\langle d_1, \ldots, d_k \rangle \in D'_1 \times \cdots \times D'_k$ *satisfies* a constraint $C$ on a sequence of variables $x_1,\ldots,x_k$, with $D'_i = D(x_i)$, if $\langle d_1, \ldots, d_k \rangle \in C$. $\langle d_1, \ldots, d_k \rangle$ is called a *solution* to $C$. Given a CSP $\langle \mathcal{C}; \mathcal{V}; \mathcal{D} \rangle$, where $\mathcal{V}$ is $x_1,\ldots,x_n$, a $n$-uple $\langle d_1, \ldots, d_n \rangle \in D_1 \times \cdots \times D_n$ is a *solution* to the given CSP if, for each constraint $C \in \mathcal{C}$ on a subsequence $S_C$ of $\mathcal{V}$, $x'_1,\ldots,x'_k$, $k \leq n$, the subsequence of $\langle d_1, \ldots, d_n \rangle$, $\langle d'_1, \ldots, d'_k \rangle$, where $d'_i = D(x'_i)$, satisfies $C$ on $S_C$. If a CSP has a solution, then it is *consistent*; otherwise, it is *inconsistent*.

In practice, in order to express constraints and the related domains we need some specific *constraint language* ($\mathcal{CL}$), which is precisely defined by its syntax and semantics. The syntax is specified by the signature and a way to compose constraints.

**Definition 2.4. (Signature)**
The *signature* $\Sigma$ of a $\mathcal{CL}$ is defined as $\Sigma = \langle \mathcal{V}, \mathcal{F}, \Pi \rangle$ where $\mathcal{V}$ is a fixed finite set of variables, $\mathcal{F}$ is the set of constant and function symbols, and $\Pi$ is the set of *constraint* predicate symbols allowed in $\mathcal{CL}$.

**Definition 2.5. ($\mathcal{CL}$-constraint)**
A *primitive $\mathcal{CL}$-constraint* is any atomic predicate built using the symbols from the signature, according to the language syntactic rules. A (non-primitive) *$\mathcal{CL}$-constraint* is formed by composing primitive $\mathcal{CL}$-constraints, possibly using the logical connectives (and, or and not) and logical quantifiers.

The semantics of a $\mathcal{CL}$ is specified, as usual, by defining the domain of discourse and by mapping each element of the signature to elements of the domain and to functions and relations over such domain.

In this paper we are mainly interested in Finite Domain constraints and Set Constraints. In *Finite Domain constraints* the domain is $\mathbb{Z}$ and variables range over finite sets of integers. The signature of the language comprises constants and function symbols that represent integer numbers and the standard arithmetic operations, as well as predicate symbols that denote membership to an integer interval (e.g., $x \in m..n$), and comparison operations between integers (e.g., $=$, $<$, $\leq$). Constraints of the form $\in m..n$, $m, n \in \mathbb{Z}, m \leq n$, are used as a way to associate finite domains with constraint variables: the interpretation of the constraint $x \in m..n$ (a *domain declaration*) is: $x \in m..n$ holds iff $m \leq x \leq n$.

In *Set Constraints* the domain is the powerset of $\mathbb{Z}$ and variables range over finite sets of sets. We call such variables *set variables*. Generally speaking, a Set Constraint language provides, besides set variables, constant and function symbols to be used as *set constructors* (e.g., element insertion/removal, set union), and predicate symbols to denote operations on sets. These operations may include classical set-theoretic operations, e.g., membership ($\in$), equality ($=$), subset ($\subseteq$, $\subset$), as well as aggregate operations involving both set and integer variables, e.g., cardinality ($\#s = n$), sum of all elements of a set (of integers) ($\mathsf{sum}(s)$). Constraints of the form $\in a..b$, $a$, $b$ finite sets, $a \subseteq b$, can be used to associate (finite) set domains with variables. The interpretation of the constraint $x \in a..b$ (a *set domain declaration*) is: $x \in a..b$ holds iff $a \subseteq x \subseteq b$, i.e., $a..b$ represents the lattice of sets induced by the subset partial ordering relation $\subseteq$, having $a$ and $b$ as the *greatest lower bound* (glb) and the *least upper bound* (lub), respectively.

Set Constraints where each set variable has a (possibly different) finite set domain associated with it are often referred to in the literature as *Finite Set (FS)* constraints.

Solving a constraint $C$ consists in transforming $C$ into a "simplified" form, whose set of solutions is the same as the one of $C$. A *constraint solver* for a constraint language $\mathcal{CL}$, $Solve_{\mathcal{CL}}$, is a procedure that is able to solve $\mathcal{CL}$-constraints. A *complete constraint solver* is such that the generated constraints $C_1, \ldots, C_k$ correspond to an "easy" way to compute solutions. In particular, if $C$ has no solution then the generated constraint is the logical constant $\mathsf{false}$. An *incomplete constraint solver*, instead, typically generates only *one* (i.e., $k = 1$) constraint which has the same set of solutions as the original constraint $C$; however, determining the satisfiability of such a resulting constraint can be potentially hard.

Very often, complete constraint solvers employ incomplete consistency algorithms combined with backtracking search procedures that make the solver itself complete. The amount of time spent by the solver in checking consistency, and the amount spent in searching the solution space, may vary drastically from one solver to another. Various levels of consistency check can be defined, each with associated costs and benefits. In particular, constraint solving for FD constraints and for FS constraints is very often done by maintaining and updating the domain of each variable using a local propagation algorithm. The local propagation algorithm attempts to enforce consistency on the values in the variable domains by removing values that cannot be part of a solution to the system of constraints. Consistency is enforced using a fixed set of inference rules specific to each constraint.

Most of the consistency algorithms used in these cases are incomplete, so they are often combined with a backtracking search procedure to produce a complete constraint solver.

## 3. The HFS Constraint Language

This section details the syntax and semantics of the constraint language that we consider in this work, namely the HFS constraint language. It also highlights the peculiar features that this language provides for treating domains, which will be exploited in the constraint solving procedure (see next section).

Note that what we are considering here is just the constraint language. This language, along with its constraint solver, can be embedded very naturally in a CLP language, but it can be integrated also within a more conventional setting, e.g. an object-oriented language like Java. A discussion of the details of possible host languages and how they interact with the constraint language and its solver are out of the scope of this paper.

## 3.1. Syntax and semantics

**Definition 3.1. (HFS signature)**
Let $\Sigma_{HFS} = \langle \mathcal{V}, \mathcal{F}_{HFS}, \Pi_{HFS} \rangle$ be the signature of the HFS constraint language. The set $\mathcal{F}_{HFS}$ of constant and function symbols is

$$\{\emptyset, \mathsf{ins}, \mathsf{int}_{\leq}, \mathsf{int}_{\subseteq}\} \cup Z \cup F_Z \cup F_U$$

where: $\emptyset$, ins, $\mathsf{int}_{\leq}$, and $\mathsf{int}_{\subseteq}$ are the *set constructors*; $Z$ is the denumerable set of constants representing the integer numbers, i.e., $Z = \{0, -1, 1, -2, 2, \dots\}$; $F_Z$ is a set of function symbols representing operations over integer numbers, e.g., $+, -, *, \mathsf{div}, \mathsf{mod}$; $F_U$ is a (possibly empty) set of user-defined constant and function symbols.

In particular, terms constructed using ins, $\mathsf{int}_{\leq}$ and $\mathsf{int}_{\subseteq}$ are called extended set terms.

**Definition 3.2. (Extended set terms)**
An *extended set term* is any $\Sigma_{HFS}$-term of the form $\emptyset$ or $\mathsf{ins}(t, s)$ or $\mathsf{int}_{\leq}(m, n, s)$ or $\mathsf{int}_{\subseteq}(a, b, s)$, where $s$ is a variable or an extended set term, $m$, $n$ are integer constants, $a$, $b$ are ground extended set terms, and $t$ is any $\Sigma_{HFS}$-term.

The set $\Pi_{HFS}$ of constraint predicate symbols is

$$\{=, \neq\} \cup \Pi_S \cup \{\mathsf{set}, \mathsf{integer}, \mathsf{not\_integer}\} \cup \Pi_R$$

where: $\Pi_S$ is a set of predicate symbols representing the usual set-theoretic operations, such as $\in$, $\subseteq$, union, inters, disj, diff, size, along with their negative counterparts; $\Pi_R$ is a set of predicate symbols representing the usual comparison relations over integer numbers, such as $\leq$ and $<$.

**Definition 3.3. (HFS-constraints)**
A *primitive HFS-constraint* is any atomic predicate built using the symbols from the signature $\Sigma_{HFS}$. A *non-primitive HFS-constraint* is a conjunction of primitive HFS-constraints.

We consider also a particular form of HFS-constraints.

**Definition 3.4. (Canonical HFS-constraints)**
A HFS-constraint that contains only predicates taken from $\{=, \neq, \in, \notin, \mathsf{union}, \mathsf{disj}, \leq, \mathsf{size}, \mathsf{set}, \mathsf{integer}, \mathsf{not\_integer}\}$ is said to be in *canonical form*.

The intuitive semantics of the various symbols is as follows[2]. The symbol $\emptyset$ represents the empty set. $\mathsf{ins}(t, s)$ represents the set composed of the element $t$ union the elements of the set $s$, i.e., $\{t\} \cup s$. For example, $\mathsf{ins}(1, s)$, where $s$ is an uninitialized variable, represents the (unbounded) set $\{1\} \cup s$. $\mathsf{int}_{\leq}(m, n, s)$, where $m$, $n$ are integer constants, represents the set composed of the elements of the set $s$

[2]A formal discussion of these concepts is out of the scope of this paper and can be found in [8].

union the elements in the interval $[m, n]$, i.e., $\{x \ : \ m \leq x \wedge x \leq n\} \cup s$. $\mathsf{int}_{\subseteq}(a, b, s)$, where $a$, $b$ are ground terms denoting sets, represents the set composed of the elements of the set $s$ union the elements in the set of sets $\{x \ : \ a \subseteq x \wedge x \subseteq b\}$. Note that if $m > n$ or $a \not\subset b$, we assume that the terms $\mathsf{int}_{\leq}(m, n, s)$ and $\mathsf{int}_{\subseteq}(a, b, s)$ denote the empty set.

Symbols in $Z$ are mapped to the elements of $\mathbb{Z}$, while functions in $F_Z$ and the predicate symbols $\leq, \geq$, etc. are mapped to functions and relations over $\mathbb{Z}$ in the natural way.

As concerns symbols in $\Pi_S$, the predicates $=$ and $\in$ represent the equality and the membership relationships, respectively; the predicate union represents the union relation: $\mathsf{union}(r, s, t)$ holds iff $t = r \cup s$; the predicate disj represents the disjoint relationship between two sets: $s$ disj $t$ holds iff $s \cap t = \emptyset$; and so on. Finally, the integer and set predicates hold for integer constants and extended set terms, respectively.

Note that, as shown in [11], canonical HFS-constraints allow all other predicates in $\Pi_S$ to be defined as non-primitive constraints.

Special notations are also introduced to express extended set terms in a more convenient way. Specifically, $\{t_1 \,|\, t\}$ is used as a shorthand for $\mathsf{ins}(t_1, t)$, while $\{t_1..t_2 \,|\, t\}$ is a shorthand for $\mathsf{int}_{\leq}(t_1, t_2, t)$ or $\mathsf{int}_{\subseteq}(t_1, t_2, t)$, depending on whether $t_1$ and $t_2$ are integer constants or set terms. Moreover, when $t$ is the empty set, the two terms above are abbreviated as $\{t_1\}$ and $\{t_1..t_2\}$, respectively. This notation is easily extended to the case of $n$ elements, possibly mixing the ins, $\mathsf{int}_{\leq}$ and $\mathsf{int}_{\subseteq}$ constructors.

**Example 3.1.** The extended set term $\mathsf{ins}(1, \mathsf{int}_{\leq}(10, 20, \mathsf{ins}(100, \emptyset)))$ can be written as $\{1, 10..20, 100\}$. It represents the set containing all integers between 10 and 20 along with the integers 1 and 100.

We will refer to a syntactic object of the form $t_1..t_2$ as an *interval*; moreover, if $t_1$ and $t_2$ are integer constants we say that $t_1..t_2$ is an *integer interval*, while if $t_1$ and $t_2$ denote sets, $t_1..t_2$ is called a *set interval*.

Elements of a set can be values of any type (not necessarily homogeneous), including variables and other sets. Hence, sets can be *nested* at any level, e.g., $\{\, 1, \{\emptyset, \{a\}\}, \{\{\{b\}\}\}\,\}$. Moreover, sets can be *partially specified*, i.e., they can contain uninitialized variables in place of enumerated elements.

It is worth noting that HFS-constraints can deal equally well with sets made of enumerated elements (either integers or not) or intervals (either integer or set intervals), and with sets where enumerated elements and intervals are mixed together.

**Example 3.2.** The following $\Sigma_{HFS}$-predicates

- $13 \in \{1..10, 15, 20..100\}$
- $\{1, 3, x\} = \{1..3\}$
- $\mathsf{union}(\{1, 5, 7\}, \{3..6\}, r)$
- $\mathsf{inters}(\{\{a\}..\{a, b, c\}\}, \{\{a, b\}, \{b, c\}\}, s)$

are all admissible HFS-constraints, which are satisfied, respectively: never, by $x = 2$, by $r = \{1, 3..7\}$, and by $s = \{\{a, b\}\}$.

The availability of extended set terms is particularly useful to represent concisely sets containing both sparse elements and intervals, e.g., the one obtained from the constraint $\mathsf{diff}(\{1..10\}, \{5\}, r)$, whose solution is $r = \{1..4, 6..10\}$.

Actually, extended set term management represents a straightforward extension of the set representation and manipulation facilities provided by CLP($\mathcal{SET}, \mathcal{FD}$), JSetL, and other FD/FS constraint solvers (see Remark 3.1 below).

## 3.2. Domains

Variables occurring in constraints that denote set-theoretic operations, e.g., union and disj, as well as in the right-hand side of membership constraints, range over the domain of HFS (*set variables*). Variables occurring in constraints that denote comparison relations between arithmetic expressions, as well as in the second argument of cardinality constraints, range over the domain of integers (*integer variables*).

Domains can be specified as HFS and associated to variables through membership constraints. $x \in d$, where $d$ is an extended set term, states that the domain of $x$ is the set denoted by $d$. This turns domains into first class abstractions of the language and naturally equips them with common set operations, thus generalizing domains from bounded intervals to very general, unbounded and potentially sparse sets.

More in detail, domains can be specified either as sets or as intervals. In the first case, a domain can be given either by enumerating all its values or it can be constructed as the result of some set operation; moreover, a domain can be partially specified. When a domain is specified as an interval, the interval itself can be given either by specifying its lower and upper bounds, $l..u$, or it can be constructed as the result of some set operation.

**Example 3.3.** The following are all valid domain specifications:

- $x \in \{1, 3, 5, 7, 9\}$ states that the domain of the integer variable $x$ is the set of the odd natural numbers less than 10;
- $x \in \{1 \mid s\}$, with $s$ a set variable, states that the domain of $x$ is a set containing the value 1 plus something else not yet specified;
- $r \in \{\{1\}..\{1, 2, 3\}, \{4\}\}$ states that the domain of the set variable $r$ is the set of sets $\{\{1\}, \{1, 2\}, \{1, 3\}, \{1, 2, 3\}, \{4\}\}$;
- $\mathsf{union}(\{1, 3, 5\}, 3..7, d\} \wedge x \in d$ defines the domain of $x$ as the interval from 1 to 7.

Domains can be either closed or open, bounded or unbounded, as follows.

**Definition 3.5. (Bounded/unbounded domains)**
A domain $d$ whose number of elements is known and finite is said *bounded domain*; otherwise it is *unbounded*.

**Definition 3.6. (Closed/open domains)**
A domain $d$ is said *closed* if it is bounded and all elements in $d$ are completely known; otherwise, $d$ is an *open domain*. In particular, the domain denoted by a ground extended set term is a closed domain.
If all elements of a closed domain $d$ are integer numbers we say that $d$ is an *integer domain*, while if all elements are sets we say that $d$ is a *set domain*.

**Example 3.4.** Sample domains.

- $\{1..10\}$ and $\{1, 3, 5, 7, 9\}$ are (bounded) closed integer domains;
- $\{1, x\}$, with $x$ variable, is a (bounded) open domain;
- $\{1 \mid s\}$, with $s$ a set variable, is an (unbounded) open domain.

Existing FD and FS constraint languages usually deal with (closed) integer and set domains only.

**Remark 3.1.** Among current FD solvers, many of them allow to specify domains with terms that represent unions of intervals, which resemble our extended set terms. For example, the extended set term $\{1..10, 15, 20..100\}$ can be input in SWI-Prolog as `1..10 \/ 15 \/ 20..100`, while in SICStus Prolog it can be written as `(1..10) \/ {15} \/ (20..100)`.

Note that the unions of intervals can contain only integer constants and must be bounded: variables may not be used and no elements different from integer constants can be employed. Moreover, the union of intervals can be used as part of membership constraints (e.g., `X in 1..10 \/ 15 \/ 20..100` in SWI-Prolog), however no operation on them can be performed.

This last limitation is partly overcome in SICStus Prolog by the introduction of FDSET terms. FDSET terms allow to represent domains as union of intervals (e.g., the domain above is represented by the FDSET term `[[1|10],[15|15],[20|100]]`), and set operators can be applied on them. For example, the union of two FDSET terms can be obtained by:

```
fdset_union([[1|1],[5|5],[7|7]],[[3|6]],R)
```

and the result is `R = [[1|1],[3|7]]` (cf. the third constraint in Example 3.2). The FDSET terms miss the semantic properties that they should support being sets, and in fact they are treated as lists. For example, the equality of two FDSET terms succeeds only if they are (syntactically) equal and not, more generally, when they represent the same set. E.g., the constraints `fdset_eq([[1|10],[11|15]],[[1|15]])` and `fdset_eq([[1|1],[2|2]],[[2|2],[1|1]])`, which should be true, are evaluated as false. The corresponding HFS-constraints are correctly rewritten in true. Moreover, FDSET terms have the limitations typical of terms representing union of intervals: they can not contain any variable and their elements can only be integer constants. Therefore both the second and fourth HFS-constraint in Example 3.2 can not be written as FDSET terms.

# 4. The Overall Solver Architecture

In order to enhance effectiveness of the constraint solving process, we decided to structure the proposed constraint solver into a *layered architecture*. Each layer achieves a different level of consistency at a different cost. Lower levels are considered more effective in proving inconsistency and reducing the search space than upper levels.

Let HFS_Solve be the procedure that implements the whole solver. More precisely HFS_Solve is structured as follows:

- Level 1 ($\mathsf{HFS\_Solve}_1$) – Single constraints are processed to generate *canonical constraints* by means of deterministic rewrite rules only. Rules are triggered by the type of the primitive constraint to be handled. A very weak form of propagation of constraints is performed at this level: variable substitution only.

- Level 2 ($\mathsf{HFS\_Solve}_2$) – Deterministic rules involving pairs of constraints are applied. These are mostly propagation rules that take advantage of the knowledge about variable domains, whenever possible, to reduce the size of domains or to reveal an inconsistency, according to the FD and FS approaches.
- Level 3 ($\mathsf{HFS\_Solve}_3$) – Nondeterministic and labeling rules are applied. This level is highly nondeterministic and it includes the labeling of variables whose domains are known. In this level rules are mostly taken from the available implementations of CLP($\mathcal{SET}$) solvers. The output of level 3 are *solved form constraints* [11].

The overall solving process repeatedly exploits all applicable rules at a given level until a fixpoint is reached. If a fixpoint is reached, i.e., the constraint store cannot be simplified any further at this level, the process steps forward to a higher lever; otherwise, it restarts from level 1.

The results of levels 1 and 2 are simplified forms of the original constraint that we cannot prove satisfiable. Conversely, the output of level 3 is guaranteed to be a *solved form constraint* of exactly the same form as the one defined in [11]. We report the definition of solved form constraint here for the reader's convenience.

**Definition 4.1.** Let $C$ be a constraint, $X$, $X_i$ variables, and $t$ a term. A literal $c$ of $C$ is in *solved form* if it satisfies one of the following conditions:

*(i)* $c \equiv X = t$ and neither $t$ nor $C \setminus \{c\}$ contain $X$;

*(ii)* $c \equiv X \neq t$ and $X$ does not occur in $t$;

*(iii)* $c \equiv t \notin X$ and $X$ does not occur in $t$;

*(iv)* $c \equiv \mathsf{union}(X_1, X_2, X_3)$, $X_1 \not\equiv X_2$, and for $i = 1, 2, 3$ there are no disequalities of the form $X_i \neq t$ or $t \neq X_i$ in $C$;

*(v)* $c \equiv X_1$ $\mathsf{disj}$ $X_2$, $X_1 \not\equiv X_2$;

*(vi)* $c \equiv \mathsf{size}(X, N)$ and there are no other literals of the form $\mathsf{size}(X, M)$, with $M \not\equiv N$, in $C$;

*(vii)* for each variable $X$, at most one among: $\mathsf{set}(X)$, $\mathsf{integer}(X)$, $\mathsf{not\_integer}(X)$ is in $C$.

A constraint $C$ is in solved form if it is empty or if all its components are simultaneously in solved form.

The main difference with respect to our previous work, however, is the layered architecture of our solver (in contrast with the monolithic structure of the CLP($\mathcal{SET}$) and CLP($\mathcal{SET}, \mathcal{FD}$) solvers). To obtain the proposed structuring, we introduce a number of new rewrite rules at the lower levels to deal with special cases and domain information, and we define suitable ordering for rule application. In particular, rules are split into deterministic and nondeterministic and all (costly) nondeterministic rules are postponed to the end of the constraint solving cycle. In this way, the solver is given more chance to detect inconsistency before branching any nondeterministic choice.

Furthermore, the proposed layered architecture allows the user deciding the depth of consistency check he/she wants to achieve. As a matter of fact, the user is free to choose to *(i)* stop at any of the first two levels, with no satisfiability warrants, or *(ii)* select which (if any) variable to label if the corresponding domain is known, or optionally *(iii)* skip level 2 in order to avoid constraint propagation, thus letting more time to be spent in searching the solution space rather than in enforcing consistency. Conversely, in CLP($\mathcal{SET}, \mathcal{FD}$) all rewrite rules are considered at the same level, and they are applied

as soon as their pre-conditions are met (except for FD labeling which is applied only as the last step of the computation). The user has no possibility to stop the solver at a certain level, or to skip intermediate levels: all rules (including labeling) are always applied before getting an answer.

## 5. Constraint Solving Rules

This section presents a taxonomy of the rewrite rules we employed for constraint solving. For each level of the solver, we provide full details for the rules related to the constraints $\in$ and size. We limit to those constraints due to space limitations. However, this should enough to give the idea of the approach.

Many of the rewrite rules of our solver are directly derived from CLP($\mathcal{SET}$) and they have been *(i)* adapted to cope with extended set terms, and *(ii)* specialized to benefit from interesting properties of special cases (e.g. ground sets). Moreover, some new rules have been introduced to deal with integer and set domains using FD- and FS-like approaches. Finally, some additional rules have been introduced to handle cardinality constraints, much like in Cardinal [3].

Most of the rewrite rules are direct application of classic set theory adapted to support HFS (see [11] for an in-depth discussion).

The rules are presented as deterministic rewrite rules that operate whenever respective pre-conditions are met:

$$\frac{\textit{pre-conditions}}{\{C_1, \ldots, C_n\} \rightarrow \{C'_1, \ldots, C'_m\}}$$

where $C_1, \ldots, C_n$ and $C'_1, \ldots, C'_m$ ($n$, $m \geq 0$) are primitive HFS-constraints and $\{C_1, \ldots, C_n\} \rightarrow \{C'_1, \ldots, C'_m\}$ represents the changes in the constraint store caused by the application of the rule.

In the rest of this section, we adopt the following notation: $t$, $t_j$ are terms (either ground or not); $a$, $b$, $c$, $c_j$ are ground terms (either integer or not); $i$, $j$, $k$, $i_j$ are integer constants; $X$, $Y$, $Z$, $N$, ... are variables; and $s$, $r$, $s_j$ are extended set terms (either ground or not).

We also introduce a few functions to deal with domains that will be useful in the subsequent presentation.

**Definition 5.1.** Let $X$ be a variable and $CS$ the current constraint store. The function $\mathsf{dom}(X)$ returns the closed domain associated with $X$ in $CS$, or $\top$ if the domain of $X$ is open or $X$ has no domain attached. $\top$ represents the universe of possible values for $X$: hence, $X \in \top$ is always true.

It is worth noting that the domain of a variable $X$ can be stored as a constraint of the form $X \in d$ in the constraint store, as well as an attribute of the variable itself, so to avoid scanning of the entire constraint store in the search for its domain. How $\mathsf{dom}(.)$ is concretely implemented is out of the scope of this paper. We only require that the implementation provides a way to efficiently associate a domain to a variable and to retrieve it when needed.

**Definition 5.2.** Let $d$ be a domain. The function $\mathsf{int\_dom}(d)$ returns $\mathsf{true}$ if $d$ is an *integer domain*; otherwise it returns $\mathsf{false}$. Similarly, the function $\mathsf{set\_dom}(d)$ returns $\mathsf{true}$ if $d$ is a *set domain*; otherwise it returns $\mathsf{false}$.

### 5.1. Level 1

Rewrite rules at level 1 are all deterministic rules and take into account one single constraint at a time. Rules at this level are basically of three kinds:

- **Simplification rules** – Completely remove the considered constraint or (deterministically) replace the constraint with a conjunction of "simpler" constraints (e.g., equalities).
- **Inference rules** – Add new constraints to the existing ones (much like in the form of CHR propagation rules [16]). They are used to ensure, e.g., type and set cardinality coherence, as well as to specify domains that are not explicitly provided.
- **Generation of canonical constraints** – Primitive constraints that are not in a canonical form and that cannot be further simplified are rewritten to semantically equivalent canonical constraints.

Figure 1 shows the complete collection of rules dealing with the membership constraint $\in$ at level 1. Rules from $(2)$ to $(5)$ are simplification rules; rules $(1)$, $(6)$ and $(7)$ are inference rules.

All cases that are not tackled by these rules are considered *irreducible* at level 1 and they are therefore shipped to upper levels. In particular, constraints like $t \in \{t_1, \ldots, t_n \mid s\}$, with $n \geq 1$, $s$ either variable or $\emptyset$, and either $t$ or $t_1, \ldots, t_n$ non-ground terms, are passed unaltered to level 2 (note that constraints of the form $X \in \{t_1, \ldots, t_n \mid s\}$ represent sort of domain declarations that are managed at upper levels).

$$\frac{}{\{t \in s\} \rightarrow \{t \in s, \mathsf{set}(s)\}} \quad (1)$$

$$\frac{}{\{t \in \emptyset\} \rightarrow \mathsf{false}} \quad (2)$$

$$\frac{t_2 \text{ not an interval}}{\{t_1 \in \{t_2\}\} \rightarrow \{t_1 = t_2\}} \quad (3)$$

$$\frac{(c_i \text{ not an interval}, c = c_i) \text{ or } (c_i \equiv i_1..i_2, c \geq i_1, c \leq i_2)}{\{c \in \{c_1, \ldots, c_i, \ldots, c_n\}\} \rightarrow \{\}} \quad (4)$$

$$\frac{(c_i \text{ not an interval}, c \neq c_i) \text{ or } (c_i \equiv i_1..i_2, c < i_1 \text{ or } c > i_2)}{\{c \in \{c_1, \ldots, c_i, \ldots, c_n \mid t\}\} \rightarrow \{c \in \{c_1, \ldots, c_{i-1}, c_{i+1}, \ldots, c_n \mid t\}\}} \quad (5)$$

$$\frac{\mathsf{int\_dom}(s)}{\{X \in s\} \rightarrow \{X \in s, \mathsf{integer}(X)\}} \quad (6)$$

$$\frac{\mathsf{set\_dom}(\{s_1 \,..\, s_2\}), |s_1| = a, |s_2| = b}{\{X \in \{s_1 \,..\, s_2\}\} \rightarrow \{X \in \{s_1 \,..\, s_2\}, \mathsf{size}(X, N), N \in \{a \,..\, b\}\}} \quad (7)$$

Figure 1. Rewrite rules for constraint $\in$ at level 1

Figure 2 shows the rules dealing with the cardinality constraint size at level 1. Rules from $(9)$ to $(11)$ are simplification rules; rules $(8)$ and $(12)$ are inference rules. The following is a simple example of application of rule $(11)$.

**Example 5.1.** If $\mathsf{size}(\{X, Y, 1, 2\}, N)$ is the current constraint, the application of rule $(11)$ adds the new constraint $N \in \{2..4\}$ to the current one. If $\mathsf{size}(\{X, Y, Z\}, N)$ is the current constraint, applying rule $(11)$ adds the constraint $N \in \{1..3\}$.

$$\overline{\{\mathsf{size}(s,t)\} \rightarrow \{\mathsf{size}(s,t), t \geq 0, \mathsf{set}(s), \mathsf{integer}(t)\}} \tag{8}$$

$$\overline{\{\mathsf{size}(\emptyset,t)\} \rightarrow \{t = 0\}} \tag{9}$$

$$\overline{\{\mathsf{size}(\{i_1..i_2\},t)\} \rightarrow \{t = i_2 - i_1 + 1\}} \tag{10}$$

$$\frac{\begin{array}{c} m = \text{ number of distinct ground terms in } \{t_1, \ldots, t_n\} \text{ or} \\ m = 1 \text{ if } t_1, \ldots, t_n \text{ are all non-ground terms} \end{array}}{\{\mathsf{size}(\{t_1, \ldots, t_n\}, t)\} \rightarrow \{\mathsf{size}(\{t_1, \ldots, t_n\}, t), t \in \{m..n\}\}} \tag{11}$$

$$\overline{\{\mathsf{size}(\{t_1 \mid s\}, t)\} \rightarrow \{\mathsf{size}(\{t_1 \mid s\}, t), t \geq 1\}} \tag{12}$$

Figure 2. Rewrite rules for constraint size at level 1

Similar rules apply to other constraints, such as union, intersection and difference, whenever their arguments are sufficiently specified. Moreover, a number of other rules at level 1 are devoted to replace non-canonical constraints with canonical ones. For example:

$$\overline{\{X \subseteq Y\} \rightarrow \{\mathsf{union}(X, Y, Y)\}} \tag{13}$$

replaces $\subseteq$ with a canonical constraint based on union. Applying similar rules to all non-canonical constraints allows the outcome of level 1 to be in canonical form. This feature eases the tasks of level 2 because it can address pairs of constraints in canonical form only.

**Remark 5.1.** While the fact that the output of level 1 is in canonical form is crucial to limit the potential explosion of rules at level 2, it may introduce some inefficiencies in the overall constraint solving process. As a matter of fact, the relation represented by a non-canonical constraint may become no longer visible after the constraint is replaced by the equivalent conjunction of primitive constraints in canonical form. Hence, some optimizations and/or inferences that apply to the original constraint are hardly applicable to the transformed constraint. For example, the non-canonical constraint $\mathsf{inters}(X, Y, Z)$ is replaced by the equivalent canonical constraint $\mathsf{union}(A, Z, X) \wedge \mathsf{union}(B, Z, Y) \wedge A \ \mathsf{disj}\ B$; if after such a rewrite, $X$, $Y$, $Z$ get instantiated to ground set terms, the solver cannot apply the efficient processing rules provided for dealing with intersection of ground sets. The identification of the appropriate trade-off between reducing the number of cases to be managed by upper levels and delaying the elimination of non-canonical constraints is left as an open problem for future work.

Efficiency of the set constraint processing performed at level 1 is crucial for the overall efficiency of our solver. As a matter of fact, in our approach the usual FD/FS reasoning on domains (implemented

by rules of level 2) is completely expressed in terms of set constraints applied to ground extended set terms that represent the domains. For example, domain reduction associated with $x \in d \ \wedge \ x \in d'$, where $d$, $d'$ are ground terms, is implemented by using the constraint $\mathsf{inters}(d, d', D)$ (see rule $(14)$). Such set constraints are completely managed by the simplification rules at level 1. Hence, the efficient implementation of these rules is crucial for obtaining an adequate level of efficiency for the FD/FS-like processing of our solver.

## 5.2. Level 2

Rules at level 2 are all deterministic and they act either upon pairs of constraints or upon one constraint and the domain(s) of some variables occurring in it (as computed by the function dom). Rules at this level are mainly intended to perform the following actions:

- **Domain management** – Add/remove elements from the domains of variables, merge sparse domain information of single variables, etc.
- **Domain utilization** – Use domain information to simplify constraints.
- **Inference** – Add new constraints (e.g., size) to specify the relation among sets occurring as arguments of irreducible set constraints.
- **Consistency check** – Check pairs of constraints to make sure they are consistent (e.g., between type constraints).

As for level 1, we show here the whole set of rules dealing with constraints $\in$ (Figure 3) and size (Figure 4) at level 2.

$$\frac{s \text{ ground}, \mathsf{dom}(X) = r, r \neq \top}{\{X \in s\} \rightarrow \{X \in D, \mathsf{inters}(s, r, D)\}} \tag{14}$$

$$\frac{\mathsf{dom}(Y) = r, r \neq \top}{\begin{array}{c}\{X \in \{t_1, \ldots, t_{i-1}, Y, t_{i+1}, \ldots, t_n\}\} \rightarrow \\ \{X \in D, \mathsf{union}(\{t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_n\}, r, D)\}\end{array}} \tag{15}$$

$$\frac{c \notin \mathsf{dom}(X)}{\{c \in \{t_1, \ldots, t_{k-1}, X, t_{k+1}, \ldots, t_n\}\} \rightarrow \{c \in \{t_1, \ldots, t_{k-1}, t_{k+1}, \ldots, t_n\}\}} \tag{16}$$

Figure 3. Rewrite rules for constraint $\in$ at level 2

The following examples show simple application of rules (15) and (16), respectively.

**Example 5.2.** Given the constraint $X \in \{1, Y, 3\} \wedge Y \in \{10..100\}$, the new domain for $X$ computed by rule $(15)$ is $\{1, 10..100, 3\}$, while given the constraint $X \in \{1, Y, 3\} \wedge Y \in \{6, 7, 8\}$, the new domain for $X$ is $\{1, 3, 6, 7, 8\}$.

**Example 5.3.** Given the constraint $1 \in \{X, Y, 10\}$, the repeated application of rule $(16)$ removes all variable elements whose domains are closed and do not contain 1. If the final constraint is ground, then it can be efficiently solved at level 1; otherwise it is delivered to level 3.

$$\frac{}{\{\mathsf{size}(X, t_1), \mathsf{size}(X, t_2)\} \rightarrow \{\mathsf{size}(X, t_1), t_1 = t_2\}} \quad (17)$$

$$\frac{W \equiv X \text{ or } W \equiv Y \text{ or } W \equiv Z}{\begin{array}{c}\{\mathsf{size}(W, N_W),\ \mathsf{union}(X, Y, Z)\} \rightarrow \\ \{\mathsf{size}(W, N_W),\ \mathsf{union}(X, Y, Z),\ \mathsf{size}(X, N_X), \mathsf{size}(Y, N_Y), \mathsf{size}(Z, N_Z), N_Z \leq N_X + N_Y\}\end{array}} \quad (18)$$

Figure 4. Rewrite rules for constraint size at level 2

Rule $(17)$ makes sure the cardinality of a set is always unique. Rule $(18)$ adds the constraints that express the relation among cardinalities of sets involved in a constraint union. As noted in [3], inferences using cardinalities can be very useful to deduce more rapidly the non-satisfiability of a set of constraints, thus improving efficiency of combinatorial search problem solving.

Similar rules apply to other canonical constraints. In particular:

- the constraint $X \neq t$, with $t$ an integer variable with closed domain $d$ is managed by the following rule:
$$\frac{t \text{ ground}, \mathsf{dom}(X) = s, s \neq \top}{\{X \neq t\} \rightarrow \{X \in D, \mathsf{diff}(s, \{t\}, D)\}} \quad (19)$$
- the constraint $X \leq e$, with $e$ an arithmetic expression and $X$ an integer variable with an integer domain, fires the application of common FD-like rules that cause an FD-like propagation;
- pairs of type constraints are checked to detect type clashes; e.g., $\mathsf{set}(X) \wedge \mathsf{integer}(X)$ immediately fails.

It is worth noting that the HFS-constraints inters, diff and union generated by the above rules are completely and efficiently solved at level 1. Moreover, note that the rules are applicable for any (ground) extended set term, since HFS-constraints (in particular, inters, diff and union) can work equally well with interval and non-interval sets, of either integer or non-integer elements.

## 5.3. Level 3

This level groups all rewrite rules that involve nondeterminism, as well as a number of different rules that, though deterministic, assign specific values to variables. Delaying such rules until this level allows the solver to perform more accurate consistency checking at lower levels rather than spending time in searching. Rules at this level, instead, implement the usual labeling phase that force the assignment of values to variables from their domains, leading to a (chronological) backtracking search of the space of solutions.

$$\frac{}{\{t \in X\} \rightarrow \{X = \{t \mid Y\}, \mathsf{set}(Y)\}} \tag{20}$$

$$\frac{t_1 \text{ not interval}}{\{X \in \{t_1 \mid s\}\} \rightarrow \{X = t_1\} \;\; \mathsf{or} \;\; \{X \in \{t_1 \mid s\}\} \rightarrow \{X \in s\}} \tag{21}$$

$$\frac{t_1 \text{ interval}}{\{X \in \{t_1 \mid s\}\} \rightarrow \{X \in t_1\} \;\; \mathsf{or} \;\; \{X \in \{t_1 \mid s\}\} \rightarrow \{X \in s\}} \tag{22}$$

$$\frac{s \text{ interval}, \; \mathsf{first}(s) = a, \; \mathsf{rest}(s) = r}{\{X \in s\} \rightarrow \{X = a\} \;\; \mathsf{or} \;\; \{X \in s\} \rightarrow \{X \in r\}} \tag{23}$$

Figure 5. Rewrite rules for constraint $\in$ at level 3

As done for the previous levels we show the rules for the constraint $\in$ (Figure 5) and size (Figure 6) provided by our solver at level 3.

Rule $(20)$ transforms a membership constraint into a set equality constraint. Rules from $(21)$ to $(23)$ deal with membership constraints of the form $X \in s$, where $s$ can be any extended set term. These rules allow all possible values from a domain to be assigned to the specified variable (labeling phase). Functions $\mathsf{first}(s)$ and $\mathsf{rest}(s)$, used in rule $(22)$, allow to obtain, respectively, the first element and the rest of $s$, where $s$ can be either an integer or a set interval.

$$\frac{k \geq 1}{\{\mathsf{size}(X,k)\} \rightarrow \{X = \{X_1 \mid Y\}\}, X_1 \notin Y, \mathsf{size}(Y,N), N = k-1\})\}} \tag{24}$$

$$\frac{}{\begin{array}{c}\{\mathsf{size}(\{t_1 \mid s\},t)\} \rightarrow \{t \geq 1, t_1 \notin s, N = t-1, \mathsf{size}(s,N)\} \\ \mathsf{or} \\ \{\mathsf{size}(\{t_1 \mid s\},t)\} \rightarrow \{t \geq 1, s = \{t_1 \mid r\}, t_1 \notin r, N = t-1, \mathsf{size}(r,N)\}\end{array}} \tag{25}$$

Figure 6. Rewrite rules for constraint size at level 3

The following are examples of applications of rules $(24)$ and $(25)$, respectively.

**Example 5.4.** Given the constraint $\mathsf{size}(X,3)$, the repeated application of rule $(24)$ generates the new constraint $X = \{X_1, X_2, X_3\} \wedge X_1 \neq X_2 \wedge X_2 \neq X_3 \wedge X_1 \neq X_3$.

**Example 5.5.** Given the constraint $\mathsf{size}(\{X,Y\},N)$, the repeated application of rule $(25)$ generates the new constraints, $N = 2 \wedge X \neq Y$ or $N = 1 \wedge X = Y$.

Other non-deterministic rules are available at level 3 to deal with general cases of other canonical constraints. For example, the following rule deals with general inequalities between sets:

$$\frac{}{\{s \neq r\} \rightarrow \{Z \notin s, Z \in r\} \;\; \mathsf{or} \;\; \{s \neq r\} \rightarrow \{Z \in s, Z \notin r\}} \tag{26}$$

where $s$ and $r$ are (any) extended set terms.

Note that other forms of solution enumeration are obtained from the treatment of other constraints, e.g., $\mathsf{union}(X, Y, \{1, 2, 3\})$ which assigns to $X$ and $Y$ all possible subsets whose union yields $\{1, 2, 3\}$. From a pragmatic point of view, the treatment of this kind of constraints, that leads to a generalized notion of labeling, could be explicitly activated or deactivated by the user upon request.

## 6. Soundness and Completeness

The constraint language we deal with is the same considered in [11, 9], except for the introduction of extended set terms. This extension, however, does not affect the notion of satisfiability defined in those papers. Conversely, the introduction of new rewrite rules and the adoption of a different, more articulated, control flow make the termination results proved in [11, 9] no longer directly applicable to our case.

We address the termination issue first, then we state correctness and completeness of the global constraint solving procedure.

Termination is proved by first considering level 1 rules ($\mathsf{HFS\_Solve}_1$), then level 2 rules ($\mathsf{HFS\_Solve}_2$), and finally level 3 rules ($\mathsf{HFS\_Solve}_3$). Proving termination of $\mathsf{HFS\_Solve}_3$ implies proving termination of the whole solver $\mathsf{HFS\_Solve}$.

**Lemma 6.1.** $\mathsf{HFS\_Solve}_1$ terminates for every input constraint $C$.

**Proof:**
[Sketch] We consider separately the three kinds of rules at level 1.

- Inference rules. We assume the implementation guarantees that each inference rule is fired only once on a given constraint $C$. Moreover, we can see, by case analysis, that the processing of constraints added by an inference rule does not generate any constraint of the same kind as those that fired the rule itself. Therefore, inference rules may add only a finite number of new constraints.
- Generation of canonical constraints. If sufficiently instantiated, non-canonical constraints are completely eliminated at level 1, namely they are rewritten to either $\mathsf{true}$ or $\mathsf{false}$ or to a conjunction of equalities. Otherwise, non-canonical constraints are replaced by the equivalent conjunction of canonical constraints that are delivered to upper levels. We can see, by case analysis, that the processing of equalities does not generate any non-canonical constraint. More generally, no canonical constraint handling procedure generates non-canonical constraints. Therefore, rules for the generation of canonical constraints may be fired only a finite number of times.
- Simplification rules. Without loss of generality, we assume that interval terms are replaced by the corresponding explicit enumeration of their elements. We introduce a simple complexity measure *size* which is recursively defined as follows:

$$size(t) = \begin{cases} 0 & \text{if } t \text{ is a variable} \\ 1 & \text{if } t \text{ is a constant} \\ 1 + \sum_{i=1}^{n} size(t_i) & \text{if } t = f(t_1, \ldots, t_n), f \in \mathcal{F} \\ \sum_{i=1}^{n} size(t_i) & \text{if } t = p(t_1, \ldots, t_n), p \in \Pi_{\mathcal{C}} \\ \sum_{i=1}^{n} size(t_i) & \text{if } t = \neg p(t_1, \ldots, t_n), p \in \Pi_{\mathcal{C}} \\ size(C_1) + size(C_2) & \text{if } t = C_1 \wedge C_2 \end{cases}$$

By case analysis, we can show that the application of each inference rule at level 1 to a constraint $C$ decreases the complexity measure *size(C)*. These assures that the repeated application of simplification rules always terminates.

If $C_1$ is the (finite) collection of constraints generated by the inference rules, $C_2$ is the (finite) collection of constraints generated by dealing with non-canonical constraints, and $C$ is the input constraint, then *size*$(C_1 \wedge C_2 \wedge C)$ decreases at each iteration of $\mathsf{HFS\_Solve}_1$ over $C_1 \wedge C_2 \wedge C$. Hence, $\mathsf{HFS\_Solve}_1$ eventually terminates. □

We can adopt the same technique to prove termination of $\mathsf{HFS\_Solve}_2$, but taking into account the more complicated interdependencies among the different rules.

**Lemma 6.2.** $\mathsf{HFS\_Solve}_2$ terminates for every input constraint $C$.

**Proof:**
[Sketch] We distinguish between inference rules and other rules, and for both of them we consider the number of variables occurring in the constraint.

- Inference rules. These rules may add new constraints to the existing ones. In particular, they can add new variables. However, we can prove that the number $v$ of new variable occurrences possibly introduced by the repeated application of inference rules within $\mathsf{HFS\_Solve}_2$ is finite. To prove this property, we need to check, by case analysis, that the processing of the added constraints does not generate any constraint of the same kind as those that fired the rule itself. This is the same check performed for inference rules at level 1 but complicated by the fact that we need to take into account both level 1 and level 2 rules. However, notice that at level 2 only canonical constraints need to be considered, since non-canonical constraints are completely eliminated by rules at level 1. The finiteness of the number of added constraints assures that only a finite number of new variable occurrences may be added by these rules.

- Non-inference rules. For the other rules at level 2 we can prove termination by introducing a suitable complexity measure, and by showing that the application of these rules produces constraints with smaller complexity.

  Let $vars{+}(C)$ be the multiset of all variable occurrences in $C$. We define the following complexity measure for a constraint $C$:

  $$mul(C) = |\{\![\, x \,:\, x \in^n vars{+}(C), n \geq 2 \,]\!\}|$$

  where $\{\![\ldots]\!\}$ denotes a multiset, $x \in^n m$ is the multiset membership, i.e., $x$ occurs $n$ times in the multiset $m$, and $|m|$ denotes the cardinality of $m$. We can prove, by case analysis, that the application of each non-inference rules at level 2 eventually reduces $mul(C)$ (note that, the fact that for a variable $X$ we have $\mathsf{dom}(X) = r, r \neq \top$, implies that $X$ has at least one occurrence in the current constraint). We show this for rule $(14)$; for other rules, we can use the same reasoning.

  The direct application of rule $(14)$ possibly increases *mul* by 1. Moreover, from $X \in D$ we get at level 1 also the constraint $\mathsf{set}(D)$, hence *mul* is increased again. However, thanks to the requirement that $s$ and $r$ must be ground, the constraint $\mathsf{inters}(s, r, D)$ is completely eliminated at

level 1, and we get $D = w$, where $w$ is a (ground) set term. By applying variable substitution for $D$ on the whole constraint, we finally obtain that the initial constraint $X \in s \wedge X \in r$ is replaced by $X \in w \wedge D = w$; hence, *mul* is decreased by 1.

If $C_1$ is the (finite) collection of constraints generated by the inference rules at level 2, and $C$ is the input constraint, then *mul*($C_1 \wedge C$) decreases at each iteration of HFS_Solve$_2$ over $C_1 \wedge C$. Hence, HFS_Solve$_2$ eventually terminates. □

Similar techniques can be used also to prove termination of HFS_Solve$_3$, hence of the whole solver. However, in this case, one has to account for very general rules, which possibly involve partially specified sets and non-determinism.

**Theorem 6.1.** HFS_Solve terminates for every input constraint $C$.

**Proof:**
[Sketch] Rules at this level are basically the rules provided by CLP($\mathcal{SET}$). The termination proof can be conducted in the same (quite complex) way presented in [11]. We start by proving that the rewriting for each different kind of constraints terminates (local termination). In particular, the proof for equality constraints can be adapted from that in [12] which is much simpler than that in [11]. After proving local termination, we show that the algorithm, deprived by the treatment of the union constraint, always terminates by making use of a suitable complexity measure. Specifically, this measure uses a level function $lev$ that computes the height of a term $t$. The intuitive idea behind this part of the proof is that it is possible to determine a bound on the height of the terms which can be generated during the constraint solving process. During this process, it is possible to show that the algorithm operates on terms which have progressively decreasing height. Finally, we insert union in the reasoning and we prove the global termination result. □

The termination proof of the whole solver HFS_Solve adapted from [11] is rather involved, and its complete description is out of the scope of this paper. Developing a simpler termination proof, that can fully take advantage of the layered structure of the solver, is a goal for future work.

**Remark 6.1.** The condition that the sets in rules $(14)$, $(15)$ and $(19)$ must be ground is important to simplify the termination analysis, but it is not strictly necessary for the rule correctness and could be advantageously relaxed in some cases. As an example of one of such cases, consider the constraint $X \in \{1, 3, 5, Y\} \wedge X \neq 3$ which states that the domain of $X$ contains the values 1, 3, 5 and possibly another value $Y$ which is left unspecified for the moment, and that $X$ must be different from 3. If we remove the groundness condition of rule $(19)$, then we could apply such a rule and we would get the new equivalent constraint (of smaller *size*) $X \in \{1, 5, Y\} \wedge Y \neq 3$. Thus, application of rule $(19)$ would allow us to get a reduced domain for $X$ although it is not completely specified.

In the general case, however, removing the groundness condition of rules like $(14)$ and $(19)$ makes the termination proof more complicated. Furthermore, the solution of constraints like diff and inters applied to general sets may lead to a substantial decrease in the efficiency of the rewriting process, due to nondeterminism. For these reasons we prefer for now to restrict applicability of these rules to the ground case only. A more precise analysis of cases in which it may be convenient to allow non-ground cases to be considered is left for future work.

Assuming termination, we can state the following result for our solver that is directly adapted from [11] and [9].

**Theorem 6.2.** Let $C$ be a constraint, $vars(C)$ the set of (all) variables in $C$, $\mathcal{V}_{\in}$ the set of variables $X$ for which there exists a constraint $X \in d$ in $C$, and $d$ be a closed domain. If $\mathcal{V}_{\in} = vars(C)$, then

1. if $C$ is satisfiable, then the output of HFS_Solve is the equisatisfiable collection of solved form constraints $C_1, \ldots, C_n$;
2. if $C$ is not satisfiable, then the output of HFS_Solve is false.

**Proof:**
[Sketch] The output of level 3 is a *solved form constraint* of the same form as the one defined in [11, 9]. Then, the proof can be conducted as in that paper. First, solved form constraints are proved to be always satisfiable in the considered interpretation structure. Then, the overall constraint solving procedure is shown to transform a constraint $C$ into a collection of constraints in solved form $C_1, \ldots, C_n$, provided all variables in $C$ have a closed domain. Finally, the disjunction of all the generated solved form constraints is proved to be equisatisfiable to the original constraint $C$ (roughly speaking, $C$ and $C_1 \vee \ldots \vee C_n$ have the same set of solutions). Thus, from the ability to transform $C$ into $C_1, \ldots, C_n$ derives the satisfiability of $C$. □

# 7. Efficiency Considerations

The HFS_Solve procedure embeds FD and FS constraint solving as particular cases. Actually, one of the main duties of level 2 is to identify FD and FS sub-problems and treat them accordingly. This fact can be exploited, at the implementation level, to reuse existing FD/FS solvers, instead of having to re-implement them from scratch.

Making use of an existing FD solver within a general set constraint solver has been investigated in detail in [8, 9]. In that proposal, the integration of the different solvers is achieved through a form of solver cooperation [21], where a master solver (actually an extension of the CLP($\mathcal{SET}$) solver) invokes a slave FD solver (namely, the FD solver of SICStus in the current implementation), whenever constraints over FD/FS domains have to be processed.[3] Whenever during the rewriting process the master solver finds out that a constraint can be managed by the embedded FD solver, it passes the constraint to that solver. The resulting simplified constraint (e.g., reduced through arc and bound consistency) is then added back to the constraint store of the master, thus propagating the simplifications performed by the FD solver to the set constraints.

The possibility of exploiting existing (efficient) FD/FS solvers within our general solver allows it to deal with HFS-constraints that can be viewed as FD/FS constraints with almost the same efficiency as they were processed by the corresponding embedded solvers. Some (constant) cost has to be payed by our solver due to the overhead required by the switching between solvers.

As a simple example of this, if $S$ is the set of the integer numbers between 1 and 1000 and $R$ is the set of the integer numbers between 1000 and 10000, solving the constraint $X \in S \wedge Y \in R \wedge X = Y$

[3]The use of a master/slave architecture for solver cooperation, akin to the one considered here, has been discussed in more general terms in [4].

in CLP($\mathcal{SET}$) requires the explicit enumeration of the possible bindings of $X$ and $Y$ to detect the unique solution $X = 1000, Y = 1000$. On the other hand, the HFS solver can detect that the domains are known sets of integers, and thus it can exploit the efficient FD solving techniques to directly jump to the desired solution.

Conversely, whenever the input constraint cannot be considered as a FD/FS constraint (e.g., the domains are not known sets of integers), our HFS-constraint solver is not able to take advantage of the efficient FD/FS solvers and it solves the constraint using its more general, though less efficient, constraint solving procedure.

As concern the computational complexity of our solver it must be noted that, since it uses set unification, it is NP in the general case. However, as observed in [12], complexity of the set unification operation depends on the allowed form of set terms (e.g., flat or nested sets, with zero, one, or more set variables). Thus, different complexity results can be obtained for different classes of set terms. For instance, while the set equivalence test of ground set terms denoting flat sets, such as $\{a, b, c\}$ and $\{b, c, a\}$, is rather easy, when the decision problem deals with nested set terms involving variables it becomes NP-complete. Moreover, various works have been proposed to study, in particular, the simple cases of matching (where variables are allowed to occur in only one of the two set terms which are compared) and unification of *bound simple* set terms, i.e., bound set terms of the form $\{s_1, \ldots, s_n\}$, where each $s_i$ is either a constant or a variable [2, 20].

Our constraint solver can take advantage of its layered structure. In particular simpler unification problems (i.e., those involving set terms of simpler forms) are dealt with at lower levels, while more complex problems are passed unchanged to upper levels. In general, some classes of problems can be solved at lower levels, without incurring in the cost of higher levels. Moreover, when the solver detects inconsistencies at lower level, the full power of set unification is not required at all, resulting in a more efficient resolution.

## 8. Related Work

This work represents a natural evolution of our previous research on programming with sets, starting from the logic language {log} [10], and later developed as the constraint logic programming language CLP($\mathcal{SET}$) [11]. An extensive comparison with related works dealing with constraint programming and logic programming in the presence of sets can be found in [11]. Notably, the current proposal offers various advantages over CLP($\mathcal{SET}$):

- $(i)$ integrating the domains of integer numbers and general sets;
- $(ii)$ allowing the definition and handling of constraints that use both domains, e.g., the set cardinality constraint size;
- $(iii)$ taking advantage of FD/FS-like constraint solving techniques to efficiently solve problems involving FD/FS-like constraints;
- $(iv)$ extending the notion of set term to that of extended set term to allow a compact representation of sets and intervals;
- $(v)$ providing a new layered architecture for the solver that allows: $(a)$ possible inconsistencies to be detected at earlier stages in the constraint solving process; $(b)$ special cases to be singled out and dealt with straightforwardly; $(c)$ the depth of consistency check to be specified by the user.

Advantages from $(i)$ to $(iii)$ are present also in [8] (though $(iii)$ is limited to FD constraints); other features are new.

As an example of $(iv)$, given the constraint

$$\mathsf{diff}(\{1..1000\}, 10, S)$$

the computed answer for $S$ is compactly represented by an extended set term

$$S = \{1..9, 11..1000\},$$

whereas in CLP($\mathcal{SET}, \mathcal{FD}$) the same is represented by (ordinary) set terms, hence requiring the explicit enumeration of all possible values of the resulting set:

$$S = \{1, 2, \ldots, 9, 11, \ldots, 999, 1000\}.$$

As a simple example of a special case which is dealt efficiently at level 1, consider the constraint

$$\mathsf{union}(\{X, 1\}, \{2, Y\}, R)$$

that is the union of two sets representing bounded (open) domains. The computed answer for $R$ is $R = \{X, 1, 2, Y\}$, whereas in CLP($\mathcal{SET}, \mathcal{FD}$), using the more general (non-deterministic) rule dealing with general sets, we get 8 (redundant) solutions.

The current proposal offers also some advantages over existing FD and FS solvers:

- $(i)$ providing a uniform framework where HFS, integers and set intervals coexist, and all are manipolable through a single uniform collection of primitive constraints endowed with a clear set-theoretical semantics;
- $(ii)$ allowing domains to be constructed dynamically, as the result of set-theoretic operations over sets and intervals;
- $(iii)$ allowing domains to contain unknown elements and, nevertheless, to be processed by the solver.

Feature $(ii)$ is supported also in SICStus Prolog through the FDSET structure. FDSET terms, however, do not allow domains to be partially specified, and they support feature $(i)$ in a very rough way. As a matter of fact, FDSET terms are basically lists and operations on FDSET terms have no clear set theoretic semantics (see Remark 3.1).

As an example of features $(ii)$ and $(iii)$, consider the following HFS-constraint:

$$D = \{5..15\}, N = \{X|D\}, X \in 1..100000, 0 \in N,$$

that is $N$ represents a bounded open domain, which is build dynamically by the addition of a new unknown element $X$. Our solver HFS_Solve is able to efficiently reduce this constraint to false, by applying rule $(16)$ and then rules $(5)$ and $(2)$:

$$0 \in \{X, 5..15\} \rightarrow 0 \in \{1..100000\} \rightarrow 0 \in \{\} \rightarrow \mathsf{false}.$$

Conversely, using FDSETterms of SICStus Prolog the same problem could be coded as follows:

```
D = [[5|15]], fdset_add_element(D,X,N), X in 1..100000, 0 in_set N.
```

However, in this form this constraint cannot be solved by the FDSET solver, due to the requirement that all arguments of the built-in `fdset_add_element` are ground (as a matter of fact, SICStus detects an instantiation error in argument 2). In order to allow the constraint to be processed, we can add an explicit labeling on the variable `X`, e.g., by `indomain(X)` before calling `fdset_add_element`. In this way the solver is able to detect the inconsistency, but through a complete, time-consuming, search of the whole solution space (with response times becoming unacceptable as the domain of $X$ becomes a very large one).

Two main drawbacks of the current version of our language and solver with respect to existing FD/FS solvers are:

$(i)$ only a prototype version has been implemented;

$(ii)$ the constraint language could be further extended: most existing FD solvers provide more extensive coverage of constraints (e.g., global constraints), which, however, could be easily accommodate for within our language of HFS-constraints.

As regards related work, a work similar in spirit to the one provided here is the proposal by Flener et al. [13], where the authors compile the language ESRA, that includes set-based primitives, into the modeling language OPL—i.e., thus transforming the high level set constructions into efficient lower-level constraints. ESRA [14] is a relational modeling language, where relations are viewed as sets of tuples, constructed using various set operations (e.g., Cartesian product, intensional sets). The language allows the construction of set and arithmetic expressions, and it includes aggregation capabilities. More recently, a project for the compilation of a subset of ESRA to SICStus Prolog has been described [24]. Such a subset disallows intensional sets, cardinality constraints, and it introduces strong restrictions on the domains (e.g., groundness at run-time).

Other modeling languages (e.g., Numerica [30] and AMPL [15]) introduce sets expressions in the context of constraint modeling, but with relatively more limited features, e.g., Numerica requires sets to be constructed from ground ranges, thus disallowing partially specified sets.

Finally, the proposal described in [17] provides a technique to enhance CLP($\mathcal{FD}$) by introducing the notion of *incrementally specified sets*, and allowing such sets to be used as domains for finite domain variables. The proposal makes explicit use of CLP($\mathcal{SET}$) to handle the construction of incrementally specified sets. This scheme can be seen as a special case of our proposal, as it allows the use of set constructs only for the construction of finite (flat) domains.

## 9. Conclusion and Future Work

In this paper we have presented a constraint language and its solver that generalize the usual FD and FS constraint proposals by allowing domains to be general sets, containing elements of any kind, possibly nested and partially specified. The presented work is also an improvement over CLP($\mathcal{SET}$) and its extension CLP($\mathcal{SET}, \mathcal{FD}$) since the proposed solutions represents a better integration of intervals and

sets, a more flexible and effective interaction with the FD solver, and, finally, it allows a larger class of set constraints to be efficiently processed, while retaining the expressive power of the original CLP($\mathcal{SET}$) proposal.

The presented work has been mainly performed on a theoretical basis and just some preliminary experiments are available. However, previous experiences with the integration of FD and CLP($\mathcal{SET}$), both within the Prolog interpreter {log} [26] and within the Java library JSetL [25], gave us some feedback about the feasibility and the possible performance improvements of this approach.

For the near future, we plan to extend the current implementations to completely include the constraint solving technique described in this paper. This will be achieved by:

- extending the current representation of sets by allowing {log} and JSetL to support extended set terms that mix enumerated set elements and intervals;
- providing efficient implementation of all constraints in the ground cases, including those intended to handle integer and set intervals;
- implementing rules for FS constraints to effectively manage set domains, as provided, e.g., by Conjunto and Cardinal; and
- structuring the whole solver in terms of the layered architecture described in Section 5.

## References

[1] APT, K.R. (2003) *Principles of Consraint Programming*. Cmbridge University Press.

[2] ARNI, N., GRECO, S., AND SACCÀ, D. (1996) Matching of Bounded Set Terms in the Logic Language LDL++. *J. of Logic Programming*, 27(1):73–87.

[3] AZEVEDO, F. (2007) Cardinal: A Finite Sets Constraint Solver. *Constraints*, 12(37):93–129.

[4] BERGENTI, F., PANEGAI, E., AND ROSSI, G. (2006) A Master-Slave Architecture to Integrate Sets and Finite Domains in Java. Presented at CILC'06 – Convegno Italiano di Logica Computazionale, Bari.

[5] BOUQUET, F., LEGEARD, B., AND PEUREUX, F. (2002) CLPS-B - a constraint solver for B. In *Tools and Algorithms for the Construction and Analysis of Systems*, Vol. 2280 of LNCS, Springer Verlag, 188–204.

[6] CODOGNET, P., AND DIAZ, D. (1996) Compiling constraints in CLP(FD). *Journal of Logic Programming*, 27(3):185–226.

[7] CUCCHIARA, R., GAVANELLI, M., LAMMA, E., MELLO, P., MILANO, M., PICCARDI, M. (online) Extending the CSP Model to Cope With Partial Information. Available at: `http://lia.deis.unibo.it/Research/Areas/icsp`

[8] DAL PALÙ, A., DOVIER, A., PONTELLI, E., AND ROSSI, G. (2003) Integrating Finite Domain Constraints and CLP with Sets. In D. Miller, ed., *Fifth ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming*, ACM Press, 219–229.

[9] DAL PALÙ, A., DOVIER, A., PONTELLI, E., AND ROSSI, G. (2006) Constraint Logic Programming Language for Effective Programming with Sets and Finite Domains. Research Report "Quaderno del Dipartimento di Matematica", 437, Università di Parma.

[10] DOVIER, A., OMODEO, E. G., PONTELLI, E., AND ROSSI, G. (1996) {log}: A Language for Programming in Logic with Finite Sets. *Journal of Logic Programming*, 28(1):1–44.

[11] DOVIER, A., PIAZZA, C., PONTELLI, E., AND ROSSI, G. (2000) Sets and Constraint Logic Programming. *ACM Transactions on Programming Languages and Systems*, 22(5):861–931.

[12] DOVIER, A., PONTELLI, E., AND ROSSI, G. (2006) Set unification. *Theory and Practice of Logic Programming*, 6:645–701.

[13] FLENER, P., HNICH, B., AND KIZILTAN, Z. (2001) Compiling high-level type constructor in constraint programming. In I.V. Ramakrishnan ed., *Practical Aspects of Declarative Languages*, Vol. 1990 of LNCS, Springer Verlag, pp. 229–244.

[14] FLENER, P., PEARSON, J., AND AGREN. M. (2004) Introducing ESRA, a relational language for modeling combinatorial problems. In M. Bruynooghe ed., *LOPSTR'03: Revised Selected Papers*, pp. 214-232, Springer-Verlag.

[15] FOURER, R., GAY, D., AND KERNIGHAN, B.W. (1993) *AMPL: a modeling language for mathematical programming.* The Scientific Press.

[16] FRÜHWIRTH, T. (1998) Theory and practice of constraint handling rules. *J. of Logic Programming*, 37(1-3):95–138.

[17] GAVANELLI M., LAMMA E., MELLO P., AND MILANO, M. (2005) Dealing with incomplete knowledge on CLP(FD) variable domains. *ACM Transactions on Programming Languages and Systems*, 27(2):236–263.

[18] GERVET, C. (1997) Interval Propagation to Reason about Sets: Definition and Implementation of a Practical Language. *Constraints*, 1(3):191–244.

[19] GERVET, C. (2006) Constraints over Structured Domains. In Rossi, F. van Beek, P., and Walsh, T. (Eds.), Handbook of Constraint Programming, Elsevier.

[20] GRECO S. (1996) Optimal Unification of Bound Simple Set Terms. In *Proc. of Conf. on Information and Knowledge Management*, 326–336, ACM Press.

[21] HOFSTEDT, P. (2000) Cooperating Constraint Solvers. In Dechter, R. (Ed.), *International Conference on Principle and Practice of Constraint Programming*, Vol. 1894 of LNCS, Springer Verlag, 520–524.

[22] IC PARC (2003) *The ECLiPSe Constraint Logic Programming System*. London. `www.icparc.ic.ac.uk/eclipse/`.

[23] LECONTE, M. (1996) A bounds-based reduction scheme for constraints of difference. In Proceedings of the Constraint-96 International Workshop on Constraint-Based Reasoning, pp. 19–28

[24] NORBERG, M. (2006) *Writing a compiler for the finite domain CSP modeling language ESRA.* Master's Thesis, Department of Information Technology, Uppsala University.

[25] PANDINI, D. (2008) *Progettazione e realizzazione in Java di un risolutore di vincoli su domini finiti*. Tesi di Laurea, Dipartimento di Matematica, Università degli Studi di Parma.

[26] ROSSI, G. (2005) *The {log} Constraint Logic Programming Language*. `prmat.math.unipr.it/~gianfr/setlog.Home.html`.

[27] ROSSI, G., PANEGAI, E., AND POLEO, E. (2007) JSetL: A Java Library for Supporting Declarative Programming in Java. *Software-Practice & Experience*, 37:115-149.

[28] VAN ROY, P. (ED.) (2005) Multiparadigm Programming in Mozart/Oz. Lecture Notes in Computer Science 3389 Springer 2005, ISBN 3-540-25079-4.

[29] SWEDISH INSTITUTE OF COMPUTER SCIENCE. *The SICStus Prolog Home Page*. `www.sics.se`.

[30] VAN HENTENRYCK, P., MICHEL, L., AND DEVILLE, Y. (1997) *Numerica: a modeling language for global optimization.* MIT Press.

[31] WIELEMAKER, J. (2004) *SWI-Prolog Reference Manual (Version 5.4).* University of Amsterdam.

[32] ZHOU, N-F. (2005) *B-Prolog User's Manual (Version 6.8).* Afany Software.