Integrating Finite Domain Constraints and CLP with Sets

A. Dal Palù, A. Dovier
Università di Udine
Dip. di Mat. e Informatica.
dalpalu@dimi.uniud.it
dovier@dimi.uniud.it

E. Pontelli
New Mexico State University
Dept. Computer Science.
epontell@cs.nmsu.edu

G. Rossi
Università di Parma
Dip. di Matematica.
gianfr@prmat.math.unipr.it

ABSTRACT

In this paper we propose a semantically well-founded combination of the constraint solvers used in the constraint programming languages CLP($\mathcal{SET}$) and CLP($\mathcal{FD}$). This work demonstrates that it is possible to provide efficient executions (through CLP($\mathcal{FD}$) solvers) while maintaining the expressive power and flexibility of the CLP($\mathcal{SET}$) language. We develop a combined constraint solver and we show how static analysis can help in organizing the distribution of constraints to the two constraint solvers.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*Constraints*; D.1.6 [**Programming Techniques**]: Logic Programming

General Terms

Languages, Algorithms, Performance

Keywords

Programming with sets, constraint logic programming

1. INTRODUCTION

The literature is rich of proposals aimed at developing declarative programming frameworks that incorporate different types of *set-based primitives* (e.g., [1, 11, 15, 5, 6, 3]). These frameworks provide a high level of data abstraction, due to the direct availability of the popular language and operations of Set Theory. These features make this type of languages particularly suited for system modeling and rapid software prototyping.

Among these proposals, CLP($\mathcal{SET}$) [6] provides very flexible and general forms of sets while preserving a clean semantics and a true declarative reading. CLP($\mathcal{SET}$) is a constraint logic programming language whose constraint domain is that of *hereditarily finite sets*. CLP($\mathcal{SET}$) allows sets to be *nested* and *partially specified*, e.g., set elements can contain unbound variables and sets can be only partially enumerated. CLP($\mathcal{SET}$) provides a collection of primitive constraint predicates sufficient to cover all the basic set-theoretic operations (along with their negative counterparts, e.g., $\in$ and $\notin$). In [6] we presented a complete constraint solver capable of deciding the satisfiability of arbitrary conjunctions of these primitive set constraints.

One of the main criticisms moved to these approaches is that the intrinsic computational costs of some set operations (e.g., the NP-completeness of unification between sets) make these languages suited for fast prototyping but not for the development of efficient solutions. On the other hand, various proposals have been made to introduce constraint-based language primitives for dealing with simple forms of sets—i.e., *finite* subsets of a predefined *domain* (typically, the set $\mathbb{Z}$ of integer numbers). Constraint-based manipulation of finite domains provides the foundations of the popular CLP($\mathcal{FD}$) framework [17, 4], that has been demonstrated well-suited for the encoding and the fast resolution of combinatorial and optimization problems [12].

The overall goal of this work is to develop a bridge between the expressive power and the high level of abstraction offered by constraint solving over generic hereditarily finite sets—as possible in CLP($\mathcal{SET}$)—and the efficient use of constraint propagation techniques provided by constraint solving over finite domains—as possible in CLP($\mathcal{FD}$). We combine the constraint logic programming languages CLP($\mathcal{SET}$) and CLP($\mathcal{FD}$) in order to use, when possible, the efficiency of the CLP($\mathcal{FD}$) constraint solver, while maintaining the expressive power and flexibility of the CLP($\mathcal{SET}$) language. Ultimately, the combination of the two models leads to a framework where high-level set operations allow flexible manipulation of domains (viewed as sets), while the constraint propagation mechanisms of CLP($\mathcal{FD}$) allow us to efficiently handle uses of simple sets.

In order to accomplish these results, we propose an extension of the CLP($\mathcal{SET}$) framework by introducing integer constants and integer intervals, as a natural extension of the existing constraint handling capabilities—the new language is called CLP($\mathcal{SET}$)$^{\mathsf{int}}$. In particular, intervals are naturally viewed as set terms and processed according to such view. We then integrate the CLP($\mathcal{FD}$) constraint solver and the CLP($\mathcal{SET}$) constraint solver, by allowing the latter to exploit the former whenever constraints over intervals or equations and disequations over integers have to be processed. We develop novel constraint solving algorithms that allow the two solvers to communicate and cooperate in verifying

PDPP'03, August 27–29, 2003, Uppsala, Sweden.

the satisfiability of constraint formulae over the combined structure (finite domains + hereditarily finite sets). We also present static analysis strategies for CLP($\mathcal{SET}$) programs which make it possible to automatically discover situations in which the CLP($\mathcal{FD}$) constraint solver can be used in place of the CLP($\mathcal{SET}$) solver to check constraint satisfiability. The technique is based on the ability of the program analyzer to determine those finite sets of integers which can be easily mapped to intervals and therefore are likely to be processed more efficiently by the CLP($\mathcal{FD}$) constraint solver. Moreover, the analyzer can detect entire fragments of CLP($\mathcal{SET}$) programs that can be completely translated to equivalent CLP($\mathcal{FD}$) programs and as such executed more efficiently.

This work represents a natural evolution of the previous research on programming with sets, starting from the logic language {log} [5], and later developed as a constraint logic programming language (CLP($\mathcal{SET}$) [6]) and endowed with more set-theoretic primitives. The methodology proposed in this paper extends the ideas in CLP($\mathcal{SET}$) in two directions. On one hand, we provide CLP($\mathcal{SET}$) with a methodology to execute programs more efficiently, by relying on the fast propagation mechanisms developed for CLP($\mathcal{FD}$); on the other hand, we allow CLP($\mathcal{SET}$) to become a high-level language to express constraint-based manipulation of CLP($\mathcal{FD}$) domains. A similar idea is behind the work by Flener et al. [7], where the authors compile the language ESRA, that includes set-based primitives, into the modeling language OPL. Gervet in [8] introduces set operations over finite domains in the CLP language ECLiPSe [9]. In [19] the authors propose a tuple-based treatment of sets in ECLiPSe and compare it with Gervet's approach. Both these two approaches, however, do not allow a complete and easy handling of programs involving sets.

The paper is organized as follows. In Section 2 we summarize the main distinguishing features of CLP($\mathcal{SET}$) and we show how intervals can be added to this framework. In Section 3 we show how the solver of CLP($\mathcal{SET}$) can be improved to deal with intervals. The syntax and main features of CLP($\mathcal{FD}$) are summarized in Section 4. In Section 5 we describe a semantically sound integration of the two constraint solvers. In Section 6 we show how extra predicates on integers (e.g., $\leq$) and on sets (e.g., cardinality) can be dealt with in this framework. In Section 7 we show how to use static analysis for obtaining useful information for the distribution of the constraints to the two constraint solvers. Conclusions and future work are reported in Section 8.

2. CLP($\mathcal{SET}$) WITH INTERVALS

The signature Σ upon which CLP($\mathcal{SET}$) [6] is based is composed of the set $\mathcal{F}$ of function symbols, that includes $\emptyset$ and $\{\cdot \,|\, \cdot\}$, the set $\Pi_c = \{=, \in, \cup_3, ||, \mathtt{set}\}$ of constraint predicate symbols, and a denumerable set $\mathcal{V}$ of variables. The intuitive semantics of the various symbols is the following: $\emptyset$ represents the empty set, while $\{t \,|\, s\}$ represents the set composed of the elements of the set s plus the element t (i.e., $\{t \,|\, s\} = \{t\} \cup s$). $=$ and $\in$ represent the equality and the membership relations. $\cup_3$ represents the union relation: $\cup_3(r, s, t)$ holds if $t = r \cup s$. The predicate $||$ captures the disjoint relationship between two sets: $s \,||\, t$ holds if $s \cap t = \emptyset$. The constraint predicate set is used to enforce the multi-sorted nature of the language; terms are separated into two sorts, one representing set terms (called `Set`) and one representing non-set terms (called `Ker`). The predicate set checks whether a term belongs to the sort `Set` (e.g., $\mathtt{set}(\{a, b\})$ holds while $\mathtt{set}(1)$ does not). We will use the notation $\{t_1, t_2, \ldots, t_n \,|\, t\}$ as a shorthand for $\{t_1 \,|\, \{t_2 \,|\, \cdots \{t_n \,|\, t\} \cdots\}\}$ and the notation $\{t_1, t_2, \ldots, t_n\}$ as a shorthand for $\{t_1 \,|\, \{t_2 \,|\, \cdots \{t_n \,|\, \emptyset\} \cdots\}\}$. Observe that one can write terms representing sets nested at any level; for instance: $\{\, X, \{\emptyset, \{a\}\}, \{\{\{b\}\}\}\,\}$.

The semantics of CLP($\mathcal{SET}$) has been described with respect to the predefined structure $\mathcal{A}_{\mathcal{SET}} = \langle \mathcal{S}, (\cdot)^{\mathcal{S}} \rangle$ [6]. The interpretation domain $\mathcal{S}$ is a subset of the set of (well-formed) ground terms built from symbols in $\mathcal{F}$, i.e., $T(\mathcal{F})$. Terms are partitioned into equivalence classes according to the set-theoretical properties expressed by the two axioms:

$$\begin{array}{llcl} (Ab) & \{X \,|\, \{X \,|\, Z\}\} & = & \{X \,|\, Z\} \\ (C\ell) & \{X \,|\, \{Y \,|\, Z\}\} & = & \{Y \,|\, \{X \,|\, Z\}\} \end{array}$$

Axiom (Ab) states that duplicates in a set do not matter (*Absorption property*). Axiom $(C\ell)$ states that the order of elements in a set is irrelevant (*Commutativity on the left*). Thus, for example, $\{b, a\}$, $\{a, a, b\}$, $\{a, b, b\}$, are all placed in the same equivalence class. Let $\cong$ be the least congruence relation $\cong$ over $T(\mathcal{F})$ which contains the equational axioms (Ab) and $(C\ell)$. This relation induces a partition of $T(\mathcal{F})$ into equivalence classes. The set of these classes will be denoted by $T(\mathcal{F})/\cong$. A total ordering $\prec$ on $T(\mathcal{F})$ can be used to identify a *representative* term from each congruence class in $T(\mathcal{F})/\cong$ [5]. Hence, each set term is mapped to a set term in a normalized form, where duplicates have been removed and elements are listed in a predefined order. E.g., $\{a, b\}$ can be the representative of the class containing itself, $\{b, a\}, \{b, a, b, a\}$, and so on. On the other hand, a non-set term which does not contain any set term, e.g., $f(a)$, is mapped to itself. The *domain* $\mathcal{S}$ is therefore defined as the the set of objects of $T(\mathcal{F})$ that behave as class representative and the *interpretation function* $(\cdot)^{\mathcal{S}}$ for terms is the function that maps each term to its representative. The function $(\cdot)^{\mathcal{S}}$ is extended to set-based predicates preserving their behavior.

The *primitive constraints* in CLP($\mathcal{SET}$) are all the positive literals built from symbols in $\Pi_c \cup \mathcal{F} \cup \mathcal{V}$. A *($\mathcal{SET}$-) constraint* is a conjunction of primitive constraints and negation of primitive constraints of CLP($\mathcal{SET}$). The class of $\mathcal{SET}$-constraints is denoted by $\mathcal{C}_{\mathcal{SET}}$. Other basic set-theoretic operations (e.g., $\cap$, $\subseteq$) are easily defined as $\mathcal{SET}$-constraints. CLP($\mathcal{SET}$) provides also a syntactic extension called *Restricted Universal Quantifiers (RUQs)*, that allows one to write subgoals of the form

$$(\forall X_1 \in S_1) \cdots (\forall X_n \in S_n)(C \wedge B)$$

where C is a $\mathcal{SET}$-constraint and B is a CLP($\mathcal{SET}$) program goal. The semantics of $(\forall X \in S)G$ is simply $\forall X\, (X \in S \rightarrow G)$. The implementation of RUQs is accomplished via program transformation—generating two recursive CLP($\mathcal{SET}$) clauses using set terms and constraints [5].

The following example is a simple program to compute the cliques of a given undirected graph $\langle V, E\rangle$:

$$\begin{array}{l} \mathit{clique}(V, E, \mathit{Clique}) \;:- \\ \quad \mathit{Clique} \subseteq V \;\wedge \\ \quad (\forall I \in \mathit{Clique})(\forall J \in \mathit{Clique})(\{I, J\} \in \{\{I\} \,|\, E\}). \end{array}$$

The ability to use nested sets and RUQs greatly facilitate the encoding of this problem.

In this paper we extend Σ by adding the constant symbols $0, 1, -1, 2, -2, \ldots$ to $\mathcal{F}$. Let us call these terms *integer constants* (or, simply, *integers*). Let us also assume that the total order $\prec$ defined on $T(\mathcal{F})$ be such that $\cdots - 2 \prec -1 \prec 0 \prec 1 \prec 2 \prec \ldots$. We say that $s \preceq t$ iff $s \prec t$ or $s = t$. When clear from the context, we denote with $x + y$ $(x - y)$ the integer constant corresponding to the sum (difference) of the values of the two integer constants x and y. We will also use $<$ and $\leq$ with the usual meaning. These terms are seen as different, uninterpreted, constant terms. Since they are not interpreted, the constraint structure $\mathcal{A}_{\mathcal{SET}}$ for CLP($\mathcal{SET}$) does not need to be changed.

We also introduce in $\mathcal{F}$ the binary function symbol int. A term $\mathsf{int}(t_i, t_f)$, where t_i and t_f are integer constants, is called an *interval term*. $\mathsf{int}(t_i, t_f)$ is a shorthand for the set term $\{t_i, t_i + 1, \ldots, t_f - 1, t_f\}$. If $t_f \prec t_i$, then $\mathsf{int}(t_i, t_f)$ is a shorthand for the set term $\emptyset$. We only allow variables or integer constants as arguments of the function symbol int. To start, however, we require the terms t_i, t_f in an interval term $\mathsf{int}(t_i, t_f)$ to be non-variable. This restriction will be relaxed in the successive sections. Terms of the form $\mathsf{int}(t_1, t_2)$ where either t_1 or t_2 is not an integer constant are considered ill-formed terms. Occurrences of ill-formed terms in a constraint cause the CLP($\mathcal{SET}$) solver to detect unsatisfiability and to return a false result.

We will refer to the language CLP($\mathcal{SET}$) with these extensions as *CLP($\mathcal{SET}$) with intervals* or, simply, CLP$(\mathcal{SET})^{\mathsf{int}}$.

3. CLP$(\mathcal{SET})^{\mathsf{int}}$: CONSTRAINT SOLVING USING THE MODIFIED $SAT_{\mathcal{SET}}$

The language CLP$(\mathcal{SET})^{\mathsf{int}}$ is endowed with the sound and complete constraint solver $SAT_{\mathcal{SET}}$ for verifying the satisfiability of $\mathcal{SET}$-constraints. Given a constraint C, $SAT_{\mathcal{SET}}(C)$ non-deterministically transforms C either to false (if C is unsatisfiable) or to a finite collection $\{C_1, \ldots, C_k\}$ of constraints in *solved form* [6]. A constraint in solved form is guaranteed to be satisfiable in the structure $\mathcal{A}_{\mathcal{SET}}$. Moreover, the disjunction of all the constraints in solved form generated by $SAT_{\mathcal{SET}}(C)$ is *equi-satisfiable* to C. For instance, let $C \equiv \{1, 2 \,|\, X\} = \{1 \,|\, Y\} \wedge 2 \notin X$. Then $SAT_{\mathcal{SET}}(C)$ returns the three solved form constraints:

- $C_1 \equiv Y = \{2 \,|\, X\} \wedge 2 \notin X \wedge \mathsf{set}(X)$
- $C_2 \equiv X = \{1 \,|\, N\} \wedge Y = \{2 \,|\, N\} \wedge \mathsf{set}(N) \wedge 2 \notin N$
- $C_3 \equiv Y = \{1, 2 \,|\, X\} \wedge 2 \notin X \wedge \mathsf{set}(X)$

where N is a new variable. Equi-satisfiability means that:

$$\mathcal{A}_{\mathcal{SET}} \models C \leftrightarrow \exists N\,(C_1 \vee C_2 \vee C_3).$$

A detailed description of the constraint solver $SAT_{\mathcal{SET}}$ can be found in [6]; its implementation is included in the $\{log\}$-interpreter [14].

3.1 Dealing with intervals

Interval terms can be treated simply as syntactic extensions and automatically removed before processing the constraints. The simplest solution is to replace each interval term $\mathsf{int}(t_i, t_f)$ with the corresponding extensional set term $\{t_i, t_i + 1, \ldots, t_f\}$. Obviously, this is a rather inefficient solution, as it may lead to an explosion in the size of the constraint.

A more effective way to handle constraints involving interval terms is to modify the rewriting procedures of the CLP($\mathcal{SET}$) constraint solver. The modifications should guarantee that atomic constraints involving intervals (e.g., $t \in \mathsf{int}(t_i, t_f)$) have the expected semantics, but without having to expand interval terms into the corresponding set terms. The modified rewriting rules for $\in$ constraints are shown in Figure 1. Similar modifications can be applied to the procedures used to handle the other constraint predicates (e.g., the new cases for the procedure handling $\cup_3$ are summarized in Figure 2). We omit the details of the other modified procedures.

(1)	$s \in \emptyset \wedge C'$	$\mapsto$	false
(2)	$r \in \{s \,|\, t\} \wedge C'$	$\mapsto$	$C' \wedge r = s$ *or* $C' \wedge r \in t$
(3)	$t \in X \wedge C'$	$\mapsto$	$X = \{t \,|\, N\} \wedge \mathsf{set}(N) \wedge C'$
(4.1)	$r \in \mathsf{int}(t_i, t_f) \wedge C'$ r *is neither a variable* *nor an integer*	$\mapsto$	false
(4.2)	$t_n \in \mathsf{int}(t_i, t_f) \wedge C'$ $t_f \prec t_i$ t_i, t_f *integers*	$\mapsto$	false
(4.3)	$t_n \in \mathsf{int}(t_i, t_f) \wedge C'$ $t_i \preceq t_f$ t_i, t_f, t_n *integers* $t_i \preceq t_n \preceq t_f$	$\mapsto$	C'
(4.4)	$X \in \mathsf{int}(t_i, t_f) \wedge C'$ $t_i \preceq t_f$ t_i, t_f *integers* *and* X *is a variable*	$\mapsto$	$C' \wedge X = t_i$ *or* $C' \wedge X \in \mathsf{int}(t_i + 1, t_f)$
(4.5)	$t_n \in \mathsf{int}(t_i, t_f) \wedge C'$ $t_i \preceq t_f$ t_i, t_f, t_n *integers* $t_n \prec t_i$ *or* $t_f \prec t_n$	$\mapsto$	false

Figure 1: $\in$-constraint rewriting in $SAT_{\mathcal{SET}}$

Although this solution avoids exploding interval terms to set terms and ensures good efficiency in processing each individual membership constraint over intervals, it is still unsatisfactory whenever the constraint involves more than one interval term. For example, consider the following $\mathcal{SET}$-constraint: $X \in \mathsf{int}(1, 1000) \wedge X \in \mathsf{int}(1001, 2000)$. According to the CLP($\mathcal{SET}$) constraint solving technique and the above definition of the membership rewriting procedure, solving this constraint will cause the constraint solver to repeatedly (through backtracking) generate at least 1000 integer constants to solve the first or the second constraint, before concluding that the constraint is unsatisfiable. This problem derives from the fact that the constraint solver does not fully exploit the semantic properties of interval constraints. This problem can be relieved by semantically coupling the CLP$(\mathcal{SET})^{\mathsf{int}}$ solver with a standard solver for finite domain constraints, as illustrated in the successive sections.

$\cup_3(s_1, s_2, \mathsf{int}(t_1, t_2)) \wedge C' \}$	$\mapsto$	$C' \wedge s_1 = \{t_1 \mid N_1\} \wedge t_1 \notin N_1 \wedge \cup_3(N_1, s_2, \mathsf{int}(t_1+1, t_2))$ *or* $C' \wedge s_2 = \{t_1 \mid N_2\} \wedge t_1 \notin N_2 \wedge \cup_3(s_1, N_2, \mathsf{int}(t_1+1, t_2))$ *or* $C' \wedge s_1 = \{t_1 \mid N_1\} \wedge s_2 = \{t_2 \mid N_2\} \wedge t_1 \notin N_1 \wedge t_2 \notin N_2 \wedge \cup_3(N_1, N_2, \mathsf{int}(t_1+1, t_2))$
$\cup_3(\mathsf{int}(t_1, t_2), s, X) \wedge C' \}$	$\mapsto$	$X = \{t_1 \mid N\} \wedge t_1 \notin N \wedge t_1 \notin s \wedge \cup_3(\mathsf{int}(t_1+1, t_2), s, N)$ *or* $X = \{t_1 \mid N\} \wedge t_1 \notin N \wedge s = \{t_1 \mid N_1\} \wedge t_1 \notin N_1 \wedge \cup_3(\mathsf{int}(t_1+1, t_2), N_1, N)$

Figure 2: Additional cases to handle $\cup_3$ constraints with int

4. CLP($\mathcal{FD}$): LANGUAGE AND CONSTRAINT SOLVING

The language CLP($\mathcal{FD}$) [17, 4, 12] is an instance of the CLP scheme, and it is particularly suited to encode combinatorial problems. The constraint domain used by CLP($\mathcal{FD}$) contains an alphabet of constraint predicate symbols $\Pi_c = \{\{\in m..n\}, =, \neq, <, \leq, >, \geq\}$ [10] and an interpretation structure $\mathcal{A}_{\mathcal{FD}}$, whose interpretation domain is the set $\mathbb{Z}$ of the integer numbers and the interpretation function $(\cdot)^{\mathcal{FD}}$ maps symbols of the signature to elements of $\mathbb{Z}$ and to functions and relations over $\mathbb{Z}$ in the natural way. In particular, the interpretation of the constraint $u \in m..n$ is: $u \in^{\mathcal{FD}} m..n$ holds if and only if $m \leq u \leq n$.[1] The constraint predicates $\{\in m..n\}$ are used to identify intervals representing the *domains* of the variables in the problem. Additional constraint predicates are often provided in CLP($\mathcal{FD}$) languages, e.g., cumulative constraints and labeling, but for the purpose of this paper we will only focus on the primitive constraints mentioned above. A *$\mathcal{FD}$-constraint* is a conjunction of literals built from symbols in the given signature. The class of $\mathcal{FD}$-constraints is denoted by $\mathcal{C}_{\mathcal{FD}}$.

Let $SAT_{\mathcal{FD}}$ be a constraint solver for the $\mathcal{FD}$-constraint domain. Here we consider $SAT_{\mathcal{FD}}$ as a black-box, i.e., a procedure that takes as input a $\mathcal{FD}$-constraint and returns a set of $\mathcal{FD}$-constraints. The resulting constraints cover the same solution space as the input constraint, but provides simplified constraints—in particular, intervals representing domains may have been narrowed. The CLP($\mathcal{FD}$) solvers adopted in SICStus [16] and ECLiPSe [9] are incomplete solving procedures; e.g., given the goal

```
| ?- X in 1..2, Y in 1..2, Z in 1..2,
     X #\= Y, X #\= Z, Y #\= Z.
```

the CLP($\mathcal{FD}$) solvers provided by SICStus and ECLiPSe are not able to find out the inconsistency (actually they leave the given constraint unchanged). A complete solver can be obtained by adding chronological backtracking to force exploration of the solution space. E.g., if we conjoin the goal above with the atom `labeling([X,Y,Z])` (`labeling([], [X,Y,Z])` in SICStus) then the constraint solver will successfully determine the unsatisfiability of the constraint.

Thus, the constraint solver of CLP($\mathcal{FD}$) with input a constraint C either returns a constraint C' implying C, or, via labeling, a solved form solution. However, C' is possibly not in solved form and it is possibly unsatisfiable. This can be compared with the case of CLP($\mathcal{SET}$), where the result obtained is either a disjunction of solved form satisfiable constraints C'_i or false (if C is unsatisfiable—see Section 3).

5. CLP($\mathcal{SET}$)$^{\mathsf{int}}$: COMBINING SOLVERS

[1]The constraint $u \in m..n$ is concretely expressed as `u in m..n` in SICStus and as `u::m..n` in ECLiPSe.

5.1 Integration of $SAT_{\mathcal{SET}}$ and $SAT_{\mathcal{FD}}$

The solution we propose is to exploit the CLP($\mathcal{FD}$) constraint solver within the CLP($\mathcal{SET}$)$^{\mathsf{int}}$ constraint solver—the resulting combined solver is called $SAT_{\mathcal{SET}+\mathcal{FD}}$. This allows us to solve membership constraints over intervals by mapping the $\in$ constraints of CLP($\mathcal{SET}$)$^{\mathsf{int}}$ to $\{\in m..n\}$ constraints of CLP($\mathcal{FD}$). We also use the CLP($\mathcal{FD}$) solver to deal with equations and disequations over integers, which may occur in a $\mathcal{SET}$-constraint. Extensions that allow also the use of $<$ and $\leq$ are discussed in the successive section. In the specific implementation described here we make use of the constraint solvers offered by SICStus Prolog [16], though the proposed technique is general and applicable to other CLP($\mathcal{FD}$) solvers. Let us assume that the two languages share both the set of variables and the representation for integer constants.

The function τ that translates atomic $\mathcal{SET}$-constraints to $\mathcal{FD}$-constraints is defined as follows:

$$\tau(c) = \begin{cases} s \in t_i..t_f & \textit{if } c \equiv s \in \mathsf{int}(t_i, t_f) \\ & (\textit{or } s \in \{t_i, t_i+1, \ldots, t_f\}) \\ s = t & \textit{if } c \equiv s = t \\ s \neq t & \textit{if } c \equiv s \neq t \\ \mathsf{true} & \textit{otherwise} \end{cases}$$

where s, t_i, t_f, t are either variables or integer constants. In concrete syntax, the $\mathcal{SET}$-constraint $s \in \mathsf{int}(m, n)$ will correspond to the SICStus CLP($\mathcal{FD}$) constraint `s in m..n`, whereas $s \neq t$ and $s = t$ will correspond to `s #\= t` and `s #= t`, respectively. The function τ can be naturally extended to conjunctions of primitive constraints. We will also refer to the inverse function τ^{-1}, where $\tau^{-1}(s \in t_i..t_f) = (s \in \mathsf{int}(t_i, t_f))$.

It is also possible to extend the function τ to deal with some other kind of constraints; for example, a CLP($\mathcal{SET}$)-constraint such as $s \in \{t_1, \ldots, t_n\}$ where $t_1, \ldots, t_n$ are integer constants, such that:

$$\{t_1, \ldots, t_n\} = i_1 \cup \cdots \cup i_k$$

and $i_1, \ldots, i_k$ are intervals, can be translated as the $\mathcal{FD}$-constraint $s \in i_1 \cup \cdots \cup i_k$—or, using SICStus Prolog's syntax, $\mathtt{s\ in\ i_1 \backslash/ \cdots \backslash/ i_k}$.

Proposition 5.1. *Let C be a $\mathcal{SET}$-constraint consisting of a conjunction of literals of the forms*

$$\ell \in \mathsf{int}(t_i, t_f), \ell = r, \ell \neq r$$

where t_i, t_f are integer constants and ℓ and r are integer constants or variables. Then

$$\mathcal{FD} \models \bar{\exists}\tau(C) \textit{ if and only if } \mathcal{SET} \models \bar{\exists}C.$$

Proof. (Sketch) Since $\mathcal{SET}$ contains a submodel isomorphic to $\mathcal{FD}$, the ($\rightarrow$) direction follows trivially. For the ($\leftarrow$) direction, assume that σ is such that $\mathcal{SET} \models \sigma(C)$. We show that there is σ' such that $\mathcal{FD} \models \sigma'(\tau(C))$. Let

M be the maximum integer involved in $\sigma(C)$. If $\sigma(X)$ is an integer constant, assign $\sigma'(X) = \sigma(X)$. Otherwise, we are sure that X does not occur in a constraint of the form $X \in \mathsf{int}(t_i, t : f)$. Assign $\sigma'(X) = M+1$ (and $\sigma'(Y) = M+1$, for all other variables Y such that $\sigma(X) = \sigma(Y)$). Update $M := M+1$ and repeat the above step for all variables. $\square$

From a *practical* point of view, we know that, given a constraint C, $SAT_{\mathcal{SET}}(C)$ always detects its satisfiability or unsatisfiability. Conversely, the solvers for CLP($\mathcal{FD}$) are typically incomplete. We can obtain a complete solver by using labeling primitives. Specifically, if C is a $\mathcal{SET}$-constraint as in Proposition 5.1 and $\mathcal{V}_\in$ is the set of variables occurring in the $\in$-constraints, if $\mathcal{V}_\in = vars(C)$ then the CLP($\mathcal{FD}$) solver of SICStus Prolog returns a ground solution (or detect unsatisfiability) for the constraint $\tau(C) \wedge \texttt{labeling}([vars(C)])$.

Using these considerations, it is possible to identify three different kinds of constraints that are dealt with by $SAT_{\mathcal{SET}+\mathcal{FD}}$:

- K^S is a $\mathcal{SET}$-constraint that can be handled only by a $\mathcal{SET}$ solver,
- K^F is a $\mathcal{FD}$-constraint that can be handled only by a $\mathcal{FD}$ solver, and
- K^C is a $\mathcal{SET}$-constraint that can be handled by both solvers—i.e., K^C is composed of those constraints for which τ produces a result different from true.

The K^C constraint effectively represents a communication conduit between the two constraint solvers. From now on, we will assume that constraints dealt with by $SAT_{\mathcal{SET}+\mathcal{FD}}$ are organized as pairs $\langle C, D\rangle$, where $C = K^S \wedge K^C$ and $D = K^F \wedge \tau(K^C)$. The procedure $SAT_{\mathcal{SET}}$ is modified to handle pairs of constraints of this form, and to properly distribute the different components to the proper solvers.

(1)	$\langle s \in \emptyset \wedge C', D\rangle$	$\mapsto$	false
(2)	$\langle r \in \{s \mid t\} \wedge C', D\rangle$	$\mapsto$	$\langle C' \wedge r = s, D \wedge \tau(r = s)\rangle$ *or* $\langle C' \wedge r \in t, D \wedge \tau(r \in t)\rangle$
(3)	$\langle t \in X \wedge C', D\rangle$	$\mapsto$	$\langle X = \{t \mid N\} \wedge \mathsf{set}(N) \wedge C', D\rangle$
(4.1)	$\langle r \in \mathsf{int}(t_i, t_f) \wedge C', D\rangle$ r *is neither a variable nor an integer* t_i, t_f *integers*	$\mapsto$	false
(4.2)	$\langle r \in \mathsf{int}(t_i, t_f) \wedge C', D\rangle$ t_i, t_f *integers* $SAT_{\mathcal{FD}}(D \wedge \tau(r \in \mathsf{int}(t_i, t_f))) = \mathsf{false}$	$\mapsto$	false
(4.3)	$\langle r \in \mathsf{int}(t_i, t_f) \wedge C', D\rangle$ t_i, t_f *integers* $SAT_{\mathcal{FD}}(D \wedge \tau(r \in \mathsf{int}(t_i, t_f))) = D'$	$\mapsto$	$\langle C' \wedge \tau^{-1}(D'), D'\rangle$

Figure 3: $\in$-constraint rewriting in $SAT_{\mathcal{SET}+\mathcal{FD}}$

The modified rewriting rules for membership constraints are illustrated in Figure 3. Whenever a domain constraint $c \equiv X \in \mathsf{int}(t_i, t_f)$ is encountered in C, the solver passes the corresponding $\mathcal{FD}$-constraint $\tau(c)$ to the $SAT_{\mathcal{FD}}$ procedure, which in turn will check the satisfiability of the constraint $\tau(c) \wedge D$. If $\tau(c) \wedge D$ turns out to be unsatisfiable, then also $SAT_{\mathcal{SET}}$ will terminate with a false result, otherwise, $SAT_{\mathcal{SET}}$ will continue. Note also that $SAT_{\mathcal{FD}}$ may return an improved constraint D' (e.g., by narrowing intervals through node and arc consistency); in this case $SAT_{\mathcal{SET}}$ will be reactivated on $\langle C \wedge \tau^{-1}(D'), D'\rangle$. Observe also that the component $\tau^{-1}(D')$ that is returned to the $\mathcal{SET}$ solver may contain different types of constraints—i.e., simplified domain constraints, equality and inequality constraints.

Similar modifications can be applied to the procedures that handle the other primitive constraints. Figure 4 shows the new cases to be added to the $\mathcal{SET}$ solver for =-constraints. The modified parts of the rewriting rules for $\neq$-constraints are shown in Figure 5. The restriction to domain variables that appears in conditions of the rules in Figure 5 is motivated by the observation that a constraint of the form $Y \neq 1 \wedge Y = f(a)$ is rewritten to false in CLP($\mathcal{FD}$) (e.g., in SICStus) whereas it is clearly satisfiable in $\mathcal{SET}$. The restriction to domain variables forces to consider only those variables that are already constrained to be integers. Thus only disequations for which we can apply the result of Proposition 5.1 are passed to $SAT_{\mathcal{FD}}$. Others constraints (e.g., $Y \neq 1$ in the above example) are instead completely handled by $SAT_{\mathcal{SET}}$.

(8′)	$\langle \mathsf{int}(s_1, s_2) = \mathsf{int}(t_1, t_2) \wedge C', D\rangle$	$\mapsto$	$\langle s_1 = t_1 \wedge s_2 = t_2 \wedge C',$ $D \wedge \tau(t_1 = s_1) \wedge \tau(t_2 = s_2)\rangle$
(9′)	$\langle \{s \mid s'\} = \mathsf{int}(t, t') \wedge C', D\rangle$ s, t, t' *integers or variables* $SAT_{\mathcal{FD}}(D \wedge t \leq s \wedge s \leq t') = D'$	$\mapsto$	$\langle C' \wedge \cup_3(s_1, s_2, s') \wedge s \notin s_2 \wedge s1 = \mathsf{int}(t, s) \wedge$ $s_2 = \mathsf{int}(s+1, t') \wedge \tau^{-1}(D'), D'\rangle$

Figure 4: =-constraint rewriting in $SAT_{\mathcal{SET}+\mathcal{FD}}$

(9)	$\langle s \neq t \wedge C', D\rangle$ s, t *are either integers or domain variables* $SAT_{\mathcal{FD}}(D \wedge \tau(s \neq t)) = D'$	$\mapsto$	$\langle C' \wedge \tau^{-1}(D'), D'\rangle$
(9′)	$\langle s \neq t \wedge C', D\rangle$ s, t *are either integers or domain variables* $SAT_{\mathcal{FD}}(D \wedge \tau(s \neq t)) = \mathsf{false}$	$\mapsto$	false

Figure 5: $\neq$-constraint rewriting in $SAT_{\mathcal{SET}+\mathcal{FD}}$

Observe that $\mathcal{FD}$-constraints could be simply accumulated in the CLP($\mathcal{FD}$) constraint store, without solving them during the $SAT_{\mathcal{SET}}$ computation. The CLP($\mathcal{FD}$) constraint store could be processed by $SAT_{\mathcal{FD}}$ at the end, after the termination of $SAT_{\mathcal{SET}}$. However, interleaving $SAT_{\mathcal{FD}}$ and $SAT_{\mathcal{SET}}$ activities, as done in the rewriting rules discussed above, provides significant pruning of the search space. Let us also observe that each new constraint that is generated during $SAT_{\mathcal{SET}}$ computation should be translated with τ and made available to $SAT_{\mathcal{FD}}$ in the second part of the constraint. For example, each time a constraint of the form $X = i$ (with i integer constant) is processed, the equation

$X = i$ has to be provided to CLP($\mathcal{FD}$) as well.

5.2 The Global Constraint Solver

The global constraint solver $SAT_{\mathcal{SET}+\mathcal{FD}}$ (Figure 6) is simply the same CLP($\mathcal{SET}$) constraint solver $SAT_{\mathcal{SET}}$, in which CLP($\mathcal{SET}$) rewriting rules are replaced by the new rules defined in the previous section. These rules allow us to exploit the $SAT_{\mathcal{FD}}$ constraint solver whenever it is possible (and convenient). $SAT_{\mathcal{SET}+\mathcal{FD}}$ uses the procedure set_check to check the set constraints and the procedure set_infer to add set constraints for those variables that are required to belong to the sort Set; set_check and set_infer are the same procedures already used in CLP($\mathcal{SET}$), suitably modified in order to account for interval terms.[2] The STEP procedure is the core part of $SAT_{\mathcal{SET}+\mathcal{FD}}$: it calls the rewriting procedures on the pair $\langle C, D\rangle$. In order to ensure termination, some application order for the procedures present in $SAT_{\mathcal{SET}}$ must be respected. Technical details are explained in [6]. At the end of the execution of the STEP procedure, non-solved primitive constraints may still occur in $\langle C, D\rangle$; therefore, the execution of STEP has to be iterated until a fixed-point is reached—i.e., the constraint cannot be simplified any further.

```
SAT_{SET+FD}(⟨C, D⟩) :
    set_infer(C);
    repeat
        ⟨C', D'⟩ := ⟨C, D⟩;
        set_check(C);
        STEP(⟨C, D⟩);
    until ⟨C, D⟩ = ⟨C', D'⟩
```

Figure 6: Global Constraint Solver

As mentioned in Section 2, if C is satisfiable, then $SAT_{\mathcal{SET}}$ non-deterministically returns satisfiable constraints in solved form. Unfortunately, this is not the case for the $SAT_{\mathcal{SET}+\mathcal{FD}}$ solver. Let $\langle C', D'\rangle$ be one of the pairs returned by the procedure $SAT_{\mathcal{SET}+\mathcal{FD}}$. $C' \wedge \tau^{-1}(D')$ is no longer guaranteed to be satisfiable in $\mathcal{SET}$. For example, $SAT_{\mathcal{SET}+\mathcal{FD}}$ on the constraint:

$$\left\langle \begin{array}{ll} \mathtt{X} \notin \mathtt{S}, & \mathtt{X} \neq \mathtt{Y} \wedge \mathtt{X} \neq \mathtt{Z} \wedge \mathtt{Y} \neq \mathtt{Z} \wedge \mathtt{X} \in 1..2 \wedge \\ & \mathtt{Y} \in 1..2 \wedge \mathtt{Z} \in 1..2 \end{array} \right\rangle$$

returns the same, unsatisfiable, constraint. On the other hand, observe that $SAT_{\mathcal{SET}+\mathcal{FD}}$ still preserves the set of solutions of the original constraint, thanks to Proposition 5.1.

A complete solver can be obtained by adding chronological backtracking to force exploration of the solution space, e.g., by using labeling predicates. However, requiring direct execution of a complete labeling within $SAT_{\mathcal{SET}+\mathcal{FD}}$ each time $SAT_{\mathcal{FD}}$ is called would cause, in practice, the loss of most of the advantages of using the $SAT_{\mathcal{FD}}$ solver—it will effectively lead to a translation of each interval into an extensional set. On the other hand, in our approach we try to achieve a behavior of the global solver as close as possible to that of the traditional $SAT_{\mathcal{SET}}$ solver; thus we do not want to require the user to explicitly request an enumeration of the solutions.

[2] In fact, well-formed interval terms $\mathsf{int}(t_i..t_f)$ are considered of sort Set.

The approach we propose consists in modifying the global solver as follows. Given a CLP($\mathcal{SET}$)$^{\mathsf{int}}$ constraint C, the resolution of C is performed according to the following steps:

step 1. The $SAT_{\mathcal{SET}+\mathcal{FD}}$ solver is initially applied to the pair $\langle C, \mathsf{true}\rangle$; this will lead either to false (in this case we can stop) or to the pair $\langle C', D'\rangle$.

step 2. Let D' be the $\mathcal{FD}$ part of the pair of constraints obtained from the last call to $SAT_{\mathcal{SET}+\mathcal{FD}}$. A constraint of the form $X \in t_1..t_2$ is selected from D' and the solutions $R_1, \ldots, R_k$ are collected from the execution of $SAT_{\mathcal{FD}}(D' \wedge \mathsf{labeling}([X]))$. Each R_i, in particular, will contain a constraint of the type $X = r_i$ for some $r_i \in t_1..t_2$. The execution of $SAT_{\mathcal{SET}+\mathcal{FD}}$ is restarted from each constraint $\langle C' \wedge \tau^{-1}(R_i), R_i\rangle$. The step is repeated until there are no constraints of the form $X \in t_1..t_2$ in D'.

Additional improvements to the resolution process can be achieved through a refinement of the sorts system used by CLP($\mathcal{SET}$)$^{\mathsf{int}}$. Currently the system recognizes two sorts, `Set` and `Ker`, describing respectively set terms and non-set terms. The sort `Ker` requires now a further refinement, by distinguishing integer constants (`Int`) from all other non-set terms (`Ker`). Enforcing the sorts requires the introduction of additional constraints: $\mathsf{integer}(s)$ and $\mathsf{ker}(s)$ are satisfiable only by terms belonging to the sort `Int` and `Ker`, respectively. During the rewriting process, each time a constraint of the type $s \in \mathsf{int}(t_i, t_f)$ is generated, the following constraint is introduced: $\mathsf{integer}(s) \wedge \mathsf{integer}(t_i) \wedge \mathsf{integer}(t_f)$. Observe that if one of the endpoints of the interval (i.e., t_i or t_f) is a variable, then the new constraint will guarantee that the variable can be assigned only integer values.

For the sake of simplicity, we omit the details of how the constraints based on integer and ker are handled. Intuitively, the integer succeeds on integer constants, fails on any non-variable non-integer term; the constraint will also need to interact with the set constraint (e.g., $\mathsf{integer}(X)$ fails if there is a $\mathsf{set}(X)$ in the constraint). Similarly for the ker constraint. Constraints $\mathsf{integer}(X)$ are also generated via static program analysis, as shown in Section 7.

6. ENCODING NEW OPERATORS

CLP($\mathcal{FD}$) allows us to properly and efficiently deal with constraints of the form $\ell \in t_i..t_f, \ell = r, \ell \neq r, \ell < r, \ell \leq r$. Moreover, arithmetic operations can be used in terms ℓ and r. The capability of dealing with arithmetic operations and with the natural ordering between integers can be used to improve the capabilities of CLP($\mathcal{SET}$)$^{\mathsf{int}}$ as well.

One simple possibility is to use arithmetics and ordering in the solver $SAT_{\mathcal{SET}+\mathcal{FD}}$, without extending the language. This means that constraints of the type $s < t$ or $s \leq t$ may be internally generated during the constraint resolution process and directly processed by $SAT_{\mathcal{FD}}$. This has already been done, for instance, in the definition of the rewriting rules for equality constraints (see Fig. 4). The same technique can be used to extend the treatment of membership constraints, in order to allow the solver to deal with int terms with unbound endpoints—since $r \in \mathsf{int}(s, t)$ can be rewritten as $s \leq r \wedge r \leq t$, where either s or t can be unbound variables. The additional rewriting rule can be defined as follows:

$$
(4.4)\quad \left.\begin{array}{r}\langle r \in \mathsf{int}(s,t) \wedge C', D\rangle\\ s,t \textit{ variables}\\ SAT_{\mathcal{FD}}(D \wedge s \le r \wedge r \le t) = D'\end{array}\right\} \mapsto \quad \langle C' \wedge \tau^{-1}(D'), D'\rangle
$$

Similar changes can be performed to other constraint handling procedures. For example, constraints of the type $r \notin \mathsf{int}(s,t)$ can be rewritten as

$$
\left.\begin{array}{r}\langle r \notin \mathsf{int}(s,t) \wedge C, D\rangle\\ SAT_{\mathcal{FD}}(D \wedge r < s) = D_1\\ SAT_{\mathcal{FD}}(D \wedge t > r) = D_2\end{array}\right\} \mapsto \quad \begin{array}{r}\langle C \wedge \tau^{-1}(D_1) \wedge \mathsf{integer}(r), D_1\rangle \;\; or\\ \langle C \wedge \tau^{-1}(D_2) \wedge \mathsf{integer}(r), D_2\rangle \;\; or\\ \langle C \wedge \mathsf{not\_integer}(r), D\rangle\end{array}
$$

where $\mathsf{not\_integer}(r)$ is defined as

$$
\mathsf{not\_integer}(r) \Leftrightarrow \mathsf{set}(r) \vee \mathsf{ker}(r).
$$

It is also possible to extend the discussion to allow $<$ and $\le$ constraints to be available at the user level in CLP$(\mathcal{SET})^{\mathsf{int}}$. The presence of constraints based on $<$ and $\le$ significantly improves the capabilities of the language and of the constraint solver.

A direct treatment of these new constraints in CLP$(\mathcal{SET})^{\mathsf{int}}$ can be realized assuming that (non-negative) integer numbers are represented by sets. For instance, if integers are encoded *á la* Von Neumann:

$$
\left\{\begin{array}{rcl}\underline{0} & = & \emptyset,\\ \underline{i+1} & = & \{\underline{i} \,|\, \underline{i}\}\end{array}\right.
$$

then the relation $t < s$ $(t \le s)$ can be simply expressed as $t \in s$ $(t \in \{s \,|\, s\})$. In this case the introduction of the predicate symbols $<$ and $\le$ in CLP$(\mathcal{SET})^{\mathsf{int}}$ could be seen as a syntactic sugar. While this approach is feasible (at least in principle) for completely specified arithmetic expressions, it turns out to be not adequate to deal with more general algebraic expressions such as $X = Y + 3 \wedge X - Z \le Y + 1$, with X, Y, Z uninstantiated variables.

A more effective solution can be achieved by avoiding any direct treatment of $<$ and $\le$ within $SAT_{\mathcal{SET}}$, and simply relying on $SAT_{\mathcal{FD}}$ to handle this type of constraints. Let us assume that the language of CLP$(\mathcal{SET})^{\mathsf{int}}$ is extended as follows: $\Pi_c = \{=, \in, \cup_3, \|, \mathsf{set}, \mathsf{integer}, \le\}$ and $\mathcal{F}$ includes the usual arithmetic function symbols (e.g., $+$, $-$). Let us call *integer terms* those terms built using arithmetic operators, integer constants and variables. Integer terms are interpreted on the subset of the domain of $\mathcal{SET}$ containing integer constants (i.e., on terms of sort $\mathtt{Int}$). Interpretation of arithmetic function symbols is the expected one. Their sort will be $\langle\{\mathtt{Int}\}, \{\mathtt{Int}\}, \{\mathtt{Int}\}\rangle$.

Integer terms can occur as arguments of primitive constraints based on $=$, $\neq$ and $\le$. Rewriting rules for these constraints are modified so that as soon as they recognize that one of the arguments of the predicate symbol at hand is an integer term, solution of the constraint is committed to $SAT_{\mathcal{FD}}$. The translation function τ has to be modified to allow s, t_i, t_f, t to be either variables or integer terms (not only constants). Moreover τ needs to be extended to handle constraints based on $\le$ so that $\tau(s \le t) = s \le t$, where s, t are either integer terms or variables (in concrete syntax of SICStus Prolog $\tau(s \le t)$ is `s #<= t`). The rewriting rules for $\le$ constraints used in $SAT_{\mathcal{SET}+\mathcal{FD}}$ are defined as follows:

(1)	$\left.\begin{array}{r}\langle s \le t \wedge C, D\rangle\\ s \textit{ or } t \textit{ not integers or}\\ SAT_{\mathcal{FD}}(D \wedge \tau(s \le t)) = \mathsf{false}\end{array}\right\} \mapsto \mathsf{false}$
(2)	$\left.\begin{array}{r}\langle s \le t \wedge C, D\rangle\\ s,t \textit{ integers or variables}\\ SAT_{\mathcal{FD}}(D \wedge \tau(s \le t)) = D'\end{array}\right\} \mapsto$ $\langle C \wedge \tau^{-1}(D'), D'\rangle$

The definition of the rewriting rules for the $=$ and $\neq$ constraints are updated accordingly. In particular, rules (9) and (9$'$) for $\neq$-constraints shown in Figure 5 need only to generalize the condition on s and t from that of being *integer constants* to that of being *integer terms*.

Note that we assume that terms built using arithmetic operators but involving also non-integer terms are considered ill-formed terms (i.e., not respecting the sorts). For instance $X + f(a)$ is an ill-formed term. In CLP$(\mathcal{SET})^{\mathsf{int}}$ the presence of ill-formed syntactic objects is detected within the constraint solver procedure by exploiting sort constraints, set, $\mathsf{integer}$, and ker, automatically generated by the solver itself or provided by the user as part of the input constraint. Whenever the solver detects an inconsistency among sort constraints it terminates and returns the result false.

The presence of $\le$ allows us also to introduce new expressive constraints in $\mathcal{SET}$. For example, it becomes possible to provide a general cardinality constraint size in the context of CLP$(\mathcal{SET})$: the constraint $\mathsf{size}(s,t)$ is satisfied if s is a set, t is an integer, and t corresponds to the cardinality of s. The constraint can be handled using the rewriting rules in Figure 7—after a preprocessing step that, for each constraint $\mathsf{size}(s,t)$, introduces the additional constraints $\mathsf{set}(s)$ $\wedge$ $\mathsf{integer}(t)$. In addition, the other constraints dealing with sets also require the addition of new rules, e.g., whenever a constraint of the form $\cup_3(r,s,t)$ is introduced, then the constraints

$$
\mathsf{size}(r, N_r) \wedge \mathsf{size}(s, N_s) \wedge \mathsf{size}(t, N_t) \wedge N_r \le N_t \wedge N_s \le N_t
$$

have to be added as well.

(1)	$\left.\begin{array}{r}\langle \mathsf{size}(X,n) \wedge C, D\rangle\\ SAT_{\mathcal{FD}}(D \wedge n = m+1) = D'\\ D' \neq \mathsf{false}\end{array}\right\} \mapsto$ $\left\langle\begin{array}{c} C \wedge \tau^{-1}(D') \wedge X = \{A \mid B\} \wedge\\ A \notin B \wedge \mathsf{size}(B,m),\;\; D'\end{array}\right\rangle$
(2)	$\left.\begin{array}{r}\langle \mathsf{size}(\emptyset,X) \wedge C, D\rangle\\ SAT_{\mathcal{FD}}(D \wedge X = 0) = D'\end{array}\right\} \mapsto$ $\langle C \wedge \tau^{-1}(D'), D'\rangle$
(3)	$\left.\begin{array}{r}\langle \mathsf{size}(\mathsf{int}(s,t),N) \wedge C, D\rangle\\ SAT_{\mathcal{FD}}(D \wedge N = t - s + 1) = D'\end{array}\right\} \mapsto$ $\langle C \wedge \tau^{-1}(D'), D'\rangle$
(4)	$\left.\begin{array}{r}\langle \mathsf{size}(\{s \mid t\},N) \wedge C, D\rangle\\ SAT_{\mathcal{FD}}(D \wedge M = N - 1) = D'\end{array}\right\} \mapsto$ $\langle \mathsf{size}(t,N) \wedge s \in t \wedge C, D\rangle$ *or* $\langle \mathsf{size}(t,M) \wedge s \notin t \wedge C \wedge \tau^{-1}(D'), D'\rangle$

Figure 7: Rewriting procedures for size

As an example of how constraints on integer terms can be conveniently exploited in CLP($\mathcal{SET}$)$^{\mathsf{int}}$ consider the problem of finding solutions for a system of linear equations. The following code shows a CLP($\mathcal{SET}$)$^{\mathsf{int}}$ implementation for this problem, while the table shows experimental results obtained running the code using $SAT_{\mathcal{SET}}$ and using $SAT_{\mathcal{SET}+\mathcal{FD}}$.

$$
\begin{array}{l}
\texttt{start}(X, Y, Z, L, H) :- \\
\quad X \in \mathsf{int}(L, H) \wedge \\
\quad Y \in \mathsf{int}(L, H) \wedge \\
\quad Z \in \mathsf{int}(L, H) \wedge \\
\quad X1 = 1 + X \wedge X1 = 2 * Y + Z \wedge \\
\quad X2 = Z - Y \wedge X2 = 3 \wedge \\
\quad X3 = X + Y \wedge X3 = 5 + Z.
\end{array}
$$

	Experimental results						
L	-10	-20	-30	-10^3	-10^4	-10^5	-10^6
H	10	20	30	10^3	10^4	10^5	10^6
$SAT_{\mathcal{SET}}$	41s	8m	38m	∞	∞	∞	∞
$SAT_{\mathcal{SET}+\mathcal{FD}}$	0s	0s	0s	0.13s	1.6s	19.5s	4m

Solving the problem using $SAT_{\mathcal{SET}}$ is feasible but with unacceptable computation time. Conversely, the solution using the $SAT_{\mathcal{SET}+\mathcal{FD}}$ solver is obtained in time comparable to that obtained using directly CLP($\mathcal{FD}$).

As another example, consider the well-known combinatorial problem of solving the SEND + MORE = MONEY puzzle (i.e., solve the equation by assigning a distinct digit between 0 and 9 to each letter). The following code shows a CLP($\mathcal{SET}$)$^{\mathsf{int}}$ possible solution for this problem.

$$
\begin{array}{l}
\texttt{solve\_puzzle}(S, E, N, D, M, O, R, Y) :- \\
\quad \{S, E, N, D, M, O, R, Y\} \subseteq \mathsf{int}(0, 9) \ \wedge \\
\quad \texttt{size}(\{S, E, N, D, M, O, R, Y\}) = 8 \ \wedge \\
\quad S \neq 0 \ \wedge \ M \neq 0 \wedge \\
\quad M * 10000 + O * 1000 + N * 100 + E * 10 + Y \ = \\
\qquad\qquad S * 1000 + E * 100 + N * 10 + D \ + \\
\qquad\qquad M * 1000 + O * 100 + R * 10 + E.
\end{array}
$$

Running the program using $SAT_{\mathcal{SET}}$ results in unacceptable computational time, while using $SAT_{\mathcal{SET}+\mathcal{FD}}$ we get the solution in 0.01 seconds.

As final remark, observe that effective solutions have been proposed to handle *cardinality* constraints for CSP, e.g., [13], and in the context of CLP($\mathcal{FD}$) [18, 2]. This concept has a semantics related to the size of a set, but it is not the general cardinality notion. Precisely, given n variables $X_1, \dots, X_n$ with domains $D_1, \dots, D_n$, a cardinality constraint for a domain element d is a bound k on the number of X_i that can be simultaneously instantiated to d. It is therefore a constraint associated to the availability of a fixed number k of resources of a certain kind d.

7. IMPROVING COMMUNICATION BETWEEN SOLVERS

The previous sections have highlighted the main aspects of an effective methodology to integrate finite domain constraint solving with resolution of constraints over hereditarily finite sets. The procedures presented are capable of distributing primitive constraints between the different solvers. Nevertheless, it is possible to further improve communication between the solvers by *statically* extracting selected properties from the programs.

7.1 Static Analysis for Nested Sets Identification

In this section we introduce a static analysis schema for CLP($\mathcal{SET}$)$^{\mathsf{int}}$ programs, aimed at capturing the following property of a term s: s is a *n*ested *set* of *i*ntegers and $\mathsf{ni\_set}(s)$ is a measure of the depth of nesting of the sets. The intuitive semantics is the following:

- $\mathsf{ni\_set}(s) = 0$ if s in an integer constant;
- $\mathsf{ni\_set}(s) = n + 1$ if s is a set and for each $t \in s$ it holds that $\mathsf{ni\_set}(t) = n$.

Observe that $\mathsf{ni\_set}(s) > 0$ implies that the constraint $\mathsf{set}(s)$ holds. If $\mathsf{ni\_set}(s) = 1$ then s is a flat set of integers (e.g., $\{1, 2, 3, 5\}$). Moreover, we allow two additional possible values for $\mathsf{ni\_set}(s)$: $\bot$, when no information is currently available for s, and $\top$, when s is not a nested set of integers.

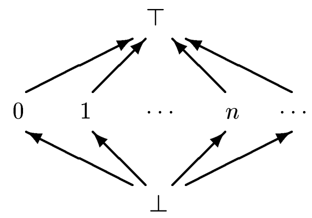

Figure 8: The lattice $\mathcal{L}$

The values for $\mathsf{ni\_set}(s)$ are elements of the lattice $\mathcal{L} = \langle\{\bot, \top, 0, 1, 2, \dots\}, \prec\rangle$ represented in Figure 8. The join operator $\sqcup$ is naturally defined on $\mathcal{L}$. We introduce two more operations on the lattice $\mathcal{L}$, respectively denoted by "+" and "−", defined in Table 1, where m, n, p denote integers, $+_{\mathbb{Z}}$ and $-_{\mathbb{Z}}$ represent the standard addition and subtraction between integers, and x is a generic element of $\mathcal{L}$.

Table 1: Semantics of + and - on $\mathcal{L}$

$\bot + \bot = \bot$	
$\bot + n = n + \bot = \bot$	
$\top + x = x + \top = \top$	
$m + n = m +_{\mathbb{Z}} n$	

$\top - x = \top$	
$m - n = m -_{\mathbb{Z}} n$	$m \geq n$
$m - n = \top$	$m < n$
$\bot - \bot = \bot - n = \bot$	
$\bot - \top = \top$	

The process of deriving the values of $\mathsf{ni\_set}(\cdot)$ starts by initializing $\mathsf{ni\_set}(X)$ to $\bot$ for each variable X occurring in the CLP($\mathcal{SET}$)$^{\mathsf{int}}$ goal G under consideration. The rules in Figure 9 are then applied on the terms and subterms of G and constraint literals until a fixpoint is reached. The case $s \,||\, t$ is missing since no information concerning the elements of s and t can be obtained by the fact that they are disjoint.

PROPOSITION 7.1. *The analysis presented is correct: for every term s occurring in a goal G, if $\mathsf{ni\_set}(s) = i$ then at each step of the computation s is a nested set of integers with i nestings of sets. The analysis terminates for every possible input, and its time complexity is $O(nm)$, where n is the number of variables in G and m is the sum of the occurrences of symbols of the terms in G.*

(1)	$\emptyset \mapsto$ $\mathsf{ni\_set}(\emptyset) := \bot$
(2)	$f(t_1,\ldots,t_k)$, $f \not\equiv \{\cdot \mid \cdot\}, f \not\equiv \mathsf{int}$, *not an integer constant* $\mapsto$ $\mathsf{ni\_set}(f(t_1,\ldots,t_k)) := \top$
(3)	*an integer constant* $t \mapsto$ $\mathsf{ni\_set}(t) := 0$
(4)	$\mathsf{int}(t_i,t_f)$, t_i, t_f *are integers and* $t_i \leq t_f \mapsto$ $\mathsf{ni\_set}(\mathsf{int}(t_i,t_f)) := 1$
(5)	$\{t \mid s\} \mapsto$ $\mathsf{ni\_set}(\{t \mid s\}) := (\mathsf{ni\_set}(t)+1) \sqcup \mathsf{ni\_set}(s)$; $\mathsf{ni\_set}(t) := \mathsf{ni\_set}(\{t \mid s\}) - 1$; $\mathsf{ni\_set}(s) := \mathsf{ni\_set}(\{t \mid s\})$
(6)	$s = t \mapsto$ $\mathsf{ni\_set}(s) := \mathsf{ni\_set}(s) \sqcup \mathsf{ni\_set}(t)$; $\mathsf{ni\_set}(t) := \mathsf{ni\_set}(s)$
(7)	$t \in s \mapsto$ $\mathsf{ni\_set}(t) := \mathsf{ni\_set}(t) \sqcup (\mathsf{ni\_set}(s) - 1))$; $\mathsf{ni\_set}(s) := \mathsf{ni\_set}(s) \sqcup (\mathsf{ni\_set}(t)+1)$
(8)	$\cup_3(r,s,t) \mapsto$ $\mathsf{ni\_set}(r) := \mathsf{ni\_set}(r) \sqcup \mathsf{ni\_set}(s) \sqcup \mathsf{ni\_set}(t)$; $\mathsf{ni\_set}(s) := \mathsf{ni\_set}(r)$; $\mathsf{ni\_set}(t) := \mathsf{ni\_set}(r)$

Figure 9: Abstract Interpretation steps to compute $\mathsf{ni\_set}$

PROOF. Correctness follows trivially by structural induction. The termination is guaranteed by the finite height of the lattice. At every goal scan ($O(m)$ time), at least one variable is updated and raised up by one level in the lattice, otherwise a fixpoint is reached. In the worst case, at each scan only one variable changes its value. Since $\mathcal{L}$ is a flat lattice with height equal to 2, after $2n$ scans all the variables will reach the $\top$ value and a fixpoint will be found. Thus the worst case time complexity is $O(nm)$. $\square$

This analysis can be further refined, by extracting additional information from the arithmetic operations. For example, from $A+B$ one can infer that $\mathsf{ni\_set}(A) = \mathsf{ni\_set}(A) \sqcup 0$ (the same for B). Similarly, the presence of interval definitions with variables as endpoints, as in $X \in \mathsf{ni\_set}(1,N)$ allows us to infer: $\mathsf{ni\_set}(N) = \mathsf{ni\_set}(N) \sqcup 0$.

As an example, let us consider the constraint:

$$\mathtt{M} \in \mathsf{int}(1,10) \wedge \mathtt{N} = \{\{\mathtt{M}\},\mathtt{M}\} \wedge \cup_3(\{\mathtt{M}\},\emptyset,\mathtt{S})$$

The analysis returns

$$\mathsf{ni\_set}(M) = 0, \mathsf{ni\_set}(S) = 1, \mathsf{ni\_set}(N) = \top.$$

7.2 Analysis of CLP$(\mathcal{SET})^{\mathsf{int}}$ Programs

The set of rules presented in the previous section can be used to analyze a complete CLP$(\mathcal{SET})^{\mathsf{int}}$ program. Let us assume w.l.o.g. that each program P is composed of a set of clauses of the form

$$p(X_1,\ldots,X_m) :\!-\ C, B$$

where $X_1,\ldots,X_m$ are distinct variables, C is a CLP$(\mathcal{SET})^{\mathsf{int}}$ constraint, and B is a conjunction of program atoms of the form $q(Y_1,\ldots,Y_k)$, where $Y_1,\ldots,Y_k$ are distinct variables. Moreover, always w.l.o.g., we can assume that each variable in the program P occurs only in one clause.

Given a program P, we want to force some equivalences between variables used in different clauses defining the same predicate. Given the n rules defining p

$$\begin{array}{rcl} p(X_1,\ldots,X_k) & :\!- & C_1, B_1. \\ & \ldots & \\ p(Y_1,\ldots,Y_k) & :\!- & C_n, B_n. \end{array}$$

the rules are renamed to

$$\begin{array}{rcl} p_1(N_1,\ldots,N_k) & :\!- & C_1[X_i/N_i], B_1[X_i/N_i]. \\ & \ldots & \\ p_1(N_1,\ldots,N_k) & :\!- & C_n[Y_i/N_i], B_n[Y_i/N_i]. \end{array}$$

where N_i are new, pairwise distinct, variables. We assume to deal with programs P obtained by this variable renaming.

The analysis gathers more information with the program analysis if we consider the program together with its input, or with a "typical" input. In other words, if we have some information on the *modes* of use of a predicate p (e.g., with one argument typically a ground integer, another typically a variable, etc.), we can further infer some properties for p. For doing that, we define a new rule obtained by a typical input.

Let us assume that the goal G contains the m variables $X_1,\ldots,X_m$ and that some information regarding some variables $X_1,\ldots,X_n$, $n \leq m$, is known in advance. For instance, we know that $X_1,\ldots,X_n$ are associated to integer numbers. Then, choosing some integer numbers $s_1,\ldots,s_n$, we can make available the information to the analysis by adding the new clause

$$g(X_1,\ldots,X_n,X_{n+1},\ldots,X_m) :\!-\ X_1 = s_1 \wedge \ldots \wedge X_n = s_n, G.$$

to the program P.

The analysis algorithm proceeds as follows. At the beginning, all the program variables are initialized with the value $\bot$ from the lattice $\mathcal{L}$. The program analysis performs the fixpoint described in the previous section to determine the $\mathsf{ni\_set}$ values, by repeatedly executing the following two steps for each clause $p(X_1,\ldots,X_k) :\!-\ C, B$ in the program (the order is irrelevant):

- *Local analysis:* The analysis described in the previous section is applied to C.
- *Propagation of information to other clauses:* For each $q(A_1,\ldots,A_k)$ in B, the information contained in the formal parameters $X_1,\ldots,X_k$ and the actual parameters $A_1,\ldots,A_k$ are updated as follows:

$$\begin{array}{rcl} \mathsf{ni\_set}(X_i) & := & \mathsf{ni\_set}(X_i) \sqcup \mathsf{ni\_set}(A_i); \\ \mathsf{ni\_set}(A_i) & := & \mathsf{ni\_set}(X_i); \end{array}$$

7.3 Domain Analysis

The static analysis scheme presented can be also used to infer domain information for those variables X such that $\mathsf{ni\_set}(X) = 0$. The new abstraction lattice $\mathcal{D}$ is a lattice whose points are integer intervals. The order in the lattice is provided by the relation $[x,y] \leq [x',y']$ iff $x' \leq x$ and $y \leq y'$. The join operation $\sqcup$ between two intervals is:

$[x,y] \sqcup [x',y'] = [\min(x,x'), \max(y,y')]$. The analysis rules are similar to those in Figure 9. For instance:

$$\begin{array}{rcl} X \in \mathsf{int}(s,t) & \mapsto & \mathtt{dom}(X) := \mathtt{dom}(X) \sqcup [s,t] \\ X = n & \mapsto & \mathtt{dom}(X) := \mathtt{dom}(X) \sqcup [n,n] \\ X = Y & \mapsto & \mathtt{dom}(X) := \mathtt{dom}(X) \sqcup \mathtt{dom}(Y); \\ & & \mathtt{dom}(Y) := \mathtt{dom}(X) \end{array}$$

As an example, consider the CLP($\mathcal{SET}$) program:

```
a(X,Y) :-                    p(Z) :-
   X ∈ int(1,3) ∧               Z = 3.
   X ≠ Y ∧                   p(Z) :-
   p(Y).                        Z = 5.
```

The program analysis determines the following properties:

$$\begin{array}{ccc} \mathsf{ni\_set}(X) = 0 & \mathsf{ni\_set}(Y) = 0 & \mathsf{ni\_set}(Z) = 0 \\ X \in \mathsf{int}(1,3) & Y \in \mathsf{int}(3,5) & Z \in \mathsf{int}(3,5) \end{array}$$

Without this piece of information, a constraint such as `X` $\neq$ `Y` would be maintained in the $\mathcal{SET}$ part of the constraint, while with the result of the static analysis it becomes possible to pass it to $SAT_{\mathcal{FD}}$, and in particular, to use `X` and `Y` as arguments of a labeling. Thus, this analysis helps in preparing the domain constraints to be sent to CLP($\mathcal{FD}$).

7.4 Application of the Static Analysis Framework

The results of the static analysis can be directly translated to new constraints. For all variables X such that $\mathsf{ni\_set}(X) = 0$ we can generate the new constraint $\mathsf{integer}(X)$. In addition, for each one of such variables it is possible to introduce the constraint $X \in \mathtt{dom}(X)$.

Moreover, in some cases it is possible to directly compile a fragment of a CLP($\mathcal{SET}$) program P into CLP($\mathcal{FD}$). This can be done, for example, when, for all terms in P we have that $\mathsf{ni\_set}(s) \in \{0,1\}$. For example, consider the following program, used to find subsets S of $\{1,\ldots,n\}$ such that, for each pair of (not necessarily distinct) numbers in S, their sum is not in S (this problem is a simplification of the problem of computing Shur numbers):

$$\begin{array}{l} shur(N,S) :- \\ \quad S \subseteq \mathsf{int}(1,N) \ \wedge \\ \quad (\forall X \in S)(\forall Y \in S)(\forall Z \in S)(Z \neq X+Y). \end{array}$$

The $\forall$ are used to encode RUQs, as discussed earlier; the $\subseteq$ predicate is a constraint predicate that can be defined using $\cup_3$—i.e., $s \subseteq t \Leftrightarrow \cup_3(s,t,t)$ [6]. The static analysis process is applied to the program obtained from the removal of the RUQs. The static analysis produces

$$\begin{array}{l} \mathsf{ni\_set}(N) = \mathsf{ni\_set}(X) = \mathsf{ni\_set}(Z) = \mathsf{ni\_set}(T) = 0 \\ \mathsf{ni\_set}(S) = 1 \end{array}$$

These results can be used to directly derive the SICStus CLP($\mathcal{FD}$) program in Figure 10. The predicate `subsetint` is a standard translation of the built-in $\subseteq$ when its second argument is an interval. The last three predicates on the right are auxiliary predicates that do not depend on the specific program. In Table 2 we report the running times obtained by running the same program in CLP($\mathcal{SET}$) and CLP($\mathcal{SET}$)$^{\mathsf{int}}$ with the given automatic transformation to CLP($\mathcal{FD}$). The programs have been executed for different values of N, using a PC, 1GHz, using SICStus Prolog 3.10. Precisely, in {log} (our implementation of CLP($\mathcal{SET}$)) the goal is `Q={A:shur(N,A)}` and in SICStus (execution of the

```
shur(N,S):-                  subsetint(S,M,N):-
  subsetint(S,1,N),            X in M .. N,
  s1(S,S),                     lengthfd(S,X),
  labeling([],S).              domain(S,M,N),
                               orderedfd(S).

s1([],S).
s1([X|R],S):-                lengthfd([],0).
  nonmemberfd(X,R),          lengthfd([_|R],N):-
  s2(S,S,X),                   N #> 0,
  s1(R,S).                     M #= N - 1,
                               lengthfd(R,M).

s2([],S,X).
s2([Y|R],S,X):-              orderedfd([]).
  nonmemberfd(Y,R),          orderedfd([_]).
  s3(S,S,X,Y),               orderedfd([A,B|R]):-
  s2(R,S,X).                   A #< B,
                               orderedfd([B|R]).

s3([],S,X,Y).
s3([Z|R],S,X,Y):-            nonmemberfd(A,[]).
  nonmemberfd(Z,R),          nonmemberfd(A,[X|R]):-
  Z #\= X+Y,                    A #\=X,
  s3(R,S,X,Y).                  nonmemberfd(A,R).
```

Figure 10: The compiled program

program automatically derived from CLP($\mathcal{SET}$)$^{\mathsf{int}}$ using static analysis) the goal is `setof(A,shur(N,A),Q)`.

Table 2: Computational results for the `shur` program

N	5	6	7	8	16	24
{log}	11.7s	54s	3m.57s	∞	∞	∞
CLP($\mathcal{FD}$)	0.01s	0.02s	0.04s	0.08s	6.5s	3m.48s
Speedup	1200	2700	6000	∞	∞	∞

8. CONCLUSIONS

In this paper we presented an extension of the CLP language on hereditarily finite sets CLP($\mathcal{SET}$). The extension provides the ability to combine manipulation of domain variables (as in CLP($\mathcal{FD}$)) with the ability to manipulate the domain themselves, seen as finite sets. On one hand, this extends the expressive power of CLP($\mathcal{FD}$), allowing domains to play the role of first-class citizens of the language. On the other hand, the ability to automatically determine domain/domain variables (one for CLP($\mathcal{SET}$) and one for CLP($\mathcal{FD}$)) allows the system to effectively distribute constraints between two constraint solvers (one for CLP($\mathcal{SET}$) and one for CLP($\mathcal{FD}$)), thus using the most efficient resolution mechanisms to solve each constraint. Static analysis mechanisms have been developed to support the execution of the extended constraint solving system. The extensions described in this paper have been implemented and are available in the current distribution of the {log} system [14].

The work presented provides a number of new avenues for further generalizations. We plan to extend the results to sets of integer elements (or sets of elements chosen from a finite universe) and to sets of sets of integer elements. These situations are precisely identified by the program analysis technique, when the outputs for a set s are $\mathsf{ni\_set}(s) = 1$ or $\mathsf{ni\_set}(s) = 2$, respectively. Several algorithms are based on operations on these two kinds of sets. For instance graphs related problems deal with sets of nodes and sets of sets of nodes (edges and/or partitions). Similarly, cooperations with constraint solvers on real numbers, as in $CLP(\mathcal{R})$, will be investigated.

9. ACKNOWLEDGMENTS

We would like to thank Sara Manzoli for the useful discussions and the anonymous referees for their useful comments. The work is partially supported by MIUR project: *Automatic Aggregate—and number—Reasoning for Computing* and by NSF grants EIA-9810732, CCR-9900320, EIA-0220590, CCR-9875279, EIA-0130887, and CCR-9820852.

10. REFERENCES

[1] Arenas-Sánchez, P., and Rodríguez-Artalejo, M. A General Framework for Lazy Functional Logic, Programming with Algebraic Polymorphic Types. *Theory and Practice of Logic Programming 2*, 1 (2001), 185–245.

[2] Beldiceanu, N., and Carlsson, M. Revisiting the Cardinality Operator and Introducing the Cardinality-Path Constraint Family. In *International Conference on Logic Programming* (2001), P. Codognet, Ed., Springer Verlag, pp. 59–73.

[3] F. Bouquet, B. Legeard, and F. Peureux. CLPS-B - a constraint solver for B. In J.-P. Katoen and P. Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 2280, pages 188–204. Springer-Verlag, 2002.

[4] Codognet, P., and Diaz, D. Compiling constraints in CLP(FD). *Journal of Logic Programming 27* (1996), 185–226.

[5] Dovier, A., Omodeo, E. G., Pontelli, E., and Rossi, G. {log}: A Language for Programming in Logic with Finite Sets. *Journal of Logic Programming 28*, 1 (1996), 1–44.

[6] Dovier, A., Piazza, C., Pontelli, E., and Rossi, G. Sets and Constraint Logic Programming. *ACM Transactions on Programming Languages and Systems 22*, 5 (2000), 861–931.

[7] Flener, P., Hnich, B., and Hiziltan, Z. Compiling high-level type constructor in constraint programming. In *PADL2001* (2001), I. Ramakrishnan, Ed., no. 1990 in Lecture Notes in Computer Science, Springer-Verlag, Berlin, pp. 229–244.

[8] Gervet, C. Interval Propagation to Reason about Sets: Definition and Implementation of a Practical Language. *Constraints 1* (1997), 191–246.

[9] IC Parc. *The ECLiPSe Constraint Logic Programming System.* London. `www.icparc.ic.ac.uk/eclipse/`.

[10] Jaffar, J., and Maher, M. J. Constraint Logic Programming: A Survey. *Journal of Logic Programming 19–20* (1994), 503–581.

[11] Jayaraman, B. Implementation of Subset-Equational Programs. *Journal of Logic Programming 12*, 4 (1992), 299–324.

[12] Marriott, K., and Stuckey, P. J. *Programming with Constraints: an Introduction.* The MIT Press, Cambridge, Mass., 1998.

[13] Régin, J.-C. Generalized Arc Consistency for Global Cardinality Constraints. In *Proceedings of the AAAI/IAAI Conference* (1996), pp. 209–215.

[14] Rossi, G. *The {log} Constraint Logic Programming Language.* `prmat.math.unipr.it/~gianfr/setlog.Home.html`.

[15] Shmueli, O., Tsur, S., and Zaniolo, C. Compilation of Set Terms in the Logic Data Language (LDL). *Journal of Logic Programming 12*, 1/2 (1992), 89–119.

[16] Swedish Institute of Computer Science. *The SICStus Prolog Home Page.* `www.sics.se`.

[17] Van Hentenryck, P. *Constraint Satisfaction in Logic Programming.* MIT Press, 1989.

[18] Van Hentenryck, P., and Deville, Y. The Cardinality Operator: a New Logical Connective for Constraint Logic Programming. In *International Conference on Logic Programming* (1991), K. Furukawa, Ed., MIT Press, pp. 745–759.

[19] Yakhno, T., and Petrov, E. Extensional Set Library for ECLiPSe. In *PSI'99* (2000), D. Bjørner, M. Broj, and A. Zamulin, Eds., no. 1755 in Lecture Notes in Computer Science, Springer-Verlag, Berlin, pp. 434–444.