International Journal on Artificial Intelligence Tools
© World Scientific Publishing Company

Automatic Generation of Rule-based Solvers for Intentionally Defined Constraints

Slim Abdennadher
Computer Science Department, Oettingenstr. 67, 80538 München, Germany
abdennad@informatik.uni-muenchen.de

Christophe Rigotti
INSA Lyon, 69621 Villeurbanne Cedex, France
Christophe.Rigotti@insa-lyon.fr

A general approach to implement propagation and simplification of constraints consists of applying rules over these constraints. However, a difficulty that arises frequently when writing a constraint solver is to determine the constraint propagation algorithm. In previous work, different methods for automatic generation of rule-based solvers for constraints defined over finite domains have been proposed [1,2,3,4]. In this paper, we present a method for generating rule-based solvers for constraint predicates defined by means of a constraint logic program, even when the constraint domain is infinite.

Keywords: Constraint Solving, Rule-based Programming, Machine Learning

1. Introduction

In rule-based constraint programming, the solving process of constraints consists of a repeated application of rules. In general, we distinguish two kinds of rules: simplification and propagation rules. Simplification rules rewrite constraints to simpler constraints while preserving logical equivalence, e.g. $X \leq Y \wedge Y \leq X \Leftrightarrow X = Y$. Propagation rules add new constraints which are logically redundant but may cause further simplification, e.g. $X \leq Y \wedge Y \leq Z \Rightarrow X \leq Z$.

However, a difficulty that arises frequently when writing a constraint solver is to determine the constraint propagation algorithm. In previous work, different methods for automatic generation of rule-based solvers for constraints defined over finite domains have been proposed [1,2,3,4]. It has also been shown in [4] that these rules generated automatically can lead to more efficient constraint reasoning than rules found by hand.

In this paper, we present an algorithm, called PropMiner, that can be used to generate propagation rules for constraint predicates defined by means of a constraint logic program, even when the constraint domain is infinite. We also show that

this algorithm can be completed with the algorithm presented in [4] to transform some propagation rules into simplification rules improving both the time and space behavior of constraint solving.
The combination of these techniques can be seen as a software engineering tool to help solver developers to find out propagation and simplification rules. Using this tool, the user only has to determine the semantics of the constraints of interest by means of their intentional definitions (i.e. a constraint logic program), and to specify the admissible syntactic form of the rules she/he wants to obtain.

Example 1 Consider the following constraint logic program, where $min(A,B,C)$ means that C is the minimum of A and B:

$$\begin{aligned} min(A,B,C) &\leftarrow A{\leq}B \ \wedge \ C{=}A. \\ min(A,B,C) &\leftarrow B{\leq}A \ \wedge \ C{=}B. \end{aligned}$$

For the predicate min, our algorithm PropMiner described in Section 3 generates the following propagation rules if the user specifies that the left hand side of the rules may consist of min constraints and equality constraints:

$$\begin{aligned} min(A,B,C) &\Rightarrow C{\leq}A \ \wedge \ C{\leq}B. \\ min(A,B,C) \ \wedge \ A{=}B &\Rightarrow A{=}C. \end{aligned}$$

For example, the second rule means that the constraint $min(A,B,C)$ when it is known that the input arguments A and B are equal can propagate the constraint that the output C must be equal to the input arguments.
If the user additionally allows disequality and less-or-equal constraints on the left hand side of the rules, the algorithm generates the following rules:

$$\begin{aligned} min(A,B,C) \ \wedge \ C{\neq}B &\Rightarrow C{=}A. \\ min(A,B,C) \ \wedge \ C{\neq}A &\Rightarrow C{=}B. \\ min(A,B,C) \ \wedge \ B{\leq}A &\Rightarrow C{=}B. \\ min(A,B,C) \ \wedge \ A{\leq}B &\Rightarrow C{=}A. \end{aligned}$$

Using the algorithm presented in [4], as illustrated in Section 6, some propagation rules can be transformed into simplification rules and we obtain the following rule-based constraint solver for min:

$$\begin{aligned} min(A,B,C) &\Rightarrow C{\leq}A \ \wedge \ C{\leq}B. \\ min(A,B,C) \ \wedge \ A{=}B &\Rightarrow A{=}C. \\ min(A,B,C) \ \wedge \ C{\neq}B &\Rightarrow C{=}A. \\ min(A,B,C) \ \wedge \ C{\neq}A &\Rightarrow C{=}B. \\ min(A,B,C) \ \wedge \ A{\leq}B &\Leftrightarrow C{=}A \ \wedge \ A{\leq}B. \\ min(A,B,C) \ \wedge \ B{\leq}A &\Leftrightarrow C{=}B \ \wedge \ B{\leq}A. \end{aligned}$$

For example, the goal $min(A, B, B)$ will be transformed into $B \leq A$ using the first propagation rule and then the last simplification rule.

The generated rules can be directly encoded in a rule-based programming language, e.g., Constraint Handling Rules (CHR) [5], to provide a running constraint solver. However, the generation techniques proposed in this paper can be adapted to any other language.

The paper is organized as follows. After a few preliminaries on notation and terminology, we present in Section 3 an algorithm to generate propagation rules for constraint predicates defined by a constraint logic program. In Section 4, we give more examples for the use of our algorithm. We discuss in Section 5 how recursive programs can be handled. Section 6 illustrates by means of an example the transformation method of propagation rules into simplification rules. Finally, we conclude with a summary and compare the proposed approach with related work. A preliminary short version of this paper was presented at ICTAI'01 [6].

2. Preliminaries

The reader is referred to [7] for a detailed introduction to Constraint Logic Programming (CLP). In this section, we give only those concepts and notation that we shall need in the following sections.

The CLP programs are parameterized by a constraint system defined by a 4-tuple $\langle \Sigma, \mathcal{D}, \mathcal{L}, T \rangle$ and a signature Π determining the predicate symbols defined by a program. Σ is a signature determining the predefined predicate and function symbols, $\mathcal{D}$ is a Σ-structure (the domain of computation), $\mathcal{L}$ is a class of Σ-formulas (the constraints), and T is a first-order Σ-theory that is an axiomatization of the properties of $\mathcal{D}$.

We require that T is satisfaction complete with respect to $\mathcal{L}$, that is, for every constraint $c \in \mathcal{L}$ either $T \models \tilde{\exists} c$ or $T \models \neg \tilde{\exists} c$, where $\tilde{\exists}(\phi)$ denotes the existential closure of ϕ. We also require that $\mathcal{D}$ and T correspond on $\mathcal{L}$, i.e. $\mathcal{D}$ is a model of T and for every c in $\mathcal{L}$ we have $\mathcal{D} \models \tilde{\exists} c$ iff $T \models \tilde{\exists} c$. Note that these requirements are fulfilled by most commonly used CLP languages.

Definition 1 An *atomic constraint* is a formula of the form $c(t_1, \ldots, t_n)$, where $t_1, \ldots, t_n$ are terms and $c \in \Sigma$ is a predefined predicate symbol.

A *constrained clause* is a rule of the form

$$H \leftarrow B_1 \wedge \ldots \wedge B_n \wedge C_1 \wedge \ldots \wedge C_m$$

where $H, B_1, \ldots, B_n$ are atoms and $C_1, \ldots, C_m$ are atomic constraints. A *goal* is a set of atoms and atomic constraints interpreted as their conjunction. An *answer* is a set of atomic constraints also interpreted as their conjunction. A *CLP program* is a finite set of constrained clauses. The logical semantics of a CLP program P is its Clark's completion and is denoted by P^*.

In programs, goals and answers, when clear from the context, we use upper case letters (resp. lower case letters and numbers) to denote variables (resp. constants).

3. Generation of Propagation Rules

In this section, we present an algorithm, called PROPMINER, to generate propagation rules for constraints using the intensional definitions of the constraint predicates. These definitions are given by means of a program in a constraint logic programming (CLP) language.

3.1. *Rules of Interest*

A *constraint pattern* over a set of atomic constraints A is a finite subset of A. A constraint pattern $C \subseteq \mathcal{A}$ is interpreted as the conjunction of the atomic constraints in C. The set of variables appearing in A is denoted by $Var(\mathcal{A})$.
A *propagation rule* is a rule of the form $C_1 \Rightarrow C_2$ or of the form $C_1 \Rightarrow false$, where C_1 and C_2 are constraint patterns. C_1 is called the *left hand side* (lhs) and C_2 the *right hand side* (rhs) of the rule. A rule of the form $C_1 \Rightarrow false$ is called *failure rule*.
To formulate the logical semantics of these rules, we use the following notation: let $\mathcal{V}$ be a set of variables then $\exists_{-\mathcal{V}}(\phi)$ denotes the existential closure of ϕ except for the variable in $\mathcal{V}$.

Example 2 The following propagation rules describe the (partial) order $<$ that can handle variable arguments.

$$\begin{aligned} \{X{<}X\} &\Rightarrow false. \\ \{X{<}Y,\ Y{<}X\} &\Rightarrow false. \\ \{X{<}Y,\ Y{<}Z\} &\Rightarrow \{X{<}Z\}. \end{aligned}$$

The first two rules are failure rules. The first rule describes the irreflexivity of $<$, the second rule defines the asymmetry of $<$ and the third one the transitivity of $<$.

Definition 2 A propagation rule $\{c_1, \ldots, c_n\} \Rightarrow \{d_1, \ldots, d_m\}$ is *valid* wrt. the constraint theory T and the CLP program P iff $P^*, T \models \bigwedge_i c_i \rightarrow \exists_{-\mathcal{V}}(\bigwedge_j d_j)$, where $\mathcal{V}$ is the set of variables appearing in $\{c_1, \ldots, c_n\}$.
A failure rule $\{c_1, \ldots, c_n\} \Rightarrow false$ is *valid* wrt. T and P if and only if $P^*, T \models \neg\tilde{\exists}(\bigwedge_i c_i)$.

To reduce the number of rules which are uninteresting to build a solver, we restrict with a syntactic bias the generation to a particular set of rules called *relevant propagation rules*. These rules must contain in their lhs at least the constraints on which we want to propagate information and all atomic constraints in the lhs must be connected with common variables. This is defined more precisely by the notion of *interesting pattern*.

Definition 3 A set of atomic constraints $\mathcal{A}$ is an *interesting pattern* wrt. a set of atomic constraints $Base_{lhs}$ if and only if the following conditions are satisfied:

1. $Base_{lhs} \subseteq \mathcal{A}$.

2. the graph defined by the relation $join_{\mathcal{A}}$ is connected, where $join_{\mathcal{A}}$ is a binary relation that holds for pairs of atomic constraints in A that share at least one variable, i.e., $join_{\mathcal{A}} = \{\langle c_1, c_2\rangle \mid c_1 \in \mathcal{A}, c_2 \in \mathcal{A}, Var(\{c_1\}) \cap Var(\{c_2\}) \neq \emptyset\}$.

A *relevant propagation rule* wrt. $Base_{lhs}$ is a propagation rule such that its lhs is an interesting pattern wrt. $Base_{lhs}$.

3.2. *The* PROPMINER *Algorithm*

In this section, we describe the PROPMINER algorithm to generate propagation rules from a program P expressed in a CLP language determined by $\langle \Sigma, \mathcal{D}, \mathcal{L}, T\rangle$. The algorithm takes as input the program P and two sets of atomic constraints $Base_{lhs}$ and $Cand_{lhs}$ used to specify the possible lhs of the rules: $Base_{lhs}$ is the constraint for which we want to obtain propagation rules and $Cand_{lhs}$ is a set of atomic constraints for which we already have a (built-in) solver. The possible lhs of the rules are the constraints that are relevant wrt. $Base_{lhs}$ and subset of $Base_{lhs} \cup Cand_{lhs}$.

3.2.1. *Principle*

From an abstract point of view, the algorithm enumerates each possible lhs constraint subset of $Base_{lhs} \cup Cand_{lhs}$ (denoted by C_{lhs}). For each C_{lhs} it computes a set of constraints noted C_{rhs} such that $C_{lhs} \Rightarrow C_{rhs}$ is valid wrt. T and P and relevant wrt. $Base_{lhs}$.
For each C_{lhs} it determines C_{rhs} by calling the CLP system to execute C_{lhs} as a goal and then

1. if C_{lhs} has no answer then it produces the failure rule $C_{lhs} \Rightarrow false$.

2. if C_{lhs} has a finite number of answers $\{Ans_1, \ldots, Ans_n\}$ then let C_{rhs} be the *least general generalization* (lgg) of $\{Ans_1, \ldots, Ans_n\}$ as defined by [8]. C_{rhs} is then in some sense the strongest constraint common to all answers as illustrated below (see Example 3). If C_{rhs} is not empty then the algorithm produces the rule $C_{lhs} \Rightarrow C_{rhs}$.

It is clear that these two criteria can be applied only if all answers can be collected in finite time. The extension of the algorithm to handle recursive programs leading to non-terminating executions is discussed in Section 5.
The algorithm is given in Figure 1. To simplify its presentation, we consider that all possible lhs are stored in a list. For efficiency reasons the concrete implementation is based on a tree and unnecessary candidates are not materialized. More details on the implementation are given in Section 3.4.
A particular ordering is used to enumerate the lhs candidates so that the more general lhs are tried before the more specific ones. Then, we use the following

begin

Let R be an empty set of rules.
Let L be a list containing all non-empty subsets
of $Base_{lhs} \cup Cand_{lhs}$ in any order.

Remove from L any element C which is not an
interesting pattern wrt. $Base_{lhs}$. Then order L with any
total ordering compatible with the subset partial ordering
(i.e., for all C_1 in L if C_2 is after C_1 in L then $C_2 \not\subset C_1$).

while L is not empty **do**
Let C_{lhs} be the first element of L.
Remove from L its first element.
Let $\mathcal{A}$ be the set of answers for the goal C_{lhs} wrt.
the program P.
if $\mathcal{A}$ is empty **then**
add the failure rule $(C_{lhs} \Rightarrow false)$ to R
and remove from L each element C such that
$C_{lhs} \subset C$.
else
if $\mathcal{A}$ is finite **then**
compute the set of constraints C_{rhs}
as the least general generalization (lgg)
of $\mathcal{A}$
if C_{rhs} is not empty **then**
add the rule $(C_{lhs} \Rightarrow C_{rhs})$ to R
endif
endif
endif
endwhile

output R

end

Figure 1: The PROPMINER Algorithm

pruning criterion which improves greatly the efficiency of the algorithm: if a rule $C_{lhs} \Rightarrow false$ is generated then there is no need to consider any superset of C_{lhs} to form other rule lhs.
We now illustrate on the following example the basic behavior of the algorithm PROPMINER. More uses of the algorithm are given in Section 4.

Example 3 *Consider the following CLP program defining p and q:*

$$\begin{array}{lcl} p(X,Y,Z) & \leftarrow & q(X,Y,Z). \\ p(X,Y,Z) & \leftarrow & X{\leq}W \ \wedge \ Y{=}W \ \wedge \ X{>}Z. \\ q(X,Y,Z) & \leftarrow & X{\leq}a \ \wedge \ Y{=}a \ \wedge \ Z{\neq}b. \end{array}$$

We use the algorithm to find rules to propagate constraints over constraints involving p. Let $Base_{lhs} = \{p(X,Y,Z)\}$ *and let for example* $Cand_{lhs}$ *be the set* $\{X{\leq}Z,\ Y{=}a,\ Z{=}b\}$.
When the while loop is entered for the first time we have

$$\begin{array}{rl} L \ = \{ & \{p(X,Y,Z)\},\ \{p(X,Y,Z),\ X{\leq}Z\}, \\ & \{p(X,Y,Z),\ Y{=}a\},\ \{p(X,Y,Z),\ Z{=}b\}, \\ & \{p(X,Y,Z),\ X{\leq}Z,\ Y{=}a\}, \\ & \{p(X,Y,Z),\ X{\leq}Z,\ Z{=}b\}, \\ & \{p(X,Y,Z),\ Y{=}a,\ Z{=}b\}, \\ & \{p(X,Y,Z),\ X{\leq}Z,\ Y{=}a,\ Z{=}b\} \ \ \} \end{array}$$

Each element in L is executed in turn as a goal and the corresponding answers are collected and used to build a rule rhs. For example, $\{p(X,Y,Z),\ Z{=}b\}$ *leads to a single answer* $Ans_1 = \{X{\leq}W,\ Y{=}W,\ X{>}Z,\ Z{=}b\}$. *The lgg is simply* Ans_1 *itself and we have the propagation rule* $\{p(X,Y,Z),\ Z{=}b\} \Rightarrow \{X{\leq}W,\ Y{=}W,\ X{>}Z,\ Z{=}b\}$. *For* $\{p(X,Y,Z), X{\leq}Z\}$ *we have again a single answer* $\{X{\leq}a, Y{=}a, Z{\neq}b,\ X{\leq}Z\}$ *and thus also a trivial lgg producing the rule*

$$\{p(X,Y,Z), X{\leq}Z\} \Rightarrow X{\leq}a, Y{=}a, Z{\neq}b, X{\leq}Z\}.$$

For the goal $\{p(X,Y,Z),\ Y{=}a\}$, *the situation is different since we have the following answers*

$$Ans_1 = \{X{\leq}a, Y{=}a, Z{\neq}b\} \text{ and } Ans_2 = \{X{\leq}a, Y{=}a, X{>}Z\}.$$

The lgg which is based on a syntactical generalization is $\{X{\leq}a, Y{=}a\}$ *and we have the rule* $\{p(X,Y,Z), Y{=}a\} \Rightarrow \{X{\leq}a, Y{=}a\}$.
The situation may be more tricky. For example, the goal $\{p(X,Y,Z)\}$ *have two answers*

$$Ans_1 = \{X{\leq}a,\ Y{=}a,\ Z{\neq}b\} \text{ and } Ans_2 = \{X{\leq}W,\ Y{=}W,\ X{>}Z\}$$

having no common atomic constraint. Fortunately, the lgg correspond in some sense to the least upper bound of $\{Ans_1, Ans_2\}$ *wrt. the* θ*-subsumption ordering* [8] *(more precisely it represents the equivalence class of constraints that corresponds to this*

least upper bound). Thus, the lgg of $\{Ans_1, Ans_2\}$ *is* $\{X{\leq}E,\ Y{=}E\}$*, where* E *is a new variable, and the algorithm produces the rule* $\{p(X,Y,Z)\} \Rightarrow \{X{\leq}E, Y{=}E\}$.

The effect of the pruning criterion is straightforward. The goal $G = \{p(X,Y,Z),\ X{\leq}Z,\ Z{=}b\}$ *has no answer and leads to the rule*

$$\{p(X,Y,Z),\ X{\leq}Z,\ Z{=}b\} \Rightarrow false.$$

Then the element $\{p(X,Y,Z),\ X{\leq}Z,\ Y{=}a,\ Z{=}b\}$ *that is a super set of* G *is simply removed from* L *and will not be considered to generate any rule.*

3.2.2. *Properties*

It is straightforward to see that the algorithm is complete in the sense that if $C_{lhs} \subseteq Base_{lhs} \cup Cand_{lhs}$ is an interesting pattern wrt. $Base_{lhs}$ and there is no $C \subset C_{lhs}$ such that $C \Rightarrow false$ is valid, then C_{lhs} is considered by the algorithm as a candidate to form the lhs of a rule.
To establish the soundness of the algorithm, we need the following results presented in [7].

Theorem 1 Let P be a program in the CLP language determined by $\langle \Sigma, \mathcal{D}, \mathcal{L}, T \rangle$, where $\mathcal{D}$ and T correspond on $\mathcal{L}$. Suppose that T is satisfaction complete wrt. $\mathcal{L}$, and that P is executed on a CLP system for this language. Then:

1. If a goal G has a finite computation tree, with answers $c_1, \ldots, c_n$ then $P^*, T \models G \leftrightarrow \exists_{-\mathcal{V}}(c_1 \vee \ldots \vee c_n)$, where $\mathcal{V}$ is the set of variables appearing in G.
2. If a goal G is finitely failed for P then $P^*, T \models \neg G$.

The soundness of PropMiner is stated by the following theorem.

Theorem 2 (Soundness) The PropMiner algorithm produces propagation rules that are relevant wrt. $Base_{lhs}$ and valid wrt. T and P.

Proof. All C_{lhs} considered are interesting pattern wrt. $Base_{lhs}$, thus only relevant rules can be generated. If a rule of the form $C_{lhs} \Rightarrow false$ is produced then by property 2 in Theorem 1 this rule is valid. Suppose a rule of the form $C_{lhs} \Rightarrow C_{rhs}$ is generated. Then C_{rhs} is the lgg of a finite set of answers $\{Ans_1, \ldots, Ans_n\}$ obtained by the execution of the goal C_{lhs} on the program P.
By property 1 in Theorem 1 we have $P^*, T \models C_{lhs} \leftrightarrow \exists_{-\mathcal{V}}(Ans_1 \vee \ldots \vee Ans_n)$, where $\mathcal{V}$ is the set of variables appearing in C_{lhs}. Since C_{rhs} is the lgg of $\{Ans_1, \ldots, Ans_n\}$ then by [8] we know that $Ans_1 \vee \ldots \vee Ans_n \rightarrow C_{rhs}$. Thus $P^*, T \models C_{lhs} \rightarrow \exists_{-\mathcal{V}}(Ans_1 \vee \ldots \vee Ans_n)$, i.e., $C_{lhs} \Rightarrow C_{rhs}$ is valid wrt. T and P.

3.3. *Interesting Rules for Constraint Solvers*

The basic form of the PropMiner algorithm given in Figure 1 produces a very large set of rules. Most of these rules are redundant (partly or completely) or propagates too weak constraints or on the contrary propagates too many stronger constraints

(inflating considerably the constraint store at runtime) and thus may be of little interest to built a constraint solver.
We present in this section mandatory complementary processing that is integrated in the basic algorithm in order to generate rules of practical interest wrt. solver construction.
Consider again the CLP program of Example 3. Let $Base_{lhs} = \{p(X,Y,Z)\}$ and let us use a richer set of atomic constraints to form the lhs of the rules $Cand_{lhs} = \{X{\leq}Z,\ Y{\leq}X,\ X{=}Z, Y{=}Z,\ X{=}b,\ Y{=}a,\ Z{=}b\}$.
Among the rules generated by the basic algorithm PROPMINER, we have:
(1) $\{p(X,Y,Z)\} \Rightarrow \{X{\leq}E,\ Y{=}E\}$
(2) $\{p(X,Y,Z),\ X{\leq}Z\} \Rightarrow \{X{\leq}a,\ Y{=}a,\ Z{\neq}b,\ X{\leq}Z\}$
(3) $\{p(X,Y,Z),\ Y{\leq}X\} \Rightarrow \{X{\leq}E,\ Y{=}E,\ Y{\leq}X\}$
(4) $\{p(X,Y,Z),\ X{=}Z\} \Rightarrow \{X{\leq}a,\ Y{=}a,\ Z{\neq}b,\ X{=}Z\}$
(5) $\{p(X,Y,Z),\ Y{=}Z\} \Rightarrow \{X{\leq}a,\ Y{=}a,\ Z{\neq}b,\ Y{=}Z\}$
(6) $\{p(X,Y,Z),\ X{=}b\} \Rightarrow \{X{\leq}E,\ Y{=}E,\ X{=}b\}$
(7) $\{p(X,Y,Z),\ Y{=}a\} \Rightarrow \{X{\leq}E,\ Y{=}E,\ Y{=}a\}$
(8) $\{p(X,Y,Z),\ Z{=}b\} \Rightarrow \{X{\leq}W,\ Y{=}W,\ X{>}Z,\ Z{=}b\}$
(9) $\{p(X,Y,Z),\ X{\leq}Z,\ Z{=}b\} \Rightarrow false$

Since the algorithm only imposes that the exploration ordering is a total ordering compatible with the subset ordering on the lhs, the real order of the rules generated may be slightly different according to implementation choices (see Section 3.4). However, the specific processing presented in this section can still be applied.

3.3.1. *Removing redundancy*

The key idea of the simplification is to remove from the rhs of a rule R all atomic constraints that can be derived from the lhs of R using the built-in solvers and the rules already generated. If the remaining rhs is empty then the whole rule can be suppressed.
For example, according to this process rule (6) is removed because its rhs is fully redundant wrt. its lhs and wrt. rule (1). For rule (2) only the rhs is modified and becomes $\{X{\leq}a,\ Y{=}a,\ Z{\neq}b\}$, since $X{\leq}Z$ is trivially entailed by the lhs of the rule. Depending on the behavior of the built-in solvers, rule (4) may be only transformed into $\{p(X,Y,Z),\ X{=}Z\} \Rightarrow \{X{\leq}a,\ Y{=}a,\ Z{\neq}b\}$ while if we know the semantics of $\leq$ we may use rule (2) to derive the same atomic constraints. If the built-in solver does not allow to discover this redundancy, the user can simply add propagation rules to derive explicitely logical consequence of the built-in constraints. In this example, one of this complementary rules can be $\{X{=}Z\} \Rightarrow \{X{\leq}Z\}$ which allows to find that rule (4) is then fully redundant wrt. rule (2).
This simplification process also applies to failure rules. Suppose that the built-in solver is able to detect that $Z{=}b \wedge Z{\neq}b$ is inconsistent, then the rule (9) is removed since it is redundant wrt. rule (2).

3.3.2. *Generating stronger rhs*

If we consider rule (6) $\{p(X,Y,Z), X{=}b\} \Rightarrow \{X{\leq}E,\ Y{=}E, X{=}b\}$ the rhs constructed from the least general generalization of the answers obtained for the goal $\{p(X,Y,Z),\ X{=}b\}$ is in some sense too general. The execution of the goal gives two answers. One containing $\{Z{\neq}b\}$ and the other $\{X{>}Z,\ X{=}b\}$. From a semantical point of view, this leads clearly to $Z{\neq}b$ in both cases, but the least general generalization is mainly syntactical and do not retains this information.
If we want a richer rhs (containing $Z{\neq}b$) then we must have at hand a (built-in) solver that propagates $\{Z{\neq}b\}$ also in the second answer. If we do not have such a solver, then here again the user can provide himself complementary propagation rules (in this example the single rule $\{X{>}Y\} \Rightarrow \{X{\neq}Y\}$).

3.3.3. *Projecting variables*

For efficiency reasons in constraint solving it is particularly important to limit the number of variables.
Then a rule like $\{p(X,Y,Z)\} \Rightarrow \{X{\leq}E,\ Y{=}E\}$ should be avoided since it will create a new variable each time it is fired.
So, we simply project out such useless variables in the following way. We consider in turn each equality in the rhs of a rule. If this equality is of the form $E{=}F$ or $F{=}E$ where E and F are variables and E does not appear in the lhs of the rule, then we suppress this equality from the rhs and we apply the substitution transforming E into F to the whole remaining rhs.
More subtle situations may arise. Suppose that the second clause of the program given in Example 3 was $p(X,Y,Z) \leftarrow X{\leq}W \wedge Y{=}W \wedge Z{\neq}a$. Then, the first rule generated would have been $\{p(X,Y,Z)\} \Rightarrow \{X{\leq}E,\ Y{=}E,\ Z{\neq}F\}$. And then projecting out E would transform it into $\{p(X,Y,Z)\} \Rightarrow \{X{\leq}Y,\ Z{\neq}F\}$. During constraint solving the firing of this rule will add to the store the atomic constraint $Z{\neq}F$ where F is a new variable. This phenomena leads in general to a rather inefficient solving process. So, we propose the following optional treatment: When all other previous processing has been performed (simplification, additional propagation and projection of variable in equalities) the user can choose to apply a strict *range restriction* criteria: all atomic constraints in the rhs containing a variable that does not appear in the lhs is removed (e.g., $Z{\neq}F$ in the previous rule). This range restriction criteria is applied in all examples presented in this paper.

3.4. *Implementation Issues*

The key aspects of our implementation of the PROPMINER algorithm are presented in this section. The prototype has been developed under SICStus Prolog 3.7.1. It is written in Prolog and takes advantage of the rule-based programming language Constraint Handling Rules (CHR) [5] supported in this environment.

Using CHR. The CHR language facilitates in two ways the implementation of

the important processing described in Section 3.3. Firstly, we can use the rules generated as CHR rules and then run CHR to decide if a rule propagates new constraints wrt. the rules we have already. Secondly, the user can directly add new rules to perform complementary propagations wrt. the built-in solvers.

Clause encoding. It should be noticed that in this environment the equality $=$ is reserved to specify unification. So in practice, we use another binary predicate to denote the equality constraint. Moreover, the bindings of the variables due to the resolution steps are not handled explicitely as equalities in the store. Suppose that the third clause of the program given in Example 3 was written under the form $q(X, a, Z) \leftarrow X \leq a \wedge Z \neq b$. Then, for the goal $\{p(X, Y, Z),\ X \leq Z\}$ we may have not collected the atomic constraint $Y=a$ explicitely and thus $Y=a$ will not appear in the rhs of rule (2). Thus, we simply preprocess the clauses so that the atom in the head of a clause does not contain functors (including constants) and coreferences. The corresponding functors and coreferences are simply encoded by equality constraints in the body of the clause. For example a head of the form $p(X, a, X)$ will be transformed into $p(X, Y, Z)$ and the constraint $X=Z \wedge Y=a$ will be added to the body.

Enumeration of lhs. The PROPMINER algorithm enumerates lhs constraints (the elements in L). The implementation of this enumeration is based on the exploration of a tree corresponding to the lhs search space as described in [9]. The tree is explored using a depth first strategy. As in [3], the branches are expanded using a partial ordering on the lhs candidates such that the more general lhs are examined before more specialized ones. The partial ordering used in our implementation is the θ-subsumption ordering [8].

4. Practical Uses of PROPMINER

In this section, we show on examples that a practical application of our approach lies in solver development. All the set of rules presented in this section have been generated in a few seconds on a PC Pentium 3 with 128 MBytes of memory and a 500 MHZ processor.

For convenience, we introduce the following notation. Let c be a constraint symbol of arity 2 and D_1 and D_2 be two sets of terms. We define $atomic(c, D_1, D_2)$ as the set of all atomic constraints built from c over $D_1 \times D_2$. More precisely, $atomic(c, D_1, D_2) = \{c(\alpha, \beta) \mid \alpha \in D_1 \text{ and } \beta \in D_2\}$.

Example 4 *Consider the maximum constraint $max(A, B, C)$ meaning that C is the maximum of A and B, and defined by the CLP program below:*

$$\begin{aligned} max(A, B, C) &\leftarrow A \leq B \ \wedge \ C=B. \\ max(A, B, C) &\leftarrow B \leq A \ \wedge \ C=A. \end{aligned}$$

If the user specifies that the left hand side of the rules may consist of max *constraints*

and equality constraints by means of the following input:

$$\begin{aligned} Base_{lhs} &= \{max(A,B,C)\} \\ Cand_{lhs} &= atomic(=,\{A,B,C\},\{A,B,C\}) \end{aligned}$$

then the PROPMINER *algorithm generates the 2 following rules:*

$$\begin{aligned} max(A,B,C) &\Rightarrow A{\leq}C \ \wedge \ B{\leq}C. \\ max(A,B,C) \ \wedge \ A{=}B &\Rightarrow C{=}A. \end{aligned}$$

If the user allows additionally disequality and less-or-equal constraints on the left hand side of the rules using the following input:

$$\begin{aligned} Base_{lhs} &= \{max(A,B,C)\} \\ Cand_{lhs} &= atomic(=,\{A,B,C\},\{A,B,C\}) \ \cup \\ &\quad\ atomic(\neq,\{A,B,C\},\{A,B,C\}) \ \cup \\ &\quad\ atomic(\leq,\{A,B,C\},\{A,B,C\}) \end{aligned}$$

then the algorithm generates the two previous propagation rules and also the following four new rules:

$$\begin{aligned} max(A,B,C) \ \wedge \ C{\neq}B &\Rightarrow C{=}A. \\ max(A,B,C) \ \wedge \ C{\neq}A &\Rightarrow C{=}B. \\ max(A,B,C) \ \wedge \ B{\leq}A &\Rightarrow C{=}A. \\ max(A,B,C) \ \wedge \ A{\leq}B &\Rightarrow C{=}B. \end{aligned}$$

It should be noticed that to be able to generate the rule $max(A,B,C) \Rightarrow A{\leq}C \wedge B{\leq}C$*, the following rules for equality and less-or-equal constraints have to be present in the built-in solver to ensure the generation of stronger rhs (as illustrated in Section 3.3):*

$$\begin{aligned} X{\leq}Y \ \wedge \ Y{\leq}Z &\Rightarrow X{\leq}Z. \\ X{=}Y &\Rightarrow X{\leq}Y. \end{aligned}$$

If these rules are not already in the built-in solver, in our implementation the user can provide them very easily by means of CHR rules (see Section 3.4). Moreover, using this possibility, PROPMINER *can incorporate additional knowledge given by the user about the constraint of interest. For example, the user can express the symmetry of max with respect to the the first and second arguments by the rule:*

$$max(A,B,C) \Rightarrow max(B,A,C).$$

If this rule is added as a CHR rule to the built-in solver, then the PropMiner *algorithm generates only the following simplified set of 4 rules:*

$$\begin{array}{rcl} max(A,B,C) & \Rightarrow & A{\leq}C \ \wedge \ B{\leq}C. \\ max(A,B,C) \ \wedge \ A{=}B & \Rightarrow & C{=}A. \\ max(A,B,C) \ \wedge \ C{\neq}B & \Rightarrow & C{=}A. \\ max(A,B,C) \ \wedge \ B{\leq}A & \Rightarrow & C{=}A. \end{array}$$

Example 5 *If we consider the minimum constraint min, a set of rules similar to the rules for max is generated by* PropMiner *(Example 1). Then the user has the possibility to add these two sets of rules to the built-in solver and to execute* PropMiner *to generate interaction rules between min and max. This execution is performed with the following input*

$$\begin{array}{rcl} Base_{lhs} & = & \{min(A,B,C) \wedge max(D,E,F)\} \\ Cand_{lhs} & = & atomic(\neq, \{A,B,C\}, \{D,E,F\}) \end{array}$$

and a CLP program consisting of the definitions of min and max. Since the propagation rules specific to min and max alone have been added to the built-in solver, PropMiner *takes advantage of these rules to simplify many redundancies. Thus only 10 propagation rules specific to the conjunction of min with max are generated. Examples of rules are:*

$$\begin{array}{rcl} min(A,B,C) \ \wedge \ max(D,E,F) \ \wedge \ C{\neq}E \ \wedge \ C{\neq}D & \Rightarrow & F{\neq}C. \\ min(A,B,C) \ \wedge \ max(D,E,F) \ \wedge \ B{\neq}D \ \wedge \ A{\neq}D & \Rightarrow & D{\neq}C. \\ min(A,B,C) \ \wedge \ max(D,E,F) \ \wedge \ C{\neq}E \ \wedge \ B{\neq}D \ \wedge \ A{\neq}F & \Rightarrow & F{\neq}C. \\ min(A,B,C) \ \wedge \ max(D,E,F) \ \wedge \ C{\neq}D \ \wedge \ B{\neq}F \ \wedge \ A{\neq}E & \Rightarrow & F{\neq}C. \end{array}$$

Now, we show on the following example that the user can easily decide that specific constants of the domain must be used in the left hand side of the rules.

Example 6 *The absolute value, abs, can be defined by the constrained clauses*

$$\begin{array}{rcl} abs(A,B) & \leftarrow & A{\leq}0 \ \wedge \ A{=}(-B). \\ abs(A,B) & \leftarrow & 0{\leq}A \ \wedge \ A{=}B. \end{array}$$

Then, the user can specify with the following input that the constant 0 *must also be considered to form the left hand side of the rules:*

$$\begin{array}{rcl} Base_{lhs} & = & \{abs(A,B)\} \\ Cand_{lhs} & = & atomic(=, \{A,B,0\}, \{A,B,0\}) \ \cup \\ & & atomic(\neq, \{A,B,0\}, \{A,B,0\}) \ \cup \\ & & atomic(\leq, \{A,B,0\}, \{A,B,0\}) \end{array}$$

In this case, using an appropriate solver for the built-in constraints, the following propagation rules are generated:

$$\begin{aligned}
abs(A,B) &\Rightarrow 0{\leq}B \wedge A{\leq}B.\\
abs(A,B) \wedge B{\neq}A &\Rightarrow A{=}(-B) \wedge A{\leq}0.\\
abs(A,B) \wedge 0{\leq}A &\Rightarrow B{=}A.\\
abs(A,B) \wedge B{\leq}0 &\Rightarrow A{=}0.\\
abs(A,B) \wedge A{\leq}0 &\Rightarrow A{=}(-B).
\end{aligned}$$

In [1,2,3] first steps towards automatic generation of propagation rules have been done. In these approaches the constraints are defined extensionally over finite domains by a truth table or their solution tuples. This paper is an extension of these previous works towards infinite domains. Over finite domains, the algorithm PROPMINER can be used to generate the rules produced by the other methods.

Example 7 *For the boolean conjunction constraint $and(X,Y,Z)$ defined by the following clauses*

$$\begin{aligned}
and(X,Y,Z) &\leftarrow X{=}0 \wedge Y{=}0 \wedge Z{=}0.\\
and(X,Y,Z) &\leftarrow X{=}0 \wedge Y{=}1 \wedge Z{=}0.\\
and(X,Y,Z) &\leftarrow X{=}1 \wedge Y{=}0 \wedge Z{=}0.\\
and(X,Y,Z) &\leftarrow X{=}1 \wedge Y{=}1 \wedge Z{=}1.
\end{aligned}$$

the algorithm PROPMINER *with the following input*

$$\begin{aligned}
Base_{lhs} &= \{and(X,Y,Z)\}\\
Cand_{lhs} &= atomic(=,\{X,Y,Z\},\{X,Y,Z,0,1\})
\end{aligned}$$

generates the following propagation rules

$$\begin{aligned}
and(0,Y,Z) &\Rightarrow Z{=}0.\\
and(X,0,Z) &\Rightarrow Z{=}0.\\
and(1,Y,Z) &\Rightarrow Y{=}Z.\\
and(X,1,Z) &\Rightarrow X{=}Z.\\
and(X,X,Z) &\Rightarrow X{=}Z.\\
and(X,Y,1) &\Rightarrow X{=}1 \wedge Y{=}1.
\end{aligned}$$

For the negation constraint, $neg(X,Y)$, the PROPMINER *algorithm generates among other rules the following failure rule:*

$$neg(X,X) \Rightarrow false.$$

5. Handling Recursive Constraint Definitions

In this section, we show informally that the algorithm PROPMINER can be adapted when the CLP program P defining the constraint predicates is recursive and may lead to non-terminating executions.

As presented in Figure 1, for each possible rule lhs in L (denoted by C_{lhs}) the algorithm needs to collect in finite time all answers to the goal C_{lhs} wrt. the program P. In general, we cannot guarantee such a termination property, but we can use standard Logic Programming solutions developed to handle recursive clauses. For example, we can prefer a resolution based on the OLDT [10] scheme that ensures finite refutations more often than a resolution following the SLD principle (e.g., with the OLDT resolution the execution always terminates for Datalog programs). We can also decide to bound the depth of the resolution to stop the execution of a goal that may cause non-termination. In this case, if the execution of goal C_{lhs} has a resolution depth exceeding a given threshold, we interrupt this execution and proceed with the next possible lhs in L. Of course this strategy may be too restrictive, in the sense that it may stop too early some terminating executions and thus may avoid the generation of some interesting rules.

Example 8 Consider the well-known ternary *append* predicate for lists, which holds if its third argument is a concatenation of the first and the second argument. It is usually implemented by these two clauses:

$$\begin{array}{lcl} append(X,Y,Z) & \leftarrow & X{=}[] \ \wedge \ Y{=}Z. \\ append(X,Y,Z) & \leftarrow & X{=}[H|X1] \ \wedge \ Z{=}[H|Z1] \ \wedge \ append(X1,Y,Z1). \end{array}$$

Then, if we bound the resolution depth to discard non-terminating executions, the algorithm PROPMINER terminates and using the appropriate input produces, among others, the following rules:

$$\begin{array}{rcl} append(A,B,C) \ \wedge \ A{=}B \ \wedge \ C{=}[D] & \Rightarrow & false. \\ append(A,B,C) \ \wedge \ B{=}C \ \wedge \ C{=}[D] & \Rightarrow & A{=}[]. \\ append(A,B,C) \ \wedge \ C{=}[] & \Rightarrow & B{=}[] \ \wedge \ A{=}[]. \\ append(A,B,C) \ \wedge \ A{=}[] & \Rightarrow & B{=}C. \end{array}$$

6. Generation of Simplification Rules

In this section, we show by means of an example that the method presented in [4] can be applied on the propagation rules generated by PROPMINER to find simplification rules.

Starting from a given terminating and confluent set of propagation rules S encoded as CHR rules[1], the general principle of the process is as follows. We consider in

[1] This means essentially that equalities in lhs are encoded as coreferences. For example, a lhs of the form $min(A,B,C) \ \wedge \ A{=}B$ is encoded as $min(A,A,C)$

turn each rule R in S. For a rule R, we try to transform it into a simplification rule R' by shifting some constraints from the lhs into the rhs of the rule (it should be noticed that there are several possibilities to transform a propagation rule into a simplification rule). If the new program where R is replaced by R' remains terminating and confluent then the transformation is accepted and the next rule of the program is considered.

It should be noticed that for most existing CHR programs (i.e. set of CHR rules) it is straightforward to prove termination, e.g., using simple well-founded orderings [11]. It should also be pointed out that for confluence of terminating CHR programs a decidable sufficient and necessary condition has been proposed in [12,13].

Due to space limitation, we only illustrate the generation of simplification rules on an example. A description of the corresponding algorithm and formal considerations can be found in [4].

Example 9 Let S be the set of propagation rules corresponding to the min constraint of Example 1:

$$min(A,B,C) \quad \Rightarrow \quad C{\leq}A \ \wedge \ C{\leq}B. \tag{1}$$
$$min(A,A,C) \quad \Rightarrow \quad A{=}C. \tag{2}$$
$$min(A,B,C) \ \wedge \ C{\neq}B \quad \Rightarrow \quad C{=}A. \tag{3}$$
$$min(A,B,C) \ \wedge \ C{\neq}A \quad \Rightarrow \quad C{=}B. \tag{4}$$
$$min(A,B,C) \ \wedge \ B{\leq}A \quad \Rightarrow \quad C{=}B. \tag{5}$$
$$min(A,B,C) \ \wedge \ A{\leq}B \quad \Rightarrow \quad C{=}A. \tag{6}$$

Obviously, S is terminating, since only equality and less-or-equal constraints appear in the right hand side of the rules.
If we transform the rule (1) into a simplification rule of the form

$$min(A,B,C) \quad \Leftrightarrow \quad C{\leq}A \ \wedge \ C{\leq}B. \tag{7}$$

then the resulting set consisting of the rules (7), (2), ..., (6) becomes non-confluent. We can see for example that the conjunction $min(A,B,C) \ \wedge \ A{=}B$ can lead to two different rewritings. The first is $C{\leq}A \ \wedge \ C{\leq}B \ \wedge \ A{=}B$ using the simplification rule (7) and the second is $C{\leq}A \ \wedge \ C{\leq}B \ \wedge \ A{=}B \ \wedge \ A{=}C$ using first (2) and then (7). Thus, the propagation rule (1) cannot be replaced by the simplification rule (7).

The propagation rule (2) can be transformed into the simplification rule

$$min(A,A,C) \quad \Leftrightarrow \quad A{=}C. \tag{8}$$

and the resulting set consisting of the rules (1), (8), (3), ..., (6) remains terminating and confluent. Thus rule (2) is replaced by rule (8).

For the rules (3) and (4), there is no possibility to transform them into simplification rules preserving the confluence or the termination.

Now, we consider the propagation rule (5) and the possibilities to transform it into a simplification rule.

1. If we transform rule (5) into the simplification rule

$$min(A,B,C) \ \wedge \ B{\leq}A \quad \Leftrightarrow \quad C{=}B. \tag{9}$$

   then the resulting set consisting of the rules (1), (8), (3), (4), (9), (6) becomes non-confluent. For example, if we consider the conjunction $min(A,B,C) \ \wedge \ B{\leq}A \ \wedge \ min(A,B,D)$, it has two different rewritings: $min(A,B,D) \wedge C{=}B$ if we apply rule (9) on the subconjunction $min(A,B,C) \wedge B{\leq}A$ and also $min(A,B,C) \wedge D{=}B$ if we apply the same rule (9) on the subconjunction $min(A,B,D) \wedge B{\leq}A$. Thus, we do not replace rule (5) by rule (9).

2. The propagation rule (5) can also be transformed into the following simplification rule performing less simplification than using rule (9)

$$min(A,B,C) \ \wedge \ B{\leq}A \quad \Leftrightarrow \quad C{=}B \ \wedge \ min(A,B,C). \tag{10}$$

   but the resulting set of rules becomes also non-confluent.

3. Finally, we can also transform the propagation rule (5) into a simplification rule of the form

$$min(A,B,C) \ \wedge \ B{\leq}A \quad \Leftrightarrow \quad C{=}B \ \wedge \ B{\leq}A. \tag{11}$$

   which leads in this case to a terminating and confluent set of rules.

Thus, rule (5) is replaced by rule (11). Then we proceed with the last rule (6), which can be replaced by the following simplification rule (12) in a similar way to the transformation of rule (5).

$$min(A,B,C) \ \wedge \ A{\leq}B \quad \Leftrightarrow \quad C{=}A \ \wedge \ A{\leq}B. \tag{12}$$

The rules in the final set are (1), (8), (3), (4), (11), (12) as given in Example 1.

7. Conclusion and Related Work

We have presented an approach to generate rule-based constraint solvers from the intentional definition of the constraint predicates given by means of a CLP program. The generation is performed in two steps. In a first step, it produces propagation rules using the algorithm PROPMINER described in Section 3, and in a second step it transforms some of these rules into simplification rules using the method proposed in [4].

In previous work, first steps towards automatic generation of propagation rules have been done [1,2,3]. The practical interest of this computer-aided generation of

rule-based constraint solvers has been confirmed in [4], where it was shown on a practical application in digital circuit design that these solvers can be more efficient than solvers written by hand. In this paper, we have extended the work presented in [1,2,3] that where limited to extensionally defined constraints over finite domains.
Other work done in different contexts is also partially related to the underlying techniques used in this paper. Computing a kind of least upper bound of a set of computed answers to a goal in order to propagate information between resolution steps has been early explored in [14]. This framework called *Generalized Constraint Propagation* does not say much about how to compute the least upper bound and when to perform propagation steps. In contrast to our approach where the generation of rules is done once at compile time, the generalized propagation is performed only at runtime.
We used techniques (e.g., the computation of the lgg [8]) also used in the fields of Inductive Logic Programming (ILP) [15] and Inductive Constraint Logic Programming (ICLP) [16,17,18,19,20]. In ILP and ICLP the user is interested to find out logic programs and CLP programs from examples. In our case, we generate constraint solvers in the form of propagation rules, using the definition of the constraint predicates given by means of a CLP program. To our knowledge, the work done in ILP and ICLP have not previously been adapted or applied to the generation of rule-based constraint solvers and it seems important to consider this issue.
Additionally, the field of Constructive Disjunction [21,22] has produced techniques to extract common information from disjunctions of constraints over finite domains. In particular, these techniques take into account the semantics of the arithmetic operators, comparison predicates, and interval constraints during this extraction. It seems interesting to investigate how these approaches can be reused to enhance the generation of propagation rules.
Future work includes the extension of the algorithm PropMiner to generate more information to be propagated in the right hand side of the rules. In the current algorithm, the computation of the least upper bound of set of answers is based on [8] which does not rely on the semantics of the constraints in the answers. As illustrated in Section 3.3 and Section 4, the user can provide by hand propagation rules to take into account (partially) this semantics, but, as it has been pointed out to us, approaches like [23] can be used to embed this semantics in a more general way and directly in the computation of the least upper bound. Another complementary aspect that needs to be investigated is the completeness of the solvers generated. It is clear that in general this property cannot be guaranteed, but in some cases it may be possible to check it, or at least to characterize the kind of consistency the solver can ensure.

References

[1] K. Apt and E. Monfroy. Automatic generation of constraint propagation algorithms for small finite domains. In *5th International Conference on Principles and Practice of*

Constraint Programming, CP'99, LNCS 1713, pages 58–72. Springer-Verlag, 1999.

[2] C. Ringeissen and E. Monfroy. Generating propagation rules for finite domains: A mixed approach. In *New Trends in Constraints*, pages 150–172. LNAI 1865, 2000.

[3] S. Abdennadher and C. Rigotti. Automatic generation of propagation rules for finite domains. In *6th International Conference on Principles and Practice of Constraint Programming, CP'00*, LNCS 1894, pages 18–34. Springer-Verlag, September 2000.

[4] S. Abdennadher and C. Rigotti. Using confluence to generate rule-based constraint solvers. In *Third International Conference on Principles and Practice of Declarative Programming*. ACM Press, September 2001.

[5] T. Frühwirth. Theory and practice of constraint handling rules, special issue on constraint logic programming. *Journal of Logic Programming*, 37(1-3):95–138, October 1998.

[6] S. Abdennadher and C. Rigotti. Generation of propagation rules for intentionally defined constraints. In *The Thirteenth IEEE International Conference on Tools with Artificial Intelligence*. IEEE, 2001.

[7] J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19-20:503–581, 1994.

[8] Gordon Plotkin. A note on inductive generalization. In *Machine Intelligence*, volume 5, pages 153–163. Edinburgh University Press, 1970.

[9] Roberto J. Bayardo, Rakesh Agrawal, and Dimitrios Gunopulos. Constraint-based rule mining in large, dense databases. In *Proceedings of the 15th International Conference on Data Engineering*, pages 188–197. IEEE Computer Society, 1999.

[10] H. Tamaki and T. Sato. Old resolution with tabulation. In *3rd International Conference on Logic Programming*, LNCS 225, pages 84–98. Springer-Verlag, 1986.

[11] T. Frühwirth. Proving termination of constraint solver programs. In *New Trends in Constraints*, pages 298–317. LNAI 1865, 2000.

[12] S. Abdennadher. Operational semantics and confluence of constraint propagation rules. In *Third International Conference on Principles and Practice of Constraint Programming, CP'97*, LNCS 1330, pages 252–266. Springer-Verlag, November 1997.

[13] S. Abdennadher, T. Frühwirth, and H. Meuss. Confluence and semantics of constraint simplification rules. *Constraints Journal, Special Issue on the Second International Conference on Principles and Practice of Constraint Programming*, 4(2):133–165, May 1999.

[14] T. Le Provost and M. Wallace. Generalized constraint propagation over the CLP scheme. *Journal of Logic Programming*, 16(3):319–359, 1993.

[15] Stephen Muggleton and Luc De Raedt. Inductive Logic Programming : theory and methods. *Journal of Logic Programming*, 19,20:629–679, 1994.

[16] C. Page and A. Frisch. Generalizing atoms in constraint logic. In *Second international conference on knowledge representaion and reasoning (KR'91)*, pages 429–440. Morgan Kaufmann, 1991.

[17] T. Kawamura and K. Furukawa. Towards inductive generalization in constraint logic programs. In *Proceedings of the IJCAI-93 workshop on inductive logic programming*, 1993.

[18] F. Mizoguchi and H. Ohwada. Constrained relative least general generalization for inducing constraint logic programs. *New Generation Computing*, 13:335–368, 1995.

[19] L. Martin and C. Vrain. Induction of constraint logic programs. In *Proceedings of Algorithms and Learning Theory*, 1996.

[20] M. Sebag and C. Rouveirol. Constraint inductive logic programming. In *Advances in ILP*, pages 277–294. IOS Press, 1996.

[21] P. Van Hentenryck, V. Saraswat, and Y. Deville. Desing, implementation, and evalua-

tion of the constraint language cc(FD). *Journal of Logic Programming*, 37(1-3):139–164, 1998.

[22] Jörg Würtz and Tobias Müller. Constructive disjunction revisited. In *20th German Annual Conference on Artificial Intelligence*, LNAI 1137, pages 377–386. Springer-Verlag, 1996.

[23] C. Page and A. Frisch. Generalization and learnability: a study of constrained atoms. In *Inductive Logic Programming*, pages 29–61. London: Academic Press, 1992.