Automatic Generation of CHR Constraint Solvers

SLIM ABDENNADHER
Computer Science Department, University of Munich
Oettingenstr. 67, 80538 München, Germany
Slim.Abdennadher@informatik.uni-muenchen.de

CHRISTOPHE RIGOTTI
Laboratoire d'Ingénierie des Systèmes d'Information
Bâtiment 501, INSA Lyon, 69621 Villeurbanne Cedex, France
Christophe.Rigotti@insa-lyon.fr

Abstract

In this paper, we present a framework for automatic generation of CHR solvers given the logical specification of the constraints. This approach takes advantage of the power of tabled resolution for constraint logic programming, in order to check the validity of the rules. Compared to previous work (Apt & Monfroy, 1999; Ringeissen & Monfroy, 2000; Abdennadher & Rigotti, 2000; Abdennadher & Rigotti, 2001a), where different methods for automatic generation of constraint solvers have been proposed, our approach enables the generation of more expressive rules (even recursive and splitting rules) that can be used directly as CHR solvers.

1 Introduction

Constraint Handling Rules (CHR) (Frühwirth, 1998) is a high-level language especially designed for writing constraint solvers. CHR is essentially a committed-choice language consisting of multi-headed rules that transform constraints into simpler ones until they are solved. CHR defines both *simplification* of and *propagation* over user-defined constraints. Simplification replaces constraints by simpler constraints while preserving logical equivalence. Propagation adds new constraints, which are logically redundant but may cause further simplifications. Consider the constraint *min*, where $min(X, Y, Z)$ means that Z is the minimum of X and Y. Then typical CHR rules for this constraint are:

$$\begin{aligned} min(X,Y,Z),\ Y{\leq}X \quad &\Leftrightarrow \quad Z{=}Y,\ Y{\leq}X. \\ min(X,Y,Z),\ X{\leq}Y \quad &\Leftrightarrow \quad Z{=}X,\ X{\leq}Y. \\ min(X,Y,Z) \quad &\Rightarrow \quad Z{\leq}X,\ Z{\leq}Y. \end{aligned}$$

The first two rules are simplification rules, while the third one is a propagation rule. The first two rules correspond to the usual definition of *min*. The first rule states that $min(X, Y, Z),\ Y{\leq}X$ can be replaced by $Z{=}Y,\ Y{\leq}X$. The second has

an analogous reading, and the third rule states that if we have $min(X, Y, Z)$ then we can add $Z \leq X$, $Z \leq Y$ to the current constraints.
If such rules are in general easy to read, in many cases it remains a hard task to find them when one wants to write a constraint solver. Thus, several methods have been proposed to automatically generate rule-based solvers for constraints given their logical specification (Apt & Monfroy, 1999; Ringeissen & Monfroy, 2000; Abdennadher & Rigotti, 2000; Abdennadher & Rigotti, 2001a). These approaches can help to find more easily interesting rules, and it has also been shown in (Abdennadher & Rigotti, 2001b) that the rules produced automatically can lead to more efficient constraint reasoning than rules found by programmers.
In this paper, we propose a new method to generate CHR propagation and simplification rules. This work extends the previous rule-based solver generation techniques described in (Apt & Monfroy, 1999; Ringeissen & Monfroy, 2000; Abdennadher & Rigotti, 2000; Abdennadher & Rigotti, 2001a). It allows to obtain more general forms of rules, even for constraints defined intensionally over infinite domains.
The intuitive principle of the generation is the following. Consider that a solver S for some constraints, called *primitive constraints*, is already available. Other constraint predicates, called *user-defined constraints*, are given and their semantics is specified by mean of a constraint logic program P (i.e., the constraint predicates are defined by clauses of P). Then we want to obtain CHR propagation and simplification rules for the user-defined constraints to extend the existing solver. The basic idea of our approach relies on the following observation: a rule of the form $C \Rightarrow D$ is valid if the execution of the goal $C, \neg(D)$ finitely fails with program P and solver S. For the execution of such goals, we will use a tabled resolution for constraint logic programming (Codognet, 1995; Cui & Warren, 2000) that terminates more often than execution based on SLD-like resolutions.
We present three algorithms that can be integrated to build an environment to help developers to write CHR rule-based constraint solvers. Two of the algorithms focus on how to generate propagation rules for constraints given their logical specification. The first algorithm generates only primitive propagation rules (i.e., rules with right hand side consisting of primitive constraints). The second algorithm extends the first one to generate more general propagation rules with right hand side consisting of both primitive and user-defined constraints. We also show that a slight modification of this algorithm allows to generate the so-called splitting rules (rule having a disjunction in their right hand side) supported by the extension of CHR called $\text{CHR}^{\vee}$ (Abdennadher & Schütz, 1998). The third algorithm focuses on transforming propagation rules into simplification rules to improve the time and space behavior of constraint solving. This article is an extended and revised version of the paper (Abdennadher & Rigotti, 2002).

Organization of the paper. In Section 2, we present an algorithm to generate primitive propagation rules. In Section 3, we describe how to modify the algorithm to generate more general propagation rules. Section 4 presents a transformation method of propagation rules into simplification rules. For clarity reasons, we will use an abstract representation for the generated rules. In Section 5, we present how these rules can be encoded in CHR. We discuss related work in Section 6, and finally we conclude with a summary and possibilities of further improvements.

2 Generation of Primitive Propagation Rules

We assume some familiarity with constraint logic programming (CLP) (Jaffar & Maher, 1994; Marriott & Stuckey, 1998). We consider two classes of constraints, *primitive constraints* and *user-defined constraints*. Primitive constraints are those constraints defined by a constraint theory CT and for which solvers are already available. User-defined constraints are those constraints defined by a constraint logic program P and for which we want to generate solvers. We assume that the set of primitive constraints is closed under negation, in the sense that the negation of each primitive constraint must be also a primitive constraint, e.g. $=$ and $\neq$ or $\leq$ and $>$. In the following, we denote the negation of a primitive constraint c by $not(c)$.
In the rest of this paper, we use the following terminology.

Definition 2.1
A *constraint logic program* is a set of clauses of the form

$$h \leftarrow b_1, \ldots, b_n, c_1, \ldots, c_m$$

where $h, b_1, \ldots, b_n$ are user-defined constraints and $c_1, \ldots, c_m$ are primitive constraints. h is called left hand side of the clause. A *goal* is a set of primitive and user-defined constraints. An *answer* is a set of primitive constraints. The logical semantics of a constraint logic program P is its Clark's completion and is denoted by P^*. A user-defined constraint is *defined* in a constraint logic program if it occurs in the left hand side of a clause.

Definition 2.2
A *primitive propagation rule* is a rule of the form $C_1 \Rightarrow C_2$ or of the form $C_1 \Rightarrow false$, where C_1 is a set of primitive and user-defined constraints, while C_2 consists only of primitive constraints. C_1 is called the *left hand side* of the rule (*lhs*) and C_2 its *right hand side* (*rhs*). A rule of the form $C_1 \Rightarrow$ *false* is called *failure rule*.

In the following we use the notation $\tilde{\exists}(\phi)$ to denote the existential closure of ϕ and $\exists_{-\mathcal{V}}(\phi)$ to denote the existential closure of ϕ except for the variables in the set $\mathcal{V}$.

Definition 2.3
A primitive propagation rule $\{d_1, \ldots, d_n\} \Rightarrow \{c_1, \ldots, c_m\}$ is *valid* with respect to the constraint theory CT and the program P if and only if $P^*, CT \models \bigwedge_i d_i \rightarrow \exists_{-\mathcal{V}}(\bigwedge_j c_j)$, where $\mathcal{V}$ is the set of variables appearing in $\{d_1, \ldots, d_n\}$. A failure rule $\{d_1, \ldots, d_n\} \Rightarrow$ *false* is *valid* with respect to CT and P if and only if $P^*, CT \models \neg\tilde{\exists}(\bigwedge_i d_i)$.

We now give an algorithm to generate valid primitive propagation rules.

2.1 *The* PRIM-MINER *Algorithm*

The PRIM-MINER algorithm takes as input the program P defining the user-defined constraints. To specify the syntactic form of the rules, the algorithm needs also as input two sets of primitive and user-defined constraints denoted by $Base_{lhs}$ and $Cand_{lhs}$, and a set containing only primitive constraints denoted by $Cand_{rhs}$. The constraints occurring in $Base_{lhs}$ are the common part that must appear in the lhs of all rules, $Cand_{lhs}$ indicates candidate constraints used in conjunction with $Base_{lhs}$ to form the lhs, and $Cand_{rhs}$ are the candidate constraints that may appear in the rhs.

Note that a syntactic analysis of the constraint logic program P can suggest functors and constraint predicates to be used to form candidate constraints.

The algorithm PRIM-MINER is presented in Figure 2.1 and generates a set of valid rules.

The basic idea of the algorithm relies on the following observation: to be able to generate a failure rule of the form $C \Rightarrow false$, we can simply check that the execution of the goal C finitely fails. Furthermore, while these rules are useful to detect inconsistencies, it is in general more interesting to propagate earlier some information that can be used for constraint solving, instead of waiting until a conjunction of constraints becomes inconsistent. Thus, for each possible lhs C (i.e., each subset of $Base_{lhs} \cup Cand_{lhs}$) the algorithm distinguishes two cases:

1. PRIM-MINER uses a CLP system to evaluate the goal C. If the goal finitely fails, then the failure rule $C \Rightarrow false$ is generated.
2. Otherwise the negation of each candidate constraint d from $Cand_{rhs}$ is added in turn to C and the goal $C \cup \{not(d)\})$ is evaluated. If the goal finitely fails, then the rule $C \Rightarrow \{d\}$ is generated.

In practice these goal evaluations are made using a bounded depth resolution procedure to avoid non-termination of the whole generation algorithm.

The algorithm PRIM-MINER uses a basic ordering to prune the search space and to avoid the generation of many uninteresting rules. This pruning relies simply on the following observation. If $C_1 \Rightarrow false$ is valid, then rules of the form $C_2 \Rightarrow false$, where $C_1 \subset C_2$ are also valid but useless. So the algorithm considers first the smallest lhs with respect to set inclusion, and when it finds a valid failure rule $C_1 \Rightarrow false$ it discards from the lhs candidates any C_2 that is superset of C_1.

At first glance, the procedure used to evaluate the goals issued by the algorithm may be considered as a classical depth-first, left-to-right CLP resolution. However, we will show in Section 2.2 and Section 3 that a tabled CLP resolution extends greatly the class of rules that can be generated, by allowing termination of the evaluation in many interesting cases. Additionally, it should be noticed that the execution on the underlying CLP system is not required to enumerate all answers since PRIM-MINER only performs a fail/succeed test, and thus the CLP system can stop after a first answer has been found.

begin

$\mathcal{R}$ the resulting rule set is initialized to the empty set.
L is a list of all subsets of $Cand_{lhs}$,
in an order 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**
  Remove from L its first element denoted C_{lhs}.
  if the goal $(Base_{lhs} \cup C_{lhs})$ fails
    with respect to the constraint logic program P **then**
      add the failure rule $(Base_{lhs} \cup C_{lhs} \Rightarrow false)$ to $\mathcal{R}$
      and remove from L all supersets of C_{lhs}.
  else
      Let rhs be initialized to the empty set.
      for all $d \in Cand_{rhs}$
        if the goal $(Base_{lhs} \cup C_{lhs} \cup \{not(d)\})$ fails
          with respect to the constraint logic program P **then**
            add d to the set rhs.
        endif
      endfor
      if rhs is not empty **then** add the rule $(Base_{lhs} \cup C_{lhs} \Rightarrow rhs)$ to $\mathcal{R}$ **endif**
  endif
endwhile

output $\mathcal{R}$.

end

Fig. 1. The PRIM-MINER Algorithm

Example 2.1

Consider the following constraint logic program which implements the predicate *min*. $min(X,Y,Z)$ means that Z is the minimum of X and Y:

$$\begin{array}{lcl} min(X,Y,Z) & \leftarrow & X{\leq}Y, \ \ Z{=}X. \\ min(X,Y,Z) & \leftarrow & Y{\leq}X, \ \ Z{=}Y. \end{array}$$

where $\leq$ and $=$ are primitive constraints with the usual meaning as total order and syntactic equality.
The algorithm with the appropriate input generates (among others) the rule

$$min(X,Y,Z),\ Y{\leq}X \quad \Rightarrow \quad Z{=}Y.$$

after having checked that the execution of the goal $min(X,Y,Z),\ Y{\leq}X,\ Z{\neq}Y$ fails by constructing the following derivation tree:

$$
\begin{tikzcd}[column sep=small]
& \mathit{min}(X,Y,Z),\ Y{\le}X,\ Z{\neq}Y \arrow[dl] \arrow[dr] & \\
X{\le}Y,\ Z{=}X,\ Y{\le}X,\ Z{\neq}Y \arrow[d] & & Y{\le}X,\ \ Z{=}Y,\ Y{\le}X,\ Z{\neq}Y \arrow[d] \\
\mathit{false} & & \mathit{false}
\end{tikzcd}
$$

Note that we assume that the constraint solver for $\leq$ and $=$ is able to detect such inconsistencies.

Soundness and Completeness. The PRIM-MINER algorithm attempts to extract all valid primitive propagation rules of the form $C_1 \Rightarrow \{d\}$ or $C_1 \Rightarrow \mathit{false}$ such that $Base_{lhs} \subseteq C_1$, $C_1 \setminus Base_{lhs} \subseteq Cand_{lhs}$, $d \in Cand_{rhs}$, and there is no other more general failure rule (i.e., no valid rule $C_2 \Rightarrow \mathit{false}$ where $C_2 \subset C_1$). In general, the algorithm cannot be complete, since the evaluation of some goals corresponding to valid rules may be non-terminating. In fact, this completeness can be achieved if more restricted classes of constraint logic programs are used to give the semantics of user-defined constraints and if the solver for the primitive constraints used by the underlying CLP system is satisfaction complete.
The soundness of the algorithm (i.e., only valid rules are generated) is guaranteed by the nice properties of standard CLP schemes (Jaffar & Maher, 1994) and tabled CLP schemes (Codognet, 1995). An important practical aspect, is that even a partial resolution procedure (e.g., bounded depth evaluation) or the use of an incomplete solver by the CLP system, does not compromise the validity of the rules generated.

2.2 Advantage of Tabled Resolution for Rule Generation

Termination of the evaluation of (constraint) logic programs has received a lot of attention. A very powerful and elegant approach based on tabled resolution has been developed, first for logic programming (e.g., (Tamaki & Sato, 1986; Warren, 1992)) and further extended in the context of CLP (e.g., (Codognet, 1995; Cui & Warren, 2000)).
The intuitive basic principle of tabled resolution is the following. Each new subgoal S is compared to the previous intermediate subgoals (not necessarily in the same branch of the resolution tree). If there is a previous subgoal I which is equivalent to S or more general than S, then no more unfolding is performed on S and answers for S are selected among the answers of I. This process is repeated for all subsequent computed answers that correspond to the subgoal I.
The use of such technique has not been widely accepted since if this leads to termination in many more cases than execution based on SLD-resolution, this should be paid by some execution overhead in general. When using the algorithm PRIM-MINER

we can accept a slight decrease of performance (since the solver is constructed once) if this gives rise to an improvement of the termination capability which enables the generation of additional rules. Thus, tabled CLP can find very interesting applications in constraint solver synthesis. This will be illustrated in the following example.

Example 2.2
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.

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

Let the input parameters of PRIM-MINER be

$$\begin{array}{rcl} Base_{lhs} & = & \{append(X,Y,Z)\} \\ Cand_{lhs} & = & \{X{=}[],Y{=}[],Z{=}[],X{=}Y,X{=}Z,Y{=}Z,X{\neq}Y,X{\neq}Z,Y{\neq}Z\} \\ Cand_{rhs} & = & Cand_{lhs}. \end{array}$$

Then the algorithm generates (among others) the following primitive propagation rule:

$$append(X,Y,Z),\ Y{=}[] \quad \Rightarrow \quad X{=}Z$$

by executing the goal $append(X,Y,Z),\ Y{=}[],\ X{\neq}Z$ with a tabled CLP resolution. For a classical CLP scheme the resolution tree will be infinite, while in case of a tabled resolution it can be sketched as follows:

$$\begin{tikzcd}[column sep=small] & append(X,Y,Z),Y{=}[],X{\neq}Z \arrow[dl] \arrow[dr] & \\ X = [],Y{=}Z,Y{=}[],X{\neq}Z \arrow[d] & & X{=}[H|X1],Z{=}[H|Z1],append(X1,Y,Z1),Y{=}[],X{\neq}Z \arrow[d] \\ false & & false \end{tikzcd}$$

The initial goal $G_1 = (append(X,Y,Z),\ Y{=}[],\ X{\neq}Z)$ is more general than the sub-goal $G_2 = (X{=}[H|X1],\ Z{=}[H|Z1],\ append(X1,Y,Z1),\ Y{=}[],\ X{\neq}Z)$, in the sense that $append(A,B,C),\ D{=}[E|A],\ F{=}[E|C],\ B{=}[],\ D{\neq}F$ entails $append(A,B,C),\ B{=}[],\ A{\neq}C$. So no unfolding is made on G_2, and the process waits for answers of G_1 to compute answers of G_2. Since G_1 has no further possibility of having answers, then G_2 fails and thus G_1 also fails. We refer the reader to (Codognet, 1995; Cui & Warren, 2000) for a more detailed presentation of goal evaluation using a tabled CLP resolution.
Using this kind of resolution PRIM-MINER is also able to produce rules such as:

$$\begin{array}{rcl} append(X,Y,Z),\ X{=}Z & \Rightarrow & Y{=}[]. \\ append(X,Y,Z),\ Y{\neq}[] & \Rightarrow & X{\neq}Z. \\ append(X,Y,Z),\ X{\neq}[] & \Rightarrow & Z{\neq}[]. \end{array}$$

We have run all examples presented in this paper, using our own implementation of tabled CLP resolution according to the description of (Cui & Warren, 2000).

2.3 Selection of Interesting Rules and Performance of Generation

Interesting rules. If we use the algorithm PRIM-MINER as presented in Figure 2.1, we obtain a set of rules that is highly redundant and thus not suitable neither for human-reading nor for its direct use as an executable constraint solver. This problem encountered in propagation rule generation has already been pointed out in (Abdennadher & Rigotti, 2000). It is taken into account by applying the following simplification technique on the set of rules produced by PRIM-MINER:

- The rules generated by PRIM-MINER are ordered in a list L using any total ordering on the rule lhs compatible with the θ-subsumption ordering (Plotkin, 1970) (i.e., a rule having a more general lhs is placed before a rule with a more specialized lhs).
- Let S be a set of rules initialized to the empty set. For each rule $C_1 \Rightarrow C_2$ in L (taken according to the list ordering) the constraint C_2 is simplified to an equivalent constraint C_{simp} by the already known solver for $Cand_{rhs}$ and by the rules in S. If C_{simp} is empty then the rule can be discarded, else add the rule $C_1 \Rightarrow C_{simp}$ to S.
- Output the set S containing the simplified set of rules.

For example, using this process the set of rules

$$\{p(X) \Rightarrow r(X), \quad p(X),\ q(X) \Rightarrow r(X), \quad s(X,Y) \Rightarrow X{=}Y,\ X{=}a,\ Y{=}a\}$$

is simplified to

$$\{p(X) \Rightarrow r(X), \quad s(X,Y) \Rightarrow X{=}a,\ Y{=}a\}.$$

For clarity reasons this simplification step is presented apart from the algorithm PRIM-MINER, but it can be incorporated in PRIM-MINER and performed during the generation of the rules. So, if the algorithm PRIM-MINER is implemented on a flexible platform (e.g., SICStus Prolog with CHR support), when a valid rule is extracted it can be immediately simplified with respect to the already generated rules. Then, if it is not redundant it can be incorporated at runtime and thus be used actively to speed up further resolution and constraint solving steps called by the generation algorithm itself.

Other performance issues. It is likely that the same constraints and their negations are candidates for both the lhs and rhs of the rules. In this case the two following optimizations can be used:

1. If a rule $C_1 \Rightarrow \{d\}$ is generated and d is also in $Cand_{lhs}$ then there is no need to consider any C_2 such that C_2 is a superset of C_1 containing d to form the lhs of another rule.
2. If a rule $C \cup \{d_1\} \Rightarrow \{d_2\}$ is produced using the evaluation of the goal $C \cup \{d_1,\ not(d_2)\}$ (which fails), and if $not(d_2) \in Cand_{lhs}$ and $not(d_1) \in$

$Cand_{rhs}$ then the same goal evaluation is also needed to produce the rule $C \cup \{not(d_2)\} \Rightarrow \{not(d_1)\}$. Thus, we can avoid for such constraints this kind of repeated goal evaluations.
For example, the execution of the goal $min(X,Y,Z),\ Y{\leq}X,\ Z{\neq}Y$ may lead to the generation of the following rules:

$$\begin{array}{rcl} min(X,Y,Z),\ Y{\leq}X & \Rightarrow & Z{=}Y. \\ min(X,Y,Z),\ Z{\neq}Y & \Rightarrow & Y{>}X. \end{array}$$

2.4 Generation of Primitive Splitting Rules

Splitting rules have been shown to be interesting in constraint solving (Apt, 1998), since they can be used to detect early alternative labeling cases or alternative solution sets. These rules are handled by CHR$^{\vee}$ an extension of CHR proposed in (Abdennadher & Schütz, 1998). Such rules that can be generated by the extension proposed in this section are for example:

$$\begin{array}{rcl} and(X,Y,Z),\quad Z{=}0 & \Rightarrow & X{=}0 \ \vee \ Y{=}0. \\ min(X,Y,Z) & \Rightarrow & X{=}Z \ \vee \ Y{=}Z. \end{array}$$

where $and(X,Y,Z)$ means that Z is the Boolean conjunction of X and Y, defined by the facts $and(0,0,0), and(1,0,0), and(0,1,0), and(1,1,1)$; and where $min(X,Y,Z)$ is defined by the constraint logic program of Example 2.1.
In the following, we restrict ourself to the generation of *primitive splitting rules*.

Definition 2.4
A *primitive splitting rule* is a rule of the form $C \Rightarrow d_1 \vee d_2$, where d_1, d_2 are primitive constraints and C is a set of primitive and user-defined constraints. $\vee$ is interpreted like the standard disjunction, and primitive splitting rules have a straightforward associated semantics.

The modification of the basic PRIM-MINER algorithm is as follows. For all C_{lhs}, it must also consider each different set $\{d_1, d_2\} \subseteq C_{rhs}$ with $d_1 \neq d_2$ and check if the goal $(Base_{lhs} \cup C_{lhs} \cup \{not(d_1), not(d_2)\})$ fails with respect to the constraint logic program P. If this is the case, it simply adds to $\mathcal{R}$ the primitive splitting rule $(Base_{lhs} \cup C_{lhs} \Rightarrow d_1 \vee d_2)$.
For example, the rule $min(X,Y,Z) \Rightarrow X{=}Z \ \vee \ Y{=}Z$ can be obtained by issuing the goal $min(X,Y,Z),\ X{\neq}Z,\ Y{\neq}Z$ and checking that its execution finitely fails.
Here again, the soundness of the generation relies on the properties of the underlying resolution used.
In practice, a huge number of splitting rules are not interesting because they are redundant with respect to some primitive propagation rules. So, to remove these uninteresting rules, the generation should preferably be done, by first using PRIM-MINER to obtain only primitive propagation rules, and then using the modified version of the algorithm to extract primitive spitting rules. Thus in this second step, redundant rules, with respect to the set resulting from the first step, can be discarded on-the-fly. For example the rule $a, b, c \ \Rightarrow \ d \vee e$ will be removed if we

have already generated the rule $a, b \Rightarrow d$. Moreover, in such trivial redundancy cases, the test of the validity of the rule $a, b, c \Rightarrow d \vee e$ can itself be avoided, and more generally the test of any rule of the form $C \Rightarrow d \vee f$ and $C \Rightarrow f \vee d$, where $\{a, b\} \subseteq C$ and $f \in Cand_{rhs}$.

3 Generation of More General Propagation Rules

In this section, we modify the algorithm presented in Section 2 to handle a broader class of rules called *general propagation rules*, that encompasses the primitive propagation rules, and that can even represent recursive rules over user-defined constraints (i.e., rules where the same user-defined constraint predicate appears in the lhs and rhs).

Definition 3.1
A *general propagation rule* is a failure rule or a rule of the form $C_1 \Rightarrow C_2$, where C_1 and C_2 are sets of primitive and user-defined constraints.

The notion of validity defined for primitive propagation rules also applies to this kind of rules.
In the PRIM-MINER algorithm, the validity test of a primitive propagation rule $Base_{lhs} \cup C_{lhs} \Rightarrow \{d\}$ is performed by checking that the goal $Base_{lhs} \cup C_{lhs} \cup \{not(d)\}$ fails. For general propagation rules, d is no longer a primitive constraint but may be defined by a constraint logic program. In this case, the evaluation should be done using a more general resolution procedure to handle negated subgoals. However, to avoid the well known problems related to the presence of negation we can simply use a different validity test based on the following theorem.

Theorem 3.1
Let $C_1 \Rightarrow C_2$ be a general propagation rule and $\mathcal{V}$ be the variables occurring in C_1. Let S_1 be the set of answers $\{a_1, \ldots, a_n\}$ to the goal C_1, and S_2 be the set of answers $\{b_1, \ldots, b_m\}$ to the goal $C_1 \cup C_2$. Then the rule $C_1 \Rightarrow C_2$ is valid if $P^*, CT \models \neg(\exists_{-\mathcal{V}}((a_1 \vee \ldots \vee a_n) \wedge \neg\exists_{-\mathcal{V}}(b_1 \vee \ldots \vee b_m)))$.

This straightforward property comes from the completeness result of standard CLP schemes (Jaffar & Maher, 1994) which ensures that if a goal G has a finite computation tree, with answers $c_1, \ldots, c_n$ then $P^*, CT \models G \leftrightarrow \exists_{-\mathcal{V}_g}(c_1 \vee \ldots \vee c_n)$, where $\mathcal{V}_g$ is the set of variables appearing in G.
So, the modification proposed in this section consists simply in the replacement in algorithm PRIM-MINER of the call to the goal $Base_{lhs} \cup C_{lhs} \cup \{not(d)\}$ to check the validity of the rule $Base_{lhs} \cup C_{lhs} \Rightarrow \{d\}$, by the following steps.

- First, collect the set of answers $\{a_1, \ldots, a_n\}$ to the goal $Base_{lhs} \cup C_{lhs}$.
- Then, collect the set of answers $\{b_1, \ldots, b_m\}$ to the goal $Base_{lhs} \cup C_{lhs} \cup \{d\}$.
- In each answer a_i (resp. b_i), rename with a fresh variable any variable that is not in $Base_{lhs} \cup C_{lhs}$ (resp. $Base_{lhs} \cup C_{lhs} \cup \{d\}$).
- Finally, perform a satisfiability test of $(a_1 \vee \ldots \vee a_n) \wedge \neg(b_1 \vee \ldots \vee b_m)$.
- If this test fails then the rule $Base_{lhs} \cup C_{lhs} \Rightarrow \{d\}$ is valid.

Since answers only contain primitive constraints and since the set of primitive constraints is closed under negation, then we can perform the satisfiability test by rewriting $(a_1 \vee \ldots \vee a_n) \wedge \neg(b_1 \vee \ldots \vee b_m)$ into an equivalent disjunctive normal form, and then use the solver for primitive constraints on each sub-conjunctions.
It should be noticed that in cases where the evaluation of one of the two goals does not terminate before the bound of resolution depth is reached then the propagation rule $Base_{lhs} \cup C_{lhs} \Rightarrow \{d\}$ is not considered as valid and the next rule is processed.

Example 3.1
Consider the following user-defined Boolean constraints: $neg(X,Y)$ imposing that Y is the Boolean complement of X and $xor(X,Y,Z)$ stating that Z is the result of the exclusive Boolean disjunction of X and Y. These two constraints are defined by a straightforward constraint logic program. Among others, the modified algorithm presented above can generate the following rules:

$$
\begin{array}{rcl}
xor(X,Y,Z),\ Z{=}1 & \Rightarrow & neg(X,Y). \\
xor(X,Y,Z),\ Y{=}1 & \Rightarrow & neg(X,Z). \\
xor(X,Y,Z),\ X{=}1 & \Rightarrow & neg(Y,Z).
\end{array}
$$

For example, to test the validity of the first rule, the process is the following. First, the answers $A_1 := X{=}1 \wedge Y{=}0 \wedge Z{=}1$ and $A_2 := X{=}0 \wedge Y{=}1 \wedge Z{=}1$ to the goal $xor(X,Y,Z),\ Z{=}1$ are computed. Then for the goal $xor(X,Y,Z),\ Z{=}1,\ neg(X,Y)$ the same answers are collected. Finally the satisfiability test of $(A_1 \vee A_2) \wedge \neg(A_1 \vee A_2)$ fails and thus establishes the validity of the rule.
One other general rule that can be generated using the modified algorithm presented is for example:

$$
and(X,Y,Z) \quad \Rightarrow \quad min(X,Y,Z).
$$

Furthermore, rules representing symmetries can be automatically detected using the modified algorithm. For example, the rules

$$
\begin{array}{rcl}
min(X,Y,Z) & \Rightarrow & min(Y,X,Z). \\
xor(X,Y,Z) & \Rightarrow & xor(Y,X,Z).
\end{array}
$$

expressing the symmetry of the *min* and the *xor* constraints with respect to the first and second arguments can be generated. In (Abdennadher & Rigotti, 2001a), it has been shown that these rules are very useful to reduce the size of a set of propagation rules since many rules become redundant when we know such symmetries.

However, it must be pointed out that if the test presented in this section allows to handle a syntactically wider class of rules, it relies on different goal calls than the test of Section 2.1. So when testing the validity of a primitive propagation rule one of the techniques may lead to terminating evaluation while the other one may not. Thus in the case of primitive propagation rule it is preferable not to replace one test by the other, but to use both in a complementary way (run one of them, and if it reaches the bound of resolution depth then apply the other).

4 Generation of Simplification Rules

Since a propagation rule does not rewrite constraints but adds new ones, the constraint store may contain superfluous information. Constraints can be removed from the constraint store using *simplification rules.*

Definition 4.1
A *simplification rule* is a rule of the form $C_1 \Leftrightarrow C_2$, where C_1 and C_2 are sets of primitive and user-defined constraints.

In this section, we show how the rule validity test used in Section 3 can be applied to transform some propagation rules into simplification rules. For a valid propagation rule of the form $C \Rightarrow D$, we try to find a proper subset E of C such that $D \cup E \Rightarrow C$ is valid too. If such E can be found, the propagation rule $C \Rightarrow D$ can be transformed into a simplification rule of the form $C \Leftrightarrow D \cup E$.
To simplify the presentation, we present the algorithm to transform (when possible) propagation rules into simplification rules independently from the algorithm presented in Section 3. Note that the algorithm for the generation of propagation rules can be slightly modified to incorporate this step and to directly generate simplification rules.
The algorithm is given in Figure 2 and takes as input the set of generated propagation rules and the common part that must appear in the lhs of all rules, i.e. $Base_{lhs}$.

---

begin

    $P' := P$
    for each propagation rule of the form $C \Rightarrow D$ in P **do**
        Find a proper subset E of C such that $Base_{lhs} \not\subseteq E$ and
        $D \cup E \Rightarrow C$ is valid (using the validity test of Section 3)
        If E exists **then**
            $P' := (P' \backslash \{C \Rightarrow D\}) \cup \{C \Leftrightarrow D \cup E\}$
        endif
    endfor

    output P'
end

Fig. 2. The Transformation Algorithm

To achieve a form of minimality based on the number of constraints, we generate simplification rules that will remove the greatest number of constraints. So, when we try to transform a propagation rule into a simplification rule of the form $C \Leftrightarrow D \cup E$ we choose the smallest set E (with respect to the number of atomic constraints in E) for which the condition holds. If such a E is not unique, we choose any one among the smallest. The condition $Base_{lhs} \not\subseteq E$ is needed to be able to transform the propagation rules into simplification rules that rewrite constraints to *simpler* ones (primitive constraints if possible), as shown in the following example.

Example 4.1
Consider the following propagation rule R generated for the *append* constraint with $Base_{lhs} = \{append(X,Y,Z)\}$:

$$append(X,Y,Z),\ X{=}[] \quad \Rightarrow \quad Y{=}Z.$$

The algorithm tries the following transformations:

1. First, it checks if rule R can be transformed into the simplification rule $append(X,Y,Z),\ X{=}[] \Leftrightarrow Y{=}Z$. This is done by testing whether the rule $Y{=}Z \Rightarrow append(X,Y,Z),\ X{=}[]$ is valid. But, this is not the case and thus the transformation is not possible.
2. Next, the algorithm finds out that the rule $Y{=}Z,\ X{=}[] \Rightarrow append(X,Y,Z)$ is a valid rule and then the propagation rule R is transformed into the simplification rule $append(X,Y,Z),\ X{=}[] \Leftrightarrow X{=}[],\ Y{=}Z$.

Note that the propagation rule $Y{=}Z,\ append(X,Y,Z) \Rightarrow X{=}[]$ is also valid but transforming rule R into

$$append(X,Y,Z),\ X{=}[] \quad \Leftrightarrow \quad append(X,Y,Z),\ Y{=}Z.$$

will lead to a simplification rule which is uninteresting for constraint solving, i.e. using this rule the *append* constraint cannot be simplified and remains in the constraint store. The algorithm disables such transformation by checking the condition that all constraints of $Base_{lhs}$ are not shifted to the right hand side of the rule ($Base_{lhs} \not\subseteq E$).

5 Implementation of the Generated Rules in CHR

The generated rules may contain constraints that are built-in constraints for the CHR system. To have a running CHR solver, these constraints have to be encoded in a specific way. First, equality constraints appearing in the left hand side of a rule are propagated all over the constraints in its left and right hand side. Then the resulting constraints are simplified. This can be performed as follows. In turn each equality constraint appearing in the lhs is removed and transformed in a substitution that is applied to the lhs and the rhs. Then the next equality constraint is processed. For example, the simplification rule $and(X,Y,Z),\ Z{=}1 \Leftrightarrow X{=}1,\ Y{=}1,\ Z{=}1$ will be transformed into $and(X,Y,1) \Leftrightarrow X{=}1,\ Y{=}1$. Secondly, for other built-in constraints the transformation leads to CHR rules consisting of a guard (Frühwirth, 1998). For example, if $\leq$ is a built-in constraint of the CHR system then the rule $min(X,Y,Z),\ X{\leq}Y \ \Leftrightarrow\ Z{=}X,\ X{\leq}Y$ is transformed into the guarded CHR rule $min(X,Y,Z) \ \Leftrightarrow\ X{\leq}Y \mid Z{=}X$. Note that guard constraints containing only variables which appear in the left hand side of a rule can be removed from its right hand side.

6 Related Work

In (Abdennadher & Rigotti, 2001a), a method has been proposed to generate propagation rules from the intentional definition of the constraint predicates (eventually over infinite domains) given by mean of a constraint logic program. It extended previous work (Apt & Monfroy, 1999; Ringeissen & Monfroy, 2000; Abdennadher & Rigotti, 2000) where different methods dedicated to the generation of propagation rules for constraints defined extensionally over finite domains have been proposed. Compared to the work described in (Abdennadher & Rigotti, 2001a) the approach presented in this paper has several advantages:

- It enables user-defined constraints to occur in the right hand side of rules, while this is not handled by (Abdennadher & Rigotti, 2001a). As a by-product rules representing symmetries as

$$min(X, Y, Z) \quad \Rightarrow \quad min(Y, X, Z).$$

  can then be automatically detected.
- It allows the generation of splitting rules like

$$append(X, Y, Z),\ Z{=}[A] \quad \Rightarrow \quad X{=}[A] \ \vee \ Y{=}[A]$$

  supported by the extension of CHR called $\mathrm{CHR}^{\vee}$ (Abdennadher & Schütz, 1998). These rules have been shown to be interesting in constraint solving (Apt, 1998), but have not been considered in (Apt & Monfroy, 1999; Ringeissen & Monfroy, 2000; Abdennadher & Rigotti, 2000; Abdennadher & Rigotti, 2001a).
- Even if we restrict our attention to the class of rules handled in (Abdennadher & Rigotti, 2001a), the approach presented in this paper leads to a more expressive set of rules. For example, the rule

$$append(X, Y, Z),\ Y{=}[] \quad \Rightarrow \quad X{=}Z.$$

  cannot be generated by the approach proposed in (Abdennadher & Rigotti, 2001a) while the algorithm described in Section 2 is able to obtain it by executing the goal $append(X, Y, Z),\ Y{=}[], X{\neq}Z$ with a tabled resolution for CLP.
- Additionally, it needs less information about the semantics of the primitive constraints. For example it generates the rule

$$min(X, Y, Z) \quad \Rightarrow \quad Z{\leq}X,\ Z{\leq}Y.$$

  simply by calling the goals $min(X, Y, Z),\ Z{>}X$ and $min(X, Y, Z),\ Z{>}Y$, and checking that their executions fail. While in this case, the algorithm presented in (Abdennadher & Rigotti, 2001a) requires some extra information concerning the semantics of $\leq$ (information that would be provided in general by the user).

In this paper, we also described a method to transform propagation rules into simplification rules. It has been shown in (Abdennadher & Rigotti, 2001b) that this transformation has a very important impact on the efficiency of the solver produced. In (Abdennadher & Rigotti, 2001b) the propagation rules are modified to obtain (when possible) simplification rules using a technique based on a confluence notion. This is a syntactical criterion that works when we have at hand the whole set of rules defining the constraint. Thus it cannot be applied safely if only a part of the propagation rules have been generated. It also requires a termination test for rule-based programs consisting of propagation and simplification rules, and this test is undecidable for some classes of programs. The new transformation method presented in this paper avoids these two restrictions.

The generation of propagation and simplification rules is also related in some aspects to *Generalized Constraint Propagation* (Le Provost & Wallace, 1993), *Constructive Disjunction* (Van Hentenryck *et al.*, 1998; Würtz & Müller, 1996), and *Inductive Logic Programming* (Muggleton & De Raedt, 1994) as briefly discussed in (Abdennadher & Rigotti, 2001a). However, it should be pointed out that to our knowledge these works have not been used for the generation of constraint solvers.

7 Conclusion and Future Work

In this paper, we have presented three algorithms that can be integrated to build an environment to help solver developers when writing CHR programs.
The approach described allows the generation of CHR propagation and simplification rules. It can be applied on constraints defined over finite and infinite domains by mean of a constraint logic program. Moreover, it enables the developer to search for rules having such user-defined constraints in both their left and right hand sides.
We have also shown that compared to the algorithms described in (Apt & Monfroy, 1999; Ringeissen & Monfroy, 2000; Abdennadher & Rigotti, 2000; Abdennadher & Rigotti, 2001a) to generate rule-based constraint solvers, this approach is able to generate more expressive rules (including recursive and splitting rules).
One interesting direction for future work is to investigate the integration of constructive negation (e.g., (Przymusinski, 1990; Stuckey, 1995)) in tabled resolution for CLP to generate constraint solvers, in order to check the validity of the propagation and simplification rules in more general cases. Another complementary aspect is the completeness of the solvers generated. It is clear that in general this property cannot be guaranteed, but in some cases it should be possible to check it, or at least to characterize the kind of consistency the solver can ensure.

References

Abdennadher, S., & Rigotti, C. (2000). Automatic generation of propagation rules for finite domains. *6th international conference on principles and practice of constraint programming, CP'00*. LNCS 1894. Springer-Verlag.

Abdennadher, S., & Rigotti, C. (2001a). Towards inductive constraint solving. *7th international conference on principles and practice of constraint programming, CP'01*. LNCS 2239. Springer-Verlag.

Abdennadher, S., & Rigotti, C. (2001b). Using confluence to generate rule-based constraint solvers. *Third international conference on principles and practice of declarative programming*. ACM Press.

Abdennadher, S., & Rigotti, C. (2002). Constraint solver synthesis using tabled resolution for constraint logic programming. *International symposium on logic-based program synthesis and transformation*. to appear.

Abdennadher, S., & Schütz, H. (1998). $\mathrm{CHR}^{\vee}$: A flexible query language. *Third international conference on flexible query answering systems*. LNAI 1495. Springer-Verlag.

Apt, K.R. (1998). A proof theoretic view of constraint programming. *Special issue of fundamenta informaticae*, **34**(3).

Apt, K.R., & Monfroy, E. (1999). Automatic generation of constraint propagation algorithms for small finite domains. *5th international conference on principles and practice of constraint programming, CP'99*. LNCS 1713. Springer-Verlag.

Codognet, P. (1995). A tabulation method for constraint logic programs. *8th symposium and exhibition on industrial applications of prolog*.

Cui, B., & Warren, D. S. (2000). A system for tabled constraint logic programming. *First international conference on computational logic*. LNCS 1861. Springer-Verlag.

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

Jaffar, J., & Maher, M. J. (1994). Constraint logic programming: A survey. *Journal of logic programming*, **19,20**, 503–581.

Le Provost, T., & Wallace, M. (1993). Generalized constraint propagation over the CLP scheme. *Journal of logic programming*, **16**(3), 319–359.

Marriott, K., & Stuckey, P. (1998). *Programming with constraints: An introduction*. MIT Press.

Muggleton, S., & De Raedt, L. (1994). Inductive Logic Programming : theory and methods. *Journal of logic programming*, **19,20**, 629–679.

Plotkin, G. (1970). A note on inductive generalization. *Pages 153–163 of: Machine intelligence*, vol. 5. Edinburgh University Press.

Przymusinski, T. (1990). On constructive negation in logic programming. *North american conference on logic programming*. MIT Press.

Ringeissen, C., & Monfroy, E. (2000). Generating propagation rules for finite domains via unification in finite algebra. *New trends in constraints*. LNAI 1865. Springer-Verlag.

Stuckey, P. (1995). Negation and constraint logic programming. *Information and computation*, **118**(1), 12–33.

Tamaki, H., & Sato, T. (1986). OLD resolution with tabulation. *3rd international conference on logic programming*. LNCS 225. Springer-Verlag.

Van Hentenryck, P., Saraswat, V., & Deville, Y. (1998). Desing, implementation, and evaluation of the constraint language cc(FD). *Journal of logic programming*, **37**(1-3), 139–164.

Warren, D. S. (1992). Memoing for logic programs. *Communications of ACM*, **35**(4), 93–11.

Würtz, J., & Müller, T. (1996). Constructive disjunction revisited. *20th german annual conference on artificial intelligence*. LNAI 1137. Springer-Verlag.