Automatic Generation of Rule-Based Constraint Solvers over Finite Domains

SLIM ABDENNADHER
University of Munich, Germany
and
CHRISTOPHE RIGOTTI
INSA-LISI Scientific and Technical University of Lyon, France

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 this paper, we propose a method for generating propagation and simplification rules for constraints over finite domains defined extensionally by e.g. a truth table or their tuples. The generation of rules is performed in two steps. First, propagation rules are generated. Propagation rules do not rewrite constraints but add new ones. Thus, the constraint store may contain superfluous constraints. Removing these constraints not only allows saving of space but also decreases the cost of constraint solving. Constraints can be removed using simplification rules. Thus, in a second step some propagation rules are transformed into simplification rules.

Furthermore, we show that our approach performs well on various examples, including Boolean constraints, multi-valued logic, and Allen's qualitative approach to temporal logic. Moreover, an application taken from the field of digital circuit design shows that our approach is of practical use.

Categories and Subject Descriptors: F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic—*Logic and constraint programming*; I.2.6 [**Artificial Intelligence**]: Learning—*Induction*

General Terms: Algorithms, Design

Additional Key Words and Phrases: Rule-Based Constraint Programming, Generation of Solvers, Finite Domains

1. INTRODUCTION

Rule-based formalisms are ubiquitous in computer science, and even more in constraint reasoning and programming. In constraint reasoning, algorithms are often specified using inference rules, rewrite rules, sequents, or first-order axioms written

Authors' addresses:
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 Blais Pascal, INSA Lyon, 69621 Villeurbanne Cedex, France, Christophe.Rigotti@insa-lyon.fr

as implications [Apt 1998; Kirchner and Ringeissen 1998; Apt and Monfroy 2001]. In the context of constraint programming, advanced programming languages like ELAN [Kirchner et al. 1993], CLAIRE [Caseau et al. 1999], and Constraint Handling Rules (CHR) [Frühwirth 1998], have shown that the concept of rules could be of major interest as a programming tool for constraint solvers.
However, a difficulty that arises frequently when specifying or programming a constraint solver in a rule-based language is to determine rules to simplify constraints and rules to propagate new constraints. The first kind of rules, called *simplification rules*, are rules that rewrite constraints to simpler constraints while preserving logical equivalence (e.g. $X{\leq}Y,\ Y{\leq}X \Leftrightarrow X{=}Y$). While the rules of the second kind, called *propagation rules*, are used to add new constraints, which are logically redundant but which may cause further simplification (e.g. $X{\leq}Y,\ Y{\leq}Z \Rightarrow X{\leq}Z$). In this work, we propose a method to generate automatically the propagation and simplification rules and to implement them in the language Constraint Handling Rules (CHR) to obtain a running rule-based constraint solver. However, the generation techniques proposed in this paper can be adapted to other languages. Using our method the user has the possibility to specify the form of the rules she/he wants to generate. The method allows any kind of constraints in the left hand side of rules and in their right hand side as well. The generation of rules is performed in two steps. In a first step, only propagation rules are generated using a method inspired by techniques used in the field of knowledge discovery and data mining [Toivonen et al. 1995]. In a second step, some propagation rules are transformed into simplification rules. In general, simplification rules reduce the number of constraints by replacing constraints by "'simpler"' ones and thus improve the time and space behavior of constraint solving. Of course this transformation step adds some cost to the whole rule generation process. However, solvers are generated once to solve different problems or different instances of the same problem. Thus the time needed for the generation step is not so crucial while it is important to produce reasonably efficient solvers.
Consider the following example: we want to generate a constraint solver for the boolean conjunction $and(X, Y, Z)$, where X and Y are the input arguments and Z is the output argument, and for the boolean negation $neg(A, B)$, where A is the input argument and B is the output argument. The relation *and* can be defined extensionally by the triples $\{(0,0,0),(0,1,0),(1,0,0),(1,1,1)\}$ and the relation *neg* can be defined by the pairs $\{(0,1),(1,0)\}$, where 1 stands for truth and 0 for falsity. In the first step, only propagation rules are generated, e.g.

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

In a second step, some propagation rules are transformed into simplification rules:

$$and(0,Y,Z) \Leftrightarrow Z{=}0.$$

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

Now, the last rule says that the constraints $and(X,Y,Z), neg(X,Y)$ can be simply replaced by the equality constraint $Z{=}0$ and the constraint $neg(X,Y)$, and thus will lead in general to more efficient constraint solving. This set of rules, generated automatically, corresponds to the well-known rules that can be found in several papers describing the propagation and simplification of boolean constraints, e.g. in form of demons [Dincbas et al. 1988], conditionals [van Hentenryck et al. 1992], CHR rules [Frühwirth 1998] or proof systems [Codognet and Diaz 1993; Apt 2000]. The paper is organized as follows: In section 2, we present the algorithm for the generation of propagation rules and give some soundness, correctness and termination results. Then, we give more examples of the use of this algorithm. In section 3, we present a method to transform propagation rules into simplification rules and give some properties of the transformation. In Section 4, we present an example showing the practical usefulness of the method developed in this paper. We review the related work in Section 5, and we conclude with a summary. This paper is a revised and homogenized presentation of separate contributions presented in [Abdennadher and Rigotti 2000] and in [Abdennadher and Rigotti 2001b].

2. GENERATION OF PROPAGATION RULES

In this section, we present the algorithm PropMiner dedicated to the generation of propagation rules for constraints over finite domains. First, before the description of the algorithm, we give a definition of the class of rules that are generated.

2.1 Class of Generated Rules

Let $\mathcal{A}$ be a set of atomic constraints. The set of all constraints over $\mathcal{A}$, i.e. the set of all non-empty finite subsets of $\mathcal{A}$, is noted $\mathcal{L}(\mathcal{A})$. The set of variables appearing in $\mathcal{A}$ is denoted by $Var(\mathcal{A})$.
Let CT be a constraint theory and let σ be a ground substitution. σ is a *solution* of a constraint C if $CT \models \sigma(C)$.

Definition 2.1. *Let $\mathcal{A}_{lhs}$ and $\mathcal{A}_{rhs}$ be two sets of atomic constraints not containing false*[1]*. The set of* propagation rules *over $\langle \mathcal{A}_{lhs}, \mathcal{A}_{rhs}\rangle$ is the set of all rules of the form $C_1 \Rightarrow C_2$, where $C_1 \in \mathcal{L}(\mathcal{A}_{lhs})$ and $C_2 \in \mathcal{L}(\mathcal{A}_{rhs}) \cup \{\{false\}\}$ and $C_1 \cap C_2 = \emptyset$. C_1 is called the left hand side (*lhs*) and C_2 the right hand side (*rhs*) of the rule. A* failure rule *is a propagation rule of the form $C_1 \Rightarrow \{false\}$.*

Definition 2.2. *A propagation rule $C_1 \Rightarrow C_2$ is* valid *if for any ground substitution σ, if σ is a solution of C_1 then σ is a solution of C_2. The rule $C_1 \Rightarrow \{false\}$ is* valid *if C_1 has no solution.*

[1] *false* will be used as a particular *rhs* for the rules.

Since the number of valid rules may become quite large, we consider that the rules that are in some sense the most general will be the most interesting to build a solver. We consider only a syntactical notion of rule generality which is inspired by the notion of structural covering used in association rule mining [Toivonen et al. 1995]. Intuitively this notion can be seen as a kind of subsumption between sets of rules.

DEFINITION 2.3. *Let $\mathcal{R}$ and $\mathcal{R}'$ be two sets of propagation rules. $\mathcal{R}'$ is a* lhs-cover *of $\mathcal{R}$ if for all $(C_1 \Rightarrow C_2) \in \mathcal{R}$ there exists $(C'_1 \Rightarrow C'_2) \in \mathcal{R}'$, such that $C'_1 \subseteq C_1$ and $C_2 \subseteq C'_2$.*

Note that all propagations that can be made using rules in $\mathcal{R}$, can also be done with rules in $\mathcal{R}'$.

EXAMPLE 2.1. *Let and be a ternary constraint corresponding to the Boolean conjunction used in Section 1. $\{\{and(X,Y,Z),\ X{=}0\} \Rightarrow \{Z{=}0\}\}$ is a* lhs-*cover of $\{\{and(X,Y,Z),\ X{=}0\} \Rightarrow \{Z{=}0\}, \{and(X,Y,Z),\ X{=}0,\ Y{=}0\} \Rightarrow \{Z{=}0\}\}$.*

The algorithm PROPMINER presented in Section 2.2 will generate a *lhs*-cover of the set of *valid propagation rules* over $\langle \mathcal{A}_{lhs}, \mathcal{A}_{rhs} \rangle$. However, many *lhs* are of little interest to build solvers based on propagation rules. To filter out such rules we consider that the user has in mind a particular *pattern* of constraints over which she/he wants to obtain propagations. Suppose that a user wants to generate interaction rules between the Boolean operations conjunction (*and*) and negation (*neg*), then we will restrict the search to rules having at least the constraint $and(X,Y,Z)$ together with $neg(A,B)$ in their *lhs*.
Additionally, it is in general useless to look for rules such that some atomic constraints are not related to the others by variable sharing. For example $\{and(X,Y,Z),\ neg(A,B),\ Y{=}1\} \Rightarrow \{Z{=}X\}$ contains an isolated atomic constraint $neg(A,B)$ in its *lhs* and is likely to be useless since it is redundant wrt. to the simpler rule $\{and(X,Y,Z),\ Y{=}1\} \Rightarrow \{Z{=}X\}$. Of course some rules having such independent atomic constraints in their *lhs* can be non-redundant, as it is the case for instance for the rule $c_1(X_1,\ldots,X_n), c_2(Y_1,\ldots,Y_m) \Rightarrow c_3(X_1,\ldots,X_n,Y_1,\ldots,Y_m)$. But it should be noticed that in practice such rules are rarely used (for example they are not used in the classical rule-based solvers implemented in CHR [Abdennadher and Saft]). So, in order to reduce a priori the search space to be explored to find the rules, we prefer to restrict the generation to rules such that all atomic constraints are related by variable sharing. However, the algorithm PROPMINER can be easily modified to drop this limitation if needed.
We now define the set of rules satisfying these restriction criteria and called *relevant propagation rules*. The examples presented in Section 2.6 and Section 4 show that this notion of relevant rules is appropriated to find interesting propagation, without being overwhelmed by huge sets of rules.

DEFINITION 2.4. *Let $Base_{lhs}$ be a set of atomic constraints. A set of atomic constraints $\mathcal{A}$ is an* interesting pattern *wrt. $Base_{lhs}$ 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* $\mathcal{A}$ *that share at least one variable, i.e.,* $join_{\mathcal{A}} = \{\langle c_1, c_2\rangle \mid c_1 \in \mathcal{A}, c_2 \in \mathcal{A}, Var(\{c_1\}) \cap Var(\{c_2\}) \neq \emptyset\}$.

DEFINITION 2.5. *The set of* relevant propagation rules *over* $\langle Base_{lhs}, \mathcal{A}_{lhs}, \mathcal{A}_{rhs}\rangle$ *is the set of propagation rules over* $\langle \mathcal{A}_{lhs}, \mathcal{A}_{rhs}\rangle$ *of the form* $C_1 \Rightarrow C_2$, *where* C_1 *is an interesting pattern with respect to* $Base_{lhs}$ *and has at least one solution.*

In this definition, the second restriction ensures that rules with an inconsistent *lhs* but having a *rhs* different from *false* are not relevant.

EXAMPLE 2.2. *Assume again that one wants to generate interaction rules between Boolean conjunction and Boolean negation, then* $Base_{lhs}$ *has the following form* $\{and(X,Y,Z),\ neg(A,B)\}$.
$\{and(X,Y,Z),\ neg(A,B),\ A{=}X,\ B{=}Y\} \Rightarrow \{Z{=}0\}$ *is then a relevant propagation rule, while* $\{and(X,Y,Z),\ Y{=}1\} \Rightarrow \{Z{=}X\}$ *and* $\{and(X,Y,Z),\ neg(A,B),\ X{=}0\} \Rightarrow \{Z{=}0\}$ *are not relevant since their* lhs *are not interesting patterns wrt.* $Base_{lhs}$. *However, it should be noticed that* $\{and(X,Y,Z),\ Y{=}1\} \Rightarrow \{Z{=}X\}$ *would have been relevant for the atomic constraint* $and(X,Y,Z)$ *alone (i.e., when* $Base_{lhs} = \{and(X,Y,Z)\}$*).*

The generation algorithm will discard any rule which is not a relevant propagation rule over a given $\langle Base_{lhs}, \mathcal{A}_{lhs}, \mathcal{A}_{rhs}\rangle$. We present in Section 2.4 additional simplifications of the set of rules generated to remove some specific redundancies.

2.2 The PROPMINER Algorithm

Using the PROPMINER algorithm the user has the possibility to specify the admissible syntactic forms of the rules. The user determines $Base_{lhs}$ and chooses the set of candidate atomic constraints (denoted $Cand_{lhs}$) used to form conjunctions with $Base_{lhs}$ in the *lhs* of the rules (usually, these candidate atomic constraints are simply equality constraints). The user also specifies the set of atomic constraints that can be used to form conjunctions in the *rhs* of the rules. This set is denoted $Cand_{rhs}$.
The semantics of the constraint $Base_{lhs}$ is determined by means of a set noted $SolBase_{lhs}$ containing all solutions of $Base_{lhs}$. This set must be finite. Additionally the semantics of the candidate atomic constraints $Cand_{lhs}$ and $Cand_{rhs}$ must be provided by two constraint theories CT_{lhs} and CT_{rhs}, respectively, and must be defined over the same domain as $Base_{lhs}$.
To compute the propagation rules the algorithm generates each possible *lhs* constraint (noted C_{lhs}) and for each determines the corresponding *rhs* constraint (noted C_{rhs}).
For each *lhs* C_{lhs} the corresponding *rhs* C_{rhs} is computed in the following way:

(1) if C_{lhs} has no solution then $C_{rhs} = \{false\}$ and we have the failure rule $C_{lhs} \Rightarrow \{false\}$.
(2) if C_{lhs} has at least one solution then C_{rhs} is the set of all atomic constraints that are candidates for the *rhs* part and are true for all solutions of C_{lhs}. If C_{rhs} is not empty we have the rule $C_{lhs} \Rightarrow C_{rhs}$.

During the exploration of the search space, the algorithm uses two main pruning strategies:

(1) (*Pruning1*) 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*.
(2) (*Pruning2*) if a rule $C_{lhs} \Rightarrow C_{rhs}$ is generated then there is no need to consider any C such that $C_{lhs} \subset C$ and $C \cap C_{rhs} \neq \emptyset$ to form other rule *lhs*.

Since the effect of *Pruning2* is not as straightforward as the effect of *Pruning1* we illustrate it by mean of the following example.

EXAMPLE 2.3. *After generating the relevant propagation rule of example 2.2:* $\{and(X,Y,Z),\ neg(A,B),\ A{=}X,\ B{=}Y\} \Rightarrow \{Z{=}0\}$, *the possible* lhs $\{and(X,Y,Z),\ neg(A,B),\ A{=}X,\ B{=}Y,\ B{=}1,\ Z{=}0\}$ *is not considered using Pruning2, while* $\{and(X,Y,Z),\ neg(A,B),\ A{=}X,\ B{=}Y,\ B{=}1\}$ *remains a* lhs *candidate and may lead to the following rule* $\{and(X,Y,Z),\ neg(A,B),\ A{=}X,\ B{=}Y,\ B{=}1\} \Rightarrow \{Z{=}0,\ A{=}0,\ X{=}0,\ Y{=}1\}$.

These pruning strategies are much more efficient if during the enumeration of all possible rule *lhs*, a given *lhs* is considered before any of its supersets. So a specific ordering for this enumeration is imposed in the algorithm. Moreover, this ordering allows to discover early covering rules avoiding then the generation of many uninteresting covered rules.
To simplify the presentation of the algorithm we consider that all possible *lhs* are stored in a list L and that unnecessary *lhs* candidates are simply removed from this list. For efficiency reasons the concrete implementation is not based on a list but on a tree containing *lhs* candidates on its nodes. More details are given in Section 2.5.

We now give an abstract description of the PROPMINER algorithm. The algorithm is given in Figure 1. It takes as input:

(1) $Base_{lhs}$: a constraint that must be included in any *lhs* of the rules.
(2) $SolBase_{lhs}$: the finite set of ground substitutions that are solutions of $Base_{lhs}$. Note that this defines the constraint $Base_{lhs}$ extensionally.
(3) $Cand_{lhs}$: a finite set of atomic constraints that are candidates to form *lhs* of the rules such that $Var(Cand_{lhs}) \subseteq Var(Base_{lhs})$.
(4) $Cand_{rhs}$: a finite set of atomic constraints that are candidates to form *rhs* of the rules such that $Var(Cand_{rhs}) \subseteq Var(Base_{lhs})$.
(5) CT_{lhs}: a constraint theory for $Cand_{lhs}$.
(6) CT_{rhs}: a constraint theory for $Cand_{rhs}$.

And it produces the following output: a *lhs*-cover of the valid relevant propagation rules over $\langle Base_{lhs}, Cand_{lhs}, Cand_{rhs}\rangle$.

It should be noticed that the algorithm needs to check if a given ground constraint is entailed by CT_{lhs} or by CT_{rhs}. This requirement is formalized by the notion of *ground decidability* defined as follows.

DEFINITION 2.6. *Let CT be a constraint theory, let* Γ *be a set of ground substitutions and* $\mathcal{A}$ *be a set of atomic constraints. CT is* ground decidable for $\langle \mathcal{A}, \Gamma\rangle$ *if*

the properties $CT \models \sigma(c)$ *and* $CT \not\models \sigma(c)$ *are decidable for any* $c \in \mathcal{A}$ *and for any substitution* $\sigma \in \Gamma$.

So, the algorithm imposes that the constraint theories CT_{lhs} and CT_{rhs} are ground decidable for $\langle Cand_{lhs}, SolBase_{lhs}\rangle$ and $\langle Cand_{rhs}, SolBase_{lhs}\rangle$ respectively. It should be noticed that this restriction is very weak, since the property holds for almost all useful classes of constraint theories. Furthermore, the algorithm also requires that the corresponding decision procedures are provided.

---

begin

Let $\mathcal{R}$ be an empty set of rules.
Let L be a list containing the elements of $\mathcal{L}(Base_{lhs} \cup Cand_{lhs})$ in any order.

Remove from L any element which is not an interesting pattern wrt. $Base_{lhs}$.
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 equal to the first element of L, and remove from L its first element.
if for all $\sigma \in SolBase_{lhs}$ we have
$CT_{lhs} \not\models \sigma(C_{lhs} \setminus Base_{lhs})$ **then**
add the failure rule $(C_{lhs} \Rightarrow \{false\})$ to $\mathcal{R}$
and remove from L each element C such that $C_{lhs} \subset C$. *//Pruning1*
else
compute C_{rhs} the rule *rhs*, defined by
$C_{rhs} = \{c | c \in (Cand_{rhs} \setminus Cand_{lhs})$ and for all $\sigma \in SolBase_{lhs}$
when $CT_{lhs} \models \sigma(C_{lhs} \setminus Base_{lhs})$ we have $CT_{rhs} \models \sigma(c)\}$.
if C_{rhs} is not empty **then**
add the rule $(C_{lhs} \Rightarrow C_{rhs})$ to $\mathcal{R}$
and remove from L each element C such that
$C_{lhs} \subset C$ and $C \cap C_{rhs} \neq \emptyset$. *//Pruning2*
endif
endif
endwhile

output $\mathcal{R}$

end

Fig. 1. The PropMiner Algorithm

In the PropMiner algorithm the list L of possible *lhs* is initialized to be a finite list. Each iteration of the while loop removes at least one element in L. This ensures the following property.

Theorem 2.1 Termination. *The algorithm* PropMiner *terminates and yields a finite set of propagation rules.*

The following results establish soundness and correctness of the algorithm.

THEOREM 2.2 SOUNDNESS. *The* PROPMINER *algorithm computes valid relevant propagation rules over* $\langle Base_{lhs}, Cand_{lhs}, Cand_{rhs}\rangle$.

PROOF. All C_{lhs} considered are interesting pattern wrt. $Base_{lhs}$, thus only relevant rules can be generated. Let C'_{lhs} be $C_{lhs} \setminus Base_{lhs}$. A rule of the form $C_{lhs} \Rightarrow \{false\}$ can be generated only if all solutions of $Base_{lhs}$ are not solutions of C'_{lhs}. So any rule $C_{lhs} \Rightarrow \{false\}$ generated is valid. A rule of the form $C_{lhs} \Rightarrow C_{rhs}$, where $C_{rhs} \neq \{false\}$ can be generated only if all solutions of $Base_{lhs}$ that are solutions of C'_{lhs}, are also solutions of all atomic constraints in C_{rhs}. Hence all generated rules of the form $C_{lhs} \Rightarrow C_{rhs}$ are valid. □

THEOREM 2.3 CORRECTNESS. *The* PROPMINER *algorithm computes a lhs-cover of the valid relevant propagation rules over* $\langle Base_{lhs}, Cand_{lhs}, Cand_{rhs}\rangle$ *when* $Var(Cand_{lhs}) \subseteq Var(Base_{lhs})$ *and* $Var(Cand_{rhs}) \subseteq Var(Base_{lhs})$.

PROOF. First, we do not consider the two pruning strategies *Pruning1* and *Pruning2*. Then the algorithm enumerates all possible rule left hand sides that are interesting patterns wrt. $Base_{lhs}$. So it generates all valid relevant failure rules. Moreover for any valid relevant rule of the form $C_1 \Rightarrow C_2$, where $C_2 \neq \{false\}$ the algorithm considers C_1 as a candidate *lhs*. Then it computes C_{rhs} containing all atomic constraints c such that all solutions of C_1 are solutions of c. Thus $C_2 \subseteq C_{rhs}$. So if we do not consider the two pruning strategies *Pruning1* and *Pruning2* the algorithm output a *lhs*-cover of the valid relevant propagation rules. Now we show that the two pruning criteria are safe.
(*Pruning1)* When a rule of the form $C_1 \Rightarrow \{false\}$ is generated all candidate *lhs* C such that $C_1 \subset C$ are discarded. However, since $C_1 \Rightarrow \{false\}$ is valid, C_1 has no solution, and thus any $C \supset C_1$ have no solution too, and can only leads to a rule of the form $C \Rightarrow \{false\}$ which will be *lhs*-covered by $C_1 \Rightarrow \{false\}$.
(*Pruning2)* When a rule of the form $C_1 \Rightarrow C_2$, where $C_2 \neq \{false\}$ is generated all candidates *lhs* C such that $C_1 \subset C$ and $C \cap C_2 \neq \emptyset$ are discarded.
To establish the safety of this pruning criterion we show by induction on the size of C that for any valid relevant rule $C \Rightarrow C_3$, where $C_3 \neq \{false\}$, there exists a candidate *lhs* $C' \subseteq C$ such that C' is not pruned and generates a valid relevant rule that *lhs*-covered $C \Rightarrow C_3$.
The claim holds trivially for $|C| = |Base_{lhs}|$. Suppose that the property holds for all C such that $|Base_{lhs}| \leq |C| \leq i$, and consider a valid relevant rule $C \Rightarrow C_3$, where $|C| = i + 1$ and $C \neq \{false\}$.
If C has not be pruned then we have the claim. If C has been pruned, then there exists $C_1 \subset C$ and C_2 such that $C_1 \Rightarrow C_2$ is a valid relevant rule and $C \cap C_2 \neq \emptyset$. Let A be any element in $C \cap C_2$ and $C_4 = C \setminus \{A\}$. Now consider the rule $C_4 \Rightarrow C_3 \cup \{A\}$. $C_1 \Rightarrow C_2$ is a propagation rule, so $C_1 \cap C_2 = \emptyset$ and then $A \notin C_1$. As $C_1 \subset C$ and $C_4 = C \setminus \{A\}$ then $C_1 \subset C_4$. So $C_4 \Rightarrow C_2$ is valid since $C_1 \Rightarrow C_2$ is valid. $C_4 \Rightarrow C_2$ is valid and $A \in C_2$ thus $C_4 \Rightarrow C_3$ is also valid because $C \Rightarrow C_3$ is valid and $C = C_4 \cup \{A\}$.
Moreover C_1 and C are interesting patterns wrt. $Base_{lhs}$ and have at-least one solution because they are the *lhs* of the relevant rules $C_1 \Rightarrow C_2$ and $C \Rightarrow C_3$. As $C_1 \subset C_4 \subset C$ then C_4 is also an interesting pattern with at-least one solution.

Thus $C_4 \Rightarrow C_3$ is a valid relevant propagation rule.
By the induction hypothesis, there exists C_5 and C_6 such that C_5 is a candidate that is not pruned and generates the valid relevant propagation rule $C_5 \Rightarrow C_6$ that *lhs*-covers $C_4 \Rightarrow C_3$, and thus also *lhs*-covers $C \Rightarrow C_3$. □

The complexity of the algorithm can be expressed in the number of ground decidability tests needed. In the worst case the number of decidability tests to perform is in the order of $O(2^{|Cand_{lhs}|}\ |SolBase_{lhs}|\ |Cand_{rhs}|)$. The most limitative factor is the number of atomic constraints that are candidates to form the *lhs* of the rules. However, as presented in Section 2.6 and Section 4, a limited number of candidates $|Cand_{lhs}|$ still allows to find interesting rules. Moreover, nearly all rules obtained can lead to pruning according to the strategy *Pruning2*, preventing the exploration of the full collection of $2^{|Cand_{lhs}|}$ possible *lhs*. The corresponding execution times reported in Section 2.6 and Section 4 show that in practice the generations can be done in a reasonable amount of time.

2.3 Consistency

K. R. Apt and E. Monfroy [1999; 2001] identified two consistency notions for rule-based constraint solvers, *membership consistency* and *rule consistency*. It has been shown that membership consistency is equivalent to arc consistency and that rule consistency coincides with arc consistency for domains consisting of at most two elements.
Rule consistency for a constraint C is the notion of consistency ensured by a solver consisting of all valid *equality rules* which are rules of the form

$$C(X_1, \ldots, X_n),\ X_i{=}v_i,\ \ldots,\ X_k{=}v_k\ \Rightarrow\ Y{\neq}v,$$

where $X_i, \ldots, X_k$ are variables, and $v_i, \ldots v_k, v$ are elements of the domain associated to variables of C, and Y is a variable occurring in C but not in $X_i, \ldots, X_k$.
If $Base_{lhs}$ consists of an atomic constraint and if we allow equality constraints (between a variable and a constant) in *lhs* and disequality constraints (between a variable and a constant) in *rhs*, then we can easily see that the class of rules generated by PropMiner corresponds to the class of equality rules defined in [Apt and Monfroy 2001]. Then according to Theorem 2.3 the rules generated enforce the same kind of consistency.
For convenience, we introduce the following notation. Let c be a constraint predicate 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, $atomic(c, D_1, D_2) = \{c(\alpha, \beta) \mid \alpha \in D_1 \text{ and } \beta \in D_2\}$.

Theorem 2.4. Let $C(X_1, \ldots, X_n)$ be an atomic constraint and let $v_1, \ldots, v_k$ be the elements of the domain associated to $X_1, \ldots, X_n$, where the three following conditions are satisfied:

$$\begin{aligned} Base_{lhs} &= \{C(X_1, \ldots, X_n)\} \\ Cand_{lhs} &= atomic(=, \{X_1, \ldots, X_n\}, \{v_1, \ldots, v_k\}) \\ Cand_{rhs} &= atomic(\neq, \{X_1, \ldots, X_n\}, \{v_1, \ldots, v_k\}) \end{aligned}$$

then the propagation rules generated by PropMiner enforce rule consistency.

Membership consistency corresponds to the consistency ensured by all valid *membership rules* which are rules of the form

$$C(X_1, \ldots, X_n),\ X_i{\in}S_i,\ \ldots,\ X_k{\in}S_k \Rightarrow Y{\neq}v,$$

where $X_i, \ldots, X_k$ are variables, $S_i, \ldots S_k$ are subsets of the domain D associated to variables of C, v is a an element of D, and Y is a variable occurring in C but not in $X_i, \ldots, X_k$.
Over finite domains, a membership rule can be written using only disequality constraints in the left hand side. For example, over a domain $D = \{a, b, c, d\}$, a membership rule $p(X, Y),\ X{\in}\{a, b\} \Rightarrow Y{\neq}c$ can be expressed as $p(X, Y), X{\neq}c, X{\neq}d \Rightarrow Y{\neq}c$. Thus according to Theorem 2.3 we have the following result:

THEOREM 2.5. Let $C(X_1, \ldots, X_n)$ be an atomic constraint and let $v_1, \ldots, v_k$ be the elements of the domain associated to $X_1, \ldots, X_n$, where the three following conditions are satisfied:

$$\begin{aligned} Base_{lhs} &= \{C(X_1, \ldots, X_n)\} \\ Cand_{lhs} &= atomic(\neq, \{X_1, \ldots, X_n\}, \{v_1, \ldots, v_k\}) \\ Cand_{rhs} &= atomic(\neq, \{X_1, \ldots, X_n\}, \{v_1, \ldots, v_k\}) \end{aligned}$$

then the propagation rules generated by PROPMINER enforce membership consistency.

In [Apt and Monfroy 2001], it has been established that rule consistency (on unary or binary domains) and membership consistency is equivalent to arc consistency. Theorem 2.4 and Theorem 2.5 show that PROPMINER can be used to generate rules that ensure the same kind of consistency. Furthermore, the input parameters $Cand_{lhs}$ and $Cand_{rhs}$ of PROPMINER can be set to supersets of the values needed in Theorem 2.4 and Theorem 2.5 (e.g., incorporating equalities between variables). Then, PROPMINER can find additional propagation rules, that will enable the corresponding rule-based solver to perform more constraint propagations, and thus ensure a stronger consistency.

2.4 Removing Redundancy

The PROPMINER algorithm computes a *lhs*-cover of the valid relevant propagation rules over $\langle Base_{lhs}, Cand_{lhs}, Cand_{rhs}\rangle$, but this cover may contain some kind of redundancies as illustrated in the following example.

EXAMPLE 2.4. *For Boolean constraints, with the appropriate input, the* PROPMINER *algorithm can produce the rule* $\{and(X, Y, Z),\ Z{=}1\} \Rightarrow \{X{=}Y,\ X{=}Z,\ Y{=}Z,\ X{=}1,\ Y{=}1\}$. *If we have already a solver to handle equality constraints, then it is desirable to simplify this rule into* $\{and(X, Y, 1)\} \Rightarrow \{X{=}1,\ Y{=}1\}$.
On another input, for the logical operation exclusive-or (xor), the PROPMINER *algorithm can produce the rule* $\{xor(X, Y, Z),\ X{=}Y\} \Rightarrow \{Z{=}0\}$ *and the rule* $\{xor(X, Y, Z),\ X{=}0,\ Y{=}0\} \Rightarrow \{Z{=}0\}$. *The second rule is clearly useless to build a solver since it cannot propagate new atomic constraints wrt. the first rule, and thus it should be discarded.*

We use an ad-hoc technique to simplify the rule *lhs* and *rhs*, and to suppress some redundant rules. This process does not lead to a precisely defined canonical

representation of the rules generated, but in practice (see Section 2.6) it produces small and readable sets of rules.
This simplification technique is incorporated in the PROPMINER algorithm and performed during the generation of the rules. For clarity reasons it is presented apart from the algorithm given in Figure 1. The simplification principle is as follows:

(1) For each rule generated by the algorithm of Figure 1 the equality constraints appearing in the *lhs* are transformed into substitutions that are applied to the *lhs* and the *rhs*. More precisely, for an equality constraint $V_1{=}V_2$, where V_1 and V_2 are variables, V_1 is simply replaced by V_2 in the whole rule. If the equality is of form $V{=}k$ or $k{=}V$ with V a variable and k a constant of the domain, then V is substituted by k. When all equality constraints of the *lhs* have been processed in this way, then the completely ground atomic constraints are removed from the *lhs*. For example, in the rule $\{and(X,Y,Z),\ Z{=}1\} \Rightarrow \{X{=}Y,\ X{=}Z,\ Y{=}Z,\ X{=}1,\ Y{=}1\}$ the variable Z is replaced by the constant 1 leading to $\{and(X,Y,1),\ 1{=}1\} \Rightarrow \{X{=}Y,\ X{=}1,\ Y{=}1\}$ which is then simplified to $\{and(X,Y,1)\} \Rightarrow \{X{=}Y,\ X{=}1,\ Y{=}1\}$). When the equality constraint $=$ is defined as the syntactic equality over first-order terms, then the transformed rule can be used in place of the original one to perform the same propagations.

(2) The new rules are then 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*).

(3) Then we remove each rule that does not propagate additional atomic constraint wrt. the other rules. This is done in the following way. Let S be a set of rules initialized to the empty set. Consider each rule $C_1 \Rightarrow C_2$ in L' (taken according to the list ordering so that more general rules are processed before more specific ones). Let C_3 be the subset of atomic constraints in C_2 defined by $C_3 = \{c \mid c \in C_2$ and c cannot be obtained from C_1 by any sequence of applications of the propagation rules in $S\}$. If C_3 is empty then remove the rule from L' else keep $C_1 \Rightarrow C_2$ in L' and add it in S. Process next rule in L'. This simplification step leads to the removal of rules that are useless wrt. the propagations that can be made by the remaining ones (e.g., deletion of the redundant rule for *xor* presented in Example 2.4).

(4) If $Cand_{rhs}$ contains equality constraints and if the solver given for $Cand_{rhs}$ handles also disequality constraints then L' is processed further to remove some redundant equalities in the *rhs* of the rules, that are likely to occur quite often. For each rule $C_1 \Rightarrow C_2$ in L' (taken in any order) C_2 is simplified non-deterministically as follows. C_{simp} is initialized to C_2. Then each equality $X{=}Y$ in C_{simp} is considered in turn. If $X{\neq}Y$ is handled by the solver for $Cand_{rhs}$, the solver is used to check the satisfiability of the constraint C_4 defined as $C_4 = (C_{simp} \setminus \{X{=}Y\}) \cup \{X{\neq}Y\}$. If it is not satisfiable then $X{=}Y$ is removed from C_{simp}. If the solver does not handle $X{\neq}Y$ or does not report that C_4 is unsatisfiable, then $X{=}Y$ is kept in C_{simp}. When all equalities in C_{simp} have been processed, then $C_1 \Rightarrow C_2$ is replaced in L' by $C_1 \Rightarrow C_{simp}$.
This simplification leads for example to the transformation of $\{and(X,Y,1)\} \Rightarrow \{X{=}Y,\ X{=}1,\ Y{=}1\}$ into $\{and(X,Y,1)\} \Rightarrow \{X{=}1,\ Y{=}1\}$. This step is mostly

cosmetic to enhance the human readability of the rules.

(5) Output the list L' containing the simplified rules.

2.5 Implementation Issues

As described in Section 2.2, the PropMiner algorithm needs to enumerate *lhs* constraints. Our implementation follows the idea of direct extraction of association rules [Agrawal et al. 1993] by exploring a tree corresponding to the *lhs* search space as described in [Bayardo et al. 1999]. This tree is expanded and explored using a depth first strategy, in a way that constructs only necessary *lhs* candidates and allows to remove uninteresting candidates by cutting whole branches of the tree. The branches of the tree are developed 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 [Plotkin 1970] ordering commonly used in Inductive Logic Programming [Muggleton and De Raedt 1994] to structure the search space.
The running prototype is implemented in SICStus Prolog 3.7.1 (900 lines of Prolog) and takes advantage of the support of propagation rules in this environment by means of Constraint Handling Rules (CHR) [Frühwirth 1998] in the following way. During the execution of the PropMiner algorithm we build incrementally a solver with the propagation rules generated and this solver is used to perform the rule simplification according to Section 2.4.

2.6 Examples

This section gives examples of applications of the PropMiner algorithm to generate propagation rules for well-known constraints. The generation of this kind of rules for some of these constraints has previously been considered in [Apt and Monfroy 1999; 2001] and this approach will be discussed in Section 5.
While we cannot – within the space limitations – introduce the whole generated set of rules, we still give a fragment of it. The complete sets of rules are available in [Abdennadher and Saft] and can be executed online. In this section, the times given for the generation of rules have been measured using the following software and hardware: SICStus Prolog 3.7.1, PC Pentium 3 with 256 MBytes of memory and a 500 MHz processor.
In the following, we assume that the constraint theories define among other equality ("=") and disequality ("≠") as syntactic equality and disequality. Furthermore, we assume that the corresponding constraints are handled by an appropriate constraint solver. For convenience, we also use the notation $atomic(c, D_1, D_2)$ as defined in Section 2.3.

2.6.1 *Boolean Constraints.* For these constraints, we used the constants 0 for falsity and 1 for truth, and predicates to represent Boolean conjunction (*and*), disjunction (*or*), negation (*neg*) and exclusive-or (*xor*). These Boolean operations are modeled as relation, e.g., $xor(X, Y, Z)$ holds if and only if Z is the result of the application of *xor* on X and Y.
For the conjunction *and* the algorithm PropMiner with the following input

$$Base_{lhs} = \{and(X, Y, Z)\}$$

$$Cand_{lhs} \;=\; Cand_{rhs} = atomic(=, \{X, Y, Z\}, \{X, Y, Z, 0, 1\})$$

generates the following rules in 0.05 seconds:

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

These propagation rules generated automatically correspond to the implementation of *and* using CHR [Frühwirth 1998].
The PropMiner algorithm can also produce failure rules. For example, in the case of the negation *neg*, we obtain, among other, the rule:

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

Propagation rules defining interactions between user-defined atomic constraints can also be found. With the following input

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

the following rules defining interaction between boolean conjunction and boolean negation are generated:

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

Since the user can specify the form of the *rhs* of the rules, propagation rules with a *rhs* consisting of more complex constraints than equality constraints can also be generated. For example, with the following input, in the case of *xor*

$$\begin{aligned} Base_{lhs} \;&=\; \{xor(X, Y, Z)\} \\ Cand_{lhs} \;&=\; atomic(=, \{X, Y, Z\}, \{X, Y, Z, 0, 1\}) \\ Cand_{rhs} \;&=\; Cand_{lhs} \cup atomic(neg, \{X, Y, Z\}, \{X, Y, Z, 0, 1\}) \end{aligned}$$

6 rules, analogous to the ones for *and*, and the following 3 rules are generated in 0.1 seconds:

$$\begin{aligned} xor(X, Y, 1) &\;\Rightarrow\; neg(X, Y). \\ xor(X, 1, Z) &\;\Rightarrow\; neg(X, Z). \\ xor(1, Y, Z) &\;\Rightarrow\; neg(Y, Z). \end{aligned}$$

2.6.2 *Full-adder.* One important application of Boolean constraints is the modeling of logic circuits. Let $fulladder(X, Y, Z, S, C)$ represents the addition of two bits X and Y, where Z is the input carry bit, S is the output bit, and C is the output carry bit. The full-adder can be defined using Boolean predicates (see, e.g. [van Hentenryck 1991]) as follows, $fulladder(X, Y, Z, S, C)$ is equivalent to $and(X, Y, C1) \wedge xor(X, Y, S1) \wedge and(Z, S1, C2) \wedge xor(Z, S1, S) \wedge or(C1, C2, C)$. Using the PROPMINER algorithm 28 rules within 0.68 seconds are generated for $fulladder$. Typical rules are:

$$fulladder(X, Y, CI, S, S) \Rightarrow X{=}S,\ Y{=}S,\ CI{=}S.$$
$$fulladder(X, Y, CI, CI, C) \Rightarrow X{=}C,\ Y{=}C.$$
$$fulladder(0, Y, CI, S, 1) \Rightarrow S = 0.$$

The first rule says for example that the constraint $fulladder(X, Y, CI, S, C)$, when the output bit S is equal to the output carry bit C, can propagate the information that the bits to be added X and Y, and the input carry bit CI are equal to the output bit S.

2.6.3 *Three Valued Logics.* We consider the equivalence relation defined by the truth table given in [Kleene 1950] and recalled below, where the value t stands for true, f for false and u for unknown.

X	Y	X ≡ Y
t	t	t
t	f	f
t	u	u
f	t	f
f	f	t
f	u	u
u	t	u
u	f	u
u	u	u

Let $eq3val$ be the ternary constraint corresponding to this table. The PROPMINER algorithm generates for $eq3val$ 16 rules within 0.3 seconds. Examples of these rules are:

$$eq3val(X, X, X) \Rightarrow X{\neq}f.$$
$$eq3val(X, Y, t) \Rightarrow X{\neq}u,\ X{=}Y.$$
$$eq3val(X, f, X) \Rightarrow X{=}u.$$

For instance, the first rule means that for the constraint $eq3val(X, Y, Z)$, when it is known that the input arguments X and Y and the output Z are equal, we can propagate that X is different from f.

2.6.4 *Temporal Reasoning.*
In [Allen 1983] an interval-based approach to temporal reasoning is presented. Allen's approach to reasoning about time is based on the notion of time intervals and binary relations on them. Given two time intervals, their relative positions can be described by exactly one of thirteen primitive interval relations, where each

primitive relation can be defined in terms of its endpoint relations. These relations are *equality*, the 6 direct relations *before*, *during*, *overlaps*, *meets*, *starts* and *finishes*, and their 6 converses.
In [Allen 1983] there is a 13×13 table defining a "composition" constraint between a triple of events X, Y and Z, e.i. if the temporal relations between the events X and Y and the events Y and Z are known, what is the temporal relation between X and Z. The composition constraint, denoted by *allenComp*, can be defined as follows: $allenComp(R_1, R_2, R_3)$ is equivalent to $R_1(X, Y) \wedge R_2(Y, Z) \rightarrow R_3(X, Z)$, where R_1, R_2, R_3 are primitive interval relations.
For *allenComp* 489 rules are generated in 83.12 seconds, provided that the user specifies that the *rhs* of the rules may consist of a conjunction of equality and disequality constraints. As in [Apt and Monfroy 1999; 2001] we denote the 6 direct relations respectively by b, d, o, m, s, f, their converses by bi, di, oi, mi, si, fi and the equality relation by e. Typical rules are:

$$\begin{aligned} allenComp(R_1, R_1, R_1) &\Rightarrow R_1{\neq}m,\ R_1{\neq}mi. \\ allenComp(R_1, R_1, e) &\Rightarrow R_1{=}e,\ R_1{\neq}m,\ R_1{\neq}mi. \\ allenComp(o, b, R_3) &\Rightarrow R_3{=}b. \end{aligned}$$

3. GENERATION OF SIMPLIFICATION RULES

Since a propagation rule does not rewrite constraints but adds new ones, the constraint store may contain superfluous information. These constraints can be removed from the constraint store using *simplification rules* reducing the number of constraints on which rules can be applied and thus improving both the time and space behavior of constraint solving.
In general, finding simplification rules implies additional cost to the generation of constraint solvers. However, since a solver can be used to solve different problems, its efficiency is more important than the efficiency of its generation process.

DEFINITION 3.1. Let $\mathcal{A}_{lhs}$ and $\mathcal{A}_{rhs}$ be two sets of atomic constraints not containing *false*. A *simplification rule* is a rule of the form $C_1 \Leftrightarrow C_2$, where $C_1 \in \mathcal{L}(\mathcal{A}_{lhs})$ and $C_2 \in \mathcal{L}(\mathcal{A}_{rhs})$. A simplification rule $C_1 \Leftrightarrow C_2$ is *valid* if for any ground substitution σ, if σ is a solution of C_1 then σ is a solution of C_2.

Often, propagation rules generated by the PROPMINER algorithm can be transformed into simplification rules. For the Boolean conjunction the propagation rules presented in Section 2.6.1 could be replaced by the following simplification rules:

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

Our aim is to provide some criteria to perform such a transformation. One simple criterion could be the following: whenever a *rhs* of a rule propagates information

making its *lhs* ground, then this rule can be implemented by means of a simplification rule. This criteria is not sufficient since it can only be applied to the last rule of the example presented above. Finding a general criterion is even more difficult when we consider propagation rules with a *lhs* consisting of more than one atomic constraint.
In [Abdennadher and Rigotti 2001b], a method has been proposed to transform some propagation rules into simplification rules. The method is a kind of post-processing approach based on a confluence test which requires the termination of the set of rules. Termination is in general undecidable, and this problem may jeopardize the practicability of this method. In the following, we propose a transformation method that does not require such a termination test.

3.1 The SimpMiner Algorithm

To simplify the presentation, we present the SimpMiner algorithm to transform (when possible) propagation rules into simplification rules independently from the algorithm presented in Section 2. 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 SimpMiner algorithm takes as input a set of propagation rules S and transforms when applicable these propagation rules into simplification rules. This transformation leads to a set of rules S' that can be used instead of S for constraint solving. The algorithm works by repeatedly selecting a rule R from S. For each rule R it tries to transform R into a simplification rule R' by duplicating 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). To test the validity of R' The SimpMiner algorithm checks if all solutions of the *rhs* of the rule are also solutions of its *lhs*. If it is the case then the transformation is accepted and the next rule is considered.
Before giving an abstract description of the SimpMiner algorithm, we illustrate it by the following example:

Example 3.1. *Let S be the set of propagation rules for the boolean conjunction and negation constraints and their interaction as generated in Section 2.6.1.*
All propagation rules with one atomic constraint in their lhs*, can be transformed into simplification rules. For example the propagation rule $and(X,X,Z) \Rightarrow X{=}Z$ can be transformed into the simplification rule $and(X,X,Z) \Leftrightarrow X{=}Z$, since the only solutions $\{X{=}0, Z{=}0\}$ and $\{X{=}1, Z{=}1\}$ of $X{=}Z$ are also solutions of $and(X,X,Z)$.*
Now, let R be the propagation rule $and(X,Y,Z),\ neg(X,Y) \Rightarrow Z{=}0$.
There are three possibilities to transform R into a simplification rule:

$$and(X,Y,Z),\ neg(X,Y) \Leftrightarrow Z{=}0.$$
$$and(X,Y,Z),\ neg(X,Y) \Leftrightarrow and(X,Y,Z),\ Z{=}0.$$
$$and(X,Y,Z),\ neg(X,Y) \Leftrightarrow neg(X,Y),\ Z{=}0.$$

Let us consider each of them.

(1) R cannot be transformed into the simplification rule

$$and(X,Y,Z),\ neg(X,Y) \Leftrightarrow Z{=}0.$$

since $\{X{=}0, Y{=}0, Z{=}0\}$ *is a solution of* $Z{=}0$ *but not of* $and(X,Y,Z),\ neg(X,Y)$.

(2) *Furthermore,* R *cannot be transformed into the simplification rule*

$$and(X,Y,Z),\ neg(X,Y)\ \Leftrightarrow\ and(X,Y,Z),\ Z{=}0.$$

since here again $\{X{=}0, Y{=}0, Z{=}0\}$ *is a solution of* $and(X,Y,Z),\ Z{=}0$ *but not of* $and(X,Y,Z),\ neg(X,Y)$.

(3) *All solutions of* $neg(X,Y),\ Z{=}0$ *are solutions of* $and(X,Y,Z),\ neg(X,Y)$, *thus* R *can be transformed into the simplification rule*

$$and(X,Y,Z),\ neg(X,Y)\ \Leftrightarrow\ neg(X,Y),\ Z{=}0.$$

Thus, this transformation is accepted and we proceed with the next propagation rule. Finally, the final program consists of the simplification rules with one atomic constraint in their lhs *for the Boolean conjunction and negation, and of the following rules for their interaction:*

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

The SimpMiner algorithm is given in Figure 2. It takes as input a set S of valid propagation rules, and the domain *Dom* over which the atomic constraints are defined.

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 R into a simplification rule R' 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. Note that transforming a propagation rule into a simplification rule just by duplicating the whole *lhs* of the propagation rule into the *rhs* of the simplification rule is not allowed since E must be a proper subset of C.

The following result establishes correctness of the SimpMiner algorithm.

Theorem 3.1. *Let* S *be a set of valid propagation rules. The* SimpMiner *algorithm applied on* S *computes a set of valid propagation and simplification rules.*

Proof. *Let* R *be a propagation rule of the form* $C \Rightarrow D$ *in* S *that has been transformed into the simplification rule* $C \Leftrightarrow D \cup E$, *where* E *is a proper subset of* C.

According to the SimpMiner *algorithm, any solution* σ *of* $D \cup E$ *is a solution of* C. *Thus* $D \cup E \Rightarrow C$ *is a valid propagation rule.*

Since $C \Rightarrow D$ *is a valid propagation rule and* E *is subset of* C, *the propagation rule* $C \Rightarrow D \cup E$ *is trivially valid. Therefore, the simplification rule* $C \Leftrightarrow D \cup E$ *is valid.*

□

begin

$S' := \emptyset$
for each rule R of the form $C \Rightarrow D$ in S **do**
If D is false
then $S' := S' \cup \{R\}$
else Find $R' := C \Leftrightarrow D \cup E$, where E is a proper subset of C such that
for all ground substitutions σ of the variables in R' over Dom
if σ is a solution of $D \cup E$ then σ is a solution of C.
If R' exists
then $S' := S' \cup \{R'\}$
else $S' := S' \cup \{R\}$
endif
endif
endfor

output S'

end

Fig. 2. The SimpMiner Algorithm

SimpMiner is implemented in SICStus Prolog 3.7.1 in 70 lines of Prolog. As for PropMiner, the complexity of the algorithm can be expressed wrt. the number of ground decidability tests used. Let $lhsSize$ be the greatest number of atomic constraints appearing in the *lhs* of a rule in S and $nbVar$ be the greatest number of variables of a rule in S then in the worst case the number of ground decidability tests performed is in the order of $O(|S|\ 2^{lhsSize}\ Dom^{nbVar})$.

This theoretical complexity is high, but in practice this algorithm is used as a post-processing step on rules generated by PropMiner. So $nbVar$ remains rather small since by construction all variables used in a rule produced by PropMiner are variables appearing in the parameters called $Base_{lhs}$. Moreover, when the algorithm considers a substitution σ such that σ is a solution of $D \cup E$ but which is not a solution of C then there is no need to enumerate other substitutions and then another E subset of C can be tried. So in practice, for a given subset E, only a few substitutions among the Dom^{nbVar} possible ones are tested, and the execution times in the application presented in Section 4 remain rather short.

Finally, if we consider the impact of $lhsSize$, as for PropMiner, Section 4 shows that interesting rules can be found even if we restrict the generation to *lhs* containing only a few atomic constraints. Additionally, it should be noticed that in practice rules having many atomic constraints in their *lhs* are not likely to be very useful for constraint solving, since a long *lhs* leads in general to an important triggering detection cost as clearly reported in the experiments presented in Section 4.

4. AN APPLICATION OF CONSTRAINT SOLVER GENERATION

In this section, we first show that when we use the propagation rules generated by the PropMiner algorithm in a Constraint Logic Programming approach we benefit of a significant search space reduction but also pay a significant overhead due to the rule triggering process. In many cases this overhead leads to an important increase of the execution time even though the search space is greatly reduced.
Then, we show that the transformation of propagation rules into simplification rules as presented in Section 3 can be used to obtain the same search space reduction but with less rule triggering overhead. In this case, the experiments show a significant reduction of the execution time.
The application we consider is in the field of digital circuit design: automatic test-pattern generation. Test generation is the process of defining the tests to apply to a circuit in order to detect faults. Among the possible faults in a circuit composed of boolean gates, a very important type of faults is the *stuck-at faults*. A stuck-at fault occurs when the output value of a gate remains constant, i.e. the output value does not change while the input values are modified.
If we consider a gate in a circuit, we cannot access directly the input and the output of the gate to test it. So we must find a way to perform the test using only the input and output pins of the whole circuit. The problem is first to find what signal should be applied on the input of the circuit so that the output of the gate of interest will change (if there is no fault). This is called the *control problem*. Secondly, we must determine how to observe the effect of that change on the output pins of the whole circuit. This is called the *observation problem*.
Several proposals have shown that constraint logic programming allows a simple and declarative formulation of the test generation and leads to an efficient solving process. In the following, we use the constraint logic programming approach for automatic test-pattern generation proposed in [van Hentenryck et al. 1992]. We shortly present this method below and then describe our experiments.

4.1 Automatic test-pattern generation

In this section, we briefly recall the approach of [van Hentenryck et al. 1992] and refer the reader to the original paper for a detailed description. Van Hentenryck et al. defined a specific six-valued logic and provided some rules expressed in the form of so-called demons to carry out the constraint propagation.
Each line in the circuit is associated with a variable constrained to take one of the six possible values. The primary inputs are constrained to be 0 or 1. The four other values, denoted by $d, \overline{d}, e$ and $\overline{e}$, are needed to materialize the propagation paths from the output of the gate of interest to the output of the whole circuit (the observation problem). For example, the boolean value 1 at the output of the gate of interest will not be propagated through the circuit as a 1 but as a symbolic value denoted by d to materialize the path from the gate of interest towards the output pins of the whole circuit.
Van Hentenryck et al. used rules generated by hand to propagate input and output values of the gates within the circuit. Such a rule is for example: if the input arguments of an *and* gate are d and 1 then the output argument is d. The intuitive meaning of this rule is the following: if the output value of the gate of interest

(materialized by d) reaches the input of an *and* gate having a 1 as second input, then the output value of the gate of interest is propagated through this *and* gate. The triggering of the rules is combined with a systematic labeling in a general *constraint and generate* search, commonly used in constraint logic programming.

4.2 Experiments

We consider the problem of finding all possible ways to test each gate in a 4-bit adder. Let $4BitAdder(X_3, X_2, X_1, X_0, Y_3, Y_2, Y_1, Y_0, V, Z_3, Z_2, Z_1, Z_0)$ represents the relation between the input and output of the circuit, encoded as follows:

—(X_3, X_2, X_1, X_0) and (Y_3, Y_2, Y_1, Y_0) are the two 4-bit binary numbers to be added and (V, Z_3, Z_2, Z_1, Z_0) is the result.

—The 4-bit adder can be implemented by the following simplification rule:

$$\begin{aligned} 4BitAdder(X_3, X_2, X_1, X_0, Y_3, Y_2, Y_1, Y_0, V, Z_3, Z_2, Z_1, Z_0) \;\Leftrightarrow\; & \\ fullAdder(X_0, Y_0, 0, Z_0, C_0), & \\ fullAdder(X_1, Y_1, C_0, Z_1, C_1), & \\ fullAdder(X_2, Y_2, C_1, Z_2, C_2), & \\ fullAdder(X_3, Y_3, C_2, Z_3, C_3), & \\ xor(C_2, C_3, V). & \end{aligned}$$

where $fulladder$ corresponds to the full-adder defined in Section 2.6.2 and containing itself 5 Boolean gates. We consider a 4-bit adder circuit built according to this specification and thus using 21 boolean gates.

We present the results of three experiments. Each experiment is performed with a different set of rules. In each experiment, we consider in turn each of the 21 gates in the circuit. For each gate, we find all possible ways to test the gate for stuck-at faults and record two different measures: the size of the search space that has been explored and the execution time. The size of the search space is measured using the number of backtracks made by the labeling predicate (i.e., the number of variable assignments that the program made to find all solutions). The exploration of the search space is implemented in SICStus Prolog 7.3.1 and the propagation and simplification rules are encoded as CHR rules. The execution time is the CPU time used on a Pentium 3 with 256 MBytes of memory and a 500 MHz processor. It should be noticed that the platform SICStus with CHR, as most constraint logic programming systems, offers a very efficient propagation and querying of equality constraints and coreferences. In the case of a constraint programming systems that do not support coreferences and do not handle equality constraints in a specific way the results presented in this section are likely to change. For example, suppose that the coreferences in the left hand side of a rule must be made explicitly by means of equality constraints and that these equalities are handled like any other binary constraints. Then we can expect that the triggering cost of a rule will increase when compared to its triggering cost on a constraint logic programming system as used in this section.

The set of rules used for the first experiment contains only propagation rules generated by the PropMiner algorithm and having a single atom in their *lhs*. This

generation has been realized in 13.4 seconds using the truth tables of the operators *and*, *or* and *xor* in the six-valued logic of Van Hentenryck et al., and allowing equalities and disequalities in the *rhs* of the rules. Examples of rules are:

$$\begin{aligned}
and(X,1,Z) &\Rightarrow X{=}Z.\\
and(X,0,Z) &\Rightarrow X{\neq}\overline{d},\ X{\neq}d,\ Z=0.\\
or(X,Y,Y) &\Rightarrow X{\neq}\overline{d},\ X{\neq}d.\\
or(X,X,Z) &\Rightarrow X{\neq}\overline{d},\ X{\neq}d,\ X=Z.\\
xor(X,Y,Y) &\Rightarrow X{=}0.\\
xor(d,1,Z) &\Rightarrow Z{=}\overline{d}.
\end{aligned}$$

These rules have been generated by PropMiner using the parameters

$$\begin{aligned}
Base_{lhs} &= \{C(X,Y,Z)\}, \text{where } C \text{ was one of the operators } and,\ or,\ xor\\
Cand_{lhs} &= atomic(=,\{X,Y,Z\},\{X,Y,Z,0,1,d,\overline{d},e,\overline{e}\})\\
Cand_{rhs} &= atomic(\neq,\{X,Y,Z\},\{0,1,d,\overline{d},e,\overline{e}\}) \cup\\
&\quad\ atomic(=,\{X,Y,Z\},\{X,Y,Z,0,1,d,\overline{d},e,\overline{e}\})
\end{aligned}$$

The rules used by Van Hentenryck et al. are slightly different. They can be seen as propagation rules with the same form of *lhs* but with equality constraints, domain constraints (e.g., $X \in \{0,1,d,e\}$) and also non-membership constraints (e.g., $X \notin \{0,1,d,e\}$) in their *rhs*. However, non-membership constraints can be trivially encoded as a conjunction of disequalities, and when the domain is finite, domain constraints can also be expressed by a finite conjunction of disequalities (e.g., $X \in \{0,1,d,e\}$ is transformed into $\{X{\neq}\overline{d},\ X{\neq}\overline{e}\}$).
Thus using the correctness of PropMiner (Theorem 2.3), we conjecture that for each rule R used by Van Hentenryck et al. (where non-membership and domain constraints have been encoded as conjunction of inequalities) there exists in the set of rules used in our first experiment a rule R' such that if the *lhs* of R is satisfied then the *lhs* of R' is also satisfied and the *rhs* of R' is a constraint at least as strong as the *rhs* of R (in some sense R' can be used to perform the same kind of propagations as R).
When exploiting the symmetry of the ternary operators with respect to the first and second arguments (e.g., $and(X,Y,Z)$ is equivalent to $and(Y,X,Z)$), the total number of rules[2] generated is 69. It should be noticed that it would have been a hard work to produce this set manually, even-though such a generation by hand remains possible (as it has been done by Van Hentenryck et al.).
In the second experiment, we used also automatically generated propagation rules, but we take rules with one or two atoms in their *lhs*. In our experiments, we only considered interaction between conjunction, disjunction and exclusive disjunction.

[2]In all experiments mentioned in this section, this kind of symmetry has been used for the three operators *and*, *or* and *xor*. The reduced sets of rules have been obtained by adding rules expressing this symmetry (e.g., $and(X,Y,Z) \Rightarrow and(Y,X,Z)$) during the third step of the simplification process presented in Section 2.4. For example the set $\{and(X,Y,Z) \Rightarrow and(Y,X,Z), and(0,Y,Z) \Rightarrow Z{=}0, and(X,0,Z) \Rightarrow Z{=}0\}$ is reduced to $\{and(X,Y,Z) \Rightarrow and(Y,X,Z), and(0,Y,Z) \Rightarrow Z{=}0\}$.

gate number	experiment 1 search space	experiment 1 CPU time	experiment 2 search space	experiment 2 CPU time	search space Δ abs.	search space Δ %	CPU time Δ abs.	CPU time Δ %
1	286	1.41	265	12.55	-21	-7.34	+11.14	+790.07
2	4552	43.14	1728	35.58	-2824	-62.04	-7.56	-17.52
3	264	1.84	165	6.06	-99	-37.50	+4.22	+229.35
4	4552	43.45	1728	29.92	-2824	-62.04	-13.53	-31.14
5	286	1.52	205	5.44	-81	-28.32	+3.92	+257.89
6	796	7.38	490	36.96	-306	-38.44	+29.58	+400.81
7	5044	46.56	1765	41.81	-3279	-65.01	-4.75	-10.20
8	810	9.00	357	19.79	-453	-55.93	+10.79	+119.89
9	11186	129.23	1904	40.64	-9282	-82.98	-88.59	-68.55
10	1264	14.79	495	29.72	-769	-60.84	+14.93	+100.95
11	2488	35.57	1014	123.81	-1474	-59.24	+88.24	+248.07
12	6008	47.05	2465	125.60	-3543	-58.97	+78.55	+166.95
13	3032	47.87	773	65.57	-2259	-74.51	+17.70	+36.98
14	11760	128.82	1944	46.17	-9816	-83.47	-82.65	-64.16
15	4368	69.95	1087	99.77	-3281	-75.11	+29.82	+42.63
16	5640	85.03	1716	86.35	-3924	-69.57	+1.32	+1.55
17	7080	61.02	2681	103.15	-4399	-62.13	+42.13	+69.04
18	5364	112.36	1539	80.01	-3825	-71.31	-32.35	-28.79
19	7104	135.27	1994	82.91	-5110	-71.93	-52.36	-38.71
20	7728	149.57	2079	94.85	-5649	-73.10	-54.72	-36.58
21	12516	192.11	2037	93.35	-10479	-83.72	-98.76	-51.41

Table I. Search space reduction with overhead (time in seconds).

The corresponding rules have been generated in 1218 seconds. Example of rules are:

$$and(X,Y,0),\ or(Z,Y,X) \ \Rightarrow\ X \neq \overline{d},\ X \neq d,\ X{=}Z,\ Y{=}0.$$
$$and(X,Y,Z),\ xor(Z,1,Y) \ \Rightarrow\ X{=}0,\ Y{=}1,\ Z{=}0.$$

Even when exploiting the symmetry of the ternary operators this set consists of 613 rules, and cannot reasonably be generated by hand.
Then, a third experiment has been made using the 613 rules of the second experiment transformed into simplification rules, when applicable, using SimpMiner. 301 propagation rules have been transformed into simplification rules in 64.2 seconds. For example, the two previous rules have been transformed into

$$and(X,Y,0),\ or(Z,Y,X) \ \Leftrightarrow\ X \neq \overline{d},\ X \neq d,\ X{=}Z,\ Y{=}0.$$
$$and(X,Y,Z),\ xor(Z,1,Y) \ \Leftrightarrow\ X{=}0,\ Y{=}1,\ Z{=}0.$$

The comparison between experiment 1 and experiment 2 is given in Table I. For each gate the table gives the following information: size of search space (number of backtracks) and CPU execution time (in seconds) for the first experiment, size

gate number	experiment 1		experiment 3		search space		CPU time	
	search space	CPU time	search space	CPU time	Δ abs.	Δ %	Δ abs.	Δ %
1	286	1.41	265	2.88	-21	-7.34	+1.47	+104.26
2	4552	43.14	1728	9.77	-2824	-62.04	-33.37	-77.35
3	264	1.84	165	1.43	-99	-37.50	-0.41	-22.28
4	4552	43.45	1728	8.44	-2824	-62.04	-35.01	-80.58
5	286	1.52	205	1.35	-81	-28.32	-0.17	-11.18
6	796	7.38	490	8.45	-306	-38.44	+1.07	+14.50
7	5044	46.56	1765	7.37	-3279	-65.01	-39.19	-84.17
8	810	9.00	357	4.78	-453	-55.93	-4.22	-46.89
9	11186	129.23	1904	12.67	-9282	-82.98	-116.56	-90.20
10	1264	14.79	495	7.19	-769	-60.84	-7.60	-51.39
11	2488	35.57	1014	31.56	-1474	-59.24	-4.01	-11.27
12	6008	47.05	2465	26.73	-3543	-58.97	-20.32	-43.19
13	3032	47.87	773	17.29	-2259	-74.51	-30.58	-63.88
14	11760	128.82	1944	14.95	-9816	-83.47	-113.87	-88.39
15	4368	69.95	1087	27.62	-3281	-75.11	-42.33	-60.51
16	5640	85.03	1716	23.35	-3924	-69.57	-61.68	-72.54
17	7080	61.02	2681	23.90	-4399	-62.13	-37.12	-60.83
18	5364	112.36	1539	26.38	-3825	-71.31	-85.98	-76.52
19	7104	135.27	1994	27.69	-5110	-71.93	-107.58	-79.53
20	7728	149.57	2079	28.67	-5649	-73.10	-120.90	-80.83
21	12516	192.11	2037	30.69	-10479	-83.72	-161.42	-84.02

Table II. Reduction of search space and of execution time (in seconds).

of search space and CPU execution time for the second experiment, variation of the size of the search space from the first to the second experiment (absolute variation, Δ abs., and relative variation in percent, Δ %), and finally variation of the CPU execution time from the first to the second experiment (absolute and relative variations). This table shows that in this application, the propagation rules with one or two atoms in their *lhs* generated automatically (experiment 2) can be used to greatly reduce the size of the search space compared to the search space explored using propagation rules with a single atom in their *lhs* (experiment 1). Unfortunately, the table shows also that in several cases, we should pay for a very important overhead in terms of execution time to handle these more complex rules. The comparison between experiment 1 and experiment 3 given in Table II shows that this overhead can be suppressed if we transform the set of propagation rules to a set of propagation and simplification rules using SimpMiner. Moreover, in nearly all cases, the execution time is reduced by more than 50%. Only a very small absolute overhead remains for gates 1 and 6.

Additionally, it is also interesting to evaluate the benefit of using a set of rules simplified using SimpMiner vs. the original set of rules, and to assess the impact of using longer rule *lhs*. The corresponding results are presented in Table III, where experiment 4 is the same as experiment 1, excepted that the 69 propagation rules

have been transformed when applicable into simplification rules using SIMPMINER (41 propagation rules have been transformed in 0.35 second). Let us denote t_i the execution time for experiment i. The relative variation from t_1 to t_4 gives the difference of execution time when using the set of propagation rules containing a single atom in their *lhs* and when using the same set transformed with SIMPMINER. For the set of rules having one or two atoms in their *lhs* this difference is given by the relative variation from t_2 to t_3. In both cases, the measures show that it is advantageous to transform propagation rules into simplification rules when applicable. A complementary aspect of interest is the impact of the length of the rule *lhs*. It is obvious (as shown in the results presented in Table I) that when a set of propagation rules having a single atom in their *lhs* (e.g., the set of experiment 1) is complete with rules where the *lhs* contains two atoms (e.g., the set of experiment 2) then more propagation is made during constraint solving, leading to an additional reduction of the search space. The puzzling point is the global influence of the length of the *lhs* on the execution time. More precisely, if we consider the sets of rules of experiments 3 and 4, the following question arise: can the additional reduction of the search space caused by the rules of experiment 3 compensate the triggering overhead due to rules with two atoms in their *lhs*? Some elements are given in Table III, where the relative variations of t_4 to t_3 indicates that in most cases the tradeoff is clearly in favor of the use of the set of rules that incorporates rules having one or two atoms in their *lhs*.

5. RELATED WORK

In the pioneering work [Apt and Monfroy 1999; 2001], two algorithms have been proposed and implemented: One generates equality rules, whereas the second one generates membership rules. In Section 2.3, we have shown that the PROPMINER algorithm is able to generate propagation rules ensuring at least the same kind of consistency.

For example, let $c1$ be the constraint defined in [Ringeissen and Monfroy 2000] by the following truth table:

$c1$	X_1	X_2	X_3
0	0	0	0
1	0	0	1
0	0	1	0
0	0	1	1
0	1	0	0
0	1	0	1
0	1	1	0
1	1	1	1

For the constraint $c1(X_1, X_2, X_3)$ defined above the following equality rules are generated by the algorithm presented in [Apt and Monfroy 1999; 2001]:

$$\begin{aligned} c1(X_1, X_2, X_3) &\Rightarrow X_3 \neq 0 \\ c1(X_1, X_2, X_3),\ X_1 = 1 &\Rightarrow X_2 \neq 0 \\ c1(X_1, X_2, X_3),\ X_1 = 0 &\Rightarrow X_2 \neq 1 \end{aligned}$$

gate	CPU time t_i of experiment i				relative variation in % from		
number	t_1	t_4	t_2	t_3	t_1 to t_4	t_2 to t_3	t_4 to t_3
1	1,41	1,21	12,55	2,88	-14,18	-77,052	+138,02
2	43,14	34,54	35,58	9,77	-19,94	-72,541	-71,71
3	1,84	1,53	6,06	1,43	-16,85	-76,403	-6,54
4	43,45	34,67	29,92	8,44	-20,21	-71,791	-75,66
5	1,52	1,29	5,44	1,35	-15,13	-75,184	+4,65
6	7,38	6,06	36,96	8,45	-17,89	-77,137	+39,44
7	46,56	37,1	41,81	7,37	-20,32	-82,373	-80,13
8	9,0	7,33	19,79	4,78	-18,56	-75,846	-34,79
9	129,23	103,62	40,64	12,67	-19,82	-68,824	-87,77
10	14,79	12,14	29,72	7,19	-17,92	-75,808	-40,77
11	35,57	29,06	123,81	31,56	-18,30	-74,509	+8,60
12	47,05	36,67	125,6	26,73	-22,06	-78,718	-27,11
13	47,87	39,17	65,57	17,29	-18,17	-73,631	-55,86
14	128,82	102,76	46,17	14,95	-20,23	-67,62	-85,45
15	69,95	59,7	99,77	27,62	-14,65	-72,316	-53,74
16	85,03	73,32	86,35	23,35	-13,77	-72,959	-68,15
17	61,02	48,01	103,15	23,9	-21,32	-76,83	-50,22
18	112,36	93,44	80,01	26,38	-16,84	-67,029	-71,77
19	135,27	111,66	82,91	27,69	-17,45	-66,602	-75,20
20	149,57	124,14	94,85	28,67	-17,00	-69,773	-76,91
21	192,11	158,67	93,35	30,69	-17,41	-67,124	-80,66

Table III. Impact of the form of the rules on the execution time (in seconds).

$$c1(X_1, X_2, X_3),\ X_2 = 1 \ \Rightarrow\ X_1 \neq 0$$
$$c1(X_1, X_2, X_3),\ X_2 = 0 \ \Rightarrow\ X_1 \neq 1$$

For the constraint $c1$ our algorithm generates the following single propagation rule, if the user specifies that the *rhs* of the rules may consist of equality constraints:

$$c1(X_1, X_2, X_3) \ \Rightarrow\ X_1 = X_2,\ \ X_3 = 1$$

The algorithm presented in [Ringeissen and Monfroy 2000] is a combination of the one described in [Apt and Monfroy 1999; 2001] and unification in finite algebra. Similar to [Apt and Monfroy 1999] the user has here no possibility to specify the form of the rules. The rules generated by this algorithm have the following form:

$$C(X_1, \ldots, X_n), X_i = v_i,\ \ldots,\ X_k = v_k \ \Rightarrow\ B,$$

where now $v_i, \ldots v_k$ are either elements of the domain or free constants to represent symbolically any element of the domain as used in unification in finite algebra [Kirchner and Ringeissen 1992]. B is a a set of of equality constraints and membership constraints (e.g. $X \in D$).
With the notion of free constants, equality between variables in the *rhs* of rules can be deduced. For the constraint $c1(X_1, X_2, X_3)$, the algorithm presented in [Ringeis-

sen and Monfroy 2000] generates the following rules:

$$\begin{aligned} c1(X_1, X_2, X_3) &\Rightarrow X_1{\in}\{0,1\},\ X_2{\in}\{0,1\},\ X_3{=}1 \\ c1(X_1, X_2, X_3),\ X_1 = x_1 &\Rightarrow X_2{=}x_1,\ X_3{=}1 \\ c1(X_1, X_2, X_3),\ X_2 = x_2 &\Rightarrow X_1{=}x_2,\ X_3{=}1 \end{aligned}$$

In contrast to the algorithms presented in [Apt and Monfroy 1999] and [Ringeissen and Monfroy 2000] our algorithm leads to a more compact and more expressive set of rules. With the rules generated by the algorithm presented in [Apt and Monfroy 1999], one deduces from $c1(X_1, X_2, X_3)$ that $X_3{=}1$. With our generated rule we also deduce that $X_1{=}X_2$. This can also be deduced from the rules generated by the algorithm presented in [Ringeissen and Monfroy 2000]. But if we consider the constraint $c2$ defined in [Ringeissen and Monfroy 2000] by the tuples $\{(0,1,0),(0,1,1),(1,0,0),(1,1,0)\}$, our algorithm can generate the rule $c2(X_1, X_1, X_3) \Rightarrow X_1{=}1,\ X_3{=}0$, that cannot be produced by the algorithms presented in [Apt and Monfroy 1999] and in [Ringeissen and Monfroy 2000], since the rule contains a coreference in its *lhs*.
Furthermore, in contrast to the algorithms presented in [Apt and Monfroy 1999] and [Ringeissen and Monfroy 2000] our algorithm is able to generate rules with a set of atomic constraints in the *lhs* of the rules which is an essential feature for non-trivial constraint handling (e.g., conjunction of *and* with *neg* used in Section 4). Another particularly useful aspect of our approach wrt. these other works, is the improvement of the solver produced, by transforming when possible the propagation rules into simplification rules. Even if this transformation comes with additional cost, in practice a solver can be used to solve different problems and thus its efficiency is crucial.

The generation of rule-based constraint solvers can also be related to the work in Inductive Logic Programming [Muggleton and De Raedt 1994], where one is interested to find out logic programs from examples. However, the generation of constraint solvers has its own specificities, that are mainly the notion of *interesting rule* to built a solver and the discovery of simplification rules (a critical performance issue for the resulting solver as shown in Section 4), and to our knowledge the work done in Inductive Logic Programming have not yet been adapted or applied to the generation of constraint solvers.

6. CONCLUSION

We have presented a method for generating rule-based constraint solvers for finite constraints given their extensional representation. The generation is performed in two steps. In a first step, propagation rules are generated using an algorithm called PropMiner. Compared to the algorithms described in [Apt and Monfroy 1999; 2001] and [Ringeissen and Monfroy 2000] the algorithm PropMiner is able to generate more general and more expressive rules. In a second step, propagation rules are transformed when possible into simplification rules using an algorithm called SimpMiner to improve the efficiency of the constraint solving.
The two generation steps have been illustrated on various examples of commonly used constraints. Finally, we have considered a non trivial application in the domain

of digital circuit design, and we have compared the rules generated by the two steps to the rules one can reasonably expect to write by hand. The experiments shown that the set of rules generated by the method presented in this paper performed additional propagations and new simplifications. The use of these rules provides a significant reduction of the search space and of the execution time, when compared to the use of the rules written by hand.
Two additional interesting aspects of the method are firstly that it has very weak requirements wrt. the constraint theory (the theory must simply be ground decidable), and secondly that it produces simplified and readable sets of rules. Its main limitation is that it is dedicated to the generation of rule-based solvers for constraints defined extensionally over finite domains. A promising direction of future research is the generation of solvers for constraints defined intensionally eventually over non finite domains. A first preliminary step in this direction has recently been proposed in [Abdennadher and Rigotti 2001a].

ACKNOWLEDGMENTS

We are grateful to the Bavarian-French cooperation center for its support. We would like to thank the anonymous referees for their constructive remarks.

REFERENCES

Abdennadher, S. and Rigotti, C. 2000. Automatic generation of propagation rules for finite domains. In *Proc. of the 6th International Conf. on Principles and Practice of Constraint Programming*. LNCS 1894. Springer-Verlag, Singapore, 18–34.

Abdennadher, S. and Rigotti, C. 2001a. Towards inductive constraint solving. In *Proc. of the 7th International Conf. on Principles and Practice of Constraint Programming*. LNCS 2239. Springer-Verlag, Paphos, Cyprus, 31–45.

Abdennadher, S. and Rigotti, C. 2001b. Using confluence to generate rule-based constraint solvers. In *Proc. of the 3rd International Conf. on Principles and Practice of Declarative Programming*. ACM Press, Firenze , Italy, 127–135.

Abdennadher, S. and Saft, M. Constraint Handling Rules online, `http://www.pms.informatik.uni-muenchen.de/~webchr/`

Agrawal, R., Imielinski, T., and Swami, A. N. 1993. Mining association rules between sets of items in large databases. In *Proc. of the ACM SIGMOD International Conf. on Management of Data*. ACM Press, Washington, D.C., USA, 207–216.

Allen, J. F. 1983. Maintaining knowledge about temporal intervals. *Commun. ACM 26,* 11, 832–843.

Apt, K. R. 1998. A proof theoretic view of constraint programming. *Fundamenta Informaticae 34*, 295–321.

Apt, K. R. 2000. Some remarks on boolean constraint propagation. In *New Trends in Constraints*. LNAI 1865. Springer-Verlag, 91–107.

Apt, K. R. and Monfroy, E. 1999. Automatic generation of constraint propagation algorithms for small finite domains. In *Proc. of the 5th International Conf. on Principles and Practice of Constraint Programming*. LNCS 1713. Springer-Verlag, Alexandria, Virginia, USA, 58–72.

Apt, K. R. and Monfroy, E. 2001. Constraint programming viewed as rule-based programming. *Theory and Practice of Logic Programming 1,* 6, 713–750.

Bayardo, R. J., Agrawal, R., and Gunopulos, D. 1999. Constraint-based rule mining in large, dense databases. In *Proc. of the 15th International Conf. on Data Engineering*. IEEE Computer Society, Sydney, Austrialia, 188–197.

Caseau, Y., Josset, F.-X., and Laburthe, F. 1999. Claire: Combining sets, search, and rules to better express algorithms. In *Proc. of the 16th International Conf. on Logic Programming*. The MIT Press, Las Cruces, New Mexico, USA, 245–259.

Codognet, P. and Diaz, D. 1993. Boolean constraint solving using clp(FD). In *Proc. of the International Symp. on Logic Programming*. The MIT Press, Vancouver, Canada, 525–539.

Dincbas, M., Van Hentenryck, P., Simonis, H., Aggoun, A., Graf, T., and Berthier, F. 1988. The constraint logic programming language CHIP. Technical Report TR-LP-37, ECRC, Munich, Germany.

Frühwirth, T. 1998. Theory and practice of constraint handling rules. *Journal of Logic Programming 37,* 1-3, 95–138.

Kirchner, C., Kirchner, H., and Vittek, M. 1993. Implementing computational systems with constraints. In *Proc. of the 1st Workshop on Principles and Practice of Constraints Programming*. MIT Press, Newport, Rode Island, USA, 156–165.

Kirchner, C. and Ringeissen, C. 1998. Rule-based constraint programming. *Fundamenta Informaticae 34*, 225–262.

Kirchner, H. and Ringeissen, C. 1992. A constraint solver in finite algebras and its combination with unification algorithms. In *Proc. of the Joint International Conf. and Symp. on Logic Programming*. MIT Press, Washington, DC, USA, 225–239.

Kleene, S. 1950. *Introduction to Metamathematics*. Van Nostrand, Princeton, New Jersey.

Muggleton, S. and 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. In *Machine Intelligence*. Vol. 5. Edinburgh University Press, 153–163.

Ringeissen, C. and Monfroy, E. 2000. Generating propagation rules for finite domains via unification in finite algebra. In *New Trends in Constraints*. LNAI 1865. Springer-Verlag, 150–172.

Toivonen, H., Klemettinen, M., Ronkainen, P., Hätönen, K., and Mannila, H. 1995. Pruning and grouping of discovered association rules. In *Proc. of the ECML Workshop on Statistics, Machine Learning, and Knowledge Discovery in Databases*. MLnet, Heraklion, Crete, Greece, 47–52.

van Hentenryck, P. 1991. Constraint logic programming. *The Knowledge Engineering Review 6,* 3, 151–194.

van Hentenryck, P., Simonis, H., and Dincbas, M. 1992. Constraint satisfaction using constraint logic programming. *Artificial Intelligence 58,* 1-3, 113–159.