Using Confluence to Generate Rule-based Constraint Solvers

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

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

ABSTRACT

A general approach to implement propagation and simplification of constraints consists of applying rules over these constraints. Recently, several proposals have been made to solve finite constraint satisfaction problems by means of automatically generated propagation rules [4, 9, 3]. Since propagation rules do not rewrite constraints but add new ones, 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. In this paper, we present a method to transform propagation rules into simplification rules using the concept of confluence. The generated rules are implemented in the language Constraint Handling Rules (CHR). An example taken from the field of digital circuit design shows that our approach is of practical use.

1. INTRODUCTION

Rule-based formalisms are ubiquitous in computer science, and even more so in constraint reasoning and programming. In constraint reasoning, algorithms are often specified using inference rules, rewrite rules, sequents, or first-order axioms written as implications. It is no wonder that recently there has been a revival of interest in rule-based programming in the context of constraint programming. Advanced programming languages, like ELAN [8],CLAIRE [5], and Constraint Handling Rules (CHR) [6], have shown that the concept of rule could be of major interest as a programming tool for constraint solvers.

A difficulty that arises frequently when writing a constraint solver is to determine the constraint propagation algorithm. Recently, several proposals have been made to solve finite constraint satisfaction problems by means of automatically generated constraint propagation algorithms [4, 9, 3]. The approach we have taken in [3] is to develop an automatic method to generate propagation rules defining some properties of constraints given their extensional definition. Using our method based on techniques used in the field of knowledge discovery the user has the possibility to specify the form of the rules she/he wants to generate. The method allows any kind of constraints on the left hand side of rules and on their right hand side as well.

Consider the following example, where the user wants to generate a constraint solver for the boolean conjunction denoted by $and(X, Y, Z)$, where X, Y are the input arguments and Z is the output argument. This ternary relation can be defined extensionally by the triples

$$\{(0,0,0),(0,1,0),(1,0,0),(1,1,1)\},$$

where 1 stands for truth and 0 for falsity.

The algorithm presented in [3] generates the following propagation rules provided the user specifies their left hand side to be the *and* constraint and their right hand side to be a conjunction of equality constraints:

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

For example, the first rule says that the constraint $and(X, Y, Z)$, when it is known that the first input argument X is equal to 0, can propagate the constraint that the output Z must be equal to 0. So, using this rule the goal $and(0, Y, Z)$ will be transformed into $and(0, Y, Z) \wedge Z{=}0$.

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. In general, removing constraints improves both the time and space behavior of constraint solving.

Often generated propagation rules can be rewritten as simplification rules. For the example presented above all propagation rules could be transformed into the following simplification rules:

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

Thus, our aim is to find some criteria to perform such a transformation. One simple criterion could be the following: whenever a right hand side of a rule propagates information making its left hand side 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 multi-headed propagation rules defining interaction between constraints, i.e. rules with left hand sides consisting of a conjunction of constraints. For example, for the negation constraint $neg(X,Y)$ defined by $\{(0,1),(1,0)\}$ and the conjunction constraint $and(X,Y,Z)$ our algorithm generates among other rules the following propagation rules:

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

It is obvious that the first two rules can be transformed into simplification rules. However, transforming the third rule to a simplification rule by simply replacing $\Rightarrow$ by $\Leftrightarrow$ leads to loss information about X and Y. In this case, we may remove the constraint $and(X,Y,Z)$, but we have to keep the negation $neg(X,Y)$ in the constraint store. This can be done by the following simplification rule:

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

In this paper, we propose a syntactical method to decide when and how to transform propagation rules into simplification rules. The method is based on the confluence notion. Confluence means that the result of a computation is independent from the order in which rules are applied to the constraints. The idea of the transformation method is to test the confluence of the resulting constraint solver after transforming a propagation rule into a simplification rule. We will show that if the resulting solver remains confluent then we can conclude that the transformation leads to an operationally equivalent constraint solver with respect to the built-in constraints. In this paper, we consider that the rules are expressed in the CHR language, and we use the decidable, sufficient and necessary condition for confluence of terminating CHR programs presented in [1, 2]. However, it should be noticed that our technique can be adapted to other rule-based programming languages.

The method presented in this paper could also be used when writing a rule-based constraint solver by hand, e.g. one can implement the propagation algorithm of constraints using only propagation rules and our method decides whether a propagation rule can be transformed into a simplification rule.

The paper is organized as follows: The next section defines a subset of the CHR language used in this paper and introduces a simplified version of our previous algorithm for the generation of propagation rules. In section 3, we present our method to transform propagation rules into simplification rules and give some properties of the transformation. In Section 4, we present an example to show the practical usefulness of our algorithm. Finally, we conclude with a summary.

2. PRELIMINARIES

The generated rules will be implemented in the language Constraint Handling Rules (CHR) [6]. In fact, we need only a small subset of CHR, i.e., rules without guards. In Section 2.1, we give an overview of syntax and semantics of this subset. In Section 2.2, we summarize previous confluence results. More detailed presentations can be found in [1, 2]. In Section 2.3, we describe a simplified version of the algorithm used to generate the propagation rules. A complete presentation of this algorithm can be found in [3].

2.1 Constraint Handling Rules

We use two disjoint kinds of predicate symbols for two different classes of constraints: *built-in constraint symbols* and *user-defined constraint symbols (CHR symbols).* We call an atomic formula with a constraint symbol a *constraint.* Built-in constraints are handled by a predefined, given constraint solver that already exists as a certified black-box solver. User-defined constraints are defined in a CHR program.

2.1.1 Syntax

A *guarded-free CHR program* is a finite set of rules. There are basically two kinds of rules:[1]

A *simplification rule* is of the form $H \Leftrightarrow B$ and a *propagation rule* is of the form $H \Rightarrow B$, where the *head* H is a non-empty conjunction of user-defined constraints and the *body* B is a goal. A *goal* is a conjunction of built-in and user-defined constraints.

2.1.2 Declarative Semantics

The logical meaning of a simplification rule $H \Leftrightarrow B$ is the logical equivalence $\forall(H \leftrightarrow \exists\bar{y}\ B)$, where $\bar{y}$ are the variables that appear only in the body B and $\forall F$ denotes the universal closure of a formula F.

The logical meaning of a propagation rule $H \Rightarrow B$ is the logical implication $\forall(H \rightarrow \exists\bar{y}\ B)$.

The logical meaning of a guarded-free CHR program P is the conjunction of the logical meanings of its rules united with a consistent *constraint theory* CT that defines the built-in constraint symbols. We require CT to define the constraint symbol $=$ as syntactic equality.

2.1.3 Operational Semantics

The operational semantics of CHR is given by a transition system. A state $G_{\mathcal{V}}$ is a goal G together with a sequence of variables, $\mathcal{V}$. Where it is clear from the context, we will drop the annotation $\mathcal{V}$.

We require that states are normalized so that they can be compared syntactically in a meaningful way. Basically, we require that the built-in constraints are in a (unique) normal form, where all syntactic equalities are made explicit and are propagated to the user-defined constraints. Furthermore,

[1] A third kind are simpagation rules. Since they are syntactic abbreviations for simplification rules there is no need to discuss them in this paper.

we require that the normalization projects out strictly local variables, i.e. variables appearing in the built-in constraints only. A precise definition of the normalization function $\mathcal{N}$ can be found in [1, 2].
Given a guarded-free CHR program P we define the transition relation $\mapsto_P$ by introducing two kinds of computation steps (Figure 1). In Figure 1, the notation G_{built} denotes the built-in constraints in a goal G. We will also use the notation G_{user} to denote the user-defined constraints in a goal G. An equation $c(t_1,\ldots,t_n)=d(s_1,\ldots,s_n)$ of two constraints is an abbreviation for $t_1=s_1 \wedge \ldots \wedge t_n=s_n$ if c and d are the same CHR symbols and for *false* otherwise. An equation $(p_1 \wedge \ldots \wedge p_n)=(q_1 \wedge \ldots \wedge q_m)$ stands for $p_1=q_1 \wedge \ldots \wedge p_n=q_n$ if $n = m$ and for *false* otherwise. Conjuncts can be permuted since conjunction is associative, commutative and idempotent.

Simplify

If	$(H \Leftrightarrow B)$ is a fresh variant of a rule with variables $\bar{x}$
and	$CT \models \forall\ (G_{built} \rightarrow \exists\bar{x}(H{=}H'))$
then	$(H' \wedge G)_{\mathcal{V}} \mapsto_P \mathcal{N}((H{=}H' \wedge B \wedge G)_{\mathcal{V}})$

Propagate

If	$(H \Rightarrow B)$ is a fresh variant of a rule with variables $\bar{x}$
and	$CT \models \forall\ (G_{built} \rightarrow \exists\bar{x}(H{=}H'))$
then	$(H' \wedge G)_{\mathcal{V}} \mapsto_P \mathcal{N}((H{=}H' \wedge B \wedge H' \wedge G)_{\mathcal{V}})$

Figure 1: Computation Steps of guarded-free CHR Programs

To **Simplify** user-defined constraints H' means to remove them from the state $H' \wedge G$ and to add the body B of a fresh variant of a simplification rule $(H \Leftrightarrow B)$ and the equation $H{=}H'$ to the resulting state, provided H' matches the head H, and finally to normalize the resulting state. In this case we say that the rule R is *applicable to* H'. A variant of a formula is obtained by renaming its variables. Matching means that H' is an instance of H, i.e. it is only allowed to instantiate (bind) variables of H but not variables of H'. In the logical notation this is achieved by existentially quantifying only over the fresh variables $\bar{x}$ of the rule to be applied in the condition.
The **Propagate** transition is like the **Simplify** transition, except that it keeps the constraints H' in the state.
$\mapsto_P^+$ denotes the transitive closure, $\mapsto_P^*$ denotes the reflexive and transitive closure of $\mapsto_P$.
An *initial state* for a goal G is the state $\mathcal{N}(G_{\mathcal{V}})$, where $\mathcal{N}$ is a function that normalizes a state as defined above and $\mathcal{V}$ is a sequence of all variables appearing in G. A *final state* is one where either no computation step is possible anymore or where the built-in constraints are inconsistent.
A *computation* of a goal G in a program P is a sequence $S_0, S_1, \ldots$ of states with $S_i \mapsto_P S_{i+1}$ beginning with the initial state for G and ending in a final state or diverging. Note that trivial nontermination caused by applying the same propagation rule again and again is avoided by applying a propagation rule at most once to the same constraints. A more complex operational semantics that addresses this issue can be found in [1]. Where it is clear from the context, we will drop the reference to the program P.

2.2 Confluence

The confluence property of a program guarantees that any computation starting from an arbitrary initial state, i.e. any possible order of rule applications, results in the same final state. Due to space limitations, we can just give an overview on confluence results for guarded-free CHR programs, for details see [2, 1].

DEFINITION 2.1. *A CHR program is called* confluent *if for all states S, S_1, S_2:*
If $S \mapsto^* S_1$ and $S \mapsto^* S_2$ then S_1 and S_2 are joinable. Two states S_1 and S_2 are called joinable *if there exist states T_1 and T_2 such that $S_1 \mapsto^* T_1$ and $S_2 \mapsto^* T_2$ and T_1, T_2 are variants of each other.*

To analyze confluence of a given CHR program we cannot check joinability starting from any given ancestor state S, because in general there are infinitely many such states. However one can restrict the joinability test to a finite number of "minimal" states based on the following observations: First, adding constraints to a state cannot inhibit the application of a rule as long as the built-in constraints remain consistent. Hence, we can restrict ourselves to ancestor states that consist of the head of two rules. Second, joinability can only be destroyed if one rule inhibits the application of another rule. Only the removal of constraints can affect the applicability of another rule, in case the removed constraint is needed by the other rule. Hence, at least one rule must be a simplification rule and the two rules must *overlap*, i.e. have at least one head atom in common in the ancestor state. This is achieved by equating head atoms in the state.

DEFINITION 2.2. *Given a simplification rule R_1 and an arbitrary (not necessarily different) rule R_2, whose variables have been renamed apart. Let $H_i \wedge A_i$ be the head of rule R_i $(i = 1, 2)$. Then a* critical ancestor state of R_1 and R_2 *is*

$$(H_1 \wedge A_1 \wedge H_2 \wedge (A_1{=}A_2))_{\mathcal{V}},$$

provided A_1 and A_2 are non-empty conjunctions and $CT \models \exists((A_1{=}A_2))$.

The application of R_1 and R_2, respectively, to a critical ancestor state of R_1 and R_2 leads to two states that form the so-called *critical pair*.

DEFINITION 2.3. *Let S be a critical ancestor state of R_1 and R_2. If $S \mapsto S_1$ using rule R_1 and $S \mapsto S_2$ using rule R_2 then the tuple (S_1, S_2) is the* critical pair *of R_1 and R_2. A critical pair (S_1, S_2) is* joinable, *if S_1 and S_2 are joinable.*

DEFINITION 2.4. *A CHR program is called* terminating, *if there are no infinite computations.*

The following theorem from [1, 2] gives a decidable, sufficient and necessary condition for confluence of a terminating CHR program. Joinability of a given critical pair is decidable for a terminating CHR program and there are only finitely many critical pairs.

THEOREM 2.1. *A terminating CHR program is confluent iff all its critical pairs are joinable.*

It should be noticed that for most existing CHR programs it is straightforward to prove termination, e.g. using simple well-founded orderings [7].

2.3 Generation of Propagation Rules

In this section, we present a simplified version of our algorithm for generating propagation rules for finite constraints given their extensional representation. A full explanation of an efficient version of the algorithm is beyond the scope of this paper and the reader is referred to [3].

Using this algorithm, the user has the possibility to specify the admissible syntactic forms of the rules. The user determines the constraint for which rules have to be generated and chooses the candidate constraints to form conjunctions together with it in the left hand side. Usually, these candidate constraints are simply equality constraints. For the right hand side of the rules the user specifies also the form of candidate constraints she/he wants to see there. Finally, the user determines the semantics of the constraint for which rules have to be generated by means of its extensional definition which must be finite, and provides the semantics of the candidate constraints by a constraint theory. Furthermore, we assume that the constraints defined by this theory are handled by an appropriate constraint solver.

To compute the propagation rules the algorithm generates each possible left hand side constraint (noted C_{lhs}) and for each one computes the corresponding right hand side constraint (noted C_{rhs}) as follows:

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 conjunction of all atomic constraints that are candidates for the right hand side part and are true for all solutions of C_{lhs}. If C_{rhs} is not empty we have the rule $C_{lhs} \Rightarrow C_{rhs}$. Otherwise, no rule is generated.

EXAMPLE 2.1. *If the user specifies that the right hand sides of the rules may consist of a conjunction of equality constraints then for the the conjunction constraint $and(X,Y,Z)$ the algorithm generates the 6 rules presented in Section 1. For the negation constraint $neg(X,Y)$ the following propagation rules are generated:*

$$\begin{array}{rcl} neg(X,0) & \Rightarrow & X{=}1. \\ neg(0,Y) & \Rightarrow & Y{=}1. \\ neg(X,1) & \Rightarrow & X{=}0. \\ neg(1,Y) & \Rightarrow & Y{=}0. \\ neg(X,X) & \Rightarrow & false. \end{array}$$

When the user specifies that the left hand side can also contain a conjunction of an and constraint together with a neg constraint, then the following rules handling their interaction are generated:

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

3. GENERATION OF SIMPLIFICATION RULES

Propagation rules are a feature that is essential in solving constraints. Propagation rules add new constraints, which are logically redundant but may cause further simplification (e.g. $X{>}Y \wedge Y{>}Z \Rightarrow X{>}Z$). However, since propagation rules do not rewrite constraints, the constraint store may contain superfluous information. Constraints can be removed from the constraint store using simplification rules. In general, removing constraints improves both the time and space behavior of constraint solving. In this section, we present a syntactical method to decide when and how to transform propagation rules into simplification rules. The method is based on the confluence notion.

3.1 The GenSimpRules Algorithm

We now introduce the algorithm GenSimpRules that given a set of propagation rules P (called *initial program*) generates a set of both propagation and simplification rules (called *final program*). We assume that the CHR program consisting of these propagation rules is terminating. We can easily show that this program is confluent, since propagation rules do not rewrite constraints and adding constraints to a state cannot inhibit the application of a rule as long as the built-in constraints remain consistent. The algorithm works by repeatedly selecting a rule R from P. Then, we try to transform R to a simplification rule. For a propagation rule with multiple atoms in its head, there are several possibilities to transform it to a simplification rule. If the resulting program remains confluent then the transformation is accepted and the next rule will be considered. Otherwise, the next transformation of R is tried.

Before giving an abstract description of the GenSimpRules algorithm, we illustrate it by the following example:

EXAMPLE 3.1. *Let P be the set of propagation rules defining the negation and the conjunction constraints and the interaction between them (Example 2.1). Obviously, P is terminating, since only equality constraints appear in the right-hand side of the rules.*

All single-headed propagation rules, i.e. rules with one atom in their heads, can be transformed into simplification rules and the resulting program remains confluent. Now, let R be the propagation rule $and(X,Y,Z) \wedge neg(X,Y) \Rightarrow Z{=}0$. There are three possibilities to transform R into a simplification rule. If we transform R into the simplification rule $and(X,Y,Z) \wedge neg(X,Y) \Leftrightarrow Z{=}0$, then the resulting program becomes non-confluent since the critical pair

$$(S_1,\ S_2) := (Z{=}0,\ \ X{=}Z \wedge neg(X,X))$$

stemming from the critical ancestor state

$$and(X,Y,Z) \wedge neg(X,Y) \wedge X{=}Y$$

of the rule $and(X,Y,Z) \wedge neg(X,Y) \Leftrightarrow Z{=}0$ and the rule $and(X,X,Z) \Leftrightarrow X{=}Z$ is not joinable, i.e. S_1 is a final state and S_2 leads to false.

If we transform the propagation rule R into a simplification rule of the form

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

then the resulting program becomes also non-confluent.

But, transforming the propagation rule R into a simplification rule of the form

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

leads to a confluent program. Thus, this transformation is accepted and we proceed with the next propagation rule. Finally, the final program consists of the single-headed simplification rules defining the conjunction and negation constraints and of the following rules defining their interaction:

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

We now give an abstract description of the GenSimpRules algorithm. Notice that it uses a termination test of CHR programs and this test is in most cases straightforward using simple well-founded orderings [7].

Definition 3.1. *Let $\bigwedge_{i=1}^{n} H_i$ be a conjunction of constraints, π a permutation on $[1,\ldots,n]$, then $\bigwedge_{i=1}^{m} H_{\pi_i}$, where $0 \leq m < n$, is a* proper subconjunction *of $\bigwedge_{i=1}^{n} H_i$.*

GenSimpRules Algorithm

INPUT: A terminating CHR program consisting of propagation rules.
OUTPUT: A built-in operationally equivalent guarded-free CHR program P' consisting of propagation and simplification rules.
ALGORITHM:

$P' := P$
for each rule R of the form $H \Rightarrow B$ in P **do**
Find $R' := H \Leftrightarrow B \wedge C$, where C is a proper subconjunction of H such that
$(P'\backslash\{R\}) \cup \{R'\}$ is terminating and all its critical pairs are joinable.
If R' exists
then $P' := (P'\backslash\{R\}) \cup \{R'\}$
endif
endfor
return P'

In the GenSimpRules algorithm, there are two degrees of nondeterminism: the choice of a rule R from P and the choice of the subconjunction C of H. In our implementation, the input CHR program is given in a text file and we take the rules according to the order in this file. To handle the second source of nondeterminism and 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 $H \Leftrightarrow B \wedge C$ we choose the smallest conjunction of constraints C (wrt. the number of constraints in C) for which the resulting program remains confluent, i.e. terminating and all its critical pairs are joinable. If such a C is not unique, we choose any one among the smallest conjunctions. Note that transforming a propagation rule into a simplification rule just by shifting the whole head of the propagation rule into the body of the simplification rule is not allowed since C must be a proper subconjunction of H.

3.2 Operational Equivalence wrt. Built-in Constraints

In the following, we prove that the final program, i.e. the one resulting form the application of the GenSimpRules algorithm, is *built-in-operationally equivalent* to the initial set of propagation rules. Two programs are built-in-operationally equivalent if for each goal, we obtain the same built-in constraints in the final state using either P_1 or P_2. In the following, we will denote the built-in constraints appearing in a state S by $builtIn(S)$.

Theorem 3.1. *Let P be a set of propagation rules and P' be the transformed program using the* GenSimpRules *algorithm. If $P \cup P'$ is terminating then for any state S, we have if $S \mapsto^*_{P'} S_1$ and $S \mapsto^*_{P} S_2$, where S_1 and S_2 are final states, then $builtIn(S_1)$ and $builtIn(S_2)$ are variants.*

To prove the theorem above, we first show that we can just take the union of the set of propagation rules and the transformed program and the resulting program will be built-in operationally equivalent to the set of propagation rules (cf. Lemma 3.2). Then, we prove that the union of the two programs and the transformed program alone have the same behavior (cf. Lemma 3.3). Therefore, we can conclude that the transformed program and the initial set of propagation rules are built-in operationally equivalent. Theorem 3.1 is then an immediate consequence of Lemma 3.2 and Lemma 3.3.
To prove these two lemmas, we establish first a preliminary lemma stating that the union of a set of propagation rules and its transformed program is confluent.

Lemma 3.1. *Let P be a set of propagation rules and P' be the transformed program using the* GenSimpRules *algorithm. If $P \cup P'$ is terminating, then $P \cup P'$ is confluent.*

Proof. *To show that $P \cup P'$ is confluent, we only have to show that all critical pairs of $P \cup P'$ are joinable, since $P \cup P'$ is terminating (according to Theorem 2.1). The set of critical pairs of $P \cup P'$ consists of all critical pairs stemming from two rules appearing in P (case 1. below), all critical pairs stemming from two rules appearing in P' (case 2. below) and all critical pairs stemming from one rule appearing in P and one rule appearing in P' (case 3. below).*

1. *P consists of a set of propagation rules. We can easily show that P is confluent, since any critical pair (S_1, S_2) stemming from two propagation rules R_1 and R_2 is joinable: Let S_i be a state resulting from the application of R_i on the critical ancestor state of R_1 and R_2 ($i = 1, 2$), then applying R_1 on S_2 and R_2 on S_1 leads to a common state. A more detailed proof can be found in [1].*

2. *P' is the transformed program using the* GenSimpRules *algorithm. Thus, all critical pairs stemming from two rules appearing in P' are joinable.*

3. *Let (S_1, S_2) be a critical pair stemming from one rule $R_1 \in P$ and one rule $R_2 \in P'$. We distinguish four cases:*

a) R_1 *and* R_2 *are propagation rules.* (S_1, S_2) *is joinable; the argumentation is analogous to the one of case 1. above.*

b) R_2 *is obtained by transforming the propagation rule* R_1 *into a simplification rule. The heads of* R_1 *and* R_2 *are the same. Let* S_i *be the state resulting from the application of* R_i *to the critical ancestor state of* R_1 *and* R_2 $(i = 1, 2)$*, then applying* R_2 *to* S_1 *leads to a variant of* S_2*. Therefore, the critical pair is joinable.*

c) R_2 *is a simplification rule but it is not obtained by transforming the propagation rule* R_1 *and* R_1 *is in* P'*. This situation is analogous to case 2.*

d) R_2 *is a simplification rule but it is not obtained by transforming the propagation rule* R_1 *and* R_1 *is not in* P'*, i.e.* R_1 *has been transformed into a simplification rule* R_1'*.*

Let R_1 *be of the form* $H_1 \wedge A_1 \Rightarrow B_1$ *and* R_2 *of the form* $H_2 \wedge A_2 \Leftrightarrow B_2$*. Then* R_1' *is of the form* $H_1 \wedge A_1 \Leftrightarrow B_1 \wedge C$*, where* C *is a subconjunction of* $H_1 \wedge A_1$*. The critical ancestor state* $(H_1 \wedge A_1 \wedge H_2 \wedge (A_1{=}A_2))$ *of* R_1 *and* R_2 *is also a critical ancestor state of* R_1' *and* R_2*.*

Since P' *is confluent, the critical pair*

$$(S_1,\ \ S_2) :=$$

$$(H_2 \wedge (A_1{=}A_2) \wedge B_1 \wedge C,\ \ H_1 \wedge (A_1{=}A_2) \wedge B_2)$$

stemming from the critical ancestor state of R_1' *and* R_2 *is joinable in* P'*. Therefore, this critical pair is also joinable in* $P \cup P'$*.*

The critical pair stemming from the critical ancestor state of R_1 *and* R_2 *is*

$$(S_3,\ \ S_2) :=$$

$$(H_1 \wedge A_1 \wedge H_2 \wedge (A_1{=}A_2) \wedge B_1,\ \ H_1 \wedge (A_1{=}A_2) \wedge B_2)$$

R_1' *can be applied to* S_3 *and leads to* S_1*. So, since* (S_1, S_2) *is joinable in* $P \cup P'$*, then* (S_3, S_2) *is also joinable in* $P \cup P'$*.* □

LEMMA 3.2.
Let P *be a set of propagation rules and* P' *be the transformed program using the* GENSIMPRULES *algorithm. If* $P \cup P'$ *is terminating then for any state* S *if* $S \mapsto_P^* S_1$ *and* $S \mapsto_{P \cup P'}^* S_2$*, where* S_1 *and* S_2 *are final states, then* $builtIn(S_1)$ *and* $builtIn(S_2)$ *are variants.*

PROOF. *Since* $S \mapsto_P^* S_1$ *and* $P \subseteq P \cup P'$*, we have* $S \mapsto_{P \cup P'}^* S_1$ *where* S_1 *is not necessarily a final state.*
$P \cup P'$ *is terminating then* $P \cup P'$ *is confluent by Lemma 3.1. Since we have* $S \mapsto_{P \cup P'}^* S_2$ *and* $S \mapsto_{P \cup P'}^* S_1$*, then* S_1 *and* S_2 *are joinable, i.e. there is a computation of the form* $S_1 \mapsto_{P \cup P'}^* S_2'$*, where* S_2 *and* S_2' *are variants.*
Since P' *consists of the rules of* P *where some propagation rules have been transformed into simplification rules, then* $P \cup P'$ *can not add new constraints to* S_1*. Moreover, a CHR program can only remove user-defined constraints and thus* $P \cup P'$ *can not remove built-in constraints in a computation starting from* S_1*. Thus, we have that* $builtIn(S_1)$ *and* $builtIn(S_2')$ *are variants. Therefore,* $builtIn(S_1)$ *and* $builtIn(S_2)$ *are variants of each other.* □

LEMMA 3.3.
Let P *be a set of propagation rules and* P' *be the transformed program using the* GENSIMPRULES *algorithm. If* $P \cup P'$ *is terminating then for any state* S*, we have if* $S \mapsto_{P \cup P'}^* S_1$ *and* $S \mapsto_{P'}^* S_2$*, where* S_1 *and* S_2 *are final states, then* S_1 *and* S_2 *are variants.*

PROOF. *Since* $S \mapsto_{P'}^* S_2$ *and* $P' \subseteq P \cup P'$*, we have* $S \mapsto_{P \cup P'}^* S_2$*.*
We prove the claim by contradiction. We assume that S_2 *is not a final state for* $P \cup P'$*. Since* S_2 *is a final state for* P'*, there exists a propagation rule* R *in* P *that can be used to perform a computation step on* S_2*. If we consider the rule* R' *in* P' *obtained from* R *using the* GENSIMPRULES *algorithm, since* R' *and* R *have the same head, then* R' *can also be used to perform a computation step on* S_2*, and thus* S_2 *cannot be a final state for* P'*, which contradicts the hypothesis.*
So, the state S_2 *must be a final state for* $P \cup P'$*. Moreover, since* $P \cup P'$ *is terminating then* $P \cup P'$ *is confluent by Lemma 3.1, and thus* S_1 *and* S_2 *are variants.* □

4. AN APPLICATION

In [3], some examples have been presented to show that our method for generating propagation rules is efficient and can be used in practice. In this section, we show first that when we use the propagation rules of [3] 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.
Next, we show that the method presented in this paper 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 can not 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 by Van Hentenryck et al [10]. We shortly present this approach below and then describe our experiments.

4.1 Automatic test-pattern generation

In this section, we briefly recall the approach of [10] 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. This adder is composed of 21 boolean gates and can be defined using four full adders:

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

$$\begin{aligned}
fullAdder(X, Y, Z, S, C) \Leftrightarrow\; & and(X, Y, C1) \wedge \\
& xor(X, Y, S1) \wedge \\
& and(Z, S1, C2) \wedge \\
& xor(Z, S1, S) \wedge \\
& or(C1, C2, C).
\end{aligned}$$

$4BitAdder(X_3, X_2, X_1, X_0, Y_3, Y_2, Y_1, Y_0, O, Z_3, Z_2, Z_1, Z_0)$ means that the result of adding the 4-bit binary number (X_3, X_2, X_1, X_0) to the 4-bit binary number (Y_3, Y_2, Y_1, Y_0) is (O, Z_3, Z_2, Z_1, Z_0) and $fullAdder(X, Y, Z, S, C)$ simply defines a full adder, i.e. addition of two bits X and Y is performed, where Z is the input carry bit, S is the output bit, and C is the output carry bit.
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 execution time is the CPU time used on a Pentium 3 with 256 MBytes of memory and a 500 MHz processor.
The set of rules used for the first experiment contains only single-headed propagation rules generated by the algorithm presented in [3]. This generation has been realized 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 body of the rules. Examples of rules are:

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

It should be noticed that this set of rules leads to the same propagations as the rules used by Van Hentenryck et al. Exploiting the symmetry of the ternary operators with respect to the the first and second argument (e.g., $and(X, Y, Z) \Leftrightarrow and(Y, X, Z)$), the number of rules can be reduced to 77 rules. It is a hard work to produce this set manually, but 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 in this experiment we take rules with one or two atoms in the head. One such rule is for example

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

Even when exploiting the symmetry of the ternary operators this set consists of 621 rules, and cannot reasonably be generated by hand.
Finally, the third experiment has been made using the 621 rules of the second experiment transformed into simplification rules according to the method presented in this paper. 308 propagation rules have been transformed into simplification rules. For example, the previous rule has been transformed into

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

The comparison between experiment 1 and experiment 2 is given in Table 1. 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 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 multiple heads generated automatically (experiment 2) can be used to greatly reduce the size of the search space compared to the search space explored using single-headed propagation rules (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 2 shows that this overhead can be suppressed

gate number	experiment 1		experiment 2		search space		CPU time	
	search space	CPU time	search space	CPU time	Δ abs.	Δ %	Δ abs.	Δ %
1	680	1.76	673	6.55	-7	-1.03	+4.79	+272.16
2	5088	44.86	1728	17.01	-3360	-66.04	-27.85	-62.08
3	620	2.45	417	3.34	-203	-32.74	+0.89	+36.33
4	5088	44.93	1728	14.52	-3360	-66.04	-30.41	-67.68
5	680	1.90	517	2.94	-163	-23.97	+1.04	+54.74
6	1780	8.73	1258	21.12	-522	-29.33	+12.39	+141.92
7	5854	48.40	1909	18.85	-3945	-67.39	-29.55	-61.05
8	1780	10.67	909	11.27	-871	-48.93	+0.60	+5.62
9	12778	133.34	1904	19.56	-10874	-85.10	-113.78	-85.33
10	2740	17.00	1263	17.15	-1477	-53.91	+0.15	+0.88
11	4960	40.00	2670	75.66	-2290	-46.17	+35.66	+89.15
12	7268	48.74	3086	62.09	-4182	-57.54	+13.35	+27.39
13	5900	53.32	1997	39.06	-3903	-66.15	-14.26	-26.74
14	13122	132.43	1944	22.16	-11178	-85.19	-110.27	-83.27
15	8340	77.35	2827	60.44	-5513	-66.10	-16.91	-21.86
16	2926	81.21	514	33.85	-2412	-82.43	-47.36	-58.32
17	7692	63.86	2681	44.99	-5011	-65.15	-18.87	-29.55
18	2822	91.08	687	31.02	-2135	-75.66	-60.06	-65.94
19	7884	124.49	1994	38.58	-5890	-74.71	-85.91	-69.01
20	4988	126.93	965	37.32	-4023	-80.65	-89.61	-70.60
21	11712	172.25	2059	43.25	-9653	-82.42	-129.00	-74.89

Table 1: Search space reduction with overhead.

if we transform the set of propagation rules to a set of propagation and simplification rules using the technique proposed in this paper. Moreover, in nearly all cases, the execution time is reduced by more than 50%. Only a very few overhead remains for gate 1. But, it should be noticed that in this particular case the search space reduction is very low and the execution very brief. So, for this gate, we could not expect to observe a real reduction of the execution time because of the fixed-cost we must pay to handle the set of rules.

5. CONCLUSION

The aim of this paper was to provide a method for transforming propagation rules into simplification rules. Simplification rules lead to a reduction of the size of the constraint store and to a speedup of constraint solving. The method has been developed based on the confluence notion of Constraint Handling Rules. We have also shown that the transformed program is operationally equivalent (wrt. the built-in constraints) to the initial set of propagation rules.

A practical application of our transformation method lies in software development. This method together with the algorithm for generating propagation rules proposed in [3] provide a framework in which rule-based solvers for constraints defined over finite domains can be automatically generated. Our transformation method can also be used while writing a constraint solver by hand, when it is not obvious whether a rule is a simplification rule or a propagation one. The user only needs to specify the propagation algorithm of constraints by propagation rules and our method transforms some of these propagation rules into simplification rules.

Furthermore, we have argued by an example taken form the field of digital circuit design that transforming propagation rules into simplification rules is of practical use.

6. REFERENCES

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

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

[3] S. Abdennadher and C. Rigotti. Automatic generation of propagation rules for finite domains. In *6th International Conference on Principles and Practice of Constraint Programming, CP00*, LNCS 1894. Springer-Verlag, 2000.

[4] K. Apt and E. Monfroy. Automatic generation of constraint propagation algorithms for small finite domains. In *5th International Conference on Principles and Practice of Constraint Programming, CP'99*, LNCS 1713. Springer-Verlag, 1999.

[5] Y. Caseau and F. Laburthe. Claire: Combining objects and rules for problem solving. In *JICSLP'96 workshop on multi-paradigm logic programming*, 1996.

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

[7] T. Frühwirth. Proving termination of constraint solver programs. In *New Trends in Constraints*. LNAI 1865, 2000.

[8] C. Kirchner, H. Kirchner, and M. Vittek. Implementing computational systems with constraints. In *Proceedings of the First Workshop on Principles*

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	680	1.76	673	1.86	-7	-1.03	+0.10	+5.68
2	5088	44.86	1728	5.90	-3360	-66.04	-38.96	-86.85
3	620	2.45	417	0.96	-203	-32.74	-1.49	-60.82
4	5088	44.93	1728	5.18	-3360	-66.04	-39.75	-88.47
5	680	1.90	517	0.86	-163	-23.97	-1.04	-54.74
6	1780	8.73	1258	5.46	-522	-29.33	-3.27	-37.46
7	5854	48.40	1909	4.71	-3945	-67.39	-43.69	-90.27
8	1780	10.67	909	3.11	-871	-48.93	-7.56	-70.85
9	12778	133.34	1904	7.59	-10874	-85.10	-125.75	-94.31
10	2740	17.00	1263	4.62	-1477	-53.91	-12.38	-72.82
11	4960	40.00	2670	20.80	-2290	-46.17	-19.20	-48.00
12	7268	48.74	3086	17.45	-4182	-57.54	-31.29	-64.20
13	5900	53.32	1997	10.89	-3903	-66.15	-42.43	-79.58
14	13122	132.43	1944	9.03	-11178	-85.19	-123.40	-93.18
15	8340	77.35	2827	17.67	-5513	-66.10	-59.68	-77.16
16	2926	81.21	514	12.66	-2412	-82.43	-68.55	-84.41
17	7692	63.86	2681	14.49	-5011	-65.15	-49.37	-77.31
18	2822	91.08	687	14.21	-2135	-75.66	-76.87	-84.40
19	7884	124.49	1994	16.06	-5890	-74.71	-108.43	-87.10
20	4988	126.93	965	15.36	-4023	-80.65	-111.57	-87.90
21	11712	172.25	2059	17.78	-9653	-82.42	-154.47	-89.68

Table 2: Search space and execution time reduction.

and Practice of Constraints Programming. MIT Press, Apr. 1993.

[9] C. Ringeissen and E. Monfroy. Generating propagation rules for finite domains via unification in finite algebra. In *ERCIM Working Group on Constraints / CompulogNet Area on Constraint Programming Workshop*, 1999.

[10] P. van Hentenryck, H. Simonis, and M. Dincbas. Constraint satisfaction using constraint logic programming. *Artificial Intelligence*, 58(1-3), Dec. 1992.