Electronic Notes in Theoretical Computer Science 86 No. 1 (2003)
URL: http://www.elsevier.nl/locate/entcs/volume86.html 16 pages

Deductive Generation of Constraint Propagation Rules

Sebastian Brand

CWI
P.O. Box 94079
1090 GB Amsterdam
The Netherlands

Eric Monfroy

IRIN
Université de Nantes
2, rue de la Houssinière
BP 92208 Nantes Cedex 03
France

Abstract

Constraint propagation can often be conveniently expressed by rules. In recent years, a number of techniques for automatic generation of rule-based constraint solvers have been developed, most of them using a generate-and-test approach. We examine a generation method that is based on deduction. A solver (i. e., a set of rules) for a complex constraint is obtained from one or several weaker solvers for simple constraints. We describe incremental solver constructions for several types of constraint modifications, including conjunction, existential and universal quantification.

1 Introduction

Rule-based methodologies have had a place in constraint processing for a long time. A number of languages allow one to program constraint solvers in this declarative way. Examples are the languages based on the concurrent constraint programming (`ccp`) paradigm [14] such as AKL [6], Oz [15], and especially `CHR` [10]. Also the term-rewriting language ELAN [11,8], and CLAIRE [7] are suitable for rule-based constraint processing. The last years have seen an increase in work on, and interesting results in, the automatic generation of rule-based constraint solvers [3,2]. Almost all of these approaches have in common that their paradigm is essentially generate-and-test. Syntactically correct rule candidates are enumerated and subjected to a validity test against the constraint definition, which is usually extensive (truth table), or a constraint logic program. In a different method [13] the conclusion is derived from the given premise, which itself, however, comes from a syntactic enumeration process. Another aspect is examined in [1], where the merging of solvers written in the full and more expressive CHR language is discussed. The main focus is on termination and confluence, while we look here at solvers consisting of propagation rules only where these two properties are of no concern.

We explore the idea of incremental, deductive, combinatorial rule generation. The key property here is that the *source definition* of the constraint is irrelevant. Instead, the input to our method is another set of rules associated to some constraints, and the relation of these constraints to the desired constraint. The rules are processed according to declarative transformation steps, *meta rules*, leading to the introduction of new or discarding of currently present rules. In that way, solvers for basic, primitive constraints are transformed or combined into solvers for complex constraints.

A number of benefits arise from this approach. When combining solvers, the resulting solver may propagate stronger than the simple union of its building blocks. That is, the consistency it enforces is stricter, which translates into less search when solving constraint satisfaction problems. Work spent on previously generated solvers is re-used. This means that the incremental method can be faster than generate-and-test. Even if it is not always the case, it may be best for certain classes of solvers. Finally, describing rule generation in a deductive way by meta rules provides a new perspective on, and helps understand the origin of, rule-based constraint solvers.

Although we introduce the deductive method in a general setting in Section 3, the main part of the paper deals with a specific type of rule: the inclusion/membership rules that were first discussed in [3]. These rules are especially interesting and relevant as sets of them (i. e., solvers) can enforce generalised arc-consistency (GAC), a central local consistency notion in constraint programming.

We discuss a number of transformations on inclusion rule sets, leading to solvers for modified constraints. The most important transformation is solver generation for the conjunctive constraint $C_1 \wedge C_2$, based on the rules for C_1 and C_2. An auxiliary modification is adding an unconstrained variable to a constraint, constraint padding. An inverse effect is achieved by existentially or universally quantifying a variable in a constraint. Furthermore, we discuss tightening a constraint by disallowing previous solutions, constraint restriction. An important special case of the latter is defining a constraint negatively by the set of its non-solutions. All these transformations yield a rule-based solver that enforces GAC when the source solver is complete with respect to the constraint in a certain sense. In Section 7 we conclude with the example of incrementally constructing a solver for the `fulladder` constraint, based only on solvers for some primitive base constraints.

2 Preliminaries

2.1 Constraint Satisfaction Problems

Consider a sequence of variables $X = x_1, \ldots, x_n$ with respective domains $D_1, \ldots, D_n$ associated to them. By a constraint C on the variables X we mean a subset of $D_1 \times \ldots \times D_n$. Given an element $d = d_1, \ldots, d_n$ of $D_1 \times \ldots \times D_n$ and a subsequence $Y = x_{i_1}, \ldots, x_{i_\ell}$ of X we denote by $d[Y]$ the sequence $d_{i_1}, \ldots, d_{i_\ell}$. In particular, for a variable x_i from X, $d[x_i]$ denotes d_i. A constraint satisfaction problem, in short CSP, consists of a finite sequence of variables $X = x_1, \ldots, x_n$ with respective domains $\mathcal{D} = D_1, \ldots, D_n$, together with a finite set $\mathcal{C}$ of constraints, each on a subsequence of X. We write it as $\langle \mathcal{C}; x_1 \in D_1, \ldots, x_n \in D_n \rangle$, in short $\langle \mathcal{C}; X \in \mathcal{D} \rangle$. By a solution to $\langle \mathcal{C}; X \in \mathcal{D} \rangle$ we mean an element $d \in \mathcal{D}$ such that for each constraint $C \in \mathcal{C}$ on the variables Y we have $d[Y] \in C$. The latter is also denoted by $\models_d C$. By extension we write $\models_d \mathcal{C}$ if $\models_d C$ for all $C \in \mathcal{C}$. Then d is a solution if $\models_d \mathcal{C}$.

2.2 Local Consistency and Propagation

A technique that is central in solving constraint satisfaction problems is constraint propagation. It is most common in its specific form of domain reduction, i. e., the domain of a variable participating in a constraint is reduced in the light of the domains of the other variables. As an example consider $\langle x < y\,;\, x \in \{1..3\}, y \in \{1..3\} \rangle$ in which the inequality constraint propagates to the smaller domains $x \in \{1..2\}, x \in \{2..3\}$. Constraint propagation is best understood as fixpoint computation of propagation operators on CSPs. We are concerned with propagation operators that are rules. A CSP that is a fixpoint of a set of rules, for instance after exhaustive application of these rules, is said to be closed under this rule set.

A measure of the strength of propagation is given by the enforced local consistency (as opposed to global consistency, which may have a prohibitive exponential complexity). An important local consistency notion is generalised arc-consistency (GAC). It holds for a satisfiable constraint if every instantiation of any variable of the constraint can be extended to an instantiation of all variables that satisfies the constraint and is permitted by the current domains.

Propagation is incomplete in the sense that it does not always identify a solution or inconsistency of a CSP. Thus, it is usually interleaved with search.

3 General Propagation Rules.

General constraint propagation rules have the form

$$\{c_1, \ldots, c_n\} \to \{c'_1, \ldots, c'_m\}$$

where both left-hand side (*lhs*) and right-hand side (*rhs*) are sets of constraints interpreted as conjunctions (allowing disjunctive *rhs*'s would amount to search). Propagation rules operate on a constraint store that is itself a set of constraints. A rule is applicable if its *lhs* is contained in, or implied by, the store. In this case its *rhs* is added to the store.

3.1 Validity

A substantial property of a rule is that it is valid with respect to its constraints: the *lhs* must imply the *rhs*. We presume that the *rhs* does not contain new variables, i. e., it simply constrains more some of the *lhs* variables. In this case validity means that every solution of the *lhs* can be projected into one of the *rhs*:

$$C \rightarrow C' \qquad \text{is valid if} \qquad \forall d.\ \models_d C \text{ implies } \models_d C' \ .$$

3.2 Feasibility

It is useful to classify rules by the satisfiability of their *lhs*'s. We call a rule feasible, if its *lhs* is satisfiable, that is

$$C \rightarrow C' \qquad \text{is feasible if} \qquad \exists d.\ \models_d C \ .$$

An infeasible rule is thus trivially valid.

3.3 Redundancy

When considering a set of constraint propagation rules, it is useful to ask whether it is a smallest such set to enforce the associated local consistency notion. We define: a rule r is redundant with respect to a rule set $\mathcal{R}$ if every set of constraints closed under $\mathcal{R}$ is also closed under r. In this case r does not contain any extra information. It is helpful that, when testing a rule for being redundant, it suffices to test its *rhs* as the set of constraints [5].

If a rule set is strong enough to detect inconsistency of sets of constraints in the language of their *lhs*'s, then infeasible rules are redundant.

4 Transformations of (General) Propagation Rules

A transformation is a sequence of atomic steps introducing or removing single propagation rules. There are also two auxiliary structural transformation steps. These deal with the *rhs* of rules, while the *lhs*'s are focused on by the transformation steps that introduce or remove a rule. We assume a fixed, finite language of constraints. For brevity, we write $\mathcal{R}, r$ for the set $\mathcal{R} \cup \{r\}$ in the remainder of the paper.

4.1 Subsumption

A rule subsumes another with the same *rhs* if its *lhs* implies the other. As a meta rule we formulate more precisely

$$\text{[general-subsume]} \quad \frac{\mathcal{R},\ C_1 \to C',\ C_2 \to C'}{\mathcal{R},\ C_1 \to C'} \text{ if } \quad \forall d.\ \models_d C_2 \text{ implies } \models_d C_1$$

A typical corresponding situation is $C_1 \subseteq C_2$. We say that a rule is subsumed by a set of rules if it is subsumed by a rule contained in the set.

4.2 Derivation

Two ancestor rules with the same *rhs* give rise to a descendant if the constraint language can express the disjunction of the two *lhs*'s, or something stronger. Formally,

$$\text{[general-derive]} \quad \frac{\mathcal{R},\ C_1 \to C',\ C_2 \to C'}{\mathcal{R},\ C_1 \to C',\ C_2 \to C',\ C_3 \to C'}$$

$$\text{if} \quad \forall d.\ \models_d C_3 \text{ implies } \models_d C_1 \text{ or } \models_d C_2$$

It is important to observe that validity is preserved – the descendant rule inherits it from its ancestors. The idea of this transformation step is to compose the first two rules, at best into one that subsumes both former rules.

To avoid trivial, subsumed descendants, C_3 should not simply imply one of C_1, C_2. Finding candidates for C_3 depends on the language. Ideally, it can be constructed from C_1, C_2, and we are able to do so in the following.

(De)composing rules
In the above meta rules, all mentioned propagation rules coincide in their *rhs*'s, while in general we deal with varying *rhs*'s. We assume suitable, implicit transformations between $C \to C'_1 \cup C'_2$ and the two rules $C \to C'_1,\ C \to C'_2$.

5 Inclusion Rules

Having the two transformation steps – *meta rules* –, it is interesting to describe the result of a derivation. We proceed with a specific language, and consider rules of the form

$$C, x_1 \in S_1, \dots, x_n \in S_n \ \to \ y \neq a$$

where each S_i is a set of constants, and a a constant. The constraint C is on the $n+1$ distinct variables $\{x_1, \dots, x_n, y\}$. The base domain of all the variables is D, which means $C \subseteq D^{n+1}$. We require $\varnothing \neq S_i \subseteq D$ for all $i \in \{1..n\}$.

We call C the constraint *associated* to the rule. In the following, all rules are regarded as being associated to the same constraint, and we omit it from the notation. Finally, with $X = (x_1, \dots, x_n)$ and $S = S_1 \times \ldots \times S_n$ we abbreviate the above inclusion rule as $X \in S \to y \neq a$.

The interest in inclusion rules arises from the possibility to express, by sets of such rules, generalised arc-consistency (GAC) for the associated constraint.

5.1 *Transformations of Inclusion Rules*

We provide now specialisations of the general subsumption and derivation transformations. Subsequently, we describe the rule set resulting from a stabilising derivation of such transformations. If certain conditions on the source rule set are met then the resulting rule set enforces generalised arc-consistency.

5.2 *Subsumption*

This meta rule discards a rule in presence of a stronger one:

$$\textsf{[subsume]} \quad \frac{\mathcal{R},\ X \in S \to y \neq a,\ X \in P \to y \neq a}{\mathcal{R},\ X \in S \to y \neq a} \qquad \text{if } S \supseteq P$$

A rule subsumes another one with tighter bounds. It is easy to see that this is an instance of [general-subsume].

Example 5.1 $x \in \{2\} \to y \neq 1$ is subsumed by $x \in \{2, 3\} \to y \neq 1$. □

5.3 *Derivation*

This meta rule creates a new rule from two present rules:

$$\textsf{[derive]} \quad \frac{\mathcal{R},\ X \in S \to y \neq a,\ X \in P \to y \neq a}{\mathcal{R},\ X \in S \to y \neq a,\ X \in P \to y \neq a,\ X \in Q \to y \neq a}$$

$$\text{if} \quad \begin{array}{rl} \text{(i)} & Q_i = S_i \cap P_i \neq \varnothing \text{ at all indices } i \in \{1..n\} \text{ except for some } k \\ \text{(ii)} & Q_k = S_k \cup P_k \\ \text{(iii)} & Q_k \supset S_k \text{ and } Q_k \supset P_k \end{array}$$

The side conditions guarantee that the derived rule is syntactically correct (1.), and that it is not subsumed by any parent (3.), although it may itself subsume one or both of them. Furthermore, it is valid (1. and 2.). It is useful to notice that $x_k \in Q_k$ is the collapsed disjunctive constraint "$x_k \in S_k$ or $x_k \in P_k$".

A [derive] step depends on k, and for two ancestor rules there may be several appropriate indices k, satisfying (1.,2.,3.). Note however, that no derived rule subsumes another with a different k.

Example 5.2 From

$$\begin{array}{l} x_1 \in \{1, 2\}, x_2 \in \{1, 3\} \to y \neq 2 \\ x_1 \in \{2, 3\}, x_2 \in \{2, 3\} \to y \neq 2 \end{array}$$

we derive the two rules

$$\begin{array}{ll} x_1 \in \{1, 2, 3\}, x_2 \in \{3\} \to y \neq 2 & \text{with } k = 1 \\ x_1 \in \{2\}, x_2 \in \{1, 2, 3\} \to y \neq 2 & \text{with } k = 2\ . \end{array}$$

□

5.4 Properties of the Transformations

We proceed by collecting properties of these inference rules. We do so in two steps: first, linking the source and the result of an exhaustive application of the transformations, and second, characterising the propagation that a fully transformed rule set achieves.

Atomic Rule, Rule Set Closure
It is convenient to have the concept of an atomic rule: $X \in S \to y \neq a$ is atomic if each S_i is a singleton set. It is useful to be aware of the 1-1 correspondence between such an atomic rule and a non-solution d of the associated constraint, namely by $\{d[X]\} = S$ and $d[y] = a$.

We denote by $closure(\mathcal{R})$ the rule set that results from an exhaustive application of [subsume, derive]. The following first finding links atomic rules and rule set closure.

Theorem 5.3 *Let $\mathcal{R}$ be a set of inclusion rules and C their associated constraint. If $\mathcal{R}$ subsumes every* atomic *rule valid for C then $closure(\mathcal{R})$ subsumes every rule valid for C.*

Proof. We argue by contradiction: Let us say that $r = (X \in S \to y \neq a)$ is valid but not subsumed by $closure(\mathcal{R})$. Without loss of generality we assume that all other rules $X \in S' \to y \neq a$ with $S' \subset S$ are subsumed.

Observe first that r is not atomic. Take then some S_k that is not a singleton, and partition it into $S_k = P_k \cup Q_k$ where neither P_k nor Q_k is empty. We construct complete bounds P, Q by defining $P_i = Q_i = S_i$ at the remaining indices $i \neq k$.

Both corresponding rules $X \in P \to y \neq a$ and $X \in Q \to y \neq a$ are valid since r is. For each of the two rules there must be a subsuming rule contained in $closure(\mathcal{R})$, as we assumed initially. Enter now these two subsuming rules into [derive]. The resulting new rule must subsume r, contradicting our assumption.

As regards [subsume], we remark that subsumption is a transitive relation. Therefore, if a rule is subsumed by a rule set then this is still the case after an application of [subsume] to the set. □

We know now which "seed rules" are necessary so that after closure there are rules for all valid propagations. Next, we establish the local consistency notion achieved by these propagations.

Theorem 5.4 *Let $\mathcal{R}$ be a set of inclusion rules valid for their associated constraint C. Let $\mathcal{R}$ subsume every rule valid for C. Then the constraint C is closed under $\mathcal{R}$ iff C is generalised arc-consistent.*

Proof. Suppose that C is not closed under $r = (X \in S \to y \neq a)$ in $\mathcal{R}$, thus $C[X] \subseteq S$ and $a \in C[y]$. Since r is valid we know that for all d we have that $d[X] \in S$ implies $d[y] \neq a$. The counter position is that, for all d, $d[y] = a$

implies $d[X] \notin S$, and in turn $d[X] \notin C[X]$. But this means that the partial instantiation $\{y \mapsto a\}$ can not be extended to a solution of C.

For the reverse direction, suppose that $\{y \mapsto a\}$ can not be extended to a solution of the constraint C. So no d exist with $d[y] = a$ and $d[X] \in C[X]$. Then $x_1 \in C[x_1], \ldots, x_n \in C[x_n] \to y \neq a$ is a valid rule; and as such is subsumed by $\mathcal{R}$. The subsuming rule in $\mathcal{R}$, however, is applicable to C. □

Both preceding results together allow us to obtain a GAC-enforcing set of inclusion rules from an initial set of all atomic rules, or corresponding subsuming rules, that is then closed in particular under [derive].

5.5 Uniqueness

The closure of a set $\mathcal{R}$ under the meta rules [derive,subsume] is unique if the base domain is finite. Then there are only finitely many syntactically correct rules. Any closure algorithm that applies [derive] at most once to any two rules in $\mathcal{R}$ for a specific k must terminate. Furthermore, the meta rule system is confluent. Every cricital pair, obtained by applying two meta rules to $\mathcal{R}$, is joinable. This is easy to see for pairs stemming from [derive]+[derive] and [subsume]+[subsume]. The somewhat more involved case of [derive]+[subsume] requires a case distinction (further applicability of [derive]) that we omit here.

5.6 Relation to RGA

The method of choice for producing sets of inclusion rules is described in [3], which introduced the notion. We compare the outcome of their Rule Generation Algorithm (called RGA here), and our closure method. The main difference lies in infeasible rules. Notice that the inclusion rule $X \in S \to y \neq a$ is infeasible exactly if $S \cap C[X] = \varnothing$.

Lemma 5.5 *Let $\mathcal{R}_{\mathrm{RGA}}$ be the inclusion rule set that RGA generates for a constraint C. Let $\mathcal{R}_{\mathrm{CLS}}$ be a set of rules valid for C and also subsuming every atomic rule valid for C. Moreover, let $\mathcal{R}_{\mathrm{CLS}}$ be closed under [derive, subsume]. Then*

- *every rule in $\mathcal{R}_{\mathrm{RGA}}$ is subsumed by a rule in $\mathcal{R}_{\mathrm{CLS}}$, and*
- *every feasible rule in $\mathcal{R}_{\mathrm{CLS}}$ subsumes a rule in $\mathcal{R}_{\mathrm{RGA}}$.*

Proof. RGA (see its presentation in [3]) enumerates all valid rules, discarding those that are subsumed. In turn, $\mathcal{R}_{\mathrm{CLS}}$ subsumes all valid rules, by Theorems 5.3,5.4.

Inversely, each feasible rule in $\mathcal{R}_{\mathrm{CLS}}$ is valid, and not strictly subsumed by another valid rule. This means that it is either in, or subsumes a rule in, $\mathcal{R}_{\mathrm{RGA}}$. (The latter case my arise due to presence of infeasible rules and [derive].) □

Example 5.6 The deductive approach may yield infeasible rules, and rules that are “partially infeasible”. Consider the constraint C_{ex} on x, y with domain $\{1,2,3\}$. RGA generates the rules $\mathcal{R} = \{(1),(2),(3),(4)\}$.

C_{ex}	x	y
	1	1
	3	1
	3	3

(1)	$y \in \{3\} \to x \neq 1$
(2)	$y \in \{1,2,3\} \to x \neq 2$
(3)	$x \in \{1,2,3\} \to y \neq 2$
(4)	$x \in \{1\} \to y \neq 3$

Let us construct $\mathcal{R}'$ by replacing (1) in $\mathcal{R}$ by the valid, feasible rule

(5) $y \in \{2,3\} \to x \neq 1$.

Observe that $\mathcal{R}'$ is closed under [derive, subsume]. The *rhs* of rule (5) contains the unsatisfiable part $y \in \{2\}$. Next, construct $\mathcal{R}''$ by adding to $\mathcal{R}$

(6) $y \in \{2\} \to x \neq 1$.

Rule (6) is valid but infeasible. Also $\mathcal{R}''$ is closed under [derive, subsume]. □

5.7 Significance of Infeasibility

For constraint propagation, rule (5) is preferable over rule (1), which is strictly subsumed. Rule (6), on the other hand, is redundant relative to $\mathcal{R}$, and so undesirable. Ideally, infeasible rules should not be derived. However, many feasible rules that RGA produces, are redundant [5]. Therefore, a finalising redundancy removal after rule generation is appropriate in either approach.

6 Applications

We demonstrate some applications of deductive rule generation.

6.1 Conjunction of Constraints

Given two constraints C_1 and C_2 on the same variables, and their associated rules $\mathcal{R}(C_1)$ and $\mathcal{R}(C_2)$, we are interested in rules for the conjunctive constraint $C_1 \wedge C_2$ (note that the solutions of this constraint, hence the constraint itself, is the intersection $C_1 \cap C_2$). In general the simple union $\mathcal{R}(C_1) \cup \mathcal{R}(C_2)$ does not propagate as strongly as possible. For example, consider the boolean variables $x, y \in \{0,1\}$ constrained by $not(x,y) \wedge (x = y)$. This CSP is closed under any rules valid for the two constraints individually, yet it is inconsistent.

By Theorems 5.3,5.4 we can state, however, that if $\mathcal{R}(C_1), \mathcal{R}(C_2)$ subsume all atomic rules valid for C_1, C_2, resp., then

$$\mathcal{R}(C_1 \wedge C_2) = closure(\mathcal{R}(C_1) \cup \mathcal{R}(C_2))$$

enforces GAC on the conjunctive constraint $C_1 \wedge C_2$. To see this, observe that any atomic rule valid for $C_1 \wedge C_2$ must be valid for at least one of C_1, C_2 as well. An obvious generalisation is $\mathcal{R}(\bigwedge_{i=1}^{m} C_i) = closure(\bigcup_{i=1}^{m} \mathcal{R}(C_i))$. For an example, we refer to Subsection 7.

6.2 Local Consistency Notion

It is interesting to observe that from the view of the set of the constituent constraints C_i, all on the same set of variables, the consistency enforced is relational $(1, m)$-consistency [9]. Since we enforce GAC on the global conjunctive constraint, an instantiation of any one variable can be extended to a solution of the global constraint, which is also a solution of each of the constituent constraints. Enforcing GAC on the constituent constraints separately is equivalent to relational $(1, 1)$-consistency, a strictly weaker local consistency.

6.3 Constraint Padding

In order to construct the rules for a conjunctive constraint as in the preceding subsection, the participating constraints must be on the same set of variables. This can be achieved by suitably extending the individual constraints to new variables, without constraining the latter. We call this padding. Consider a constraint $C \subseteq D^n$ and a variable $v \in D$ not constrained by C. We define $C'(X, v) = \{d \mid d[X] \in C \wedge d[v] \in D\}$, or concisely $C' = C \times D$, and find its rules by:

$$\begin{aligned}\mathcal{R}(C') = \mathit{closure}(\ & \{X \in S, v \in D \to y \neq a \mid (X \in S \to y \neq a) \in \mathcal{R}(C)\} \\ & \cup \\ & \{X \in S, y \in \{a\} \to v \neq b \mid \\ & \qquad\qquad (X \in S \to y \neq a) \in \mathcal{R}(C) \wedge b \in D\}\ )\end{aligned}$$

The first set in the union pads the input rules by simply adding a redundant test. In that way, all valid atomic rules with *rhs*'s on the variables of C are constructed. This set is closed under [derive, subsume] if $\mathcal{R}(C)$ is. The second set creates the rules for the new variable. Since v is not actually constrained, there can be no valid rule with a *rhs* on v. Hence all rules in the second set must be infeasible. Also observe that *exactly* all valid, atomic, infeasible rules on v are subsumed by this set. Although infeasible rules are redundant (with respect to a GAC-enforcing set of inclusion rules), it is essential to incorporate them here if the generated rule set is to serve as the input to another meta rule closure, such as for a conjunctive constraint.

In conclusion we can state that $\mathcal{R}(C')$ subsumes all atomic rules that are valid for C', if this is true for $\mathcal{R}(C)$ and C. Moreover, $\mathcal{R}(C')$ achieves then GAC on C'. The pre-closure processing is linear in the size of the set $\mathcal{R}(C)$.

Example 6.1 We pad the boolean constraint $not(x, y)$ to $not(x, y, z)$ with $z \in \{0, 1\}$.

$not(x, y)$:	$not(x, y, z)$:
$y \in \{0\} \to x \neq 0$	$y \in \{0\}, z \in \{0, 1\} \to x \neq 0$
$y \in \{1\} \to x \neq 1$	$y \in \{1\}, z \in \{0, 1\} \to x \neq 1$
$x \in \{0\} \to x \neq 0$	$x \in \{0\}, z \in \{0, 1\} \to x \neq 0$
$x \in \{1\} \to x \neq 1$	$x \in \{1\}, z \in \{0, 1\} \to x \neq 1$
	$x \in \{0\}, y \in \{0\} \to z \neq 0, z \neq 1$
	$x \in \{1\}, y \in \{1\} \to z \neq 0, z \neq 1$

6.4 Defining a Constraint by its Non-Solutions

A constraint for which rules should be generated can be defined *positively* by stating its solutions. This is the input of the RGA algorithm [3]. Sometimes, however, it may be more natural to define a constraint *negatively* by stating the tuples that are *not* solutions. Suppose we are given a set N of such non-solutions, or no-goods, defining the constraint C. We can write this as $C = D^{n+1} - N$ if C is on $n+1$ variables (recall $C \subseteq D^{n+1}$).

It is simple to obtain the corresponding GAC-enforcing rules as we can straightforwardly construct all valid atomic rules. Recall that every atomic rule corresponds to a non-solution of the constraint, and vice versa. Consequently, we find the seed rule set of all valid atomic rules by

$$\mathcal{R}_N = \{x_1 \in \{t_1\}, \ldots, x_n \in \{t_n\} \to x_0 \neq t_0\} \;|\; (t_0, \ldots, t_n) \text{ is a permutation of } t \in N\}$$

For instance, take a binary constraint $C(x, y)$ of which only $(1, 2)$ is not a solution. Then we obtain $\mathcal{R}_{\{(1,2)\}} = \{x \in \{1\} \to y \neq 2,\ y \in \{2\} \to x \neq 1\}$.

The closure $\mathcal{R}(C) = \mathit{closure}(\mathcal{R}_N)$ achieves GAC by Theorems 5.3,5.4. Constructing $\mathcal{R}_N$ is linear in the size of N: it produces $|N| \cdot (n+1)$ atomic rules.

Example 6.2 Define C over $x, y, z \in \{1..10\}$ as the constraint that at least one number in the ordered sequence x, y, z is not prime. The 4 non-solutions $(2, 3, 5), (2, 3, 7), (2, 5, 7), (3, 5, 7)$ are easily found. The rules are in Table 1.

Rule generation from the negative definition is particularly appropriate when the non-solutions are few, as in this example. Our implementation of the closure method took much less than a second to find the rules. In contrast, the RGA algorithm supplied with the 996 solution tuples did not return within 30 minutes. □

12 atomic rules:	8 rules after closure:	
$x \in \{2\}, y \in \{3\} \to z \neq 5$	$x \in \{2,3\},$	$y \in \{5\} \to z \neq 7$
$x \in \{2\}, z \in \{5\} \to y \neq 3$	$x \in \{2\},$	$y \in \{3\} \to z \neq 5$
$y \in \{3\}, z \in \{5\} \to x \neq 2$	$x \in \{2\},$	$y \in \{3,5\} \to z \neq 7$
$\vdots$	$x \in \{2,3\},$	$z \in \{7\} \to y \neq 5$
$y \in \{5\}, z \in \{7\} \to x \neq 3$	$x \in \{2\},$	$z \in \{5,7\} \to y \neq 3$
	$y \in \{3\},$	$z \in \{5,7\} \to x \neq 2$
	$y \in \{5\},$	$z \in \{7\} \to x \neq 3$
	$y \in \{3,5\},$	$z \in \{7\} \to x \neq 2$

Table 1
Rules of Example 6.2

6.5 Restricting a Constraint

Suppose a constraint is defined by restricting another reference constraint by discarding solutions. That is, $C'(X) = C(X) \wedge X \notin N$ where N collects the solutions of C that cease to be solutions of C'. Such a situation may occur in a dynamic setting, where propagation with a relaxation should take place before the actual constraint is learned. Assume thus that we know N and the rules $\mathcal{R}(C)$. We construct the seed rule set $\mathcal{R}_N$ precisely as in the preceding Subsection 6.5. Then we combine to

$$\mathcal{R}(C') = \mathit{closure}(\mathcal{R}(C) \cup \mathcal{R}_N)$$

The properties of $\mathcal{R}(C')$ depend on those of $\mathcal{R}(C)$, cf. Theorems 5.3,5.4.

Example 6.3 Suppose the boolean constraint *or* on $x, y, z \in \{0,1\}$ is the base constraint for or', defined by $or'(x,y,z) = or(x,y,z) \wedge (x,y,z) \neq (1,1,1)$, or alternatively, $or' = or - \{(1,1,1)\}$. We find $\mathcal{R}(or') = \mathit{closure}(\mathcal{R}(or) \cup \mathcal{R}_N)$ where $\mathcal{R}_N$ consists of the three rules $x \in \{1\}, y \in \{1\} \to z \in \{1\}$; $x \in \{1\}, z \in \{1\} \to y \in \{1\}$; $y \in \{1\}, z \in \{1\} \to x \in \{1\}$. $\square$

6.6 Universal Quantification

Assume that C constrains the variables Y and another variable $x \in D$. Consider the constraint $C' = \forall x.C$ on the variables Y. It has the solutions $\{t \mid \forall a \in D.\ \exists t' \in C.\ t'[x] = a \wedge t'[Y] = t\}$. We can derive rules for C' based on those for C by simply discarding all references to x:

$$\mathcal{R}(C') = \mathit{closure}(\{Z \in S \to y \neq a \mid (Z \in S, x \in S_x \to y \neq a) \in \mathcal{R}(C)\})$$

If $\mathcal{R}(C)$ contains or subsumes all atomic rules valid for C then the equivalent holds true for $\mathcal{R}(C')$ and C'. So again, using Theorems 5.3,5.4, we obtain a

GAC-enforcing rule set.

We argue for this as follows. Take a rule of C and a non-solution d excluded by it. d is a non-solution of C' as well, since the partial solution $d[Y]$ does not allow x to be all-quantified, $d[x]$ being the counter example. This means that we can correctly transform the rules of C into rules of C' as above.

It remains to consider completeness, that is, whether *all* atomic rules valid for C' are subsumed. Take such a rule, $Z \in S \to y \neq a$. But then some rule $Z \in S, x \in S_x \to y \neq a$ must be subsumed by $\mathcal{R}(C)$. For, otherwise all d with $\{d[Z]\} = S, d[y] = a$ were solutions, meaning that x could be all-quantified.

Again, constructing the pre-closure rule set is linear in the size of $\mathcal{R}(C)$.

Automatic rule generation for quantified constraints has been identified as desirable by [4], where a number of boolean constraints and associated rules for arc-consistency are discussed, for mixed sequences of universal and existential quantification. In the subsection following the example we deal with existential quantification.

Example 6.4 Consider the constraint *rcc8comp*, the composition constraint from the Region Connection Calculus [12]. $\mathit{rcc8comp}(R_{\mathsf{AB}}, R_{\mathsf{BC}}, R_{\mathsf{AC}})$ describes the possible spatial relations between three regions $\mathsf{A}, \mathsf{B}, \mathsf{C}$. The 8 relations are disjoint, meet, overlap, coveredby, covers, contains, inside, equal. Two examples for allowed tuples are (contains, inside, equal), (contains, inside, overlap), meaning that if region A contains region B and region B is inside (not touching the border of) region C then region A can be exactly equal to, or overlap with, region C.

The meaning of the universally quantified constraint $\forall R_{\mathsf{AC}}.\mathit{rcc8comp}(R_{\mathsf{AB}}, R_{\mathsf{BC}}, R_{\mathsf{AC}})$ is then exactly those pairs of relations between A/B, and B/C, such that *any* relation is possible between A/C. Only three such pairs of region relations are legal: $(R_{\mathsf{AB}}, R_{\mathsf{BC}}) \in \{(\mathsf{disjoint}, \mathsf{disjoint}), (\mathsf{inside}, \mathsf{inside}), (\mathsf{overlap}, \mathsf{overlap})\}$.

rcc8comp is defined by 193 solutions tuples, from which, via set complement to get the non-solutions, and closure, 912 GAC-enforcing rules are generated. For the all-quantified constraint one obtains 7 rules by the above procedure.

6.7 Existential Quantification

Existential quantification, or projection, is the dual to introduction of variables, Subsection 6.3. Assume that C constrains the variables Y and another variable $x \in D$. Consider the constraint $C' = \exists x.C$ on the variables Y that has the solutions $C' = \{d[Y] \mid d \in C\} = C[Y]$. Rule construction in this situation is different in the sense that it requires closure prior to modification of the rules:

$$\mathcal{R}(C') = \{Z \in S \to y \neq a \mid (Z \in S, x \in D \to y \neq a) \in \mathit{closure}(\mathcal{R}(C))\}$$

If $\mathcal{R}(C)$ subsumes all atomic rules valid for C, then so does $\mathcal{R}(C')$ for C', and $\mathcal{R}(C')$ enforces GAC on C'.

Consider $Z \in S, x \in D \to y \neq a$ from the set $closure(\mathcal{R}(C))$. It means that it is correct to conclude $y \neq a$ from $Z \in S$, independent of the value of x. Then, clearly, the rule $Z \in S \to y \neq a$ is valid for C'. Inversely, consider some atomic rule $Z \in S \to y \neq a$ valid for C'. It means that there does *not* exist any solution d of C with $\{d[Z]\} = S$ and $d[y] = a$. So the rule $Z \in S, x \in D \to y \neq a$ is valid for C, and must be subsumed by $closure\mathcal{R}(C)$.

Finally, notice that $\mathcal{R}(C')$ is closed under [subsume, derive], since any transformation possible in $\mathcal{R}(C')$ would have been possible in $\mathcal{R}(C)$ using the corresponding ancestor rules.

7 An Example

We implemented the closure method as a naive meta-rule fixpoint computation algorithm in the CLP system ECL^{i}PS^e. The program accepts expressions that describe the construction of inclusion rule sets according to the transformations in the preceding section. The output is a list of `CHR` rules.

FullAdder as a Composite Constraint

The basic rule set constructions of Section 6 enable us to derive a rule set for a composite constraint based only on the rules of its subconstraints, and without any reference to the extensional definition of any of the constraints. We demonstrate this with the example of the *fulladder* constraint. The *fulladder* gate adds two bits and a carry bit and produces the sum bit and output carry bit. It is often defined with the help of the primitive gates *and*, *or*, *xor* and three auxiliary variables:

$$\begin{aligned} fulladder(x,y,z,s,c) \quad \equiv \quad \exists\, c_1,c_2,s_1.\ & xor(x,y,s_1)\ \wedge \\ & and(x,y,c_1)\ \wedge \\ & and(z,s_1,c_2)\ \wedge \\ & or(c_1,c_2,c)\ \wedge \\ & xor(z,s_1,s) \end{aligned}$$

To indicate the individual transformations, we make the padding visible:

$$\begin{aligned} fulladder(x,y,z,s,c) \quad \equiv \quad & \exists\, c_1,c_2,s_1.\quad \exists\, h_1,\ldots,h_{25}. \\ & xor(x,y,h_1,s_1,h_2,h_3,h_4,h_5)\ \wedge \\ & and(x,y,h_6,h_7,c_1,h_8,h_9,h_{10})\ \wedge \\ & and(h_{11},h_{12},z,s_1,h_{13},c_2,h_{14},h_{15})\ \wedge \\ & or(h_{16},h_{17},h_{18},h_{19},c_1,c_2,c,h_{20})\ \wedge \\ & xor(h_{21},h_{22},z,s_1,h_{23},h_{24},h_{25},s) \end{aligned}$$

The input are the three base rule sets of *xor*, *and*, *or*. The rules are first padded so as to associate them to the same (auxiliary) variables. Then the rules corresponding to the conjunctive constraint are computed, and finally existential quantification removes the auxiliary variables again. This process generates 66 rules in about 0.5 *s*. The generated solver enforces GAC on the *fulladder* constraint.

It is useful to note that this transformation order requires the computation of only one rule set closure. Padding and joining the rules for the conjunctive constraint requires a post-processing closure, while existential quantification needs a pre-processing closure and yields a rule set that is closed. Recall that all transformation steps other than closure are just linear in the size of their input set.

The rules of *fulladder* propagate strictly stronger than the rules of its individual subconstraints. The CSP $\langle \mathit{fulladder}(x, y, z, s, c)\,;\, x, z, c \in \{0, 1\}, y \in \{1\}, s \in \{0\}\rangle$ is closed under the subconstraint rules, but a *fulladder* deduced rule propagates $c \in \{1\}$.

To compare with RGA, it is first necessary to create the entire *fulladder* extensional definition – the truth table – by a separate algorithm. The definition can then be given to RGA. It generates 52 rules. Since some of our 66 rules are infeasible while RGA does not produce such rules, we subjected both to a removal of redundant rules [5]. This reduced both rule sets to a unique set of 28 nonredundant rules.

8 Final Remarks

We made a first step into exploring deductive and incremental generation of rule-based constraint solvers. However, a number of topics deserve more work.

It would be desirable to obtain results for types of rules other than inclusion rules. There the problem of finding a candidate condition in derive arises.

For any rule type it would be very interesting to apply derive not exhaustively but only when it is *promising* to do so, i. e., to find strong descendant rules. This would mean a trade-off between the strength of the resulting consistency notion versus the cost of generating the solver and applying it.

The current implementation of the closure computation is unrefined. A good internal representation of rule sets should lead to substantial improvements in efficiency. Currently, unordered lists of inclusion rules are used, while an appropriate ordered data structure could support the filtering of the potential/interesting partners when considering subsume, derive for some rule. A related question is the one for the best strategy in finding the closure.

Acknowledgement

This work benefited from discussions with Lucas Bordeaux. We are grateful for the helpful suggestions of the reviewers.

References

[1] Slim Abdennadher and Thom Frühwirth. Using program analysis for integration and optimization of rule-based constraint solvers. In *Proc. of Journées Francophones de Progr. Logique et Progr. par Contraintes*, 2002.

[2] Slim Abdennadher and Christophe Rigotti. Automatic generation of rule-based solvers for intentionally defined constraints. *International Journal on Artificial Intelligence Tools*, 11(2), 2002.

[3] Krzysztof R. Apt and Eric Monfroy. Constraint programming viewed as rule-based programming. *Theory and Practice of Logic Programming*, 1(6), 2001.

[4] Lucas Bordeaux and Eric Monfroy. Beyond NP: Arc-consistency for quantified constraints. In *Proc. of Principles and Practice of Constraint Programming, Ithaka*, 2002.

[5] Sebastian Brand. A note on redundant rules in rule-based constraint programming. In *Recent Advances in Constraints*, volume 2627 of *Lecture Notes in Artifical Intelligence*. Springer, 2003.

[6] Björn Carlson, Mats Carlsson, and Sverker Janson. The implementation of AKL(FD). In *Proc. of International Symposium on Logic Programming*, 1995.

[7] Yves Caseau, François-Xavier Josset, and François Laburthe. CLAIRE: Combining sets, search and rules to better express algorithms. *Theory and Practice of Logic Programming*, 2(6), 2002.

[8] Carlos Castro. Building constraint satisfaction problem solvers using rewrite rules and strategies. *Fundamenta Informaticae*, 34(3), 1998.

[9] Rina Dechter and Peter van Beek. Local and global relational consistency. *Theoretical Computer Science*, 173(1), 1997.

[10] Thom Frühwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming*, 1998.

[11] Claude Kirchner and Christophe Ringeissen. Rule-based constraint programming. *Fundamenta Informaticae*, 34(3), 1998.

[12] David A. Randell, Zhan Cui, and Anthony Cohn. A spatial logic based on regions and connection. In *Proc. of Principles of Knowledge Representation and Reasoning*. Morgan Kaufmann, 1992.

[13] Christophe Ringeissen and Eric Monfroy. Generating propagation rules for finite domains via unification in finite algebras. In *New Trends in Constraints*, volume 1865 of *Lecture Notes in Artifical Intelligence*. Springer, 2000.

[14] Vijay Anand Saraswat. *Concurrent Constraint Programming*. MIT Press, 1993.

[15] Gert Smolka. The Oz programming model. In *Computer Science Today*, volume 1000 of *Lecture Notes in Computer Science*. Springer, 1995.