Schedulers and Redundancy for a Class of Constraint Propagation Rules

Sebastian Brand and Krzysztof R. Apt

Abstract

We study here schedulers for a class of rules that naturally arise in the context of rule-based constraint programming. We systematically derive a scheduler for them from a generic iteration algorithm of (Apt 2000). We apply this study to so-called membership rules of (Apt and Monfroy 2001). This leads to an implementation that yields a considerably better performance for these rules than their execution as standard CHR rules. Finally, we show how redundant rules can be identified and how appropriately reduced sets of rules can be computed.

KEYWORDS: constraint programming, rule-based programming, constraint propagation

1 Introduction

In this paper we identify a class of rules that naturally arise in the context of constraint programming represented by means of rule-based programming and study efficient schedulers for these rules. We call these rules *propagation* rules, in short *prop* rules. An important class of *prop* rules are the *membership rules*, introduced in (Apt and Monfroy 2001). An example of a membership rule is

$$x \in \{3,4,8\}, y \in \{1,2\} \rightarrow z \neq 2.$$

Informally, it should be read as follows: if the domain of x is included in $\{3,4,8\}$ and the domain of y is included in $\{1,2\}$, then 2 is removed from the domain of z.

In the computations of constraint programs the variable domains gradually shrink. So if the domain of x is included in $\{3,4,8\}$, then it will remain so during the computation. In turn, if 2 is removed from the domain of z, then this removal operation does not need to be repeated. The *prop* rules generalize these observations to specific conditions on the rule condition and body.

In the resulting approach to constraint programming the computation process is limited to a repeated application of the *prop* rules intertwined with splitting (labeling). So the viability of this approach crucially depends on the availability of efficient schedulers for such rules. This motivates the work here reported. We provide an abstract framework for such schedulers and use it as a basis for an implementation.

More precisely, to obtain appropriate schedulers for the *prop* rules we use the generic approach to constraint propagation algorithms introduced in (Apt 1999)

and (Apt 2000). In this framework one proceeds in two steps. First, a generic iteration algorithm on partial orderings is introduced and proved correct in an abstract setting. Then it is instantiated with specific partial orderings and functions to obtain specific constraint propagation algorithms. In this paper, as in (Apt 2000), we take into account information about the scheduled functions, which are here the *prop* rules. This yields a specific scheduler in the form of an algorithm called R.

We then show by means of an implementation how this abstract framework can be used to obtain a scheduler for the membership rules. The relevance of the membership rules for constraint satisfaction problems (CSPs) with finite domains stems from the following observations made in (Apt and Monfroy 2001):

- constraint propagation can be naturally achieved by repeated application of the membership rules;
- in particular the notions of arc consistency and hyper-arc consistency can be characterized in terms of the membership rules;
- for constraints explicitly defined on small finite domains all valid membership rules can be automatically generated; (For a more referent work on the subject of an automatic generation of such rules see (Abdennadher and Rigotti 2001).)
- many rules of the CHR language (Constraint Handling Rules) of (Frühwirth 1998) that are used in specific constraint solvers are in fact membership rules. In the logic programming approach to constraint programming CHR is the language of choice to write constraint solvers.

The implementation is provided as an ECL^{i}PS^e program that accepts a set of membership rules as input and constructs an ECL^{i}PS^e program that is the instantiation of the R algorithm for this set of rules. Since membership rules can be naturally represented as CHR propagation rules, one can assess this implementation by comparing it with the performance of the standard implementation of membership rules in the CHR language. By means of various benchmarks we found that our implementation is considerably faster than CHR. It is important to stress that this implementation was obtained by starting from "first principles" in the form of a generic iteration algorithm on an arbitrary partial ordering. This shows the practical benefits of studying the constraint propagation process on an abstract level.

Additionally, we clarify how to identify *prop* rules that are redundant for the considered computations and how to compute appropriately reduced sets of rules. The concept of redundancy is formalized here in a "semantic" sense that takes into account the type of computations performed by means of the considered rules. We provide a simple test for redundancy that leads to a natural way of computing minimal sets of rules in an appropriate sense. The computation of a specific minimal set for the membership rules is then implemented in ECL^{i}PS^e.

CHR is available in a number of languages including the ECL^{i}PS^e and the Sicstus Prolog systems. In both cases CHR programs are compiled into the host language, so either ECL^{i}PS^e or the Sicstus Prolog. There is also a recent implementation in Java, see (Abdennadher et al. 2001). To make CHR usable it is important that its

implementation does not incur too much overhead. And indeed a great deal of effort was spent on implementing CHR efficiently. For an account of the most recent implementation see (Holzbaur et al. 2001). Since, as already mentioned above, many CHR rules are membership rules, our approach provides a better implementation of a subset of CHR. This, hopefully, may lead to new insights into a design and implementation of languages appropriate for writing constraint solvers.

The paper is organized as follows. In the next section we briefly recall the original generic iteration algorithm of (Apt 2000) and modify it successively to obtain the iteration algorithm R for *prop* rules. An important novelty is the preprocessing phase during which we analyze the mutual dependencies between the rules. This allows us to remove permanently some rules during the iteration process. This permanent removal of the scheduled rules is particularly beneficial in the context of constraint programming where it leads to accumulated savings when constraint propagation is intertwined with splitting.

In Section 3 we recall the membership rules of (Apt and Monfroy 2001) and show that they are *prop* rules. Then in Section 4 we recall the relevant aspects of the CHR language, discuss the implementation of the R algorithm and present several benchmarks. Finally, in Section 5 we deal with the subject of redundancy of *prop* rules.

2 Revisions of the Generic Iteration Algorithm

2.1 The Original Algorithm

Let us begin our presentation by recalling the generic algorithm of (Apt 2000). We slightly adjust the presentation to our purposes by assuming that the considered partial ordering also has the greatest element $\top$.

So we consider a partial ordering $(D, \sqsubseteq)$ with the least element $\bot$ and the greatest element $\top$, and a set of functions $F := \{f_1, \ldots, f_k\}$ on D. We are interested in functions that satisfy the following two properties.

Definition 2.1
- f is called *inflationary* if $x \sqsubseteq f(x)$ for all x.
- f is called *monotonic* if $x \sqsubseteq y$ implies $f(x) \sqsubseteq f(y)$ for all x, y. $\qquad\square$

The following algorithm is used to compute the least common fixpoint of the functions from F.

GENERIC ITERATION ALGORITHM (GI)

$d := \bot;$
$G := F;$
while $G \neq \varnothing$ **and** $d \neq \top$ **do**
    choose $g \in G;$
    $G := G - \{g\};$
    $G := G \cup update(G, g, d);$
    $d := g(d)$
end

where for all G, g, d the set of functions $update(G, g, d)$ from F is such that

A $\{f \in F - G \mid f(d) = d \ \wedge \ f(g(d)) \neq g(d)\} \subseteq update(G, g, d)$,

B $g(d) = d$ implies that $update(G, g, d) = \varnothing$,

C $g(g(d)) \neq g(d)$ implies that $g \in update(G, g, d)$.

Intuitively, assumption **A** states that $update(G, g, d)$ contains at least all the functions from $F - G$ for which the "old value", d, is a fixpoint but the "new value", $g(d)$, is not. So at each loop iteration such functions are added to the set G. In turn, assumption **B** states that no functions are added to G in case the value of d did not change. Assumption **C** provides information when g is to be added back to G as this information is not provided by **A**. On the whole, the idea is to keep in G at least all functions f for which the current value of d is not a fixpoint.

The use of the condition $d \neq \top$, absent in the original presentation, allows us to leave the **while** loop earlier. Our interest in the GI algorithm is clarified by the following result.

Theorem 2.2 (Correctness)

Suppose that all functions in F are inflationary and monotonic and that $(D, \sqsubseteq)$ is finite and has the least element $\bot$ and the greatest element $\top$. Then every execution of the GI algorithm terminates and computes in d the least common fixpoint of the functions from F.

Proof. Consider the predicate I defined by:

$$I := (\forall f \in F - G \ f(d) = d) \ \wedge \ (\forall f \in F \ f(\top) = \top).$$

Note that I is established by the assignment $G := F$. Moreover, it is easy to check that by virtue of the assumptions **A**, **B** and **C** the predicate I is preserved by each **while** loop iteration. Thus I is an invariant of the **while** loop of the algorithm. Hence upon its termination

$$(G = \varnothing \ \vee \ d = \top) \ \wedge \ I$$

holds, which implies

$$\forall f \in F \ f(d) = d.$$

This implies that the algorithm computes in d a common fixpoint of the functions from F.

The rest of the proof is the same as in (Apt 2000). So the fact that this is the least common fixpoint follows from the assumption that all functions are monotonic.

In turn, termination is established by considering the lexicographic ordering of the strict partial orderings $(D, \sqsupset)$ and $(\mathcal{N}, <)$, defined on the elements of $D \times \mathcal{N}$ by

$$(d_1, n_1) <_{lex} (d_2, n_2) \quad \text{iff} \quad d_1 \sqsupset d_2 \ \text{ or } \ (d_1 = d_2 \ \text{ and } \ n_1 < n_2).$$

Then with each **while** loop iteration of the algorithm the pair $(d, card \ G)$, where $card \ G$ is the cardinality of the set G, strictly decreases in the ordering $<_{lex}$. $\quad \square$

4

2.2 Removing Functions

We now revise the GI algorithm by modifying dynamically the set of functions that are being scheduled. The idea is that, whenever possible, we remove functions from the set F. This will allow us to exit the loop earlier which speeds up the execution of the algorithm.

To this end we assume that for each function $g \in F$ and each element $d \in D$, two lists of functions from F are given, $friends(g, d)$ and $obviated(g, d)$, to be instantiated below. We then modify the GI algorithm in such a way that each application of g to d will be immediately followed by the applications of all functions from $friends(g, d)$ and by a removal of the functions from $friends(g, d)$ and from $obviated(g, d)$ from F and G. Below we identify a condition, (1), on $friends(g, d)$ and $obviated(g, d)$ that ensures correctness of this scheduling strategy. Informally, this condition states that after an application of all the functions from $friends(g, d)$ the functions from $friends(g, d)$ and from $obviated(g, d)$ will not change anymore the subsequent values of d.

This modified algorithm has the following form.[1]

REVISED GENERIC ITERATION ALGORITHM (RGI)

$d := \perp$;
$F_0 := F$;
$G := F$;
while $G \neq \varnothing$ **and** $d \neq \top$ **do**
  choose $g \in G$;
  $G := G - \{g\}$;
  $F := F - (friends(g, d) \cup obviated(g, d))$;
  $G := G - (friends(g, d) \cup obviated(g, d))$;
  $G := G \cup update(G, h, d)$,
    where $friends(g, d) = [g_1, \ldots, g_k]$ and $h = g \circ g_1 \circ \ldots \circ g_k$;
  $d := h(d)$
end

We now formalize the condition under which the Correctness Theorem 2.2 holds with the GI algorithm replaced by the RGI algorithm. To this end we consider the following property.

Definition 2.3
Suppose $d \in D$ and $f \in F$. We say that f is *stable above* d if $d \sqsubseteq e$ implies $f(e) = e$. We then say that f is *stable* if it is stable above $f(d)$, for all d. □

That is, f is stable if for all d and e, $f(d) \sqsubseteq e$ implies $f(e) = e$. So stability implies idempotence, which means that $f(f(d)) = f(d)$, for all d. Moreover, if d and $f(d)$ are comparable for all d, then stability implies inflationarity. Indeed, if $d \sqsubseteq f(d)$, then the claim holds vacuously. And if $f(d) \sqsubseteq d$, then by stability $f(d) = d$.

---

[1] We need in it lists instead of sets since the considered functions will be applied in a specific order. But in some places, for simplicity, we identify these lists with the sets.

Consider now the following condition

$$\forall d\, \forall e \sqsupseteq g \circ g_1 \circ \ldots \circ g_k(d)\ \forall f \in friends(g,d) \cup obviated(g,d)\ (f(e) = e), \qquad (1)$$

where $friends(g,d) = [g_1, \ldots, g_k]$. That is, for all elements d, each function f in $friends(g,d) \cup obviated(g,d)$ is stable above $g \circ g_1 \circ \ldots \circ g_k(d)$, where $friends(g,d)$ is the list $[g_1, \ldots, g_k]$. The following result holds.

Theorem 2.4
Suppose that all functions in F are inflationary and monotonic and that $(D, \sqsubseteq)$ is finite and has the least element $\perp$ and the greatest element $\top$. Additionally, suppose that for each function $g \in F$ and $d \in D$ two lists of functions from F are given, $friends(g,d)$ and $obviated(g,d)$ such that condition (1) holds.

Then the Correctness Theorem 2.2 holds with the GI algorithm replaced by the RGI algorithm.

Proof. In view of condition (1) the following statement is an invariant of the **while** loop:

$$\forall f \in F - G(f(d) = d) \wedge \forall f \in F\ (f(\top) = \top) \wedge \forall f \in F_0 - F\ \forall e \sqsupseteq d\ (f(e) = e). \qquad (2)$$

So upon termination of the algorithm the conjunction of this invariant with the negation of the loop condition, i.e.,

$$G = \varnothing \ \vee \ d = \top$$

holds, which implies that $\forall f \in F_0\ (f(d) = d)$.

The rest of the proof is the same. $\qquad\square$

2.3 Functions in the Form of Rules

In what follows we consider the situation when the scheduled functions are of a specific form $b \to g$, where b is a *condition* and g a function, that we call a *body*. We call such functions *rules*.

First, we explain how rules are applied. Given an element d of D a condition b is evaluated in d. The outcome is either *true*, that we denote by $holds(b,d)$, or *false*.

Given a rule $b \to g$ we define then its application to d as follows:

$$(b \to g)(d) := \begin{cases} g(d) & \text{if } holds(b,d) \\ d & \text{otherwise} \end{cases}$$

We are interested in a specific type of conditions and rules.

Definition 2.5
Consider a partial ordering $(D, \sqsubseteq)$.

- We say that a condition b is *monotonic* if for all d, e we have that $holds(b,d)$ and $d \sqsubseteq e$ implies $holds(b,e)$.
- We say that a condition b is *precise* if the least d exists such that $holds(b,d)$. We call then d the *witness* for b.
- We call a rule $b \to g$ a *prop* rule if b is monotonic and precise and g is stable.
$\qquad\square$

To see how natural this class of rules is consider the following example.

6

Example 2.6
Take a set A and consider the partial ordering

$$(\mathcal{P}(A), \subseteq).$$

In this ordering the empty set $\varnothing$ is the least element and A is the greatest element. We consider rules of the form

$$B \to G,$$

where $B, G \subseteq A$.

To clarify how they are applied to subsets of A we first stipulate for $E \subseteq A$

$$holds(B, E) \text{ iff } B \subseteq E.$$

Then we view a set G as a function on $\mathcal{P}(A)$ by putting

$$G(E) := G \cup E.$$

This determines the rule application of $B \to G$.

It is straightforward to see that such rules are *prop* rules. In particular, the element B of $\mathcal{P}(A)$ is the witness for the condition B. For the stability of G take $E \subseteq A$ and suppose $G(E) \subseteq F$. Then $G(E) = G \cup E$, so $G \cup E \subseteq F$, which implies $G \cup F = F$, i.e., $G(F) = F$.

Such rules can be instantiated to many situations. For example, we can view the elements of the set A as primitive constraints. Then each rule $B \to G$ is a natural operation on the constraint store: if all constraints in B are present in the store, then add to it all constraints in G.

Alternatively, we can view A as a set of some atomic formulas and each rule $B \to G$ as a proof rule, usually written as

$$\frac{B}{G}$$

The minor difference with the usual proof-theoretic framework is that rules have then a single conclusion. An axiom is then a rule with the empty set $\varnothing$ as the condition. A closure under such a set of rules is then the set of all (atomic) theorems that can be proved using them.

The algorithm presented below can in particular be used to compute efficiently the closure under such proof rules given a finite set of atomic formulas A. $\qquad\square$

We now modify the RGI algorithm for the case of *prop* rules. In the algorithm below we take into account that an application of a rule is a two step process: testing of the condition followed by a conditional application of the body. This will allow us to drop the parameter d from the lists *friends*(g, d) and *obviated*(g, d) and consequently to construct such lists before the execution of the algorithm. The list *friends*(g) will be constructed in such a way that we shall not need to evaluate the conditions of its rules: they will all hold. The specific construction of the lists *friends*(g) and *obviated*(g) that we use here will be provided in the second algorithm, called FRIENDS AND OBVIATED ALGORITHM.

RULES ALGORITHM (R)

$d := \perp$;
$F_0 := F$;
$G := F$;
while $G \neq \varnothing$ **and** $d \neq \top$ **do**
  choose $f \in G$; suppose f is $b \to g$;
  $G := G - \{b \to g\}$;
  if $holds(b, d)$ **then**
    $F := F - (friends(b \to g) \cup obviated(b \to g))$;
    $G := G - (friends(b \to g) \cup obviated(b \to g))$;
    $G := G \cup update(G, h, d)$,
      where $friends(b \to g) = [b_1 \to g_1, \ldots, b_k \to g_k]$ and $h = g \circ g_1 \circ \ldots \circ g_k$;
    $d := h(d)$
  else if $\forall e \sqsupseteq d \ \neg holds(b, e)$ **then**
    $F := F - \{b \to g\}$
  end
end

Again, we are interested in identifying conditions under which the Correctness Theorem 2.2 holds with the GI algorithm replaced by the R algorithm. To this end, given a rule $b \to g$ in F and $d \in D$ define as follows:

$$friends(b \to g, d) \quad := \quad \begin{cases} friends(b \to g) & \text{if } holds(b, d), \\ [\,] & \text{otherwise} \end{cases}$$

and

$$obviated(b \to g, d) \quad := \quad \begin{cases} obviated(b \to g) & \text{if } holds(b, d), \\ [b \to g] & \text{if } \forall e \sqsupseteq d \ \neg holds(b, e), \\ [\,] & \text{otherwise} \end{cases}$$

We obtain the following counterpart of the Correctness Theorem 2.2.

Theorem 2.7 (Correctness)
Suppose that all functions in F are *prop* rules of the form $b \to g$, where g is inflationary and monotonic, and that $(D, \sqsubseteq)$ is finite and has the least element $\perp$ and the greatest element $\top$. Further, assume that for each rule $b \to g$ the lists $friends(b \to g, d)$ and $obviated(b \to g, d)$ defined as above satisfy condition (1) and the following condition:

$$\forall d(\ b' \to g' \in friends(b \to g) \ \wedge \ holds(b, d) \ \to \ \forall e \sqsupseteq g(d) \ holds(b', e) \ ). \quad (3)$$

Then the Correctness Theorem 2.2 holds with the GI algorithm replaced by the R algorithm.

Proof. It suffices to show that the R algorithm is an instance of the RGI algorithm. On the account of condition (3) and the fact that the rule bodies are inflationary functions, $holds(b, d)$ implies that

$$((b \to g) \circ (b_1 \to g_1) \circ \ldots \circ (b_k \to g_k))(d) \ = \ (g \circ g_1 \circ \ldots \circ g_k)(d),$$

where $friends(b \to g) = [b_1 \to g_1, \ldots, b_k \to g_k]$. This takes care of the situation in which $holds(b, d)$ is true.

In turn, the definition of $friends(b \to g, d)$ and $obviated(b \to g, d)$ and assumption **B** take care of the situation when $\neg holds(b, d)$. When the condition b fails for all $e \sqsupseteq d$ we can conclude that for all such e we have $(b \to g)(e) = e$. This allows us to remove at that point of the execution the rule $b \to g$ from the set F. This amounts to adding $b \to g$ to the set $obviated(b \to g, d)$ at runtime. Note that condition (1) is then satisfied. $\qquad\square$

We now provide an explicit construction of the lists *friends* and *obviated* for a rule $b \to g$ in the form of the following algorithm. $\mathtt{GI}(d)$ stands here for the $\mathtt{GI}$ algorithm activated with $\perp$ replaced by d and the considered set of rules as the set of functions F. Further, given an execution of $\mathtt{GI}(e)$, we call here a rule g *relevant* if at some point $g(d) \neq d$ holds after the "choose $g \in G$" action.

FRIENDS AND OBVIATED ALGORITHM (F & O)

$w :=$ witness of b;
$d := \mathtt{GI}(g(w))$;
$friends(b \to g) :=$ list of the relevant rules $h \in F$ in the execution of $\mathtt{GI}(g(w))$;
$obviated(b \to g) := [\ ]$;
for each $(b' \to g') \in F - friends(b \to g)$ **do**
    if $g'(d) = d$ **or** $\forall e \sqsupseteq d \ \neg holds(b', e)$ **then**
        $obviated(b \to g) := [\ b' \to g' \mid obviated(b \to g)\ ]$
    end
end

Note that $b \to g$ itself is not contained in $friends(b \to g)$ as it is a *prop* rule, however it is in $obviated(b \to g)$, since by the stability of g $g(d) = d$ holds.

The following observation now shows the adequacy of the **F & O** algorithm for our purposes.

Lemma 2.8
Upon termination of the **F & O** algorithm conditions (1) and (3) hold, where the lists $friends(b \to g, d)$ and $obviated(b \to g, d)$ are defined as before Theorem 2.7. $\square$

Let us summarize now the findings of this section that culminated in the **R** algorithm. Assume that all functions are of the form of the rules satisfying the conditions of the Correctness Theorem 2.7. Then in the **R** algorithm, each time the evaluation of the condition b of the selected rule $b \to g$ succeeds,

- the rules in the list $friends(b \to g)$ are applied directly without testing the value of their conditions,
- the rules in $friends(b \to g) \cup obviated(b \to g)$ are permanently removed from the current set of functions G and from F.

2.4 Recomputing of the Least Fixpoints

Another important optimization takes place when the R algorithm is repeatedly applied to compute the least fixpoint. More specifically, consider the following sequence of actions:

- we compute the least common fixpoint d of the functions from F,
- we move from d to an element e such that $d \sqsubseteq e$,
- we compute the least common fixpoint above e of the functions from F.

Such a sequence of actions typically arises in the framework of CSPs, further studied in Section 3. The computation of the least common fixpoint d of the functions from F corresponds there to the constraint propagation process for a constraint C. The moving from d to e such that $d \sqsubseteq e$ corresponds to splitting or constraint propagation involving another constraint, and the computation of the least common fixpoint above e of the functions from F corresponds to another round of constraint propagation for C.

Suppose now that we computed the least common fixpoint d of the functions from F using the RGI algorithm or its modification R for the rules. During its execution we permanently removed some functions from the set F. These functions are then not needed for computing the least common fixpoint above e of the functions from F. The precise statement is provided in the following simple, yet crucial, theorem.

Theorem 2.9
Suppose that all functions in F are inflationary and monotonic and that $(D, \sqsubseteq)$ is finite. Suppose that the least common fixpoint d_0 of the functions from F is computed by means of the RGI algorithm or the R algorithm. Let F_{fin} be the final value of the variable F upon termination of the RGI algorithm or of the R algorithm.

Suppose now that $d_0 \sqsubseteq e$. Then the least common fixpoint e_0 above e of the functions from F coincides with the least common fixpoint above e of the functions from F_{fin}.

Proof. Take a common fixpoint e_1 of the functions from F_{fin} such that $e \sqsubseteq e_1$. It suffices to prove that e_1 is common fixpoint of the functions from F. So take $f \in F - F_{fin}$. Since condition (2) is an invariant of the main **while** loop of the RGI algorithm and of the R algorithm, it holds upon termination and consequently f is stable above d_0. But $d_0 \sqsubseteq e$ and $e \sqsubseteq e_1$, so we conclude that $f(e_1) = e_1$. □

Intuitively, this result means that if after splitting we relaunch the same constraint propagation process we can disregard the removed functions.

In the next section we instantiate the R algorithm by a set of rules that naturally arise in the context of constraint satisfaction problems with finite domains. In Section 4 we assess the practical impact of the discussed optimizations.

3 Concrete Framework

We now proceed with the main topic of this paper, the schedulers for the rules that naturally arise in the context of constraint satisfaction problems. First we recall briefly the necessary background on the constraint satisfaction problems.

3.1 Constraint Satisfaction Problems

Consider a sequence of variables $X := x_1, \ldots, x_n$ where $n \geqslant 0$, with respective domains $D_1, \ldots, D_n$ associated with them. So each variable x_i ranges over the domain D_i. By a *constraint* C on 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.

Recall that a *constraint satisfaction problem*, in short CSP, consists of a finite sequence of variables X with respective domains $\mathcal{D}$, together with a finite set $\mathcal{C}$ of constraints, each on a subsequence of X. We write it as $\langle \mathcal{C} \; ; \; x_1 \in D_1, \ldots, x_n \in D_n \rangle$, where $X := x_1, \ldots, x_n$ and $\mathcal{D} := D_1, \ldots, D_n$.

By a *solution* to $\langle \mathcal{C} \; ; \; x_1 \in D_1, \ldots, x_n \in D_n \rangle$ we mean an element $d \in D_1 \times \ldots \times D_n$ such that for each constraint $C \in \mathcal{C}$ on a sequence of variables X we have $d[X] \in C$. We call a CSP *consistent* if it has a solution. Two CSPs with the same sequence of variables are called *equivalent* if they have the same set of solutions.

3.2 Partial Orderings

With each CSP $\mathcal{P} := \langle \mathcal{C} \; ; \; x_1 \in D_1, \ldots, x_n \in D_n \rangle$ we associate now a specific partial ordering. Initially we take the Cartesian product of the partial orderings $(\mathcal{P}(D_1), \supseteq), \ldots, (\mathcal{P}(D_n), \supseteq)$. So this ordering is of the form

$$(\mathcal{P}(D_1) \times \ldots \times \mathcal{P}(D_n), \supseteq)$$

where we interpret $\supseteq$ as the Cartesian product of the reversed subset ordering. The elements of this partial ordering are sequences $(E_1, \ldots, E_n)$ of respective subsets of $(D_1, \ldots, D_n)$ ordered by the component-wise reversed subset ordering. Note that in this ordering $(D_1, \ldots, D_n)$ is the least element while

$$\underbrace{(\varnothing, \ldots, \varnothing)}_{n \text{ times}}$$

is the greatest element. However, we would like to identify with the greatest element all sequences that contain as an element the empty set. So we divide the above partial ordering by the equivalence relation $R_\varnothing$ according to which

$$(E_1, \ldots, E_n) \; R_\varnothing \; (F_1, \ldots, F_n) \text{ iff } (E_1, \ldots, E_n) = (F_1, \ldots, F_n)$$
$$\text{or } (\exists i \; E_i = \varnothing \text{ and } \exists j \; F_j = \varnothing).$$

It is straightforward to see that $R_\varnothing$ is indeed an equivalence relation.

In the resulting quotient ordering there are two types of elements: the sequences $(E_1, \ldots, E_n)$ that do not contain the empty set as an element, that we continue to present in the usual way with the understanding that now each of the listed sets is non-empty, and one "special" element equal to the equivalence class consisting of all sequences that contain the empty set as an element. This equivalence class is the greatest element in the resulting ordering, so we denote it by $\top$. In what follows we denote this partial ordering by $(D_{\mathcal{P}}, \sqsubseteq)$.

11

3.3 Membership Rules

Fix now a specific CSP $\mathcal{P} := \langle \mathcal{C} \; ; \; x_1 \in D_1, \ldots, x_n \in D_n \rangle$ with finite domains. We recall the rules introduced in (Apt and Monfroy 2001). They are called *membership rules* and are of the form

$$y_1 \in S_1, \ldots, y_k \in S_k \;\; \to \;\; z_1 \neq a_1, \ldots, z_m \neq a_m,$$

where

- $y_1, \ldots, y_k$ are pairwise different variables from the set $\{x_1, \ldots, x_n\}$ and $S_1, \ldots, S_k$ are subsets of the respective variable domains,
- $z_1, \ldots, z_m$ are variables from the set $\{x_1, \ldots, x_n\}$ and $a_1, \ldots, a_m$ are elements of the respective variable domains.[2]

Note that we do not assume that the variables $z_1, \ldots, z_m$ are pairwise different.

The computational interpretation of such a rule is:

if for $i \in [1..k]$ the current domain of the variable y_i is included in the set S_i, then for $j \in [1..m]$ remove the element a_i from the domain of z_i.

When each set S_i is a singleton, we call a membership rule an *equality rule*.

Let us mention here that in (Apt and Monfroy 2001) the interpretation of the conditions of an equality rule is slightly different, as it is stipulated that the current domain of the variable y_i is to be equal to the singleton set S_i. However, in the discussed algorithms the membership rules are applied only when all variable domains are non-empty and then both interpretations coincide.

Let us reformulate this interpretation so that it fits the framework considered in the previous section. To this end we need to clarify how to

- evaluate the condition of a membership rule in an element of the considered partial ordering,
- interpret the conclusion of a membership rule as a function on the considered partial ordering.

Let us start with the first item.

Definition 3.1

Given a variable y with the domain D_y and $E \in \mathcal{P}(D_y)$ we define

$$holds(y \in S, E) \quad \text{iff} \quad E \subseteq S,$$

and extend the definition to the elements of the considered ordering $(D_{\mathcal{P}}, \sqsubseteq)$ by putting

$$holds(y \in S, (E_1, \ldots, E_n)) \quad \text{iff} \quad E_k \subseteq S, \quad \text{where we assumed that } y \text{ is } x_k, \text{ and}$$
$$holds(y \in S, \top).$$

---

[2] In (Apt and Monfroy 2001) it is also assumed that the lists $y_1, \ldots, y_k$ and $z_1, \ldots, z_m$ have no variable in common. We drop this condition so that we can combine the membership rules.

Furthermore we interpret a sequence of conditions as a conjunction, by putting

$$holds((y_1 \in S_1, \ldots, y_k \in S_k), (E_1, \ldots, E_n))$$
$$\text{iff} \quad holds(y_i \in S_i, (E_1, \ldots, E_n)) \quad \text{for } i \in [1..k].$$

$\square$

Concerning the second item we proceed as follows.

Definition 3.2
Given a variable z with the domain D_z we interpret the atomic formula $z \neq a$ as a function on $\mathcal{P}(D_z)$, defined by:

$$(z \neq a)(E) := E - \{a\}.$$

Then we extend this function to the elements of the considered ordering $(D_\mathcal{P}, \sqsubseteq)$ as follows:

- on the elements of the form $(E_1, \ldots, E_n)$ we put

$$(z \neq a)(E_1, \ldots, E_n) := (E_1', \ldots, E_n'),$$

where

— if $z \equiv x_i$, then $E_i' = E_i - \{a\}$,
— if $z \not\equiv x_i$, then $E_i' = E_i$.

If the resulting sequence $(E_1', \ldots, E_n')$ contains the empty set, we replace it by $\top$,

- on the element $\top$ we put $(z \neq a)(\top) := \top$.

Finally, we interpret a sequence $z_1 \neq a_1, \ldots, z_m \neq a_m$ of atomic formulas by interpreting each of them in turn. $\square$

As an example take the CSP

$$\mathcal{P} := \langle \mathcal{C} \; ; \; x_1 \in \{a, b, c\}, x_2 \in \{a, b, c\}, x_3 \in \{a, b, c\}, x_4 \in \{a, b, c\} \rangle$$

and consider the membership rule

$$r := \quad x_1 \in \{a, b\}, x_2 \in \{b\} \quad \rightarrow \quad x_3 \neq a, x_3 \neq b, x_4 \neq a.$$

Then we have

$$
\begin{aligned}
r(\{a\}, \{b\}, \{a, b, c\}, \{a, b\}) &= (\{a\}, \{b\}, \{c\}, \{b\}), \\
r(\{a, b, c\}, \{b\}, \{a, b, c\}, \{a, b\}) &= (\{a, b, c\}, \{b\}, \{a, b, c\}, \{a, b\}), \\
r(\{a, b\}, \{b\}, \{a, b\}, \{a, b\}) &= \top.
\end{aligned}
$$

In view of the Correctness Theorem 2.7 the following observation allows us to apply the R algorithm when each function is a membership rule and when for each rule $b \rightarrow g$ the lists *friends*$(b \rightarrow g)$ and *obviated*$(b \rightarrow g)$ are constructed by the F & O algorithm.

13

Note 3.3

Consider the partial ordering $(D_\mathcal{P}, \sqsubseteq)$.

(i) Each membership rule is a *prop* rule.

(ii) Each function $z_1 \neq a_1, \ldots, z_m \neq a_m$ on $D_\mathcal{P}$ is

- inflationary,
- monotonic.

$\square$

To be able to instantiate the algorithm R with the membership rules we still need to define the set $update(G, g, d)$. In our implementation we chose the following simple definition:

$$update(G, b \rightarrow g, d) := \begin{cases} F - G & \text{if } holds(b, d) \text{ and } g(d) \neq d, \\ \varnothing & \text{otherwise.} \end{cases}$$

To illustrate the intuition behind the use of the lists $friends(b \rightarrow g)$ and $obviated(b \rightarrow g)$ take the above CSP

$$\mathcal{P} := \langle \mathcal{C} \; ; \; x_1 \in \{a, b, c\}, x_2 \in \{a, b, c\}, x_3 \in \{a, b, c\}, x_4 \in \{a, b, c\} \rangle$$

and consider the membership rules

$$
\begin{aligned}
r_1 &:= & x_1 \in \{a, b\} & & \rightarrow & \quad x_2 \neq a, x_4 \neq b, \\
r_2 &:= & x_1 \in \{a, b\}, x_2 \in \{b, c\} & & \rightarrow & \quad x_3 \neq a, \\
r_3 &:= & x_2 \in \{b\} & & \rightarrow & \quad x_3 \neq a, x_4 \neq b.
\end{aligned}
$$

Then upon application of rule r_1 rule r_2 can be applied without evaluating its condition and subsequently rule r_3 can be deleted without applying it. So we can put rule r_2 into $friends(r_1)$ and rule r_3 into $obviated(r_1)$ and this is in fact what the F & O algorithm does.

4 Implementation

In this section we discuss the implementation of the R algorithm for the membership rules and compare it by means of various benchmarks with the CHR implementation in the ECL^{i}PS^e system.

4.1 Modelling of the Membership Rules in CHR

Following (Apt and Monfroy 2001) the membership rules are represented as CHR propagation rules with one head. Recall that the latter ones are of the form

$$H ==> G_1, \ldots, G_l \mid B_1, \ldots, B_m.$$

where

- $l \geqslant 0, m > 0$,
- the atom H of the *head* refers to the defined constraints,
- the atoms of the *guard* $G_1, \ldots, G_l$ refer to Prolog relations or built-in constraints,

14

- the atoms of the *body* $B_1, \ldots, B_m$ are arbitrary atoms.

Further, recall that the CHR propagation rules with one head are executed as follows. First, given a query (that represents a CSP) the variables of the rule are renamed to avoid variable clashes. Then an attempt is made to match the head of the rule against the first atom of the query. If it is successful and all guards of the instantiated version of the rule succeed, the instantiated version of the body of the rule is executed. Otherwise the next rule is tried.

Finally, let us recall the representation of a membership rule as CHR a propagation rule used in (Apt and Monfroy 2001). Consider the membership rule

$$y_1 \in S_1, \ldots, y_k \in S_k \;\rightarrow\; z_1 \neq a_1, \ldots, z_m \neq a_m.$$

related to the constraint c on the variables $X_1, \ldots, X_n$. We represent its condition by starting initially with the atom $c(X_1, \ldots, X_n)$ as the head. Each atomic condition of the form $y_i \in \{a\}$ is processed by replacing in the atom $c(X_1, \ldots, X_n)$ the variable y_i by a. In turn, each atomic condition of the form $y_i \in S_i$, where S_i is not a singleton, is processed by adding the atom $in(y_i, LS_i)$ to the guard of the propagation rule. The in/2 predicate is defined by

```
in(X,L) :- dom(X,D), subset(D,L).
```

So in(X,L) holds if the current domain of the variable X (yielded by the built-in dom of ECL^{i}PS^e) is included in the list L. In turn, LS_i is a list representation of the set S_i.

Finally, each atomic conclusion $z_i \neq a_i$ translates to the atom z_i ## a_i of the body of the propagation rule.

As an example consider the membership rule

$$X \in \{0\}, Y \in \{1, 2\} \;\rightarrow\; Z \neq 2$$

in presence of a constraint c on the variables X, Y, Z. It is represented by the following CHR propagation rule:

```
c(0,Y,Z) ==> in(Y,[1,2]) | Z##2.
```

In ECL^{i}PS^e the variables with singleton domains are automatically instantiated. So, assuming that the variable domains are non-empty, the condition of this membership rule holds iff the head of the renamed version of the above propagation rule matches the atom c(0,Y,Z) *and* the current domain of the variable Y is included in [1,2]. Further, in both cases the execution of the body leads to the removal of the value 2 from the domain of Z. So the execution of both rules has the same effect when the variable domains are non-empty.

Execution of CHR. In general, the application of a membership rule as defined in Section 3 and the execution of its representation as a CHR propagation rules coincide. Moreover, by the semantics of CHR, the CHR rules are repeatedly applied until a fixpoint is reached. So a repeated application of a finite set of membership rules coincides with the execution of the CHR program formed by the representations

15

of these membership rules as propagation rules. An important point concerning the standard execution of a CHR program is that, in contrast to the R algorithm, every change in the variable domains of a constraint causes the computation to restart.

4.2 Benchmarks

In our approach the repeated application of a finite set of membership rules is realized by means of the R algorithm of Section 2 implemented in ECL^{i}PS^e. The compiler consists of about 1500 lines of code. It accepts as input a set of membership rules, each represented as a CHR propagation rule, and constructs an ECL^{i}PS^e program that is the instantiation of the R algorithm for this set of rules. As in CHR, for each constraint the set of rules that refer to it is scheduled separately.

In the benchmarks below for each considered CSP we used the sets of all minimal valid membership and equality rules for the "base" constraints which were automatically generated using a program discussed in (Apt and Monfroy 2001). In the first phase the compiler constructs for each rule g the lists $friends(g)$ and $obviated(g)$. Time spent on this construction is comparable with the time needed for the generation of the minimal valid equality and membership rules for a given constraint. For example, the medium-sized membership rule set for the rcc8 constraint, containing 912 rules, was generated in 166 seconds while the construction of all $friends$ and $obviated$ lists took 142 seconds.

To see the impact of the accumulated savings obtained by permanent removal of the rules during the iteration process, we chose benchmarks that embody several successive propagation steps, i.e., propagation interleaved with domain splitting or labelling.

In Table 1 we list the results for selected single constraints. For each such constraint, say C on a sequence of variables $x_1, \ldots, x_n$ with respective domains $D_1, \ldots, D_n$, we consider the CSP $\langle C \; ; \; x_1 \in D_1, \ldots, x_n \in D_n \rangle$ together with randomized labelling. That is, the choices of variable, value, and action (assigning or removing the value), are all random. The computation of simply one or all solutions yields insignificant times, so the benchmark program computes and records also all intermediate non-solution fixpoints. Backtracking occurs if a recorded fixpoint is encountered again. In essence, this benchmark computes implicitly all possible search trees. As this takes too much time for some constraints, we also impose a limit on the number of recorded fixpoints.

In turn, in Table 2 we list the results for selected CSPs. We chose here CSPs that formalize sequential automatic test pattern generation for digital circuits (ATPG), see (Brand 2001). These are rather large CSPs that employ the and constraints of Table 1 and a number of other constraints, most of which are implemented by rules.

We measured the execution times for three rule schedulers: the standard CHR representation of the rules, the generic chaotic iteration algorithm GI, and its improved derivative R. The codes of the latter two algorithms are both produced by our compiler and are structurally equal, hence allow a direct assessment of the improvements embodied in R.

An important point in the implementations is the question of when to remove

16

Constraint	rcc8	fork	and3	and9	and11
MEMBERSHIP					
relative	37% / 22%	58% / 46%	66% / 49%	26% / 15%	57% / 25%
absolute	147/396/686	0.36/0.62/0.78	0.27/0.41/0.55	449/1727/2940	1874/3321/7615
EQUALITY					
relative	97% / 100%	98% / 94%	92% / 59%	95% / 100%	96% / 101%
absolute	359/368/359	21.6/21.9/22.9	0.36/0.39/0.61	386/407/385	342/355/338

Table 1. Randomized search trees for single constraints

Logic	3-valued	9-valued	11-valued
MEMBERSHIP			
relative	61% / 44%	65% / 29%	73% / 29%
absolute	1.37/2.23/3.09	111/172/385	713/982/2495
EQUALITY			
relative	63% / 29%	40% / 57%	36% / 51%
absolute	0.77/1.22/2.70	2.56/6.39/4.50	13.8/38.7/26.7

Table 2. CSPs formalizing sequential ATPG

solved constraints from the constraint store. The standard CHR representation of membership rules as generated by the algorithm of (Apt and Monfroy 2001) does so by containing, beside the propagation rules, one CHR simplification rule for each tuple in the constraint definition. Once its variables are assigned values that correspond to a tuple, the constraint is solved, and removed from the store by the corresponding simplification rule. This 'solved' test takes place interleaved with propagation. The implementations of GI and R, on the other hand, check after closure under the propagation rules. The constraint is considered solved if all its variables are fixed, or, in the case of R, if the set F of remaining rules is empty (see the following subsection). Interestingly, comparing CHR and GI, the extra simplification rules sometimes constitute a substantial overhead while at other times their presence allows earlier termination.

We mention briefly that our specific implementation deviates slightly from the description of R inside the else branch. The test $\forall e \sqsupseteq d \; \neg holds(b, e)$ in the case of a membership condition $y \in S$ corresponds to testing whether the intersection

$D_y \cap S$ is empty. Performing this always turned out to be more costly than doing so only when D_y is a singleton set.

The platform for all benchmarks was a Sun Enterprise 450 with 4 UltraSPARC-II 400 MHz processors and 2 GB memory under Solaris, and ECL^{i}PS^e 5.5 (non-parallel). In the tables we provide for each constraint or CSP the ratio of the execution times in seconds between, first, R and GI, and second, R and CHR. This is followed by the absolute times in the order R / GI / CHR.

Recently, we have been experimenting with various ways of optimizing our implementation of the R algorithm. In particular, we considered a better embedding into the constraint-handling mechanism of ECL^{i}PS^e, for instance by finer control of the waking conditions and a joint removal of the elements from the same domain. At this stage we succeeded in achieving an average speed-up by a factor of 4. This work is in progress but already shows that further improvements are possible.

4.3 Recomputing of the Least Fixpoints

Finally, let us illustrate the impact of the permanent removal of the rules during the least fixpoint computation, achieved here by the use of the lists $friends(g)$ and $obviated(g)$. Given a set F of rules call a rule $g \in F$ *solving* if $friends(g) \cup obviated(g) = F$.

Take now as an example the equivalence relation $\equiv$ from three valued logic of (Kleene 1952, page 334) that consists of three values, t (true), f (false) and u (unknown). It is defined by the truth table

$\equiv$	t	f	u
t	t	f	u
f	f	t	u
u	u	u	u

The program of (Apt and Monfroy 2001) generates for it 26 minimal valid membership rules. Out of them 12 are solving rules. For the remaining rules the sizes of the set *friends* $\cup$ *obviated* are: 17 (for 8 rules), 14 (for 4 rules), and 6 (for 2 rules).

In the R algorithm a selection of a solving rule leads directly to the termination $(G = \varnothing)$ and to a reduction of the set F to $\varnothing$. For other rules, also a considerable simplification in the computation takes place. For example, one of the 8 rules with 17 rules in the set *friends* $\cup$ *obviated* is

$$r := x \in \{f\}, z \in \{f, u\} \;\rightarrow\; y \neq f.$$

Consider now the CSP $\langle \equiv \; ; \; x \in \{f\}, y \in \{f, t, u\}, z \in \{f, u\} \rangle$. In the R algorithm the selection of r is followed by the application of the rules from *friends* and the removal of the rules from *friends* $\cup$ *obviated*. This brings the number of the considered rules down to $26 - 17 = 9$. The R algorithm subsequently discovers that none of these nine rules is applicable at this point, so this set F remains upon termination. Then in a

18

subsequent constraint propagation phase, launched after splitting or after constraint propagation involving another constraint, the fixpoint computation by means of the R algorithm involves only these nine rules instead of the initial set of 26 rules. For solving rules this fixpoint computation immediately terminates.

Interestingly, as Table 3 shows, the solving rules occur quite frequently. We list there for each constraint and each type of rules the number of solving rules divided by the total number of rules, followed in a new line by the average number of rules in the set $friends(g) \cup obviated(g)$.

Constraints	and2	and3	and9	and11	fork	rcc8	allen
equality	6/6	13/16	113/134	129/153	9/12	183/183	498/498
	6	14	130	148	11	183	498
membership	6/6	4/13	72/1294	196/4656	0/24	0/912	n.a./26446
	6	7	810	3156	9	556	n.a.

Table 3. Solving rules

The fork constraint is taken from the Waltz language for the analysis of polyhedral scenes. The rcc8 is the composition table for the Region Connection Calculus with 8 relations from (Egenhofer 1991). It is remarkable that all its 183 minimal valid equality rules are solving. While none of its 912 minimal valid membership rule for rcc8 is solving, on the average the set $friends(g) \cup obviated(g)$ contains 556 membership rules. Also all 498 minimal valid equality rules for the allen constraint, that represents the composition table for Allen's qualitative temporal reasoning, are solving. The number of minimal valid membership rules exceeds 26,000 and consequently they are too costly to analyze.

Simplification rules. The CHR language supports besides propagation rules also so-called simplification rules. Using them one can remove constraints from the constraint store, so one can affect its form. In (Abdennadher and Rigotti 2001) a method is discussed that allows one to automatically transform CHR propagation rules into simplification rules that respects their semantics. It is based on identifying or constructing propagation rules that are solving.

In contrast, our method captures the *degree* to which a rule is solving, by the ratio of the sizes of $U(r) = friends(r) \cup obviated(r)$ and the full rule set. If the sets are equal, then the ratio is 1 and r is a solving rule. Consider now two non-solving rules r_1, r_2, that means with $U(r_1) \subset \mathcal{R}$ and $U(r_2) \subset \mathcal{R}$, but let also $U(r_1) \cup U(r_2) = \mathcal{R}$. Suppose that during a fixpoint computation the conditions of both rules have succeeded, and their bodies have been applied. The R algorithm would now immediately detect that the constraint is solved, and consequently terminate. CHR, for which r_1 and r_2 are ordinary (propagation) rules, cannot detect this possibility for immediate termination.

5 Redundancy of *prop* Rules

The cost of a fixpoint computation by the GI algorithm or one of its derivatives depends on the number of functions involved, in particular in absence of a good strategy for selecting the functions, represented in the algorithms by the "choose" predicate. It is therefore important to identify functions or rules that are not needed for computing fixpoints. In the following we shall examine the issue of rule redundancy. We shall again start with arbitrary functions before moving on to (*prop*) rules. The redundancy concept we employ is based on fixpoints. In the following, for brevity, we drop the word "common" when referring to common fixpoints of a set of functions.

Definition 5.1

- Consider a set $F \cup \{f\}$ of functions on a partial ordering. A function f is called *redundant with respect to* F if the sets of fixpoints of F and $F \cup \{f\}$ are equal.
- A set of functions F is called *minimal with respect to redundancy* (or simply *minimal*), if no function $f \in F$ is redundant with respect to $F - \{f\}$. $\quad\square$

Equivalently, we can say that a function f is redundant w.r.t. F if every fixpoint of F is also a fixpoint of f.

5.1 Redundant Rules

We now focus on the subject of redundancy for *prop* rules. The following simple test is then useful.

Theorem 5.2

Consider a set F of *prop* rules and a *prop* rule $r := b \to g$ with the witness w for b. Let e be the least fixpoint of F greater than or equal to w. If $g(e) = e$, then the rule r is redundant with respect to F.

Proof. We show that $g(e) = e$ implies that an arbitrary fixpoint d of F is a fixpoint of r by a case condition.

b **holds for** d: We have $w \sqsubseteq d$ since w is the witness for b. Also, $w \sqsubseteq e \sqsubseteq d$ since e is the least fixpoint of F greater than or equal to w. From $e \sqsubseteq d$, $g(e) = e$, and the stability of g we conclude $g(d) = d$. Hence $r(d) = (b \to g)(d) = g(d) = d$.

b **does not hold for** d: Then $r(d) = (b \to g)(d) = d$. $\quad\square$

This test is of interest to us since it allows us to compute only one fixpoint of F instead of all fixpoints. It is effective if

- the witness can be computed,
- the equality $g(e) = e$ can be determined, and
- the fixpoint computations are effective.

For the sake of fixpoint computations a rule $r = b \to g$ with a body $g = g_1, \ldots, g_n$ (describing a function composition) such that any two different g_i, g_j commute can

be identified with the collection $b \rightarrow g_1, \ldots, b \rightarrow g_n$ of the rules, and vice versa. Indeed, the respective fixpoints and the rule properties are the same. We consider here these two representations as equivalent. If a rule with such a "compound" body is not redundant it might be so in part. That is, some part of its body might be redundant or, in other words, some sub-rules of its decomposition might be. This is what we mean below by saying that a rule is *partially* redundant.

Let us consider now the task of computing minimal sets of *prop* rules. Such sets can of course be generated by a simple bounded loop: select an untested rule, test whether it is redundant and, if so, remove it from the current set. In general, however, the obtained minimal sets depend on the selection order for testing; see an example below. In our experiments we used a strategy that selects first the rules the execution of which is computationally expensive, for instance due to conditions on many variables. In this way we hope to obtain a set of computationally cheap rules.

5.2 An Example: Redundant Membership Rules

Let us illustrate now a number of issues by means of an example. Consider the constraint $c(x, y, z, u)$ defined by

x	y	z	u
0	1	0	1
1	0	0	1
1	1	1	0

The underlying domain for all its variables is $\{0, 1\}$. Hence the induced corresponding partial order is

$$(\{(A, B, C, D) \mid A, B, C, D \subseteq \{0, 1\}\}, \supseteq).$$

The algorithm of (Apt and Monfroy 2001) generates eleven membership rules listed in Figure 1. Since the rule conditions are only equality tests, we use an alternative notation that should be self-explanatory.

For example, rule (11) states that if $c(x, y, z, u)$, then it is correct to conclude from $x = 1$ and $u = 1$ that $y \neq 1$ (validity), and furthermore that neither $x = 1$ nor $u = 1$ suffices for this conclusion (minimality).

Suppose we are interested in computing the smallest fixpoint greater than or equal to $E_1 = \{1\} \times \{0, 1\} \times \{0, 1\} \times \{1\}$. Suppose rule (11) is considered. Its application yields $E_2 = \{1\} \times \{0\} \times \{0, 1\} \times \{1\}$ from where rule (4) leads to $E_3 = \{1\} \times \{0\} \times \{0\} \times \{1\}$. This is indeed a fixpoint since for each rule either its condition does not apply or the application of its body results again in E_3.

A second possible iteration from E_1 that stabilises in E_3 is by rule (5) followed by rule (10). Rule (11) can be applied at this point but its body does not change E_3.

$$c(x, y, z, 0) \quad \rightarrow \quad x \neq 0, y \neq 0, z \neq 0 \qquad (1)$$
$$c(x, y, 1, u) \quad \rightarrow \quad u \neq 1, x \neq 0, y \neq 0 \qquad (2)$$
$$c(0, y, z, u) \quad \rightarrow \quad u \neq 0, y \neq 0, \underline{z \neq 1} \qquad (3)$$
$$c(x, 0, z, u) \quad \rightarrow \quad u \neq 0, x \neq 0, \underline{z \neq 1} \qquad (4)$$
$$c(x, y, z, 1) \quad \rightarrow \quad z \neq 1 \qquad (5)$$
$$c(x, y, 0, u) \quad \rightarrow \quad u \neq 0 \qquad (6)$$
$$c(1, 1, z, u) \quad \rightarrow \quad u \neq 1, \underline{z \neq 0} \qquad (7)$$
$$c(x, 1, 0, u) \quad \rightarrow \quad x \neq 1 \qquad (8)$$
$$c(x, 1, z, 1) \quad \rightarrow \quad \underline{x \neq 1} \qquad (9)$$
$$c(1, y, 0, u) \quad \rightarrow \quad y \neq 1 \qquad (10)$$
$$c(1, y, z, 1) \quad \rightarrow \quad \underline{y \neq 1} \qquad (11)$$

Fig. 1. Membership rules for the constraint c

Indeed, E_3 is a fixpoint of all rules including rule (11). We conclude that rule (11) is redundant – we just performed the test of Theorem 5.2.

The process of identifying redundant rules can then be continued for the rule set $\{(1), \ldots, (10)\}$. One possible outcome is depicted in Figure 1, where redundant parts of rules are underlined. From the 20 initial atomic conclusions 13 remain, thus we find here a redundancy ratio of 35%.

Consider now the justification for the redundancy of rule (11), and observe that rule (11) has no effect since rule (10), which has the same body, was applied before. Suppose now that the process of redundancy identification is started with rule (10) instead of rule (11). This results in identifying rule (10) as redundant, with a relevant application of rule (11).

Note moreover that one of the rules (10), (11) must be present in *any* minimal set since their common body $y \neq 1$ occurs in no other rule. It would seem difficult to find a criterion that prefers one rule over the other as their structure is the same.

5.3 Experiments

We implemented in $\mathrm{ECL}^i\mathrm{PS}^e$ an algorithm that computes minimal sets of membership rules. The results for some benchmark rule sets are listed in Table 4.

For each constraint the set of minimal membership or equality rules (indicated respectively by the subscript "$_M$" or "$_E$") was computed by the rule generation algorithm of (Apt and Monfroy 2001). The constraints are taken from the experiments discussed in Table 1. Additionally a 5-ary constraint fula (standing for the well-known fulladder constraint) is analyzed.

The table shows the size of the rule set, the number of fully and, in parentheses, partially redundant rules. The redundancy ratio for the entire rule set shows the percentage of the atomic disequalities that are removed from the rule conclusions on the account of redundancy.

22

	and11_M	and11_E	and3_M	equ3_M	fula2_E	fork_E	fork_M
total	4656	153	18	26	52	12	24
redundant (partially)	4263 (2)	0 (6)	5 (0)	8 (0)	24 (0)	0 (9)	6 (6)
redundancy ratio	81%	4%	30%	26%	35%	35%	40%

Table 4. Minimizing rule sets

Computation times are negligible in so far as they are considerably smaller than the corresponding rule generation times.

5.4 Schedulers and Minimal Rule Sets

There is no simple connection between redundancy and the rule sets *friends* and *obviated* of the R scheduler. For instance, it is *not* the case that a rule is redundant if it is contained in *friends*(r) $\cup$ *obviated*(r) of every rule r. Nor is a redundant rule necessarily contained in *friends*(r) $\cup$ *obviated*(r) of every rule r. To examine this in an example, recall the rules in Figure 1. All except (5) and (6) are solving rules, i.e., each respective set *friends* $\cup$ *obviated* is the complete set $\{(1), \ldots, (11)\}$ of rules, while for rules (5) and (6) this set is $\{(1), (3), (5), (6)\}$. Further, neither (5) nor (6) is redundant with respect to all other rules, whereas (10) and (11) are.

Benchmarks. We reran the benchmarks from Tables 1 and 2 with all involved rule sets subjected to a removal of redundant rules and subsequent recomputation of the sets *friends* and *obviated*. The results are reported in Tables 5 and 6 below. The rule sets of rcc8 were already minimal; therefore this constraint is omitted.

Constraint	fork	and3	and9	and11
MEMBERSHIP				
relative	60% / 46%	69% / 48%	28% / 18%	50% / 29%
absolute	0.32/0.53/0.70	0.27/0.39/0.56	167/589/924	157/316/543
EQUALITY				
relative	97% / 93%	97% / 64%	96% / 101%	96% / 101%
absolute	21.6/22.2/23.2	0.37/0.38/0.58	386/404/384	341/353/339

Table 5. Randomized search trees for single constraints (without redundant rules)

Logic	3-valued	9-valued	11-valued
MEMBERSHIP			
relative	66% / 46%	62% / 33%	68% / 35%
absolute	1.32/2.00/3.05	37/59/114	70/103/199
EQUALITY			
relative	61% / 26%	40% / 58%	33% / 48%
absolute	0.72/1.18/2.73	2.57/6.41/4.46	13.8/41.0/28.6

Table 6. CSPs formalizing sequential ATPG (without redundant rules)

When comparing the redundancy and non-redundancy benchmarks versions we observe that the absolute execution times are enormously reduced in the case of the constraints on higher-valued logics. This is in line with the much smaller sizes of the reduced rule sets. The ratios of the execution times, however, are barely affected. The type of a scheduler and minimality w.r.t. redundancy appear to be rather orthogonal issues.

It is interesting to examine in one case the distribution of the solving degrees, i.e., the ratios of the sizes of *friends* $\cup$ *obviated* and the full rule set. Recall that a ratio of 1 means that the constraint is solved when the rule body has been executed. Such a rule could be represented as a simplification rule in CHR (see Section 4.3).

In Figure 2 two membership rule sets for the constraint and9 are compared. One set contains redundant rules, the other set is minimal w.r.t. redundancy. The rules in the minimal set are solving to a lesser degree. In particular, none is a proper solving rule. The good performance of the R algorithm in the benchmarks of Tables 5,6 may thus be attributed not to distinguishing solving (simplification) rules and non-solving propagation rules, but to the *accumulated* effect of removing rules from the fixpoint computation.
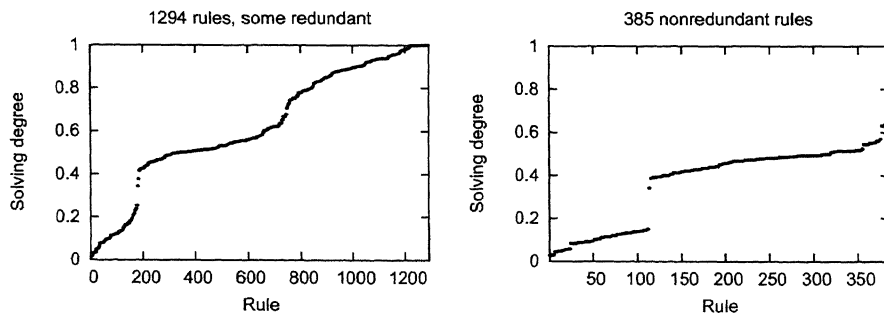

Fig. 2. and9$_M$: Solving degree and redundancy

Acknowledgments

References

ABDENNADHER, S., KRÄMER, E., SAFT, M., AND SCHMAUS, M. 2001. JACK: A Java Constraint Kit. In *International Workshop on Functional and (Constraint) Logic Programming (WFLP 2001), Technical Report No. 2017.* University of Kiel, Kiel, Germany.

ABDENNADHER, S. AND RIGOTTI, C. 2001. Using confluence to generate rule-based constraint solvers. In *Proceedings of the 3rd Int. Conf. on Principles and Practice of Declarative Programming (PPDP 2001).* ACM, Firenze, Italy, 127–135.

APT, K. R. 1999. The essence of constraint propagation. *Theoretical Computer Science 221,* 1–2, 179–210. Available via http://arXiv.org/archive/cs/.

APT, K. R. 2000. The role of commutativity in constraint propagation algorithms. *ACM Transactions on Programming Languages and Systems 22,* 6, 1002–1036. Available via http://arXiv.org/archive/cs/.

APT, K. R. AND MONFROY, E. 2001. Constraint programming viewed as rule-based programming. *Theory and Practice of Logic Programming 1,* 6, 713–750. Available via http://arXiv.org/archive/cs/.

BRAND, S. 2001. Sequential automatic test pattern generation by constraint programming. In *CP2001 Post Conference Workshop Modelling and Problem Formulation.* Available via http://homepages.cwi.nl/~sbrand/.

EGENHOFER, M. 1991. Reasoning about binary topological relations. In *Proceedings of the 2nd International Symposium on Large Spatial Databases (SSD),* O. Günther and H.-J. Schek, Eds. Lecture Notes in Computer Science, vol. 525. Springer-Verlag, 143–160.

FRÜHWIRTH, T. 1998. Theory and practice of Constraint Handling Rules. *Journal of Logic Programming 37,* 1–3 (October), 95–138. Special Issue on Constraint Logic Programming (P. J. Stuckey and K. Marriot, Eds.).

HOLZBAUR, C., DE LA BANDA, M. G., JEFFERY, D., AND STUCKEY, P. J. 2001. Optimizing compilation of constraint handling rules. In *Proceedings of the 2001 International Conference on Logic Programming.* Lecture Notes in Computer Science, vol. 2237. Springer-Verlag, 74–89.

KLEENE, S. C. 1952. *Introduction to Metamathematics.* van Nostrand, New York.