Schedulers for Rule-Based Constraint Programming

Krzysztof R. Apt[*]
CWI, P.O. Box 94079
1090 GB Amsterdam, the Netherlands
and University of Amsterdam, the Netherlands
K.R.Apt@cwi.nl

Sebastian Brand
CWI, P.O. Box 94079
1090 GB Amsterdam, the Netherlands
S.Brand@cwi.nl

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 [4]. We apply this study to so-called membership rules of Apt and Monfroy [5]. This leads to an implementation that yields for these rules a considerably better performance than their execution as standard CHR rules.

Keywords

Constraint propagation, rule-based programming

1. INTRODUCTION

In this paper we are concerned with schedulers for a class of rules that naturally arise in the context of constraint programming represented by means of rule-based programming. An example of such rules are so-called *membership rules*, introduced in Apt and Monfroy [5]. Their relevance stems from the following observations there made for constraint satisfaction problems (CSP's) with finite domains:

- constraint propagation can be naturally achieved by repeated application of the membership rules;

- in particular the notion of 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 most recent reference on the subject of such an automatic rule generation see Abdennadher and Rigotti [2].);

- many rules of the CHR language (Constraint Handling Rules) of Frühwirth [7] that are used in specific constraint solvers are in fact membership rules. Now,

---

[*]Currently on leave at School of Computing, National University of Singapore

in the logic programming approach to constraint programming CHR is the language of choice to write constraint solvers.

In the resulting approach to constraint programming the computation process is limited to a repeated application of the 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.

The abstract framework is based on an appropriate modification of the generic approach to constraint propagation algorithms introduced in Apt [3] and Apt [4]. 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 [4], we take into account information about the scheduled functions. Here we consider functions in the form of the rules $b \to g$, where b and g satisfy a number of natural conditions. We call such functions *good rules*. The relevant observation is that membership rules are good rules. Then we propose a specific scheduler in the form of an algorithm R, appropriate for good rules.

The implementation is provided as an ECL^{i}PS^e program that accepts as input a set of membership rules and constructs an ECL^{i}PS^e program that is the instantiation of the R algorithm for this set of rules. As 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. We found by means of various benchmarks that our implementation is considerably faster than CHR.

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 source language. There is also a recent implementation in Java, see [1]. A great deal of effort was spent on implementing CHR efficiently. For an account of the most recent implementation see Holzbaur et al. [8]. Since, as already mentioned above, many CHR rules are membership rules, our approach provides a better implementation of a subset of CHR. While being stricly smaller

than full CHR, the actual class of relevant rules is wider than the class of membership rules. The essential properties, such as monotonicity of condition and conclusion, are enjoyed by many rules that describe constraint propagation. This, hopefully, may lead to new insights into design and implementation of languages appropriate for writing constraint solvers.

It is important to stress that the discussed 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.

2. REVISIONS OF THE GENERIC ITERATION ALGORITHM

2.1 The Original Algorithm

Let us begin our presentation with recalling the generic algorithm of Apt [4]. 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 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. $\quad\square$

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

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

Figure 1: Generic Iteration Algorithm (GI)

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 $update(G, g, d) = \emptyset$,

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 1 (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. (Sketch). The following statement is an invariant of the **while** loop of the algorithm:

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

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

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

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 $<_{lex}$. $\quad\square$

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 realize this idea we proceed as follows. First, we introduce the following property that will be satisfied by the considered functions.

Definition 2. 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. $\quad\square$

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 $f(d) = d$.

Next, 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) that satisfy the following condition

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

where *friends*$(g,d) = [g_1, \ldots, g_k]$.

That is, for all d, each function f in *friends*$(g,d) \cup$ *obviated*(g,d) is stable above $g \circ g_1 \circ \ldots \circ g_k(d)$.

Now, we 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) both from F and G. This modified algorithm is shown in Fig. 2. To keep the notation uniform we identified at some places the lists *friends*(g,d) and *obviated*(g,d) with the sets.

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

Figure 2: Revised Generic Iteration Algorithm (RGI)

The following result then shows correctness of this algorithm.

THEOREM 2. *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 $\top$. 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 1 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.,

$$G = \emptyset \ \vee \ d = \top$$

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

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

In the next section we shall focus on functions that are in a special form. For these functions we shall show how to construct specific lists *friends*(g,d) and *obviated*(g,d).

2.3 Functions in the Form of Rules

In what follows we consider the situation when the scheduled functions are of a specific form $b \rightarrow g$, 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 evaluates in d to either true or false, denoted $\textit{Holds}(b,d)$ and $\neg\textit{Holds}(b,d)$, resp.

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

$$(b \rightarrow g)(d) := \begin{cases} g(d) & \text{if } \textit{Holds}(b,d) \\ d & \text{if } \neg\textit{Holds}(b,d) \ . \end{cases}$$

The rules introduced in the next section will be of a specific type.

Definition 3. Consider a partial ordering $(D, \sqsubseteq)$.

- We say that a condition b is *monotonic* if $\textit{Holds}(b,d)$ and $d \sqsubseteq e$ implies $\textit{Holds}(b,e)$, for all d, e.

- We say that a condition b is *precise* if the least d exists such that $\textit{Holds}(b,d)$. We call then d the *witness* for b.

- We call a rule $b \rightarrow g$ *good* if b is monotonic and precise and g is stable. $\square$

When all rules are good, we can modify the RGI algorithm by taking 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 construct the lists *friends*(g,d) and *obviated*(g,d) before the execution of the algorithm, without using the parameter d. Moreover, the list *friends*(g) can be constructed in such a way that the conditions of its rules do not need to evaluated at the moment they are applied, as they will all hold. The details of a specific construction that we shall use here will be given in a moment, once we identify the condition that is crucial for the correctness. This revision of the RGI algorithm is given in Fig. 3.

Again, we are interested in identifying conditions under which the Correctness Theorem 1 holds with the GI algorithm replaced by the R algorithm. To this end, given a rule $b \rightarrow g$ in F and $d \in D$, define as follows:

$$\textit{friends}(b \rightarrow g, d) := \begin{cases} \textit{friends}(b \rightarrow g) & \text{if } \textit{Holds}(b,d) \\ [\,] & \text{if } \neg\textit{Holds}(b,d) \end{cases}$$

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

We now have the following counterpart of the Correctness Theorem 1.

THEOREM 3 (CORRECTNESS). *Suppose that all functions in F are good rules of the form $b \rightarrow g$, where g is inflationary and monotonic, and that $(D, \sqsubseteq)$ is finite and has*

```
d := ⊥;
F₀ := F;
G := F;
while G ≠ ∅ and d ≠ ⊤ do
    choose f ∈ G; suppose f is b → g;
    G := G - {b → g};
    if Holds(b, d) then
        F := F - (friends(b → g) ∪ obviated(b → g));
        G := G - (friends(b → g) ∪ obviated(b → g));
        G := G ∪ update(G, h, d),
            where h = g ∘ g₁ ∘ ... ∘ gₖ
            and friends(b → g) = [b₁ → g₁, ..., bₖ → gₖ];
        d := h(d)
    else
        if ∀e ⊒ d ¬Holds(b, e) then
            F := F - {b → g}
        end
    end
end
```

Figure 3: Rules Algorithm (R)

the least element ⊥ and the greatest element ⊤. 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:

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

Then the Correctness Theorem 1 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 (4) and the fact that the rule bodies are inflationary functions, $Holds(b, d)$ implies that

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

where friends$(b \to g) = [b_1 \to g_1, ..., b_k \to g_k]$. This takes care of the situation when if $Holds(b, d)$.

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 if $\neg Holds(b, d)$. When the condition b fails for all $e \sqsupseteq d$, then 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. □

We now provide an explicit construction of the lists friends and obviated for a rule $b \to g$ in the form of the algorithm in Fig. 4. GI(d) stands here for the GI algorithm activated with ⊥ replaced by d and the considered set of rules as the set of functions F. Further, given an execution of 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.

Note that $b \to g \notin$ friends$(b \to g)$ since $b \to g$ is a good rule, while $b \to g \in$ obviated$(b \to g)$ since by the stability of g $g(e) = e$ holds.

```
e := witness of b;
e := GI(g(e));
friends(b → g) := list of the relevant rules h ∈ F
                in the execution of GI(g(e));
obviated(b → g) := [ ];
for each (b' → g') ∈ F - friends(b → g) do
    if g'(e) = e or ∀e' ⊒ e ¬Holds(b', e') then
        obviated(b → g) := [b' → g'|obviated(b → g)]
    end
end
```

Figure 4: Friends and Obviated Algorithm (F & O)

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

LEMMA 1. *Upon termination of the F & O algorithm conditions (1) and (4) hold, where the lists friends$(b \to g, d)$ and obviated$(b \to g, d)$ are defined as before Theorem 3.* □

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 3. 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 CSP's, 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. Then

these functions are 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 4. *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 $f(e_1) = e_1$. $\square$

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 $n \geq 0$, 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 CSP's 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 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 componentwise reversed subset ordering. Note that in this ordering $(D_1, \ldots, D_n)$ is the least element while

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

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 $R_\emptyset$ according to which

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

It is straightforward to see that $R_\emptyset$ 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 $(D_{\mathcal{P}}, \sqsubseteq)$.

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 now recall the rules introduced in Apt and Monfroy [5].[1] They are called *membership rules* and are of the form

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

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.

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:

---

[1]In our presentation we slightly relax the original syntactic restrictions.

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 reformulate this interpretation so that it fits the framework considered in the previous section. To this end we need to clarify how to evaluate a condition, and how to interpret a conclusion. We start with the first item.

Definition 4. Given a variable y with the domain D_y and $E, S \supseteq D_y$ 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

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

$$Holds(y \in S, \top).$$

Then we interpret a sequence $y_1 \in S_1, \ldots, y_k \in S_k$ of conditions as a conjunction, so by putting

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

and

$$Holds(y_1 \in S_1, \ldots, y_k \in S_k, \top). \qquad \Box$$

Concerning the second item we proceed as follows.

Definition 5. 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 use "padding", that is we interpret it as the identity on the other components. If the resulting sequence 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. $\qquad \Box$

In view of the Correctness Theorem 3 the following observation allows us to apply the R algorithm when each function

is a membership rule and when for each rule $b \to g$ the lists $friends(b \to g)$ and $obviated(b \to g)$ are constructed by the F & O algorithm.

Note 1. Consider the partial ordering $(D_\mathcal{P}, \sqsubseteq)$.

(i) Each membership rule is good.

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

- inflationary,
- monotonic. $\qquad \Box$

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:

$$update(G, b \to g, d) := \begin{cases} F - G & \text{if } Holds(b, d) \wedge g(d) \neq d \\ \emptyset & \text{otherwise.} \end{cases}$$

To illustrate the intuition behind the use of the lists $friends(b \to g)$ and $obviated(b \to g)$ take the CSP $\mathcal{P} :=$

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

and consider the membership rules

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

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 in fact is 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 [5] 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

- $l \geq 0$, $m > 0$,

- 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,

- 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 the guard of the instantiated version of the rule succeeds, 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 a CHR propagation rule used in Apt and Monfroy [5]. 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 it as a CHR rule with the single head atom $c(X_1, \ldots, X_n)$ and guard atoms $\text{in}(y_i, S_i)$ where the in/2 predicate is defined by in(X,L) :- dom(X,D), subset(D,L). The body consists of atomic calls $z_i\ \#\#\ a_i$.

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 of these membership rules as propagation rules.

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 f membership rules, each represented as a CHR propagation ule, 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 ach constraint the set of rules that refer to it is scheduled eparately.

For each considered constraint we use rules generated by a program discussed in [5]. Our compiler constructs then for each rule g the lists $friends(g)$ and $obviated(g)$ by executing the F & O algorithm (essentially computing a fixpoint for each rule). Time spent on this construction is comparable with rule generation time.

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 a variable, value, and an assignment or a removal of the value, are random. The computation of only the solutions yields times that are insignificant, so the test program computes also all intermediate fixpoints, where some domains are not singleton sets. Branching at these recorded points takes place only once, that is, backtracking occurs immediately if a recorded point is encountered again. In Table 2 we report the results for CSP's that formalize sequential automatic test pattern generation for digital circuits (ATPG).

These are rather large CSP's that employ the and constraints of Table 1 and a number of other constraints. They are taken from a recent study by the first author that will be reported elsewhere.

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 solved constraints from the constraint store. The standard CHR representation of membership rules 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 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.

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 for R and GI / CHR.

Const.	rcc8	fork	and3	and9	and11
MEM					
rel.	26%/11%	43%/40%	58%/47%	13%/6%	13%/3%
abs.	109	0.23	0.22	70	55.6
	419/950	0.54/0.58	0.38/0.47	534/1096	427/2077
EQU					
rel.	95%/100%	95%/89%	82%/74%	94%/97%	89%/94%
abs.	323	18.9	0.31	286	299
	341/324	19.9/21.2	0.38/0.42	303/294	335/318

Table 1: Randomized search trees for constraints

Logic	3-valued	9-valued	11-valued 7
MEM			
relative	64% / 35%	71% / 24%	85% / 86%
absolute	1.39 2.16/4.01	124 175/509	797 933/3120
EQU			
relative	63% / 70%	44% / 59%	39% / 48%
absolute	0.72 1.15/2.58	2.40 5.50/4.09	12.3 31.6/25.7

Table 2: CSP's formalizing sequential ATPG

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 [9][page 334] that consists of

20

three values, t (true), f (false) and u (unknown). It is defined by the truth table

≡	t	f	u
t	t	f	u
f	f	t	u
u	u	u	u

The program of Apt and Monfroy [5] 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* ∪ *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 $(G = \emptyset)$ and to a reduction of the set F to $\emptyset$. For other rules also a considerable simplification in the computation takes place. For example, one of the 8 rules with 17 rules in its set *friends* ∪ *obviated* is

$$r := x \in \{0\}, z \in \{0, u\} \rightarrow y \neq 0.$$

Consider the CSP $\langle \equiv ; x \in \{0\}, y \in \{0, 1, u\}, z \in \{0, u\}\rangle$. In the R algorithm the selection of r is followed by the application of the rules in *friends* and the removal of the rules in *friends* ∪ *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 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) ∪ *obviated*(g).

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

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 [6]. 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) ∪ *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.

The savings obtained by means of the lists *friends*(g) and *obviated*(g) are orthogonal to the ones obtained by a transformation of the CHR propagation rules into the simplification rules discussed in Abdennadher and Rigotti [2]. We think that there is a relation between two approaches that we plan to study closer.

Acknowledgments

5. REFERENCES

[1] S. Abdennadher, E. Krämer, M. Saft, and M. Schmaus. JACK: A Java Constraint Kit. In *International Workshop on Functional and (Constraint) Logic Programming (WFLP 2001), Kiel, 2001*, 2001.

[2] S. Abdennadher and C. Rigotti. Using confluence to generate rule-based constraint solvers. In *Proceedings of the 3rd Int. Conf. on Principles and Practice of Declarative Programming (PPDP 2001), Firenze, Italy*, September 2001.

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

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

[5] K. R. Apt and E. Monfroy. Constraint programming viewed as rule-based programming. *Theory and Practice of Logic Programming*, 1(6):713–750, 2001. Available via http://arXiv.org/archive/cs/.

[6] M. Egenhofer. Reasoning about binary topological relations. In O. Günther and H.-J. Schek, editors, *Proceedings of the 2nd International Symposium on Large Spatial Databases (SSD)*, volume 525, pages 143–160. Springer-Verlag, 1991.

[7] T. Frühwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming*, 37(1–3):95–138, October 1998. Special Issue on Constraint Logic Programming (P. Stuckey and K. Marriot, Eds.).

[8] C. Holzbaur, M. García de la Banda, D. Jeffery, and P. J. Stuckey. Optimizing compilation of constraint handling rules. In *Proceedings of the 2001 International Conference on Logic Programming*, volume 2237 of *Lecture Notes in Computer Science*, pages 74–89. Springer-Verlag, 2001.

[9] S. C. Kleene. *Introduction to Metamathematics*. van Nostrand, New York, 1952.

21