Rule-based constraint propagation: theory and applications
Brand, S.

Citation for published version (APA):

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: http://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
Chapter 3

Rule Schedulers

3.1 Introduction

In the rule-based approach to constraint programming, the computation process is limited to a repeated application of the propagation rules intertwined with splitting (labelling). The viability of this approach crucially depends on the availability of efficient schedulers for such rules. This motivates the work reported here. We provide an abstract framework for such schedulers and instantiate it for a case of constraint propagation rules, the membership rules. This leads to an implementation that yields a considerably better performance for these rules than the execution of their standard representation as rules in CHR [Frühwirth, 1998].

More precisely, we study schedulers for a generic class of rules which we call prop rules. Our approach is explained by the fact that constraint propagation rules, and hence membership rules, are instances of this class. 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 chapter, 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 membership rules. The implementation is provided as a program in ECLiPSe [Wallace et al., 1997] that accepts a set of membership rules as input and constructs an ECLiPSe 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 several benchmarks we found
that our implementation is considerably faster than CHR.

It is important to stress that this implementation is 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.

3.1.1. Example. To see the kind of information we use, consider the membership rule

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

Recall that, 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. Thus, 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 concept of a prop rule generalises these observations to conditions on the rule premise and body.

Constraint Handling Rules. The runtime system of the CHR language provides a rule scheduler. 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, 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.

An important novelty in our approach is the expanded, 'semantic' preprocessing phase during which we analyse 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.

3.2 Generic Iteration Algorithm

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

\[ \text{Gl} : \text{ function set } F \rightarrow \text{ least common fixpoint} \]

\[
d := \bot \\
G := F \\
\text{while } G \neq \emptyset \text{ and } d \neq T \text{ do} \\
\quad \text{choose } g \in G \\
\quad G := G \setminus \{g\} \\
\quad G := G \cup \text{update}(G, g, d) \\
\quad d := g(d) \\
\text{end} \\
\text{return } d
\]

Figure 3.1: Generic Iteration Algorithm Gl

3.2.1. Definition.

- \( f \) is called inflationary if \( x \subseteq f(x) \) for all \( x \).
- \( f \) is called monotonic if \( x \subseteq y \) implies \( f(x) \subseteq f(y) \) for all \( x, y \).

The Gl algorithm in Fig. 3.1 is used to compute the least common fixpoint of the functions from \( F \). We assume that for all \( G, g, d \) the set of functions \( \text{update}(G, g, d) \) from \( F \) is such that

A. \( \{ f \in F - G \mid f(d) = d \land f(g(d)) \neq g(d) \} \subseteq \text{update}(G, g, d) \),

B. \( g(d) = d \) implies that \( \text{update}(G, g, d) = \emptyset \),

C. \( g(g(d)) \neq g(d) \) implies that \( g \in \text{update}(G, g, d) \).

Intuitively, assumption A states that \( \text{update}(G, g, d) \) 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 those functions \( f \) for which the current value of \( d \) is not a fixpoint.

The use of the condition \( d \neq T \), absent in the original presentation [Apt, 2000], allows us to leave the while loop earlier.

Our interest in the Gl algorithm is clarified by the following result.

3.2.2. Theorem (Correctness). Suppose that all functions in \( F \) are inflationary and monotonic and that \( (D, \subseteq) \) is finite and has least element \( \bot \) and greatest element \( T \). Then every execution of the Gl algorithm terminates and computes in \( d \) the least common fixpoint of the functions from \( F \).
Chapter 3.  Rule Schedulers

PROOF. Consider the predicate $I$ defined by:

$$\forall f \in F - G. f(d) = d \quad \land \quad \forall f \in F. f(T) = T. \quad (I)$$

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

$$(G = \emptyset \lor d = T) \land I$$

holds, which implies

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

This means 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 $d$ is the least common fixpoint follows from the assumption that all functions are monotonic.

Termination is established by considering the lexicographic ordering of the strict partial orderings $(D, \sqsubseteq)$ and $(\mathbb{N}, <)$, defined on the elements of $D \times \mathbb{N}$ by

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

With each while loop iteration of the algorithm, the pair $(d, |G|)$ strictly decreases in the well-founded ordering $<_{\text{lex}}$. \qed

3.3 Revised Generic Iteration Algorithm

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 and can also simplify the update operations, 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 sequences\footnote{We need in it sequences instead of sets since the considered functions will be applied in a specific order. For simplicity, we regard these sequences as sets in some places.} of functions from $F$ are given, $\text{friends}(g, d)$ and $\text{obviated}(g, d)$, to be instantiated below. 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 $\text{friends}(g, d)$ and by a removal of the functions from $\text{friends}(g, d)$ and from $\text{obviated}(g, d)$ from $F$ and $G$. The modified algorithm is called RGI; see Fig. 3.2.

We now formalise the condition that ensures correctness of this scheduling strategy, that is, under which the Correctness Theorem 3.2.2 holds with the GI
3.3. Revised Generic Iteration Algorithm

\[ \text{RGII} : \text{ set of functions } F \rightarrow \text{ their least common fixpoint} \]

\[
d := \bot \\
G := F \\
\textbf{while } G \neq \emptyset \textbf{ and } d \neq \top \hspace{1em} \textbf{do} \\
\hspace{1em} \text{choose } g \in G \\
G := G - \{g\} \\
\hspace{1em} \text{let } \text{Del} = \text{friends}(g, d) \cup \text{obviated}(g, d) \\
\hspace{1em} \text{let } h = g \circ g_1 \circ \cdots \circ g_k \text{ where } \text{friends}(g, d) = \langle g_1, \ldots, g_k \rangle \\
F := F - \text{Del} \\
G := G - \text{Del} \\
G := G \cup \text{update}(G, h, d) \\
d := h(d) \\
\textbf{end} \\
\textbf{return} \ d
\]

Figure 3.2: Revised Generic Iteration Algorithm RGII

The algorithm replaced by the RGII algorithm. 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 never change subsequent values of \(d\). We use the notion of stability.

**3.3.1. Definition.** Suppose \(f \in F\) and \(d \in D\).

- We say that \(f\) is **stable above** \(d\) if \(d \sqsubseteq e\) implies \(f(e) = e\).
- We say that \(f\) is **stable** if it is stable above \(f(d)\) for all \(d \in D\).

That is, \(f\) is stable if for all \(d\) and \(e\), \(f(d) \sqsubseteq e\) implies \(f(e) = e\). Hence, 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 also implies inflationarity. Indeed, if \(d \sqsubseteq f(d)\), then the claim holds vacuously. If \(f(d) \sqsubseteq d\), then by stability \(f(d) = d\).

Consider now the condition

\[
\forall d. \forall e \sqsubseteq h(d). \forall f \in \text{friends}(g, d) \cup \text{obviated}(g, d). f(e) = e
\]

where \(h = g \circ g_1 \circ \cdots \circ g_k\) and \(\text{friends}(g, d) = \langle g_1, \ldots, g_k \rangle\). \hspace{1em} (3.2)

That is, for all elements \(d\), each function \(f\) in \(\text{friends}(g, d) \cup \text{obviated}(g, d)\) is stable above \(g \circ g_1 \circ \cdots \circ g_k(d)\). The following result holds.
3.3.2. **Theorem.** 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 sequences of functions from $F$ are given, friends$(g, d)$ and obviated$(g, d)$, such that condition (3.2) holds.

Then the Correctness Theorem 3.2.2 holds with the GI algorithm replaced by the RGI algorithm.

**Proof.** Denote by $F_0$ the initial value of $F$. In view of condition (3.2), the following statement is an invariant of the while loop:

\[
\begin{align*}
\forall f \in F - G. f(d) &= d \land \\
\forall f \in F. f(\top) &= \top \land \\
\forall f \in F_0 - F. \forall e \sqsubseteq d. f(e) &= e.
\end{align*}
\]

(3.3)

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

\[G = \emptyset \lor d = \top\]

holds, which implies that $\forall f \in F_0, f(d) = d$. The rest of the proof is the same. \(\square\)

### 3.4 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, which 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, which we denote by $\text{holds}(b, d)$, or false. Given a rule $b \rightarrow g$ we define its application to $d$ by

\[
(b \rightarrow g)(d) = \begin{cases} 
g(d) & \text{if } \text{holds}(b, d), \\
d & \text{otherwise}. \end{cases}
\]

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

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

- We say that a condition $b$ is monotonic if for all $d, e$ we have that, if $d \sqsubseteq e$ then $\text{holds}(b, d)$ implies $\text{holds}(b, e)$.
- We say that a condition $b$ is precise if the least $d$ exists such that $\text{holds}(b, d)$. We call then $d$ the witness for $b$.
- We call a rule $b \rightarrow g$ a prop rule if $b$ is monotonic and precise and $g$ is stable. \(\square\)
3.4.1 Rules over Sets

To see how natural this class of rules is consider the following case. Take a set $A$ and consider the partial ordering

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

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

$$B \rightarrow G,$$

where $B, G \subseteq A$.

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

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

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

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

This determines the application of $B \rightarrow G$.

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 constraints and obtain constraint propagation rules in this way. Alternatively, we can view $A$ as a set of some atomic formulas and each rule $B \rightarrow G$ 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 a rule with the empty set $\emptyset$ as the condition. A closure under such a set of rules is 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 $A$.

3.4.2 The R Algorithm

We now modify the RGl algorithm for the case of prop rules. In the R algorithm in Fig. 3.3 below, we take into account that an application of a rule is a two
\[ R \] : set of rules \( F \) --- their least common fixpoint

\[
\begin{align*}
  d &:= \bot \\
  G &:= F \\
\text{while } &G \neq \emptyset \text{ and } d \neq T \text{ do} \\
  \text{choose } & (b \rightarrow g) \in G \\
  G &:= G \setminus \{b \rightarrow g\} \\
  \text{if } & \text{holds}(b,d) \text{ then} \\
  \text{let } & Del = \text{friends}(b \rightarrow g) \cup \text{obviated}(b \rightarrow g) \\
  \text{let } & h = g \circ g_1 \circ \cdots \circ g_k \\
  \quad & \text{where } \text{friends}(b \rightarrow g) = \langle (b_1 \rightarrow g_1), \ldots, (b_k \rightarrow g_k) \rangle \\
  F &:= F - Del \\
  G &:= G - Del \\
  G &:= G \cup \text{update}(G,h,d) \\
  d &:= h(d) \\
\text{else if } & \forall e \supseteq d. \neg \text{holds}(b,e) \text{ then} \\
  F &:= F \setminus \{b \rightarrow g\} \\
\text{end} \\
\text{end} \\
\text{return } & d
\end{align*}
\]

Figure 3.3: Rule scheduling algorithm \( R \)

step process: testing of the condition followed by a conditional application of the body. This allows us to drop the parameter \( d \) from the sequences friends\((g,d)\) and obviated\((g,d)\) and consequently to construct such sequences before the execution of the algorithm. The sequence 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 sequences friends\((g)\) and obviated\((g)\) is provided in a second algorithm below, the Friends and Obviated Algorithm.

Again, we are interested in identifying conditions under which the Correctness Theorem 3.2.2 holds with the Gl algorithm replaced by the R algorithm. To this end, given a rule \( b \rightarrow g \) in \( F \) and \( d \in D \), define as follows:

\[
\text{friends}(b \rightarrow g,d) = \begin{cases} 
\text{friends}(b \rightarrow g) & \text{if holds}(b,d), \\
\emptyset & \text{otherwise}
\end{cases}
\quad (3.4)
\]

and

\[
\text{obviated}(b \rightarrow g,d) = \begin{cases} 
\text{obviated}(b \rightarrow g) & \text{if holds}(b,d), \\
\{b \rightarrow g\} & \text{if } \forall e \supseteq d. \neg \text{holds}(b,e), \\
\emptyset & \text{otherwise}
\end{cases}
\quad (3.5)
\]
3.4. Functions in the Form of Rules

We obtain the following counterpart of the Correctness Theorem 3.2.2.

3.4.2. Theorem (Correctness). Suppose that all functions in F are prop rules of the form \( b \rightarrow g \), where \( g \) is inflationary and monotonic, and that \( (D, \sqsubseteq) \) is finite and has the least element \( \bot \) and the greatest element \( \top \). Further, assume that for each rule \( b \rightarrow g \) the sequences \( \text{friends}(b \rightarrow g, d) \) and \( \text{obviated}(b \rightarrow g, d) \), defined as above, satisfy condition (3.2) and the following condition:

\[
\forall d. (b' \rightarrow g' \in \text{friends}(b \rightarrow g) \land \text{holds}(b, d) \rightarrow \forall e \sqsubseteq g(d). \text{holds}(b', e)).
\] (3.6)

Then the Correctness Theorem 3.2.2 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 account of condition (3.6) and the fact that the rule bodies are inflationary functions, \( \text{holds}(b, d) \) implies that

\[
((b \rightarrow g) \circ (b_1 \rightarrow g_1) \circ \cdots \circ (b_k \rightarrow g_k))(d) = (g \circ g_1 \circ \cdots \circ g_k)(d),
\]

where \( \text{friends}(b \rightarrow g) = \langle (b_1 \rightarrow g_1), \ldots, (b_k \rightarrow g_k) \rangle \). This takes care of the situation in which \( \text{holds}(b, d) \) is true.

In turn, the definition of \( \text{friends}(b \rightarrow g, d) \) and \( \text{obviated}(b \rightarrow g, d) \) and assumption B take care of the situation when \( \neg \text{holds}(b, d) \). When the condition \( b \) fails for all \( e \sqsubseteq d \) we can conclude that for all such \( e \) we have \( (b \rightarrow g)(e) = e \). This allows us to remove at that point of the execution the rule \( b \rightarrow g \) from the set \( F \). This amounts to adding \( b \rightarrow g \) to the set \( \text{obviated}(b \rightarrow g, d) \) at runtime. Note that condition (3.2) is then satisfied.

We now provide an explicit construction of the sequences \( \text{friends} \) and \( \text{obviated} \) for a rule \( b \rightarrow g \) in the form of the F&O algorithm in Fig. 3.4. GI\((F, e)\) stands there for the GI algorithm invoked with \( \bot \) replaced by \( e \). We call a rule \( r \) relevant in an execution of GI\((F, e)\) if at some point in this execution \( r(d) \neq d \) holds after the "choose \( r \in G"\) action.

The F&O algorithm needs the witness of the rule condition \( b \), that is, the least \( d \) for which \( \text{holds}(b, d) \). For the rules we are interested in most, the witness can be easily extracted from the condition; see Section 3.4.1 for rules over sets and Section 3.6.2 for membership rules.

Note that the rule \( r = b \rightarrow g \) itself is not in \( \text{friends}(r) \) as it is a prop rule. It is contained in \( \text{obviated}(r) \), however, since \( g(d) = d \) holds by the stability of \( g \). In Section 3.6.2, we give a concrete example for the sequences \( \text{friends} \), \( \text{obviated} \) using membership rules.

The following observation shows the adequacy of the F&O algorithm for our purposes.
\[F \& O\] : rule \(r\) in rule set \(F \rightarrow (\text{friends}(r), \text{obviated}(r))\)

\begin{verbatim}
let \(r = b \rightarrow g\)
let \(w\) be the witness of \(b\)
\(d := G_l(F, g(w))\)
\(\text{friends} :=\) sequence of the relevant rules \(h \in F\) in the preceding execution of \(G_l\)
\(\text{obviated} := \langle \rangle\)
for each \((b' \rightarrow g') \in F - \text{friends}\) do
  if \(g'(d) = d\) or \(\forall e \supseteq d. \neg \text{holds}(b', e)\) then
    \(\text{obviated} := (b' \rightarrow g'), \text{obviated}\)
  \end
\end
\text{return} \((\text{friends, obviated})\)
\end{verbatim}

Figure 3.4: Friends and Obviated Algorithm \(F \& O\)

### 3.4.3. Note

Upon termination of the \(F \& O\) algorithm, conditions (3.2) and (3.6) hold, where the sequences \(\text{friends}(b \rightarrow g, d)\) and \(\text{obviated}(b \rightarrow g, d)\) are defined as in Equations (3.4) and (3.5).

Let us summarise the findings of this section that culminated in the \(R\) algorithm. Assume that all functions are in the form of rules satisfying the conditions of the Correctness Theorem 3.4.2. Then in the \(R\) algorithm, each time the evaluation of the condition \(b \rightarrow g\) succeeds,

- the rules in the sequence \(\text{friends}(b \rightarrow g)\) are applied directly without testing the value of their conditions,
- the rules in \(\text{friends}(b \rightarrow g) \cup \text{obviated}(b \rightarrow g)\) are permanently removed from the currently active set of functions \(G\) and from \(F\).

### 3.5 Recomputing Least Fixpoints

Another substantial benefit of the sequences \(\text{friends}(b \rightarrow g)\) and \(\text{obviated}(b \rightarrow g)\) surfaces 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 \(d \subseteq e\),
- 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 3.6. There, the computation of the least common fixpoint $d$ of the functions from $F$ corresponds to the constraint propagation process for a constraint $C$. The move 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 a subsequent round of constraint propagation for $C$.

Suppose now that we computed the least common fixpoint $d$ of the functions from $F$ using the RG1 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.

### 3.5.1. Theorem

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 RG1 or R algorithm. Let $F_{\text{fin}}$ be the final value of the variable $F$ upon termination of the RG1 or R algorithm.

Suppose now that $d_0 \subseteq e$. 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 $F_{\text{fin}}$.

**Proof.** Take a common fixpoint $e_1$ of the functions from $F_{\text{fin}}$ such that $e \sqsubseteq e_1$. It suffices to prove that $e_1$ is a common fixpoint of the functions from $F$.

So take $f \in F - F_{\text{fin}}$. Since condition (3.3) is an invariant of the main **while** loop of the RG1 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. We illustrate this important effect with a concrete example in Section 3.7.4.

### 3.6 Concrete Framework

We now proceed with our main goal, namely an instantiation of the scheduler algorithm framework for the case of membership rules. We have indicated in Section 3.4.1 a possible instantiation of the prop rule framework to constraint propagation rules, of which membership rules are a special case. We set up a different instantiation for these rules now, however. This specialised instantiation is more natural as it is based on domains, and membership rules essentially deal only with domains.
3.6.1 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

\[
(\mathcal{P}(D_1) \times \cdots \times \mathcal{P}(D_n), \supseteq)
\]

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 \( (D_1, \ldots, D_n) \) is the least element in this ordering while

\[
(\emptyset, \ldots, \emptyset)
\]

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

\[
(E_1, \ldots, E_n) R_\emptyset (F_1, \ldots, F_n) \text{ if } (E_1, \ldots, E_n) = (F_1, \ldots, F_n)
\]

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, and which we continue to present in the usual way with the understanding that now each of the listed sets is non-empty;

- 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 \( T \).

In what follows we denote this partial ordering by \( (D_\mathcal{P}, \subseteq) \).

3.6.2 Membership Rules

Fix a specific CSP \( \mathcal{P} = \langle \mathcal{C}; x_1 \in D_1, \ldots, x_n \in D_n \rangle \) with finite domains. Let \( C \) be one of its constraints on the variables \( y_1, \ldots, y_k, z_1, \ldots, z_m \). We recall the notion of membership rule from Section 2.2.2. The rule

\[
C, y_1 \in S_1, \ldots, y_k \in S_k \rightarrow z_1 \neq a_1, \ldots, z_m \neq a_m
\]
3.6. \textit{Concrete Framework}

is a membership rule associated with $C$. $a_1, \ldots, a_m$ are constants, and $S_1, \ldots, S_k$ are constant subsets of the respective variable domains. We drop here the condition that the sequences $y_1, \ldots, y_k$ and $z_1, \ldots, z_m$ have no variable in common so that we can combine membership rules.

Let us reformulate the interpretation of such rules so as to fit 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.

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

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

and extend the definition to the elements of the ordering $(D_P, \sqsubseteq)$ by putting

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

and

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

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

$$holds((y_1 \in S_1, \ldots, y_k \in S_k), (E_1, \ldots, E_n))$$

if $holds(y_i \in S_i, (E_1, \ldots, E_n))$ for $i \in [1..k]$.

\[\square\]

It is not difficult to see what the witness of a membership rule condition is. Consider the CSP $(C; x_1 \in D_1, \ldots, x_n \in D_n)$ and its associated partial ordering. The witness of $y_1 \in S_1, \ldots, y_k \in S_k$ is $(E_1, \ldots, E_n)$ where $E_i = S_k$ if $x_i = y_k$, and $E_i = D_k$ if $x_i$ does not occur in the condition.

Concerning the second item we proceed as follows.

3.6.2. DEFINITION. 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 $(D_P, \sqsubseteq)$ 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

$$E'_i = \begin{cases} E_i - \{a\} & \text{if } z \equiv x_i, \\ E_i & \text{otherwise.} \end{cases}$$

If the resulting sequence $(E'_1, \ldots, E'_n)$ contains the empty set, then we replace it by $T$,

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

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.

### 3.6.3. Example.

Take the CSP $(C; x_1, \ldots, x_4 \in \{a, b, c\})$ and consider the membership rule

$$x_1 \in \{a, b\}, \ x_2 \in \{b\} \rightarrow x_3 \neq a, \ x_3 \neq b, \ x_4 \neq a.$$  \(r\)

Then we have

$$r(\{a\}, \{b\}, \{a, b, c\}, \{a, b\} ) = (\{a\}, \{b\}, \{c\}, \{b\}),$$

$$r(\{a, b, c\}, \{b\}, \{a, b, c\}, \{a, b\} ) = (\{a, b, c\}, \{b\}, \{a, b, c\}, \{a, b\}),$$

$$r(\{a, b\}, \{b\}, \{a, b\}, \{a, b\} ) = T.$$

The witness of $r$ is $(\{a, b\}, \{b\}, \{a, b, c\}, \{a, b, c\})$.  \(\Box\)

In view of the Correctness Theorem 3.4.2, 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 sequences $\text{friends}(b \rightarrow g)$ and $\text{obviated}(b \rightarrow g)$ are constructed by the F & O algorithm.

### 3.6.4. Note. Consider the partial ordering $(D_P, \subseteq)$.

1. Each membership rule is a prop rule.

2. Each function $z_1 \neq a_1, \ldots, z_m \neq a_m$ on $D_P$ is
   - inflationary,
   - monotonic.  \(\Box\)
3.7. Implementation

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

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

Note that assumptions A, B, C in (3.1) hold.

3.6.5. Example. Let us illustrate the intuition behind the use of the sequences $\text{friends}(b \rightarrow g)$ and $\text{obviated}(b \rightarrow g)$. Take again $\langle C; x_1, \ldots, x_4 \in \{a, b, c\} \rangle$ and consider the membership rules

$$
\begin{align*}
& x_1 \in \{a, b\} \rightarrow x_2 \neq a, x_4 \neq b, & (r_1) \\
& x_1 \in \{a, b\}, x_2 \in \{b, c\} \rightarrow x_3 \neq a, & (r_2) \\
& x_2 \in \{b\} \rightarrow x_3 \neq a, x_4 \neq b, & (r_3) \\
& x_2 \in \{a\} \rightarrow x_1 \neq a. & (r_4)
\end{align*}
$$

Upon application of rule $r_1$, rule $r_2$ can be applied without evaluating its condition. Subsequently, rule $r_3$ can be deleted without applying it since its body has become irrelevant; the same holds for $r_1$ itself. Finally, rule $r_4$ can be deleted since its condition can now never succeed. Hence, we can have

$$
\begin{align*}
\text{friends}(r_1) &= \langle r_2 \rangle, \quad \text{and} \\
\text{obviated}(r_1) &= \langle r_1, r_3, r_4 \rangle,
\end{align*}
$$

which is in fact what the F&O algorithm computes. \hfill \square

3.7 Implementation

In this section we discuss the implementation of the R algorithm for the membership rules and compare it by means of several benchmarks with the CHR implementation in the ECL\(')PS\(') system.

3.7.1 Modelling Membership Rules in CHR

Following [Apt and Monfroy, 2001], membership rules are represented as CHR propagation rules with a single head. Recall from Section 2.2.1 that the latter ones are of the form

$$
H \Rightarrow G_1, \ldots, G_l \mid B_1, \ldots, B_m
$$

where the atom $H$ of the rule head is a defined constraint, the atoms $G_1, \ldots, G_l$ of the rule guard are built-in constraints, and the atoms $B_1, \ldots, B_m$ of the rule body are arbitrary constraints.
Let us also review how CHR propagation rules with one head are executed. 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 a CHR propagation rule from [Apt and Monfroy, 2001]. We assume that the host language is ECL'PS°. Consider the membership rule

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

associated with the defined 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 \( \text{in}(y_i, \text{LS}_i) \) to the guard of the propagation rule. The \text{in}/2 predicate is defined by

\[
\text{in}(X, L) :- \text{dom}(X, D), \text{subset}(D, L).
\]

So \( \text{in}(X,L) \) holds if the current domain of the variable \( X \) (yielded by the built-in \text{dom} of ECL'PS°) is included in the list \( L \). In turn, \( \text{LS}_i \) 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

\[ X \in \{0\}, Y \in \{1,2\} \rightarrow Z \neq 2 \]

associated with a constraint \( c \) on the variables \( X, Y, Z \). It is represented by the following CHR propagation rule:

\[
c(0,Y,Z) \implies \text{in}(Y, \{1,2\}) \land Z \#\# 2.
\]

In ECL'PS°, variables with singleton domains are automatically instantiated. So, assuming that the variable domains are non-empty, the condition of this membership rule holds if and only if 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.
3.7. Implementation

Execution of CHR

In general, the application of a membership rule as defined in Section 3.6 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. 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.

3.7.2 Benchmarks

In our approach, the repeated application of a finite set of membership rules is realised by means of the R algorithm of Section 3.3 implemented in ECL\textsuperscript{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\textsuperscript{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, we used for each considered CSP the sets of all subsumption-free valid membership and equality rules for the ‘base’ constraints. These rule sets were automatically generated using a program discussed in [Apt and Monfroy, 2001]. In the first phase, the compiler constructs for each rule \( g \) the sequences \( \text{friends}(g) \) and \( \text{obviated}(g) \). Time spent on this construction is comparable with the time needed for the generation of the equality and membership rules for a given constraint. For example, the medium-sized membership rule set for the \text{rcc8} constraint, containing 912 rules, was generated in 166 seconds while the construction of all \text{friends} and \text{obviated} sequences took 142 seconds time. It is important to note that generating the rules and the sequences \text{friends}, \text{obviated} takes place once, at compile-time, while the resulting constraint propagation procedure is typically used many times; hence fast generation is not a critical issue.

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 search (domain splitting or labelling).

In Table 3.1, we list the results for selected single constraints. For each constraint, say \( C \) on the 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 randomised labelling; i.e., 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
Chapter 3. Rule Schedulers

Table 3.1: Randomised search trees for single constraints

<table>
<thead>
<tr>
<th>Constr.</th>
<th>rc8</th>
<th>fork</th>
<th>and3</th>
<th>and9</th>
<th>and11</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rel.</td>
<td>37% / 22%</td>
<td>58% / 46%</td>
<td>66% / 49%</td>
<td>26% / 15%</td>
<td>57% / 25%</td>
</tr>
<tr>
<td>abs.</td>
<td>147/396/686</td>
<td>0.36/0.62/0.78</td>
<td>0.27/0.41/0.55</td>
<td>449/1727/2940</td>
<td>1874/3321/7615</td>
</tr>
<tr>
<td>EQU</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rel.</td>
<td>97% / 100%</td>
<td>98% / 94%</td>
<td>92% / 59%</td>
<td>95% / 100%</td>
<td>96% / 101%</td>
</tr>
<tr>
<td>abs.</td>
<td>359/368/359</td>
<td>21.6/21.9/22.9</td>
<td>0.36/0.39/0.61</td>
<td>386/407/385</td>
<td>342/355/338</td>
</tr>
</tbody>
</table>

Table 3.2: CSPs formalising sequential ATPG

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

non-solution fixpoints. Backtracking occurs if a recorded fixpoint is encountered again. In essence, all possible search trees are traversed. In some cases, this takes too much time; we then limit the number of visited nodes.

In Table 3.2, we list the results for practically motivated CSPs. We chose here CSPs that formalise the problem of automatic test pattern generation for sequential digital circuits (ATPG), to be discussed in Chapter 6. These are large CSPs that employ the and $N$ constraints of Table 3.1 and a number of other constraints, almost all of which are implemented by rules. The constraint $\text{and} N(x, y, z)$ expresses the conjunction $x \land y = z$ in an $N$-valued logic.

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 both the latter two algorithms are produced by our compiler and are thus structurally very similar, which allows a direct assessment of the improvements embodied in R.

In the tables, we provide for each constraint or CSP the ratio of the execution times between, first, R and GI, and second, R and CHR. This is followed by the absolute times in seconds in the order R / GI / CHR.
3.7. Implementation

The platform for all benchmarks was a Sun Enterprise 450 with four UltraSPARC-II 400 MHz processors and 2 GB memory under Solaris, and ECL\textsuperscript{PS} 5.5 (in single processor mode).

We find a substantial speedup in many cases when using R, both comparing R and Gl, and R and CHR.

Possibilities for Improving the Implementation

We examined some of the various ways of optimising our implementation of the R algorithm in ECL\textsuperscript{PS}. In particular, we considered a better embedding into the constraint-handling mechanism of ECL\textsuperscript{PS}, for instance by finer control of the waking conditions and a joint removal of the elements from the same variable domain instead of several disequality constraints resulting from larger sequences friends. Using such techniques, we succeeded in achieving an additional average speed-up by a factor of 4.

This open-ended work indicates that further improvements are possible. For example, an unrealised improvement with a plausible gain in efficiency is a better choice of the data structures for handling the rule sets $F$ and $G$. We use lists (plain lists and Prolog difference lists), in which, e.g., element finding has linear cost, while in a balanced tree this cost is only logarithmic.

3.7.3 Detecting When a Constraint is Solved

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 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 solution, the constraint is solved, and removed from the store by the corresponding simplification rule. This ‘solved’ test takes place interleaved with executing the propagation rules.

The implementations of Gl and R, on the other hand, check after closure under the propagation rules. The constraint is considered solved if all its variables are fixed (necessarily to a solution), or, in the case of R, if the set $F$ of remaining rules is empty; this is discussed in the following subsection. Interestingly, comparing CHR and Gl, the additional simplification rules sometimes constitute a substantial overhead while at other times their presence allows earlier termination.

3.7.4 Recomputing Least Fixpoints

Let us finally illustrate the impact of the permanent removal of rules during the least fixpoint computation, achieved here by the use of the sequences $\text{friends}(r)$ and $\text{obviated}(r)$.
3.7.1. DEFINITION. Given a set $F$ of rules, we call a rule $g \in F$ solving if $\text{friends}(g) \cup \text{obviated}(g) = F$.

Take as an example the ternary equivalence relation $\equiv$ from the three-valued logic of [Kleene, 1952, p. 334] that consists of the values, 0 (true), 1 (false) and $u$ (unknown). For instance, we have $\equiv(1,u,u)$. The full definition is given by the following truth table:

<table>
<thead>
<tr>
<th>$\equiv$</th>
<th>1</th>
<th>0</th>
<th>$u$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>$u$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>$u$</td>
</tr>
<tr>
<td>$u$</td>
<td>$u$</td>
<td>$u$</td>
<td>$u$</td>
</tr>
</tbody>
</table>

The program of [Apt and Monfroy, 2001] generates 26 minimal valid membership rules for the $\equiv$ constraint. Out of them, 12 are solving rules. For the remaining rules the sizes of the set $\text{friends} \cup \text{obviated}$ 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 termination, $G = \varnothing$, and to a reduction of the set $F$ to $\varnothing$. For other rules, a considerable simplification in the computation takes place. For example,

$$x \in \{0\}, z \in \{0, u\} \rightarrow y \neq 0 \quad (r)$$

is one of the 8 rules of which the set $\text{friends}(r) \cup \text{obviated}(r)$ has size 17.

Consider now the CSP

$$\langle \equiv (x, y, z); x \in \{0\}, y \in \{0, 1, u\}, z \in \{0, u\} \rangle.$$ 

In the $R$ algorithm, the selection of $r$ is followed by the application of the rules from $\text{friends}(r)$ and the removal of the rules from $\text{friends}(r) \cup \text{obviated}(r)$. This brings the number of the considered rules down to $26 - 17 - 9$. The $R$ algorithm subsequently discovers that none of these rules is applicable at this point. So the nine rules remain upon termination.

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 26!

For solving rules, this fixpoint computation terminates immediately.

**Solving Rules**

Interestingly, as Table 3.3 shows, solving rules occur quite frequently for equality rules, but less often so for non-equality membership rules. We list 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
3.7. Implementation

<table>
<thead>
<tr>
<th>Constraints</th>
<th>and2</th>
<th>and3</th>
<th>and9</th>
<th>and11</th>
<th>fork</th>
<th>rcc8</th>
<th>allen</th>
</tr>
</thead>
<tbody>
<tr>
<td>EQUALITY</td>
<td>6/6</td>
<td>13/16</td>
<td>113/134</td>
<td>129/153</td>
<td>9/12</td>
<td>183/183</td>
<td>498/498</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>14</td>
<td>130</td>
<td>148</td>
<td>11</td>
<td>183</td>
<td>498</td>
</tr>
<tr>
<td>MEMBERSHIP</td>
<td>6/6</td>
<td>4/13</td>
<td>72/1294</td>
<td>196/4656</td>
<td>0/24</td>
<td>0/912</td>
<td>n.a./26446</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>7</td>
<td>810</td>
<td>3156</td>
<td>9</td>
<td>556</td>
<td>n.a.</td>
</tr>
</tbody>
</table>

Table 3.3: Solving rules

the set \( \text{friends}(r) \cup \text{obviated}(r) \). The rule sets were computed using the program of [Apt and Monfroy, 2001].

The fork constraint is taken from the Waltz language for the analysis of polyhedral scenes. The rcc8 constraint represents the composition table of the Region Connection Calculus with 8 relations from [Egenhofer, 1991] (which we revisit in Chapter 9). It is remarkable that all its 183 equality rules are solving. While none of the 912 membership rule for rcc8 is solving, on average the set \( \text{friends}(r) \cup \text{obviated}(r) \) contains 556 membership rules. Also all 498 equality rules for the allen constraint, which represents the composition table of Allen’s thirteen qualitative temporal relations [Allen, 1983], are solving. The number of membership rules exceeds 26,000 and consequently they are too costly to analyse.

CHR Simplification Rules

The CHR language supports besides propagation rules also so-called simplification rules. Using such rules, one can remove constraints from the constraint store, so one can freely affect its form. In [Abdennadher and Rigotti, 2001], a method is discussed that allows one to automatically transform CHR propagation rules into simplification rules such that the semantics of the rule set is respected. The method is based on identifying or constructing propagation rules that are solving.

In contrast, our method captures the degree to which a rule is solving. We define

\[
\text{solving degree of } r \in \mathcal{R} = \frac{|\text{friends}(r) \cup \text{obviated}(r)|}{|\mathcal{R}|}.
\]

If this ratio is 1 then \( r \) is a solving rule. More typically, the ratio will be less than 1. Consider Figure 3.5 for the distribution of the solving degree of the rules for \( \text{and9} \). Only 72 rules have degree 1 and correspond thus to simplification rules.

Let \( \text{Del} \) abbreviate \( \text{friends}(r) \cup \text{obviated}(r) \). Consider now two non-solving rules \( r_1, r_2 \), that is, such that \( \text{Del}(r_1) \neq \mathcal{R} \) and \( \text{Del}(r_2) \neq \mathcal{R} \). But let also \( \text{Del}(r_1) \cup \text{Del}(r_2) = \mathcal{R} \). Suppose that during a fixpoint computation the conditions of both rules have succeeded, and their bodies have been applied. The R algorithm detects immediately that the constraint is solved, and terminates.
consequently. CHR, for which $r_1$ and $r_2$ are ordinary propagation rules, cannot detect this possibility for immediate termination. In R, we observe an \textit{accumulated effect} of removing rules from the fixpoint computation.

We revisit the issue of the relevance of solving rules for the R scheduler in Section 4.6 of the following chapter.

### 3.8 Final Remarks

We studied the problem of efficient scheduling of constraint propagation rules. Starting from a generic iteration algorithm for functions, we obtained the R scheduler by step-wise refinement. The central observation exploited in the R algorithm is that an application of some constraint propagation rule in which its condition succeeds may provide the justification to immediately apply other rules without testing their condition, or to remove other rules from the iteration. Removing a rule in the R algorithm is ultimate in the sense that a removed rule need not be reconsidered in subsequent propagation rounds, which can therefore be expected to be faster. This is important, as a constraint propagation algorithm is typically executed repeatedly, interleaved both with other propagation and search.

We described an implementation of the R scheduler for membership rules, and we gave experimental evidence for the value of the efficiency improvements by comparing the R scheduler with a generic iteration algorithm and a CHR implementation by way of benchmarks. We found substantial speedups in many cases. Finally, we argued that the increase in efficiency is not due to implicitly distinguishing solving rules from non-solving rules, but by accumulating the effect of removing rules.