High-Level Implementations of Constraint Handling Rules*

Thom Frühwirth
ECRC, Arabellastrasse 17,
D-81925 Munich, Germany
thom@ecrc.de

Pascal Brisset
ENAC, 7, av. Edouard Belin
31055 Toulouse Cedex, France
pbrisset@eis.enac.dgac.fr

ECRC Technical Report ECRC-TR-95-20, June 1995

Abstract

Constraint handling rules (CHRs) are a high level logic concurrent committed-choice language for writing constraint systems. Rapid prototyping of novel applications for constraint techniques is encouraged by the high level of abstraction and declarative nature of CHRs. In this paper we describe basic principles of implementing CHRs in logic programming languages and show how they actually have been implemented in the CHRs library of ECL^{i}PS^e, ECRC's constraint logic programming platform. All three types of multi-headed CHRs can be transformed into single-headed simplification rules. These rules correspond to guarded rules used in typical logic concurrent committed-choice languages. We then show how to implement these guarded rules in sequential (constraint) logic programming languages. The report contains three appendices involving a generic example and its translation.

1 Introduction

Constraint handling rules (CHRs) [Fru92, Fru94, FrHa95, Fru95] are a high-level language extension to write constraint systems. CHRs are essentially a logic concurrent committed-choice language consisting of guarded rules with multiple heads. CHRs can be embedded in a given host language (e.g. Prolog, Lisp, ML) to enrich it with constraint reasoning capabilities.

CHRs provide for the two essential ways of handling constraints. *Simplification* replaces constraints by simpler constraints while preserving logical equivalence

*Part of this work was supported by ESPRIT Project 5291 CHIC.

(e.g. `X>Y,Y>X <=> false`). *Propagation* adds new constraints which are logically redundant but may cause further simplification (e.g. `X>Y,Y>Z ==> X>Z`). Repeatedly applying CHRs incrementally simplifies and finally solves constraints (e.g. `A>B,B>C,C>A` leads to `false`). A third, hybrid kind of rules, called simpagation CHRs is usefull for expressing subsumption (e.g. `X>Y\X>=Y <=> true`) and relative simplification (e.g. `X=T1\X=T2 <=> T1=T2`).

The usual abstract formalism to describe a constraint system, i.e. inference rules, rewrite rules, sequents, formulas expressing axioms and theorems, can be written as CHRs in a straightforward way. Starting from the executable specification obtained from the formalism, the rules can be refined and adapted to the specifics of the application.

In the next section, we give the syntax and semantics of constraint handling rules. Readers familiar with CHRs can skip this section. Then we discuss basic principles of for their sequential or concurrent implementations in (concurrent) (constraint) logic programming languages (including Prolog) [Sha89, VH91, Sar93, JaMa94]. The compilation from CHRs into clauses of the logic host language does not effect any atoms other than the user-defined constraints. The basic translation proceeds rule by rule and can thus be used for incremental compilation.

We first show that all three types of CHRs can be transformed into multi-headed and further into single-headed simplification rules, i.e. in the guarded rules of a typical logic concurrent committed-choice language - provided it can access delayed goals and has deep guards. Guards are deep if they allow for user-defined predicates. Then we implement such guarded rules in a logic programming language without guards and committed-choice constructs, i.e. in a CLP language. We concentrate on languages with a delay-mechanism (coroutining), since the constraint goals will be modeled as goals that can delay.

The implementation scheme given in this technical report is somewhat biased towards the most advanced implementation of CHRs utilizing advanced features of ECL^{i}PS^e. In the appendix a comprehensive generic example of the result of compilation in the actual CHRs library of ECL^{i}PS^e [B*95] is given and explained. It differs from the translation scheme described by a number of optimizations, mainly to exploit head matching and produce more deterministic code. We also show the result of applying the transformations proposed in this paper to a simple example in appendix 3. Last but not least, appendix 2 lists the abstracted code of the first interpreter for CHRs.

2 Syntax and Semantics

In this section we give syntax and semantics for constraint handling rules that extend a constraint logic programming language (including Prolog) following [Fru95]. We include syntax and semantics of built-in labeling for the first time. It

should be stressed that the host language for CHRs need not be a CLP language. Indeed, work has been done at DFKI with LISP as the host language [Her93].

2.1 Syntax

We assume some familiarity with constraint logic programming (CLP). There are two classes of distinguished predicates, built-in constraints and user-defined constraints (those written in CHRs). In most CLP languages there is a binary built-in constraint for syntactic equality over terms, `=/2`, performing unification. The built-in constraint `true`, which is always satisfied, can be seen as an abbreviation for `1=1`. `false` (short for `1=2`) is the built-in constraint representing inconsistency.

A CLP+CHR program is a finite set of clauses from the CLP language and from the language of CHRs. A *CLP clause* is of the form

$$H \texttt{ :- } B_1, \ldots B_n. \quad (n \geq 0)$$

where the head H is an atom but not a built-in constraint, the body $B_1, \ldots B_n$ is a conjunction of literals called *goals*. A *query* is a CLP clause without head.

There are two basic kinds of CHRs. A *simplification* CHR is of the form

$$H_1, \ldots H_i \texttt{ <=> } G_1, \ldots G_j \mid B_1, \ldots B_k.$$

where $(i > 0, j \geq 0, k \geq 0)$ and the multi-head $H_1, \ldots H_i$ is a conjunction of user-defined constraints and the guard $G_1, \ldots G_j$ is a conjunction of literals.

A *propagation* CHR is of the form

$$H_1, \ldots H_i \texttt{ ==> } G_1, \ldots G_j \mid B_1, \ldots B_k.$$

A third, hybrid kind is called *simpagation* CHR and is of the form

$$H_1, \ldots H_i \setminus \ldots H_l \texttt{ <=> } G_1, \ldots G_j \mid B_1, \ldots B_k. \quad (0 < i < l)$$

where '\' separates the head atoms into two non-empty groups.

When embedded in logic languages with backtracking, CHRs can provide *built-in labeling*. A *labeling declaration* for a user-defined constraint H_L is of the form

$$\text{label\_with } H_L \text{ if } G_1, \ldots G_j.$$

The labeling declaration restricts the use of CLP clauses of user-defined constraints for built-in labeling. There can be several labeling declarations for a constraint.

2.2 Declarative Semantics

Declaratively, CLP programs are interpreted as formulas in first order logic. Extending a CLP language with CHRs preserves its declarative semantics[1].

A CLP+CHR program P is seen as a conjunction of universally quantified clauses. A CLP clause is an implication

$$H \leftarrow B_1 \wedge \ldots B_n.$$

Since we assume that a predicate is defined completely, we can strengthen the above using Clark's completion. Let $(H_1 \texttt{ :- } B_{11}, \ldots B_{n1}), \ldots, (H_s \texttt{ :- } B_{1s}, \ldots B_{ns})$, $(1 \leq s)$ be all the clauses with the same predicate p in the head. Then the logical reading of the predicate p is:

$$H \leftrightarrow (H = H_1 \wedge B_{11} \wedge \ldots B_{n1}) \vee \ldots \vee (H = H_s \wedge B_{1s} \wedge \ldots B_{ns}).$$

H is of the form $p(X_1, \ldots, X_r)$ where $X_1, \ldots, X_r$ are new, different variables.

A simplification CHR is a logical equivalence provided the guard is satisfied

$$(G_1 \wedge \ldots G_j) \rightarrow (H_1 \wedge \ldots H_i \leftrightarrow B_1 \wedge \ldots B_k).$$

A propagation CHR is an implication provided the guard is satisfied

$$(G_1 \wedge \ldots G_j) \rightarrow (H_1 \wedge \ldots H_i \rightarrow B_1 \wedge \ldots B_k).$$

A simpagation CHR is a logical equivalence provided the guard is satisfied

$$(G_1 \wedge \ldots G_j) \rightarrow (H_1 \wedge \ldots H_i \setminus \ldots H_l \leftrightarrow H_1 \wedge \ldots H_i \wedge B_1 \wedge \ldots B_k).$$

A labeling declaration is a precondition on the CLP clauses defining a constraint

$$(H_L = H \wedge G_1 \wedge \ldots G_j \wedge \texttt{labeling}) \rightarrow (H \leftrightarrow (B_1 \vee \ldots B_s)).$$

where $(H \leftrightarrow B_1 \vee \ldots B_s)$ is Clark's completion of the constraint predicate. The labeling phase is entered by calling the built-in predicate `labeling/0` (that is why it appears in the premise of the implication).

2.3 Operational Semantics

The *operational semantics* of CLP+CHR program can be given by a transition system. A *computation state* is a tuple

$$< Gs, C_U, C_B >,$$

[1] Even though guarded rules in general cannot be given a first order declarative semantics, CHRs admit one when we restrict their use to handling user-defined constraints, see also [Mah87, Smo91].

where Gs is a set of goals, C_U and C_B are constraint stores for user-defined and built-in constraints respectively. Let a set of atoms represent a conjunction of atoms. A *constraint store* is a set of constraints.

The *initial state* consists of a query Gs and empty constraint stores,

$$< Gs, \{\}, \{\} >.$$

A *final state* is either *failed* (due to an inconsistent built-in constraint store represented by the unsatisfiable constraint `false`),

$$< Gs, C_U, \{\texttt{false}\} >,$$

or *successful* (no goals left to solve),

$$< \{\}, C_U, C_B >.$$

The union of the constraint stores in a successful final state is called *conditional (qualified) answer* for the query Gs, written $answer(Gs)$, meaning that the query is true under the condition that the conjunction of constraints is true.

The following *computation steps* are possible to get from one computation state to the next.

Solve
$< \{C\} \cup Gs, C_U, C_B > \longmapsto < Gs, C_U, C'_B >$
if $(C \wedge C_B) \leftrightarrow C'_B$

The built-in constraint solver updates the constraint store C_B if a new constraint C was found in Gs. To *update* the constraint store means to produce a new constraint store C'_B that is logically equivalent to the conjunction of the new constraint and the old constraint store.

We will write $H =_{set} H'$ to denote equality between the sets H and H', i.e. $H = \{A_1, \ldots, A_n\}$ and there is a permutation of H', $\mathrm{perm}(H') = \{B_1, \ldots, B_n\}$, such that $A_i = B_i$ for all $1 \leq i \leq n$.

Introduce
$< \{H\} \cup Gs, C_U, C_B > \longmapsto < Gs, \{H\} \cup C_U, C_B >$
if H is a user-defined constraint

Simplify
$< Gs, H' \cup C_U, C_B > \longmapsto < Gs \cup B, C_U, C_B >$
if $(H$ `<=>` $G \mid B) \in P$ and $C_B \rightarrow (H =_{set} H') \wedge answer(G)$

Propagate
$< Gs, H' \cup C_U, C_B > \longmapsto < Gs \cup B, H' \cup C_U, C_B >$
if $(H$ `==>` $G \mid B) \in P$ and $C_B \rightarrow (H =_{set} H') \wedge answer(G)$

Simpagate
$< Gs, H'_P \cup H'_S \cup C_U, C_B > \longmapsto < Gs \cup B, H'_P \cup C_U, C_B >$
if $(H_P \setminus H_S$ `<=>` $G \mid B) \in P$ and $C_B \rightarrow ((H_P \cup H_S) =_{set} (H'_P \cup H'_S)) \wedge answer(G)$

The rules are applied to user-defined constraints in C_U and Gs whenever they match (they are instances of) the head atoms and the guard is satisfied. A guard G is *satisfied* if the result of its local execution, $answer(G)$, is entailed (implied) by the built-in constraint store C_B. To *introduce* a user-defined constraint means to take it from the goal literals Gs and put it into the user-defined constraint store C_U. To *simplify* user-defined constraints H' means to replace them by B if H' matches the head H of a simplification rule (`H <=> G | B`) and the guard G is satisfied. To *propagate from* user-defined constraints H' means to add B to Gs if H' matches the head H of a propagation rule (`H ==> G | B`) and G is satisfied. To *simpagate from* user-defined constraints H' means to add B to Gs if H' matches the head composed of H_P and H_S of a simpagation rule ($H_P \setminus H_S$ `<=> G | B`) and to remove the constraints from H' that match H_S, provided G is satisfied.

The last two transitions deal with don't know indeterminism in the CLP+CHR language.

Unfold
$< \{H'\} \cup Gs, C_U, C_B > \longmapsto < Gs \cup B, C_U, \{H = H'\} \cup C_B >$
if $(H$ `:-` $B) \in P$ and H is not a user-defined constraint

To *unfold* an atomic goal H' in Gs means to look for a CLP clause $(H\text{:}-B)$ and to replace the H' by $(H = H')$ and B. As there are usually several clauses for a goal, unfolding is nondeterministic and thus a goal can be solved in different ways using different clauses.

The clauses for user-defined constraints can only be unfolded during built-in labeling to produce choices. The built-in labeling is invoked by calling the CHR built-in predicate `labeling/0` (no arguments).

Label
$<$`labeling`$\cup Gs, \{H'\} \cup C_U, C_B > \longmapsto$
$<$`labeling`$\cup Gs \cup B, C_U, \{H = H'\} \cup C_B >$
if $(H$ `:-` $B) \in P$ and (`label_with` H'' `if` $G) \in P$ and
$C_B \rightarrow (H' = H'') \wedge answer(G)$

3 Embedding CHRs in CHRs

The operational semantics are still far from the actual workings of an efficient implementation. In this section we show that every type of CHRs can be transformed into single-headed simplification rules. We require that the concurrent host language has deep guards and allows to access delayed goals. For simplicity of presentation, we will transform CHRs with exactly two head atoms. The case of one head atom is a simple specialization of it, the case of more than two head atoms a simple generalization. Consequently, we have to deal with the following three CHRs, one for each kind:

```
% Simplification CHRs
Head1,Head2 <=> Guard | Body.
% Simpagation CHR
Head1\Head2 <=> Guard | Body.
% Propagation CHRs
Head1,Head2 ==> Guard | Body.
```

An example application of the transformations described in this section can be found in appendix 3.

3.1 Embeddings

Simplification and propagation rules can embed each other. First, assume that we want to implement all kinds of CHRs with propagation rules only. Just replacing simplification by propagation rules preserves failure and logical equivalence. However, such a naive translation effects efficiency and termination, since constraints are no longer removed. The solution is to ignore constraints that should have been removed with the help of a variable `KF` representing a *kill flag* that is added to each user-defined constraint. We denote the constraint `Head` with one extra argument `KF` added by `Head(KF)`[2]. The predicate `var/1` checks if its argument is a free (unbound, uninstantiated) variable, `kill/1` just binds the kill flag variable.

```
% Head1,Head2 <=> Guard | Body.
Head1(KF1),Head2(KF2) ==>           % Kill flags not set so far
        var(KF1),var(KF2),
        Guard
        |
        kill(KF1),kill(KF2),        % Bind kill flags to kill head constraints
        Body.

% Head1\Head2 <=> Guard | Body.
Head1(KF1),Head2(KF2) ==>
        var(KF1),var(KF2),
        Guard
        |
        kill(KF2),                  % Kill second head constraint only
        Body.

% Head1,Head2 ==> Guard | Body.
Head1(KF1),Head2(KF2) ==>
        var(KF1),var(KF2),
        Guard
        |
        Body.
```

[2] Actually, this is HiLog [CKW89] syntax, where arbitrary Herbrand terms can be function and predicate symbols.

In the converse case, which is the typical one for an implementation in a logic language, we implement every kind of CHR with simplification rules. Logically, a propagation rule ($H \rightarrow B$) is the same as the simplification rule ($H \leftrightarrow B \wedge H$). However, just adding the head constraint again in the rule body would cause looping, since the same head constraint is recursively called again and again. To avoid such trivial non-termination it is remembered - in the constraint - that a propagation rule fired. We add a list[3] `PL` to remember applications of propagation rules to each user-defined constraint. Furthermore, each rule gets a unique identifier, `n`. Initially, the constraints are called with an empty list `Head([])`. Simplification rules stay unchanged.

```
% Head1\Head2 <=> Guard | Body.
Head1(PL1),Head2(PL2) <=>
        Guard
        |
        Body,
        Head1(PL1).                % no looping, since Head2 is removed

% Head1,Head2 ==> Guard | Body.
Head1(PL1),Head2(PL2) <=>
        not_member(n-Head2-2,PL1), % rule n with second head Head2 applied ?
        not_member(n-Head1-1,PL2), % rule n with first head Head1 applied ?
        Guard
        |
        Body,
        Head1([n-Head2-2|PL1]),    % rule n with second head Head2 applied
        Head2([n-Head1-1|PL2]).    % rule n with first head Head1 applied
```

The auxiliary predicate `not_member(E,L)` fails if `E` is an element of the list `L` and succeeds otherwise.

```
not_member(E,[]) <=> true.
not_member(E,[E1|L]) <=> not (E=E1), not_member(E,L).
```

As an optimization the head constraints `Head1, Head2` in the propagation list `PL` can be replaced by their identifiers.

3.2 Multiple Head Atoms

The difficult part of a CHRs implementation is multiple head atoms, which constraint logic programming languages usually do not support. To illustrate the implementation idea, let us first assume that the concurrent host language provides for don't know indeterminism in the form of backtracking that can be used in guards. The built-in predicate `delayed_constraint(C)` unifies `C` with a delayed constraint goal that matches `C`. If there are more such goals, it returns

[3]Whereever we use a list, in practice a more sophisticated data structure can be used to minimize the cost of searching for elements.

them on backtracking. note that in a concurrent implementation we have to make sure that constraints are returned even if their guards are currently tried for satisfaction. The predicate `remove/1` removes a delayed constraint. It can be implemented using the kill flag approach from above, this time really removing killed constraints with the rule:

```
Head(KF) <=> not var(KF) | true. % remove killed constraint
```

Two-headed CHRs are replaced by single-headed ones, one for each head atom in a rule.

```
% Head1,Head2 <=> Guard | Body.
Head1(PL1) <=>
        delayed_constraint(Head2(PL2)), % find delayed partner constraint
        Guard
        |
        remove(Head2(PL2)),             % remove partner constraint
        Body.
Head2(PL2) <=>                          % same for second head constraint
        delayed_constraint(Head1(PL1)),
        Guard
        |
        remove(Head1(PL1)),
        Body.

% Head1\Head2 <=> Guard | Body.
Head1(PL1) <=>
        delayed_constraint(Head2(PL2)),
        Guard
        |
        remove(Head2(PL2)),             % remove second head constraint
        Body,
        Head1(PL1).                     % revive first head constraint
Head2(PL2) <=>
        delayed_constraint(Head1(PL1)),
        Guard
        |
        Body.

% Head1,Head2 ==> Guard | Body.
Head1(PL1) <=>
        delayed_constraint(Head2(PL2)),
        not_member(n-Head2-2,PL1),
        not_member(n-Head1-1,PL2),
        Guard
        |
        Body,
        Head1([n-Head2-2|PL1]).         % revive first head constraint
Head2(PL2) <=>
        delayed_constraint(Head1(PL1)),
```

```
    not_member(n-Head2-2,PL1),
    not_member(n-Head1-1,PL2),
    Guard
    |
    Body,
    Head2([n-Head1-1|PL2]).         % revive second head constraint
```

Now we do away with the don't know indeterminism of `delayed_constraint/1`. This means we have to program the search for a partner constraint ourselves. If the concurrent host language provides for disjunction, this is trivial. Otherwise, it complicates the translation. The idea is to create a sub-process for each potential partner, to check it for applicability, and to quit all processes once a partner has been found by one of the processes. As soon as one process find a partner, it sets a shared flag, so that all the other processes can finish and the main process is notified.

The predicate `delayed_constraints(L)` returns a list of all delayed constraints. For each rule `n`, an instance of the recursive predicate `try_each_partner/5` is introduced. The predicate goes through the list of partner constraints and tries to apply the rule to them. If head matching succeeds and the guard is satisfiable, the partner constraint found is returned. The guards from the code above,

```
    delayed_constraint(Head2),
    Guard                           % including optional not_member/2 checks
```

are changed into

```
    delayed_constraints(Head2List),
    not Head2List=[],               % at least one partner candidate
    try_each_partner(n,Head1,Head2List,Head2,FoundFlag),
    not var(FoundFlag)              % wait for FoundFlag to be set
```

with

```
try_each_partner(N,Head1,[Head2|Head2L],Partner,Found) <=>
    try_one_partner(N,Head1,Head2,Partner,Found),    % try next
    try_each_partner(N,Head1,Head2L,Partner,Found).
try_each_partner(N,Head1,[],Partner,Found) <=> true.  % all tried
try_each_partner(N,Head1,[],Partner,Found) <=>
    not var(Found) | true.           % partner already found

try_one_partner(_N,Head1,Candidate,Partner,Found) <=>
    not var(Found) | true.           % partner already found
try_one_partner(n,Head1,Head2,Partner,Found) <=>      % one for each CHR n
    var(Found),                      % partner not found yet
    Guard
    |
    Found=true,                      % set FoundFlag to notify others
    Partner=Head2.                   % return partner constraint found
```

What is missing from the above implementation is the treatment of the case that no partner at all has been found. Then the partner search should fail. For this reason, we introduce an additional argument to `try_one_partner/5`, a flag that is set if the candidate is not a partner.

```
try_one_partner(n,Head1,Candidate,Partner,Found,NotFound) <=>
     not (                          % cannot be partner or already found
     Candidate=Head2,
     var(Found),
     Guard)
     |
     NotFound=true.                 % set NotFoundFlag
try_one_partner(n,Head1,Head2,Partner,Found,NotFound) <=>
     ...                            % same as before
```

The predicate `try_one_partner/5` could also be implemented using a simple conditional construct if available (see later section).

In the predicate `try_each_partner/6`, a `NotFoundFlag` variable for each subprocess `try_one_partner/6` is created and kept in a list.

```
try_each_partner(n,Head1,[Head2|Head2L],Partner,Found,NFL) <=>
     NFL=[NF|NFL1],                 % collect NotFoundFlags in list NFL
     try_one_partner(n,Head1,Head2,Partner,Found,NF),
     try_each_partner(n,Head1,Head2L,Partner,Found,NFL1).
try_each_partner(n,Head1,[],Partner,Found,NFL) <=> NFL=[]. % close list
```

To the initial guard we add a negated check that the list consists of set flags (i.e. `true`) only. In an actual implementation, the head constraints passed as arguments can often be replaced by the list of their variables. If available, `try_one_partner/6` can also be implemented using a if-then-else construct.

For propagation rules (and the second rule resulting from simpagation rules) the coding can be substantially optimized by taming the recursive calls of the head constraint. First note that through this recursion a propagation rule eventually is correctly applied to all constraints that qualify as a partner, not to just one. We can therefore collect all partners in a revised predicate `try_each_partner/6` and execute all the associated bodies after the commit. The collection can be implemented using a list of fixed length (one element for each candidate) as stream on which the subprocesses either return a matching partner or a notification that none has been found.

The recursive call of the head constraint also reconsiders all previous rules again, whereas one could continue just after the propagation rule that was tried in the previous round. If the rules are tried in the order of their identifiers, this behavior can be achieved by only allowing CHRs with the same or higher identifier in the recursive, continued execution of the head constraint. Optimizing further this leads away from rule by rule compilation to a global compilation of the whole

rule set. See the ECL^{i}PS^{e}implementation in appendix 1 for the final outcome and appendix 3 for an example following the transformations proposed here.

Regarding program size, the translation scheme only incurres an overhead for multi-headed CHRs. In that case it introduces a guarded rule (single-headed simplification CHR) for each head constraint in the CHR and two rules defining the instance of `try_one_partner/6` for each head of multi-headed rules. This means at three rules for each head constraint in a multi-headed CHRare resulting from the transformation.

3.3 Propagation CHRs as Conditionals

In this subsection we discuss an alternative way to implement propagation CHRs. However, in the end it will turn out that it leads to basically the same final translation. The idea is that propagation CHRs with a single head can be implemented by conditionals. Such a construct is available in most concurrent logic languages. A simple conditional is of the form

```
Condition -> Consequence
```

where `Condition` is a guard and `Consequence` a body. If `Condition` is satisfied, the `Consequence` is executed, if `Condition` does not hold, the conditional succeeds without further computation. A conditional can be implemented with simplification CHRs:

```
(Condition -> Consequence) <=> Condition | Consequence.
(Condition -> Consequence) <=> not Condition | true.
```

Depending on the overall implementation, the second rule can be specialized or dropped. The problem with this simple definition is that it makes each variable occurring in `Condition` global, since it also occurs in the head of the simplification CHR. However, the actual global variables of the conditional are only those appearing both in the conditional and the surrounding context. To overcome this problem, we introduce an argument for the global variables and use a predicate `rename_local/3` to rename the remaining, local variables into new variables.

```
GlobalVars:(Condition -> Consequence) <=>
        rename_local(GlobalVars,Condition,Condition1),  % rename local vars
        Condition
        |
        Condition=Condition1,        % unify old and new local variables
        Consequence.
GlobalVars:(Condition -> Consequence) <=> not ...  % analogous positive case
```

In another solution, each call to a conditional, `Condition ->Consequence`, can be replaced by a new, auxiliary constraint whose arguments are the global variables. In the following, for simplicity, we do not mention the global variables of a conditional explicitly.

A set of `n` single-headed propagation rules for the constraint `c/m`

```
Head1 ==> Guard1 | Body1.
...
Headn ==> Guardn | Bodyn.
```

can be rewritten as a conjunction of conditionals and placed in the body of a simplification rule

```
Head <=> Head', (Head=Head1,Guard1 -> Body1),...,(Head=Headn,Guardn -> Bodyn).
```

`Head` is of the form `c(X1,...Xm)` where `X1,...Xm` are new, disjoint variables. `Head'` is the same as `Head` except that `c/m` is renamed to `c'/m` to avoid a trivial loop. Consequently, the same renaming has to be applied to the heads of all simplification rules. Note that the global variables of the conditionals are exactly the variables occurring in `Head`.

In the original CHRs, once a simplification rule has been applied to a constraint, no subsequent propagation involving this constraint is possible, since it has been removed by the simplification. This is not the case in the translation above, since only `Head'` will be removed, but not the conditionals associated with the constraint `Head`. To simulate the original behavior, we introduce a kill flag variable in an additional argument of `c'/m`. When a simplification rule applies to `c'/m+1`, the kill flag variable is bound. The translation is now as follows:

```
Head <=> Head'(KF), (var(KF),Head=Head1,Guard1 -> Body1),
                ..., (var(KF),Head=Headn,Guardn -> Bodyn).
```

With the kill flag, we can specialize the second simplification rule used to define the conditional into a more efficient, but more lazy rule:

```
(Condition -> Consequence) <=> not var(KF) | true.
```

We have already shown how to implement multi headed CHRs. It may seem that for propagation rules, conditionals would result in a different translation. However it turns out that this is not really the case. In the `Condition` we need a predicate to try each partner constraint. That means for each potential partner given by `delayed_constraints/2` the predicate creates a new conditional. The predicate is very similar thus to `try_each_partner/6` for propagation CHRs, except that the rule bodies are not collected but used to form the `Consequence` parts of the conditionals. Since `delayed_constraints/2` may return new candidates on a later call, we have to replace `Head'(KF)` by a direct recursive call `Head(KF)` and once again use a propagation list to avoid trivial loops. Another possibility would be a variant of `delayed_constraints/2` that returns a stream of delayed constraints. The main difference with the previous approach is that the conjunctive treatment of propagation CHRs with many delayed conditionals is "more concurrent". Therefore such a translation seems to be more suitable for a inherently concurrent logic language, while in sequential CLP languages the cost of delaying goals is high as compared to backtracking.

3.4 Built-In Labeling

Last but not least, we show how to implement *built-in labeling* in a CHR. Labeling is the only point which requires the host language to offer don't know indeterminism. Assume that a form of disjunction denoted by the binary operator `or/2` is available. Let $(H \leftrightarrow B_1 \vee \ldots B_s)$ be Clark's completion of the constraint predicate. From a labeling declaration

```
label_with Head if Guard.
```

and Clark's completion of the associated constraint predicate, a simplification rule involving the built-n predicate `labeling/0` is produced:

```
labeling, Head <=> Guard | Head=H, (B1 or ... Bs), labeling.
```

Note the use of recursion in `labeling/0` to enforce further labeling after executing the disjunction which has introduced some choices and subsequent constraint handling. This formulation relies on the left-to-right execution model common to logic programming languages. A simpagation CHR with the same declarative semantics as the above simplification CHR can be written. However, the operational semantics differ, since there is no guarantee that the simpagation rule is executed only after all other rules for all constraints have been tried.

4 Implementing Guarded Rules in CLP

In this section we show how to implement guarded rules (corresponding to single-headed simplification CHRs), i.e. a committed-choice language, in a CLP language without guards. Such translations have been investigated before, i.e. compilation of matching in committed-choice languages, L. Naish's successive implementations of delaying declarations [Nai85], S. K. Debray's efficient implementation of QD-Janus [Deb93] in Prolog. The translation proposed in this section is based on ideas of Joachim Schimpf and is geared towards ECL^{i}PS^e and the actual implementation. It requires that the CLP language is equipped with a delay-mechanism.

A delay-mechanism can be implemented in any logic programming language by passing the list of delayed goals around in additional arguments of each predicate (a DCG grammar could be used). A complete delay mechanism can be implemented this way - at the cost of efficiency, of course.

The only built-in predicates needed are for delaying a goal on variables and for accessing the delayed goals. The built-in predicate `delay(L,G)` delays a goal `G` on the variables in the list `L` until one of the variables is touched. A variable is *touched* if takes part in a unification or if it gets more constrained by built-in constraints.

In a sequential CLP implementation, backtracking is efficient while delaying is usually more expensive than in inherently concurrent languages. Therefore it is more efficient to reexecute guards instead of delaying them and executing them incrementally. In our ECL^{i}PS^e implementation we also found that there is no gain in distinguishing between failure and delaying of a guard. If a guard is not satisfiable, it simply fails. Overall, using this approach in ECL^{i}PS^e we gained about one order of magnitude in speed as compared to a fully concurrent implementation we were initially aiming at. The efficiency tradeoff may no longer hold for very complex guards or other host languages.

Under these assumptions, a constraint goal fails if no rule was applicable (all guards failed). In such a case, we redelay the goal on its variables. When a variable is touched, the goal will be resumed and reexecuted. To achieve this behavior, for each constraint `Head`, the last clause is:

```
Head :- extract_vars(Head,VarList),delay(VarList,Head).
```

where the predicate `extract_vars(T,L)` returns the list `L` of free variables of the term `T`.

We now implement head matching and guard execution. Head matching can be made explicit by adding the goal `Goal=Head` to the guard. Instead of the guarded rule

```
c(t1,...tn) <=> Guard | Body.
```

we use the guarded rule

```
c(X1,...Xn) <=> c(X1,...Xn)=c(t1,...tn), Guard | Body.
```

where `X1,..Xn` are new, disjoint variables. If we do not delay guards, the equality can be optimized by using a built-in predicate like `instance(Goal,Head)` that checks if `Goal` is an instance (i.e. matches) `Head` and then unifies them. Since `Head` is known at compile-time the call to `instance/2` can be further optimized. In ECL^{i}PS^e, there is no need for a transformation, since head matching is directly supported.

Clearly if the execution of a guard further constrains global variables (those from the head(s) of the rule), it cannot be satisfied at the moment and has to delay. A variable is more constrained if it is touched or if new goals delay on it. Since we also fail a delayed guard, we would like to fail in those cases.

One way to protect the global variables from being touched is to replace them with new variables in the execution of the guard. The predicate `copy_term/2` copies a term with new variables. Then we could use the following translation

```
HeadC <=> copy_term(HeadC,Head), Guard, instance(HeadC,Head) | Body.
```

where `HeadC` is a copy of `Head` with new variables. Once again, the instance check can be optimized. The problem with this translation is that the whole `Guard` is executed before it is checked that global variables have been touched. Since touching global variables may cause a cascade of constraint handling, this solution is too expensive. Remember that if a variable is touched, all the goals that delay on it are woken. Thus we can delay a failing goal, i.e. simply `false`, on the global variables to avoid that they are touched.

```
extract_vars(Head,GlobalVars),delay(GlobalVars,false),Guard,remove(false)
```

Note that the two goals will prefix every `Guard` and thus can be factored out using an auxiliary predicate `Head'` for the rest of the code.

```
Head :- extract_vars(Head,GlobalVars), delay(GlobalVars,false), Head'.
```

To detect if delayed goals have been added, we check whether the list of delayed goals is still the same. We use the built-in predicate `delayed_constraints/1` to compare the list of delayed goals before and after the execution of the guard. At this point, we reach the border-line of where a high-level implementation can go, since a low-level check will be considerably more efficient and independent of the size of the list of delayed goals.

```
c(X1,...Xn)' :-
        c(X1,...Xn)'=Head',          % match the head with the actual goal
        delayed_constraints(CL)      % get all delayed constraint goals
        Guard,                       % execute guard
        delayed_constraints(CL)      % no new delayed constraint added
        remove(false),               % no global vars have been touched
        !,                           % commit by cutting
        Body.
```

5 Existing Implementations

The first implementation of CHRs in 1991 was an interpreter written in ECRC's constraint logic programming platform ECL^{i}PS^e(see appendix 1). At the moment, there exist two sequential implementations, one prototype in LISP [Her93], and one fully developed CHRs library in ECL^{i}PS^e[B*95]. At DFKI Saarbrücken, an implementation of CHRsin the concurrent object-oriented language OZ [SmTr94] is on the way.

The LISP implementation does not provide for simpagation rules, but offers some interesting extensions. First, rules can be given priorities (encoded as integers). Second, indeterminism is introduced by disjunction in rule bodies. This extension also allows to express Prolog clauses. Rules with disjunction are translated into simplification rules explicitly creating choice-points and performing backtracking. Rules with disjunction usually get the lowest priority. The

algorithm for executing CHRs is somewhat similar to the first implementation of CHRs in Prolog (see appendix 2). However, matching a head constraint in a rule with several heads dynamically adds a new rule with the matched head removed and the variables instantiated as in the matching. In [B*95], constraint handlers for terminological reasoning with negation and concrete domains, further equality over Herbrand terms, inequalities, finite domains, linear polynomial inequalities using Fouriers algorithm and an implementation of the terminological language TAXLOG are described as applications.

In the CHRs library in ECL^{i}PS^e, ECL^{i}PS^e and CHRs statements can be freely combined. A complete committed-choice language is available as a side-effect. The library includes a compiler, a run-time system with two debuggers, many example solvers as well as a full color demo using geometric constraints in a real-life application for wireless telecommunication. The compiler is about 450 clauses, 2700 lines, 26kB of code, the run-time system is about 360 clauses, 1900 lines, 17kB of code including comments. The code produced by the compiler from a comprehensive rule set can be found in the appendix. About 20 constraint solvers currently come with the release (see figure 1) - among them solvers for finite domains over arbitrary ground terms, reals and pairs, incremental path consistency, temporal reasoning (quantitative and qualitative constraints over time points and intervals [Fru94]), for solving linear polynomials over the reals and rationals, and last but not least for terminological reasoning [FrHa95]. A successful real-life application making essential use of CHRs is described in [MBF95].

Typically it took only a few days to produce a reasonable prototype solver, since one can directly express how constraints simplify and propagate without worrying about implementation details. The average number of rules in a constraint solver is as low as 24.

To reflect the complexity of a program in the number of CHRs, at most two head constraints are allowed in a rule. This forces the programmer to rewrite a rule with more than two head constraints into several two-headed rules. The restriction to two head atoms makes complexity for search of the head constraints of a single CHR quadratic in the worst case. On average, linear complexity can be achieved based on the observation that usually the head atoms are connected through common variables appearing in both head atoms, which means that only the constraint goals that delay on a particular variable have to be searched. Complexity can be reduced by using a more sophisticated data structure than lists for the delaying constraints.

On a range of solvers and examples, the slow-down for our declarative and high-level approach turned out to be within an order of magnitude in comparison to dedicated built-in solvers (if available). On some examples (e.g. those involving finite domains with the element-constraint), our approach is faster, since one can exactly define the amount of constraint handling that is needed. For performance and simplicity the solver can be kept as incomplete as the application allows it. Some solvers (e.g. disjunctive geometric constraints in the phone demo) would

be very hard to recast in existing CLP languages.

Domain	Algorithm	C?	Library File	Si	Sp	Pr
Term Manipul.		yes	term	10	8	7
Terminologies		no	kl-one	25	4	13
Rational Trees	Unification	no	tree	9	2	1
Lists	Extend. Unification	no	list	9	0	0
Sets	Consistency	no	set	18	10	13
Comparisons	Algebraic Laws	yes	minmax	11	22	6
Equalities	Gaussian Elimin.	yes	math-gauss	1	1	0
Inequalities	Gaussian + Slacks	no	math-lazy	19	6	0
Inequalities	Gaussian + Slacks	no	math-eager	19	6	0
Inequalities	Gaussian + Fourier	yes	math-fourier	21	6	1
Booleans	Value Propagation	no	bool	56	19	0
Finite Domains	Forward Checking	no	domain	61	7	14
Binary Relations	Path Consistency	no	time-pc	10	1	3
Time	Path Consistency	no	time-point	4	0	2
Time	Path Consistency	no	time	0	2	0
Space		yes	geons	0	1	0
Prime Numbers			primes	11	3	0
Sound Control			control	6	0	0
Rounded Average		no		16	5	3

Figure 1 The constraint solvers of the CHRs library in ECL^{i}PS^e.[4]

6 Conclusions

Constraint handling rules (CHRs) are a language extension for implementing user-defined constraints. We have given basic principles on how to implement CHRs in logic programming languages and we have shown what the result of compiling CHRs into ECRC's constraint logic programming platform ECL^{i}PS^e is. It turned out that CHRs can be easily implemented in any constraint logic programming language, be it concurrent or sequential.

According to our experience, efficiency depends mainly on updating delayed constraint goals and the search for a partner constraint. Both issues can be tackled by using a more sophisticated data structure than a list of delayed goals. To avoid redundant computations in the guards, they could be compiled into decision graphs. Furthermore, the constraints generated by propagation CHRs could be garbage collected (i.e. removed from the constraint store) when the constraints they were generated from have been rewritten or unfolded.

[4] C? stands for Complete?; Si, Sp, Pr are the numbers of Simplification, Simpagation and Propagation rules respectively.

The CHRs language offers a high potential for implementation on multi-processor systems, as guards can be processed and rules be applied concurrently and different choices can be processed independently in or-parallel mode. The latter is the topic of some ongoing experiments with the new parallel release of ECL^{i}PS^e.

Acknowledgements

Joachim Schimpf contributed with ideas and suggestions. Comments from anonymous referees on a short version of this paper were taken into account.

References

[B*95] P. Brisset et al., ECL^{i}PS^e 3.5.1 Extensions User Manual, ECRC Munich, Germany, April 1995.

[CKW89] Chen, W. and Kifer, M. and Warren, D. S., HiLog: A First-Order Semantics for Higher-Order Logic Programming Constructs, Proceeding of the North American Conference on Logic Programming, Cleveland, Ohio, October 1989, pp. 1090-1114.

[Deb93] S. K. Debray, QD-Janus : A Sequential Implementation of Janus in Prolog, Software—Practice and Experience, Volume 23, Number 12, December 1993, pp. 1337-1360.

[Fru92] T. Frühwirth, Constraint Simplification Rules, Technical Report ECRC-92-18, ECRC Munich, Germany, July 1992 (revised version of Internal Report ECRC-LP-63, October 1991), available by anonymous ftp from ftp.ecrc.de, directory pub/ECRC_tech_reports/reports, file ECRC-92-18.ps.Z).

[Fru94] T. Frühwirth, Temporal Reasoning with Constraint Handling Rules, Technical Report ECRC-94-05, ECRC Munich, Germany, February 1994 available by anonymous ftp from ftp.ecrc.de, directory pub/ECRC_tech_reports/reports, file ECRC-94-5.ps.Z.

[FrHa95] T. Frühwirth and P. Hanschke, Terminological Reasoning with Constraint Handling Rules, Chapter in Principles and Practice of Constraint Programming (P. Van Hentenryck and V.J. Saraswat, Eds.), MIT Press, April 1995, (revised version of Technical Report ECRC-94-6, ECRC Munich, Germany, February 1994, available by anonymous ftp from ftp.ecrc.de, directory pub/ECRC_tech_reports/reports, file ECRC-94-6.ps.Z).

[Fru95] T. Frühwirth, Constraint Handling Rules, Chapter in "Constraint Programming: Basics and Trends" (A. Podelski, ed.), Springer LNCS 910, March 1995, pp. 90ff.

[Her93] B. Herbig, Eine homogene Implementierungsebene fuer einen hybriden Wissensrepräsentationsformalismus, Master Thesis, in German, University of Kaiserslautern, Germany, August 1993.

[JaMa94] J. Jaffar and M. J. Maher, Constraint Logic Programming: A Survey, Journal of Logic Programming, 1994:19,20:503-581.

[Mah87] Maher M. J., Logic Semantics for a Class of Committed Choice Programs, Proc of the Fourth Intl Conf on Logic Programming MIT Press 1987, pp. 858-876.

[MBF95] J.-R. Molwitz, P. Brisset and T. Frühwirth, Planning Cordless Business Communication Systems, IEEE Expert Magazine, Special Track on Intelligent Telecommunications, to appear December 1995.

[Nai85] Naish L., Prolog control rules, Proceedings of the Ninth International Joint Conference on Artificial Intelligence, Los Angeles, California, September 1985, pp. 720-722.

[Sar93] V. A. Saraswat, Concurrent Constraint Programming, MIT Press, Cambridge, 1993.

[Sha89] E. Shapiro, The Family of Concurrent Logic Programming Languages, ACM Computing Surveys, 21(3):413-510, September 1989.

[Smo91] G. Smolka, Residuation and Guarded Rules for Constraint Logic Programming, Digital Equipment Paris Research Laboratory Research Report, France, June 1991.

[SmTr94] Gert Smolka and Ralf Treinen (ed.), DFKI Oz Documentation Series, Deutsches Forschungszentrum für Künstliche Intelligenz, Stuhlsatzenhausweg 3, D–66123 Saarbrücken, Germany, 1994, available via WWW from http://ps-www.dfki.uni-sb.de/oz/.

[VH91] P. van Hentenryck, Constraint Logic Programming, The Knowledge Engineering Review, Vol 6:3, 1991, pp 151-194.

Appendix 1 - Complete Compilation Example

Compiling the following generic CHR code (which contains all types of rules)

```
% all options are turned off for simplicity
?- nodbgcomp.                          % no code for debugger produced
option(check_guard_bindings, off).     % simple guard check
option(already_in_store, off).
option(already_in_heads, off).

constraints p/3,q/3.

rule1 @ p(a,X,Y) ==> guard(a,X,Y,G) | body(a,X,Y,G,B).
rule2 @ p(b,X,Y) <=> guard(b,X,Y,G) | body(b,X,Y,G,B).
rule3 @ p(c,X,Y),q(c,Y,Z) ==> guard(c,X,Y,Z,G) | body(c,X,Y,Z,G,B).
rule4 @ p(d,X,Y),q(d,Y,Z) <=> guard(d,X,Y,Z,G) | body(d,X,Y,Z,G,B).
rule5 @ p(e,X,Y)\q(e,Y,Z) <=> guard(e,X,Y,Z,G) | body(e,X,Y,Z,G,B).

label_with p(f,X,Y) if guard(f,X,Y,G).
p(g,X,Y) :- body(g,X,Y,B).
```

yields the code given below (edited for readability, all directives have been removed, some predicates renamed, comments have been added, variables have been renamed automatically). Turning the option `check_guard_bindings` off means that it is not checked if global variables are touched. The optional '`rule1 @`' piece of syntax allows to give names to rules.

Note that in the compiled code the order of the rules has changed, single head atoms are moved ahead of multiple head atoms and simplification CHRs ahead of propagation CHRs for efficiency reasons. The code is cluttered since introduces a number of auxiliary predicates due to optimizations like exploiting head matching and indexing as much as possible and avoiding nondeterministic code. Furthermore, conjunctions are kept is short is possible by moving right hand side subgoals down into the definitions of left hand side subgoals where possible. The implementation of built-in labeling has not been optimized.

The built-in predicates used are `=/2, var/1` and `nonvar/1`. The low-level predicates used are `execute_guard/1, delay/2, get_delayed_goals/2` and `check_and_mark_applied/2`. Their code is not given here. `execute_guard/1` basically wraps a low-level check (that the delayed goals did not change) around the execution of a guard. To optimize the search for a partner constraint, `get_delayed_goals/2` gets only the goals that delay on a variable occurring in the first argument. The code of `labeling/0` is not given here, it makes use of the `label_with/3` clauses produced for each constraint. Code starts on next page.

```
%%% The following code has been produced by the CHR compiler

% constraints p/3,q/3.
p(A, B, C) :-                           % entry point for constraint call
        p_3(p(A, B, C), KillFlag, FiredPropagationCHRsList, Identifier).
                                        % Identifier used in debuggers only

q(A, B, C) :-
        q_3(q(A, B, C), D, E, F).

%%% Label_with declaration for p / 3

% label_with p(f,X,Y) if guard(f,X,Y,G).
label_with(p(f, A, B), C, D) ?-
        execute_guard(guard(f, A, B, E)),       % check the guard
        !,
        C = clause_p(f, A, B).          % return associated Prolog predicate

%%% Prolog clauses for p / 3

% p(g,X,Y) :- body(g,X,Y,B).
clause_p(g, A, B) :-                    % Prolog clause for constraint
        body(g, A, B, C).

%%% CHR Rules for p / 3

p_3(p(A, B, C), D, E, F) :-
        nonvar(D),                      % KillFlag set, constraint removes itself
        !.
% rule2 @ p(b,X,Y) <=> guard(b,X,Y,G) | body(b,X,Y,G,B).
p_3(p(b, A, B), C, D, E) ?-
        execute_guard(guard(b, A, B, F)),
        !,
        C = true,                       % set KillFlag
        body(b, A, B, F, G).            % execute body
% rule4 @ p(d,X,Y),q(d,Y,Z) <=> guard(d,X,Y,Z,G) | body(d,X,Y,Z,G,B).
p_3(p(d, A, B), C, D, E) ?-
        get_delayed_goals(B, F),        % get constraints delaying on B
        p_3_1(F, [B], [G], H),          % look for partner constraint
        execute_guard(guard(d, A, B, G, I)),
        !,
        C = true,
        body(d, A, B, G, I, J).
p_3(p(A, B, C), D, E, F) :-             % go for propagation CHRs
        p_3_0(p(A, B, C), D, E, F).

 p_3_1([q_3(q(d, A, B), C, D, E)|F], [A], [G], H) ?-   % found partner in list
        var(C),                         % KillFlag of partner has not been set
        [C, B, E] = [true, G, H].       % kill partner, return its arguments, id
 p_3_1([A|B], C, D, E) :-               % search for partner in constraints list
        p_3_1(B, C, D, E).
```

```
% rule1 @ p(a,X,Y) ==> guard(a,X,Y,G) | body(a,X,Y,G,B).
p_3_0(p(a, A, B), C, D, E) ?-
        var(C),                         % KillFlag has not been set
        check_and_mark_applied(p_3_0, D), % check if rule has been applied
                                        % before, if not, add info to list D
        execute_guard(guard(a, A, B, F)),
        !,
        p_3_2(p(a, A, B), C, D, E),     % try other CHRs
        body(a, A, B, F, G).
p_3_0(A, B, C, D) ?-                    % previous propagation CHR not applicable
        p_3_2(A, B, C, D).              % try other propagation CHRs

% rule5 @ p(e,X,Y)\q(e,Y,Z) <=> guard(e,X,Y,Z,G) | body(e,X,Y,Z,G,B).
p_3_2(p(e, A, B), C, D, E) ?-
        var(C),
        !,
        get_delayed_goals(B, F),        % get constraints delaying on B
        p_3_2_4(F, C, p(e, A, B), D, E). % look for partner constraints
p_3_2(p(A, B, C), D, E, F) :-           % previous propagation CHR not applicable
        p_3_2_5(p(A, B, C), D, E, F).    % try other propagation CHRs

p_3_2_4([q_3(q(e, A, B), C, D, E)|F], G, p(e, H, A), I, J) ?- % found partner
        var(C),                         % KillFlag of partner has not been set
        execute_guard(guard(e, H, A, B, K)),
        !,
        C = true,                       % kill partner
        p_3_2_4(F, G, p(e, H, A), I, J), % try to apply rule to other partners
        body(e, H, A, B, K, L).
p_3_2_4([A|B], C, D, E, F) :-  % search for partner in list of constraints
        p_3_2_4(B, C, D, E, F).
p_3_2_4([], A, B, C, D) :-     % all constraints tried, continue with next CHR
        p_3_2_5(B, A, C, D).

% rule3 @ p(c,X,Y),q(c,Y,Z) ==> guard(c,X,Y,Z,G) | body(c,X,Y,Z,G,B).
p_3_2_5(p(c, A, B), C, D, E) ?-
        var(C),
        !,
        get_delayed_goals(B, F),
        p_3_2_5_6(F, C, p(c, A, B), D, E).
p_3_2_5(p(A, B, C), D, E, F) :-
        p_3_2_5_7(p(A, B, C), D, E, F).

p_3_2_5_6([q_3(q(c, A, B), C, D, E)|F], G, p(c, H, A), I, J) ?-
        var(C),
        check_and_mark_applied(rule3, G, C, I, D), % check if rule has been
                            % applied before, if not, add info to lists I and D
        execute_guard(guard(c, H, A, B, K)),
        !,
        p_3_2_5_6(F, G, p(c, H, A), I, J),
        body(c, H, A, B, K, L).
```

```
p_3_2_5_6([A|B], C, D, E, F) :-
        p_3_2_5_6(B, C, D, E, F).
p_3_2_5_6([], A, B, C, D) :-
        p_3_2_5_7(B, A, C, D).

% last clause for redelaying the constraint
p_3_2_5_7(p(A, B, C), D, E, F) :-
        (
            var(D)                    % KillFlag still not set
        ->
            delay([D, A, B, C], p_3(p(A, B, C), D, E, F)) % delay constraint
        ;
            true
        ).


%%% Rules handling for q / 3

% Compiled for q/3 are rule3, rule4 and rule5
% Analogous to p/3 except for rule5

% rule5 @ p(e,X,Y)\q(e,Y,Z) <=> guard(e,X,Y,Z,G) | body(e,X,Y,Z,G,B).
q_3(q(e, A, B), C, D, E) ?-
        get_delayed_goals(A, F),
        q_3_10(F, [A], [G], H),
        execute_guard(guard(e, G, A, B, I)),
        !,
        C = true,
        body(e, G, A, B, I, J).
q_3(q(A, B, C), D, E, F) :-
        q_3_8(q(A, B, C), D, E, F).   % continue...

        q_3_10([p_3(p(e, A, B), C, D, E)|F], [B], [G], H) ?-
                var(C),
                [A, E] = [G, H].
        q_3_10([A|B], C, D, E) :-
                q_3_10(B, C, D, E).


% In the run-time system, built-in labeling is defined

labeling :-
        ( delayed_constraint(Constraint, KF),
          label_with(Constraint, Goal, Nb),
          !,
          KF = true,
          call(Goal),
          labeling
        ;
          true
        ).
```

Appendix 2 - First Implementation

Here we shortly present an abstracted Prolog code for the first - now obsolete - implementation of CHRs, a combination of a simple compiler and an interpreter written in ECL^{i}PS^e in summer 1991. There were no simpagation CHRs. First simplification and propagation CHRs are preprocessed as follows, distinguishing between single- and multi-headed rules:

```
Propagation Chrs
        Single-headed
                Head => Guard | Body
                chr(propag,Guard,Body)
        Multi-headed
                Head,Partner => Guard | Body
                chr(propag,CommonVar,Partner,Guard,Body)

Simplification Chrs
        Single-headed
                Head <=> Guard | Body
                chr(simplif,Guard,Body)
        Multi-headed
                Head,Partner <=> Guard | Body
                chr(simplif,CommonVar,Partner,Guard,Body)
```

For each user-defined constraint occurring as a head of a CHR, the following constraint goal is produced

```
constraint(ConstraintGoal,Schrs,Mchrs,Call,flags(Fired,Multi,Choice))
```

where `Schrs` is the list of single-headed rules, and `Mchrs` the list of multi-headed rules in the `chr` format as given above.

A constraint goal is activated if a variable in it or one of the flags `Fired, Multi, Choice` gets bound.

```
% Fired flag got bound

constraint(Goal,Schrs,Mchrs,Call,flags(Fired,Multi,Choice)):-
        nonvar(Fired),
        !.

% Choice flag got bound

constraint(Goal,Schrs,Mchrs,Call,flags(Fired,Multi,Choice)):-
        nonvar(Choice),
        !,
        (label_with_ok(Call) ->
                Fired=fired,
                call(Call)
        ;
```

```
                true
        ),
        constraint(Goal,Schrs,Mchrs,Call,
                        flags(Fired,Multi,Choice1))).

% Variable in constraint got bound

constraint(Goal,Schrs,Mchrs,Call,flags(Fired,Multi,Choice)):-
        got_bound(Goal),
        !,
        do_single(Schrs,Fired,Schrs1),
        constraint(Goal,Fired,Schrs1,Mchrs,Call,
                                flags(Fired,Multi,Choice)).

 do_single(Schrs,Fired,Schrs1):- nonvar(Fired),!,
        Schrs1=[].
 do_single([],Fired,Schrs1):-
        Schrs1=[].
 do_single([Schr|Schrs],Fired,Schrs1):-
        Schr=chr(Kind,Guard,Body),
        evaluate(Guard,Result),
        (Result=success ->
                        (Kind=simplif ->
                                Fired=fired
                        ;
                                true
                        ),
                        Schrs1=Schrsb,
                        call(Body)
        ;Result=suspend ->
                        Schrs1=[Schr|Schrs2])
        ;Result=failure ->
                        Schrs1=Schrs2
        ),
        do_single(Schrs,Fired,Schrs2).

% Multi flag got bound

constraint(Goal,Schrs,Mchrs,Call,flags(Fired,Multi,Choice)):-
        nonvar(Multi),
        !,
        do_multi(Mchrs,Fired,Multi,Mchrs1),
        constraint(Goal,Schrs,Mchrs1,Call,
                                flags(Fired,Multi1,Choice)).

 do_multi(Mchrs,Fired,Multi,Mchrs1):-
        nonvar(Fired),
        !,
        Mchrs1=[].
 do_multi([],Fired,Multi,Mchrs1):-
        Mchrs1=[].
```

```
do_multi([Mchr|Mchrs],Fired,Multi,Mchrs1):-
        Mchr=chr(Kind,Var,Partner,Guard,Body),
        copy_term(Mchr,MchrCopy),
        delayed_constraints(Var,Constraints),
        find_goal(Partner,FiredPartner,Constraints),
        evaluate(Guard,Result),
        (Result=success ->
                        Multi=multi(fired),
                        (Kind=simplif ->
                                Fired=fired,FiredPartner=fired,
                                Mchrs1=Mchrs2
                        ;Kind=propag ->
                                MchrCopy=chr(Kind,Var,PartnerC,GuardC,BodyC),
                                GuardC1=(PartnerC=\=Partner,GuardC),
                                Mchrs1=
                                  [chr(Kind,Var,PartnerC,GuardC1,BodyC)|Mchrs2]
                        ),
                        call(Body)
        ;
                        Mchrs1=[Mchr|Mchrs2]
        ),
        do_multi(Mchrs,Fired,Multi,Mchrs2).
```

In the interpreter, first all single-headed CHRs are executed, then all mutli-headed rules and last the built-in labeling routine. This is achieved by a goal for `schedule/0` that is added to the end of each query and that activates constraint goals to reduce with multi-headed rules or by built-in labeling by setting the appropriate flags.

```
% Scheduling CHRs and Built-In Labeling (Making Choices)
% ?- Query,schedule.

schedule:- wake_multi,make_choice.

% Activate multi_headed Chrs

wake_multi:-
        delayed_constraints(Constraints),
        wake_multi(Signal,Constraints).    % activate a constraint to reduce
                                           % with multi-headed rules

wake_multi(Signal,Constraints):-
        get_candidate(flags(Fired,Multi,Choice),Constraints,Constraints1),
        var(Fired),          % constraint not killed yet
        var(Multi),          % multi-headed rules not applied yet
        !,
        Multi=multi(Signal), % activate constraint for multi-headed rules
        wake_multi(Signal,Constraints1).      % look for more constraints
wake_multi(Signal,_Constraints):-             % no more constraints found
        (var(Signal) -> true ; wake_multi).   % restart if a rule fired
```

```
% Make a choice
% analoguous to wake_multi/0

make_choice:-
        delayed_constraints(Constraints),
        make_choice(Constraints).

make_choice(Constraints):-
        get_candidate(flags(Fired,Multi,Choice),Constraints,Constraints1),
        var(Fired),
        var(Choice),
        !,
        Choice=choice,
        (var(Fired) -> make_choice(Constraints1) ; schedule). % If constraint
        % not killed, find other constraint to label, else restart schedule
make_choice(_Constraints).      % no more constraints for labeling found
```

Appendix 3 - Example

In this appendix we show the result of applying the translations to guarded rules proposed in section 3 to three CHRs taken from a solver for inequalities (`minmax`). The translation may differ in minor, unessential details from the one proposed in the main body of the paper. All code is written in ECL^{i}PS^e using the CHRs library.

```
handler trchr.                          % declare name of constraint handler

% original set of sample CHRs for inequalities ------------------------------

constraints lt/2,le/2.                  % declare constraints

lt(X,Y),le(Y,X) <=> writeln(fail) | fail.
lt(X,Y)\le(X,Y) <=> writeln(true) | true.
lt(X,Y),le(Y,Z) ==> writeln(trans) | lt(X,Z).

% a test query
:- subcall((lt(A,B),le(B,C),le(A,C),(true;le(C,A))),R),writeln(R),fail ; true.


% CHRs embedded in propagation rules -----------------------------------------
% KillFlag introduced

lt(A,B):- lt(A,B,_).
le(A,B):- le(A,B,_).

constraints lt/3,le/3.

% Head1,Head2 <=> Guard | Body.
lt(X,Y,KF1),le(Y,X,KF2) ==>             % Kill flags not set so far
        var(KF1),var(KF2),
        writeln(fail)
        |
        dead=(KF1),dead=(KF2),          % Bind kill flags to kill head constraints
        fail.

% Head1\Head2 <=> Guard | Body.
lt(X,Y,KF1),le(X,Y,KF2) ==>
        var(KF1),var(KF2),
        writeln(true)
        |
        dead=(KF2),                     % Kill second head constraint only
        true.

% Head1,Head2 ==> Guard | Body.
lt(X,Y,KF1),le(Y,Z,KF2) ==>
        var(KF1),var(KF2),
        writeln(trans)
        |
        lt(X,Z,KF3).


% CHRs embedded in simplification rules --------------------------------------
% PropagationList introduced

lt(A,B):- lt(A,B,[]).
le(A,B):- le(A,B,[]).

constraints lt/3,le/3.
```

```
lt(X,Y,PL1),le(Y,X,PL2) <=>
        writeln(fail)
        |
        fail.

% Head1\Head2 <=> Guard | Body.
lt(X,Y,PL1),le(X,Y,PL2) <=>
        writeln(true)
        |
        true,
        lt(X,Y,PL1).

% Head1,Head2 ==> Guard | Body.
lt(X,Y,PL1),le(Y,Z,PL2) <=>
        not_member(trans-le(Y,Z)-2,PL1), % rule n with second head Head2 applied ?
        not_member(trans-lt(X,Y)-1,PL2), % rule n with first head Head1 applied ?
        writeln(trans)
        |
        lt(X,Z,[]),
        lt(X,Y,[trans-le(Y,Z)-2|PL1]),
        le(Y,Z,[trans-lt(X,Y)-1|PL2]).

    not_member(E,[]) ?- true.
    not_member(E,[E1|L]) ?- not (E==E1), not_member(E,L).


% CHRs as guarded rules with search by backtracking in guard -----------------
% delayed_constraint/2 introduced

lt(A,B):- lt(A,B,[]).
le(A,B):- le(A,B,[]).

constraints lt/3,le/3.

lt(X,Y,PL1) <=>
delayed_constraint(le(Y,X,PL2),KF),
        writeln(fail)
        |
        dead=KF,
        fail.

le(Y,X,PL2) <=>
delayed_constraint(lt(X,Y,PL1),KF),
        writeln(fail)
        |
        dead=KF,
        fail.

% Head1\Head2 <=> Guard | Body.
lt(X,Y,PL1) <=>
delayed_constraint(le(X,Y,PL2),KF),
        writeln(true)
        |
        dead=KF,
        true,
lt(X,Y,PL1).

le(X,Y,PL2) <=>
delayed_constraint(lt(X,Y,PL1),_KF),
        writeln(true)
        |
        true.

% Head1,Head2 ==> Guard | Body.
lt(X,Y,PL1) <=>
```

```
delayed_constraint(le(Y,Z,PL2),_KF),
        not_member(trans-le(Y,Z)-2,PL1),
        not_member(trans-lt(X,Y)-1,PL2),
        writeln(trans)
        |
        lt(X,Z,[]),
        lt(X,Y,[trans-le(Y,Z)-2|PL1]).

le(Y,Z,PL2) <=>
delayed_constraint(lt(X,Y,PL1),_KF),
        not_member(trans-le(Y,Z)-2,PL1),
        not_member(trans-lt(X,Y)-1,PL2),
        writeln(trans)
        |
        lt(X,Z,[]),
        le(Y,Z,[trans-lt(X,Y)-1|PL2]).


    not_member(E,[]) ?- true.
    not_member(E,[E1|L]) ?- not (E==E1), not_member(E,L).

    delayed_constraint(Constraint, KF) :-
        delayed_goals(DG),
        member(C, DG),
        C =.. [_Pred, Constraint, KF, _PA, _Nb].



% CHRs as guarded rules with explicit search for partner constraint -----------
% delayed_constraints/1, try_each_partner/4, try_one_partner/4 introduced

option(check_guard_bindings, off).               % needed for nested guards

lt(A,B):- lt(A,B,[]).
le(A,B):- le(A,B,[]).

constraints lt/3,le/3.

fail @ lt(X,Y,PL1) <=>
delayed_constraints(List),
try_each_partner(fail,lt(X,Y,PL1),List,le(Y,X,PL2)-KF),nonvar(PL2)
        |
dead=KF,
        fail.

fail @ le(Y,X,PL2) <=>
delayed_constraints(List),
try_each_partner(fail,le(Y,X,PL2),List,lt(X,Y,PL1)-KF),nonvar(PL1)
        |
dead=KF,
        fail.

% Head1\Head2 <=> Guard | Body.
true @ lt(X,Y,PL1) <=>
delayed_constraints(List),
try_each_partner(true,lt(X,Y,PL1),List,le(X,Y,PL2)-KF),nonvar(PL2)
        |
dead=KF,
        true,
lt(X,Y,PL1).

true @ le(X,Y,PL2) <=>
delayed_constraints(List),
try_each_partner(true,le(X,Y,PL2),List,lt(X,Y,PL1)-KF),nonvar(PL1)
        |
```

```
        true.

% Head1,Head2 ==> Guard | Body.
trans @ lt(X,Y,PL1) <=>
delayed_constraints(List),
try_each_partner(trans1,lt(X,Y,PL1),List,le(Y,Z,PL2)-KF),nonvar(PL2)
        |
        lt(X,Z,[]),
lt(X,Y,[trans-le(Y,Z)-2|PL1]).

trans @ le(Y,Z,PL2) <=>
delayed_constraints(List),
try_each_partner(trans2,le(Y,Z,PL2),List,lt(X,Y,PL1)-KF),nonvar(PL1)
        |
        lt(X,Z,[]),
le(Y,Z,[trans-lt(X,Y)-1|PL2]).


    not_member(E,[]) ?- true.
    not_member(E,[E1|L]) ?- not (E==E1), not_member(E,L).

    delayed_constraints(List) :-
delayed_goals(DG),
        delayed_constraints(DG,List).

delayed_constraints([],[]).
    delayed_constraints([C|DG],[Constraint-KF|List]) :-
C =.. [_Pred, Constraint, KF, _PA, _Nb],
!,
    delayed_constraints(DG,List).
    delayed_constraints([C|DG],List) :-
    delayed_constraints(DG,List).


constraints try_each_partner/4, try_one_partner/4.

   try_each_partner(N,Head1,[H|HL],Partner) <=>
        try_one_partner(N,Head1,H,Partner),        % try next candidate
        try_each_partner(N,Head1,HL,Partner).
   try_each_partner(N,Head1,[],Partner) <=> true.  % all candidates tried

isfree(le(_,_,PL)-_KF) ?- var(PL).
isfree(lt(_,_,PL)-_KF) ?- var(PL).

   try_one_partner(N,Head1,Head2,Partner) <=>
        not isfree(Partner) | true.         % partner already found

   try_one_partner(fail,lt(X,Y,PL1),le(Y,X,PL2)-KF,Partner) <=> isfree(Partner),
        writeln(fail)
        |
        Partner=le(Y,X,PL2)-KF.             % return partner constraint found
   try_one_partner(fail,lt(X,Y,PL1),H-KF,Partner) <=>
        not (                               % H was not the appropriate partner
        H==le(Y,X,PL2),
        writeln(fail))
        |
        true.
   try_one_partner(fail,le(X,Y,PL1),lt(Y,X,PL2)-KF,Partner) <=> isfree(Partner),
        writeln(fail)
        |
        Partner=lt(Y,X,PL2)-KF.
   try_one_partner(fail,le(X,Y,PL1),H-KF,Partner) <=>
        not (
        H==lt(Y,X,PL2),
        writeln(fail))
```

```
        |
        true.

    try_one_partner(true,lt(X,Y,PL1),le(X,Y,PL2)-KF,Partner) <=> isfree(Partner),
        writeln(true)
        |
        Partner=le(X,Y,PL2)-KF.
    try_one_partner(true,lt(X,Y,PL1),H-KF,Partner) <=>
        not (
        H==le(X,Y,PL2),
        writeln(true))
        |
        true.
    try_one_partner(true,le(X,Y,PL1),lt(X,Y,PL2)-KF,Partner) <=> isfree(Partner),
        writeln(true)
        |
        Partner=lt(X,Y,PL2)-KF.
    try_one_partner(true,le(X,Y,PL1),H-KF,Partner) <=>
        not (
        H==lt(X,Y,PL2),
        writeln(true))
        |
        true.

    try_one_partner(trans1,lt(X,Y,PL1),le(Y,Z,PL2)-KF,Partner) <=> isfree(Partner),
        not_member(trans-le(Y,Z)-2,PL1),
        not_member(trans-lt(X,Y)-1,PL2),
        writeln(trans)
        |
        Partner=le(Y,Z,PL2)-KF.
    try_one_partner(trans1,lt(X,Y,PL1),H-KF,Partner) <=>
        not (
        not_member(trans-le(Y,Z)-2,PL1),
        not_member(trans-lt(X,Y)-1,PL2),
        H==le(Y,Z,PL2),
        writeln(trans))
        |
        true.
    try_one_partner(trans2,le(Y,Z,PL2),lt(X,Y,PL1)-KF,Partner) <=> isfree(Partner),
        not_member(trans-le(Y,Z)-2,PL1),
        not_member(trans-lt(X,Y)-1,PL2),
        writeln(trans)
        |
        Partner=lt(X,Y,PL1)-KF.
    try_one_partner(trans2,le(Y,Z,PL2),H-KF,Partner) <=>
        not (
        not_member(trans-le(Y,Z)-2,PL1),
        not_member(trans-lt(X,Y)-1,PL2),
        H==lt(X,Y,PL1),
        writeln(trans))
        |
        true.


% Propagation CHRs as conditionals ----------------------------------------

    % Simple Conditional

    % does not provide for local variables
    constraints ifthen/2.

    ifthen(Condition,Consequence) <=> call(Condition) | call(Consequence).

    % does provide for local variables
    constraints ifthen/3.
```

```
    ifthen(GlobalVars,Condition,Consequence) <=>
copy_term(GlobalVars-Condition,GlobalVars-Condition1), % new local vars
call(Condition1)
|
Condition=Condition1, % unify old and new local variables
call(Consequence).


constraints lt/2,le/2.
constraints lt1/2,le1/2.         % internal names

lt1(X,Y) <=>
delayed_constraint(le1(Y,X),KF),
        writeln(fail)
        |
dead=KF,
        fail.

le1(Y,X) <=>
delayed_constraint(lt1(X,Y),KF),
        writeln(fail)
        |
dead=KF,
        fail.

% Head1\Head2 <=> Guard | Body.
lt1(X,Y) <=>
delayed_constraint(le1(X,Y),KF),
        writeln(true)
        |
dead=KF,
        true,
lt1(X,Y).

le1(X,Y) <=>
delayed_constraint(lt1(X,Y),_KF),
        writeln(true)
        |
        true.

% Head1,Head2 ==> Guard | Body.
lt(A,B) <=>
lt1(A,B),
ifthen(
 lt(A,B),
 (
  lt(A,B)=lt(X,Y),
  delayed_constraint(le1(Y,Z),_KF),
          writeln(trans)
         ),
          lt(A,Z)
).

le(A,B) <=>
le1(A,B),
ifthen(
 le(A,B),
 (
          le(A,B)=le(Y,Z),
  delayed_constraint(lt1(X,Y),_KF),
          writeln(trans)
         ),
          lt(X,Z)
).
```

```
    delayed_constraint(Constraint, KF) :-
delayed_goals(DG),
member(C, DG),
C =.. [_Pred, Constraint, KF, _PA, _Nb].


% Built-In Labeling ----------------------------------------------------------

constraints labeling/0.

% label_with le(X,Y) if writeln(label).
% le(A,B):- A=B ; lt(A,B).

labeling, le1(X,Y) <=> writeln(label) | le(X,Y)=le(A,B), (A=B ; lt(A,B)), labeling.


% End of handler trchr ========================================================
```