Constraint Handling Rules*

Thom Frühwirth

ECRC, Arabellastrasse 17, D-81925 Munich, Germany
email: thom@ecrc.de

Abstract. We are investigating the use of a class of logical formulas to define constraint theories and implement constraint solvers at the same time. The representation of constraint evaluation in a declarative formalism greatly facilitates the prototyping, extension, specialization and combination of constraint solvers. In our approach, constraint evaluation is specified using multi-headed guarded clauses called *constraint handling rules* (CHRs). CHRs define determinate conditional rewrite systems that express how conjunctions of constraints propagate and simplify.
In this paper we concentrate on CHRs as an extension for constraint logic programming languages. Into such languages, the CHRs can be tightly integrated. They can make use of any hard-wired solvers already built into the host language. Program clauses can be used to specify the non-deterministic behavior of constraints, i.e. to introduce search by constraints. In this way our approach merges the advantages of constraints (eager simplification by CHRs) and predicates (lazy choices by clauses).

1 Introduction

The advent of constraints in logic programming is one of the rare cases where both theoretical and practical aspects of a programming language have been improved. *Constraint logic programming* [JaLa87, VH89, VH91, F*92, JaMa94] combines the advantages of logic programming and constraint handling. In logic programming, problems are stated in a declarative way using rules to define relations (predicates). Problems are solved by the built-in logic programming engine (LPE) using chronological backtrack search. In constraint solving, efficient special-purpose algorithms are employed to solve sub-problems involving distinguished relations referred to as constraints.

Constraint logic programming (CLP) can be characterized by the interaction of a logic programming engine (LPE) with a constraint solver (CS). During program execution, the LPE incrementally sends constraints to the CS. The CS tries to solve the constraints. In the LPE the results from the CS cause *a priori* pruning of branches in the search tree spawned by the program. Unsatisfiability of the constraints means failure of the current branch, and thus reduces the number of possible branches, i.e. choices, to be explored via backtracking.

A practical problem remains: Constraint solving is usually 'hard-wired' in a built-in constraint solver written in a low-level language. While efficient, this approach

* Part of this work is supported by ESPRIT Project 5291 CHIC. This paper is a revised version of the technical report [Fru92].

makes it hard to modify a CS or build a CS over a new domain, let alone reason about it. As the behavior of the CS can neither be inspected by the user nor explained by the computer, debugging of real life constraint logic programs is hard. It has been demanded for a long time that "constraint solvers must be completely changeable by users" (p. 276 in [CAL88]). The lack of declarativeness and flexibility becomes a major obstacle if one wants to

- build a new CS,
- extend the CS with new constraints,
- specialize the CS for a particular application,
- combine constraint solvers.

Our proposal to overcome this problem is a high-level language especially designed for writing constraint solvers, called *constraint handling rules* (CHRs) [Fru92, Fru93a, Fru93b, Fru94, B*94, FrHa95]. With CHRs, one can introduce *user-defined* constraints into a given high-level host language. In this extended abstract the host language is Prolog, a CLP language with equality over Herbrand terms as the only built-in constraint. We claim that using our logic based language allows for reasoning about, inspection and modification of a CS.

CHRs define *simplification* of and *propagation* over user-defined constraints. Simplification replaces constraints by simpler constraints while preserving logical equivalence, 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.
```

When repeatedly applied by a constraint handling engine (CHE) the constraints are incrementally solved as in a CS, e.g.

`A>B,B>C,C>A` results in `false.`

CHIP was the first CLP language to introduce constructs (demons, forward rules, conditionals) [VH89] for user-defined *constraint handling* (like constraint solving, simplification, propagation). These various constructs have been generalized into CHRs. CHRs are based on guarded rules, as can be found in concurrent logic programming languages [Sha89], in the Swedish branch of the Andorra family [HaJa90], Saraswats cc-framework of concurrent constraint programming [Sar93], and in the 'Guarded Rules' of [Smo91]. However all these languages (except CHIP) lack features essential to define non-trivial constraint handling, namely for handling conjunctions of constraints and defining constraint propagation. CHRs provide these two features using multi-headed rules and propagation rules.

In the next section, we introduce constraint handling rules by example. Then we give the syntax, semantics and describe an implementation of CHRs. In section 4, we give extensive examples of the use of CHRs for writing constraint solvers. Last but not least we discuss related work in more detail.

2 CHRs by Example

We define a user-defined constraint for less-than-or-equal, =<. In Prolog, the built-in *predicate* =< can only be evaluated if the arguments are known, while the user-defined *constraint* =< will also handle variable arguments.

```
   % Constraint Declaration
(1a) constraints =</2.
(1b) label_with X=<Y if ground(X).
(1b) label_with X=<Y if ground(Y).

   % Constraint Labeling
(2a) X=<Y :- leq(X,Y).
(2b) leq(0,Y).
(2c) leq(s(X),s(Y)) :- leq(X,Y).

   % Constraint Handling
(3a) X=<Y <=> X=Y | true. % reflexivity
(3b) X=<Y,Y=<X <=> X=Y. % identity
(3c) X=<Y,Y=<Z ==> X=<Z. % transitivity
```

The CHRs of (3) specify how =< simplifies and propagates as a constraint. They implement reflexivity, identity and transitivity in a straightforward way. CHR (3a) states that `X=<X` is logically true. Hence, whenever we see the constraint `X=<X` we can simplify it to `true`. Similarly, CHR (3b) means that if we find `X=<Y` as well as `X=<Y` in the current constraint, we can replace it by the logically equivalent `X=Y`. CHRs (3a) and (3b) are called *simplification* CHRs. CHR (3a) detects satisfiability of a constraint, and CHR (3b) solves a conjunction of constraints returning an equality constraint. CHR (3c) states that the conjunction `X=<Y,Y=<Z` implies `X=<Z`. Operationally, we add logical consequences as a redundant constraint. This kind of CHR is called *propagation* CHR.

Redundancy produced by propagation CHRs is useful, as the following example shows. Given the query `A=<B,C=<A,B=<C`. The first two constraints cause CHR (3c) to fire and add `C=<B` to the constraint goal. This new constraint together with `B=<C` matches the head of CHR (3b). So the two constraints are replaced by `B=C`. The equality is applied to the rest of the constraint goal, `A=<B,C=<A`, resulting in `A=<B,B=<A` where `B=C`. CHR (3b) applies, resulting in `A=B`. The constraint goal contains no more inequalities, the simplification stops. The constraint solver we built has solved `A=<B,C=<A,B=<C` and produced the answer `A=B,B=C`:

```
   :-  A=<B,C=<A,B=<C.
% C=<A,A=<B propagates C=<B by 3c.
% C=<B,B=<C simplifies to B=C by 3b.
% B=<A,A=<B simplifies to A=B by 3b.
      A=B,B=C.
```

Note that CHRs (3b) and (3c) have multiple head atoms, a feature that is essential

in solving conjunctions of constraints. With single-headed CHRs alone, unsatisfiability of a conjunction of constraints (e.g. `A<B,B<A`) could never be detected and global constraint satisfaction (e.g. `A=<B,C=<A,C=<B` reduces to `A=B,A=C`) could not be achieved.

If no simplification and propagation is possible anymore, a constraint is chosen for automatic labeling. The labeling declaration (1b) and (1c) state that we may label using `X=<Y` if either `X` or `Y` are ground. Labeling is performed by using the CLP clauses of the constraint as labeling routine. In clause (2a), labeling using `=<` relies on a predicate `leq` which is defined by the two CLP clauses (2b) and (2c). For example, the query `4=<A,A=<3` propagates `4=<3` by CHR (3c). Then no more simplification is possible. `4=<3` is a constraint available for labeling. Executing its labeling routine produces a failure and so we know that `4=<A,A=<3` is unsatisfiable. A similar example is:

```
   :-  s(s(0))=<A,A=<s(s(s(0))).
% s(s(0))=<A,A=<s(s(s(0))) propagates s(s(0))=<s(s(s(0))).
% Labeling using s(s(0))=<s(s(s(0))) succeeds.
% Labeling using s(s(0))=<A succeeds with A=s(s(X)).
% Labeling using A=<s(s(s(0))) succeeds with X=0.
      A=s(s(0)).
% On backtracking A=<s(s(s(0))) succeeds with X=s(0).
      A=s(s(s(0))).
% On backtracking A=<s(s(s(0))) fails.
      false.
```

When CHRs are integrated into a logic programming language, we can regard any predicate as a labeling routine of a constraint and add some CHRs for it. Seen this way, CHRs are lemmas that allow us to express the determinate information contained in a predicate. As a result, predicates and constraints are just alternate views. CHRs define "shortcuts" which allow us to arrive at an answer without backtracking and quicker than by executing the predicate. To see the power of such lemmas consider

```
append(X,[],L) <=> X=L,list(L).
```

A recursion on the list `X` in the usual definition of `append` is replaced by a simple unification `X=L` and a type check `list(L)`.

3 Syntax, Semantics and Implementation

In this paper we assume that constraint handling rules extend a given constraint logic programming language. The syntax and semantics given here reflect this choice. It should be stressed, however, that the host language for CHRs need not be a CLP language. Indeed, work has been done at DFKI in the context of LISP [Her93]. This section follows [FrHa95].

3.1 Syntax

A CLP+CH program is a finite set of clauses from the CLP language and from the language of CHRs. Clauses are built from atoms of the form $p(t_1, ...t_n)$ where p is a predicate symbol of arity n ($n \geq 0$) and $t_1, ...t_n$ is a n-tuple of terms. A term is a variable, e.g. X, or of the form $f(t_1, ...t_n)$ where f is a function symbol of arity n ($n \geq 0$) applied to a n-tuple of terms. Function symbols of arity 0 are also called constants. Predicate and function symbols start with lowercase letters while variables start with uppercase letters. Infix notation may be used for specific predicate symbols (e.g. $X = Y$) and functions symbols (e.g. $-X + Y$). There are two classes of distinguished atoms, built-in constraints and user-defined constraints. In most CLP languages there is a built-in constraint for syntactic equality over Herbrand terms, `=`, 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 clause* is of the form

H `:-` $B_1, \ldots B_n$. $(n \geq 0)$

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*. The empty body ($n = 0$) of a CLP clause may be denoted by the built-in constraint `true`. A *query* is a CLP clause without head.

There are two kinds of CHRs[2]. A *simplification* CHR is of the form

$H_1, \ldots H_i$ `<=>` $G_1, \ldots G_j$ | $B_1, \ldots B_k$.

A *propagation* CHR is of the form

$H_1, \ldots H_i$ `==>` $G_1, \ldots G_j$ | $B_1, \ldots B_k$.

A *labeling declaration* for a user-defined constraint H is of the form

label_with H if $G_1, \ldots G_j$

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 which neither are, nor depend on, user-defined constraints.

3.2 Declarative Semantics

Declaratively, CLP programs are interpreted as formulas in first order logic. Extending a CLP language with CHRs preserves its declarative semantics. A CLP+CH *program* P is a conjunction of universally quantified clauses. A *predicate definition for* p is the set of all clauses in a program with the same predicate p in the head.

A CLP clause is an implication

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

[2] A third, hybrid kind as well as options and more declarations are described in [B*94].

Since we assume that a predicate definition defines a predicate completely, we strengthen the above using Clark's completion. Let H_1 :- $B_{11}, \ldots B_{n1}, \ldots, H_s$:- $B_{1s}, \ldots B_{ns}$, $(1 \leq s)$ be the clauses of the predicate definition for p. Then their logical reading 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 different variables.

A simplification CHR is a logical equivalence provided the guard is true in the current context

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

A propagation CHR is an implication provided the guard is true

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

Procedurally, a CHR can fire if its guard allows it. A firing simplification CHR *replaces* the head constraint by the body, a firing propagation CHR *adds* the body to the head constraints.

3.3 Operational Semantics

The *operational semantics* of CLP+CH can be described by a transition system.

A *computation state* is a tuple

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

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

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 answer* for the query Gs, written $answer(Gs)$.

The built-in constraint solver (CS) works on built-in constraints in C_B and Gs, the user-defined CS on user-defined constraints in C_U and Gs using CHRs, and the logic programming engine (LPE) on goals in Gs and C_U using CLP clauses. 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 CS 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 $C_B \rightarrow (H =_{set} H') \wedge answer(G)$

The constraint handling engine (CHE) applies CHRs to user-defined constraints in Gs and C_U whenever all user-defined constraints needed in the multi-head are present and the guard is satisfied. A guard G is *satisfied* if its local execution does not involve user-defined constraints and the result $answer(G)$ is entailed (implied) by the built-in constraint store C_B. Equality is entailed between two terms if they match. 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 CHR H <=> $G \mid B$ 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 CHR H ==> $G \mid B$ and G is satisfied.

Unfold
$< \{H'\} \cup Gs, C_U, C_B > \longmapsto < Gs \cup B, C_U, \{H = H'\} \cup C_B >$
if $(H$:- $B) \in P$.

The logic programming engine (LPE) unfolds goals in Gs. To *unfold* an atomic goal H' means to look for a clause $H :- B$ 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. There can be CLP clauses for user-defined constraints. Thus they can be unfolded as well. This unfolding is called *(built-in) labeling*. Details can be found in [B*94].

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

Note that any constraint solver written with CHRs will be *determinate*, *incremental* and *concurrent*. By "determinate" we mean that the user-defined CS commits to every constraint simplification it makes. Otherwise we would not gain anything, as the CS would have to backtrack to undo choices like in a Prolog program. By "incremental" we mean that constraints can be added to the constraint store one at a time using the "introduce"-transition. Then CHRs may fire and simplify the user-defined constraint store. The rules can be applied concurrently, even using chaotic iteration (i.e. the same constraint can be simplified by different rules at the same time), because logically correct CHRs can only replace constraints by equivalent ones or add redundant constraints.

3.4 Implementation

The operational semantics are still far from the actual workings of an efficient implementation. At the moment, there exist two implementations, one prototype in LISP [Her93], and one fully developed compiler in a Prolog extension.

The compiler for CHRs together with a manual is available as a library of ECLiPSe [B*94], ECRC's advanced constraint logic programming platform, utilizing its delay-mechanism and built-in meta-predicates to create, inspect and manipulate delayed goals. All ECLiPSe documentation is available by anonymous ftp from ftp.ecrc.de, directory /pub/eclipse/doc. In such a sequential implementation, the transitions are tried in the textual order given before. To reflect the complexity of a program in the number of CHRs, at most two head constraints are allowed in a rule. A rule with more head constraints can be rewritten into several two-headed rules. This restriction also makes complexity for search of the head constraints of a CHR linear in the number of constraints on average (quadratic in the worst case) by using partitioning and indexing methods. Termination of a propagation CHR is achieved by never firing it a second time with the same pair of head constraints.

The CHRs library includes a debugger and a visual tracing toolkit as well as a full color demo using geometric constraints in a real-life application for wireless telecommunication. About 20 constraint solvers currently come with the release - for booleans, finite domains (similar to CHIP [VH89]), also 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 (similar to CLP(R) [J*92]) and rationals, for lists, sets, trees, terms and last but not least for terminological reasoning [FrHa95]. The average number of rules in a constraint solver is as low as 24. Typically it took only a few days to produce a reasonable prototype solver, since the usual formalisms to describe a constraint theory, i.e. inference rules, rewrite rules, sequents, first-order axioms, can be expressed as CHRs programs in a straightforward way. Thus one can directly express how constraints simplify and propagate without worrying about implementation details. Starting from this executable specification, the rules then can be refined and adapted to the specifics of the application.

On a wide range of solvers and examples, the run-time penalty for our declarative and high-level approach turned out to be a constant factor in comparison to dedicated built-in solvers (if available). Moreover, the slow-down is often within an order of magnitude. On some examples (e.g. those involving finite domains with the

element-constraint), our approach is faster, since we can exactly define the amount of constraint simplification and propagation that is needed. This means that 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.

4 Examples

4.1 Booleans

This example is taken from [F*92]. In the domain of boolean constraints, the behavior of an and-gate may be informally described by rules such as

- **If** one input is 0 **then** the output is 0,
- **If** the output is 1 **then** both inputs are 1.

We can define the and-gate with constraint handling rules as:

```
and(X,Y,Z) <=> X=0 | Z=0.
and(X,Y,Z) <=> Y=0 | Z=0.
and(X,Y,Z) <=> X=1 | Y=Z.
and(X,Y,Z) <=> Y=1 | X=Z.
and(X,Y,Z) <=> Z=1 | X=1,Y=1.
and(X,Y,Z1),and(X,Y,Z2) ==> Z1=Z2.
```

The first rule says that the constraint goal `and(X,Y,Z)`, when it is known that the first input argument `X` is 0, can be reduced to asserting that the output `Z` must be 0. Hence the query `and(X,Y,Z),X=0` will result in `X=0, Z=0`. The last rule says that if a goal contains both `and(X,Y,Z1)` and `and(X,Y,Z2)` then a consequence is that `Z1` and `Z2` must be the same.

Consider the following predicate from the well-known full-adder circuit:

```
add(I1,I2,I3,O1,O2):-
    xor(I1,I2,X1),
    and(I1,I2,A1),
    xor(X1,I3,O1),
    and(I3,X1,A2),
    or(A1,A2,O2).
```

The query `add(I1,I2,0,O1,1)` will produce `I1=1,I2=1,O1=0`. The computation proceeds as follows: Because `I3=0`, the output `A2` of the and-gate with input `I3` must be 0. As `O2=1` and `A2=0`, the other input `A1` of the or-gate must be 1. Because `A1` is also the output of an and-gate, its inputs `I1` and `I2` must be both 1. Hence the output `X1` of the first xor-gate must be 0, and therefore also the output `O1` of the second xor-gate must be 0.

4.2 Maximum

We extend our solver for the inequality =< with a user-defined constraint over numbers, max(X,Y,Z), which holds if Z is the maximum of X and Y.

```
label_with max(X,Y,Z) if ground(X),ground(Y).
max(X,Y,Y):- X=<Y.
max(X,Y,X):- Y=<X.

max(X,X,Z) <=> X=Z.
max(X,Y,X) <=> Y=<X.
max(X,Y,Y) <=> X=<Y.
max(X,Y,Z),X=<Y <=> Y=Z,X=<Y.
max(X,Y,Z),Y=<X <=> X=Z,Y=<X.
max(X,Y,Z) ==> X=<Z,Y=<Z. % invariant and approximation
max(X,Y,Z1),max(X,Y,Z2) ==> Z1=Z2. % functional dependency
```

In the query max(A,B,C), max(A,C,D), the first constraint propagates A=<C, B=<C. The constraints A=<C, max(A,C,D) are simplified into C=D, A=<C. The new constraint goal is max(A,B,C), B=<C, C=D, A=<C. At this point, no more application of CHRs is possible. There is also no constraint that could be labeled. Therefore the conditional answer to our query max(A,B,C), max(A,C,D) is max(A,B,C), B=<C, C=D, A=<C.

Let $\leq$ be a built-in constraint, i.e. there is a built-in CS for inequalities (the user-defined constraint =< is no longer needed). Then we can replace the CHR

```
max(X,Y,Z),X=<Y <=> Y=Z,X=<Y
```

by

```
max(X,Y,Z) <=> X≤Y | Y=Z.
```

As a consequence, the first CHR becomes obsolete, as the built-in constraint X≤Y in the guard naturally covers the case when X=Y. Contrast this with the *user-defined* constraint =< in the head of the original CHR that clearly cannot match =. Now max can be defined by CHRs as follows.

```
max(X,Y,Z) <=> X≤Y | Y=Z.
max(X,Y,Z) <=> Y≤X | X=Z.
max(X,Y,X) <=> Y≤X.
max(X,Y,Y) <=> X≤Y.
max(X,Y,Z) ==> X≤Z,Y≤Z.
max(X,Y,Z1),max(X,Y,Z2) ==> Z1=Z2.
```

However, the CS for max is not complete, i.e. there are satisfiable or (worse) unsatisfiable constraint goals which are neither simplifiable nor available for labeling.

For example, the query `max(X,7,9)` results in `max(X,7,9),X≤9`, but it is not reduced to `X=9`. In practice, a CS is often not complete for efficiency reasons [JaMa94]. If the application requires it, we can always add CHRs to cover the incomplete cases or modify the labeling declaration, while built-in constraint solvers cannot be as easily adopted. In our example, new CHRs of the form

```
max(X,Y,Z) <=> Y<Z | X=Z.
```

or an extended labeling declaration

```
label_with max(X,Y,Z) if ground(X),ground(Y).
label_with max(X,Y,Z) if ground(X),ground(Z).
label_with max(X,Y,Z) if ground(Y),ground(Z).
```

will help.

4.3 Temporal Time Point Constraints

In order to define a constraint solver for temporal constraints over time points we exploit the natural relationship of these constraints with ordering constraints in general. Therefore, we can start from the constraint solver for the less-than-or-equal constraint `=<`. We extend the inequality to the form `X+N=<Y`, where N is a given positive number, meaning that the distance in time of the two time points `X` and `Y` is at least `N`.

```
label_with XN =< Y if ground(XN),ground(Y).
XN=<Y :- XN ≤ Y.

X+N=<X <=> N=0.
X+N=<Y,X+M=<Y <=> NM is max(N,M) | X+NM=<Y.
X+N=<Y,Y+M=<X <=> N = 0, M = 0, X = Y.
X+N=<Y,Y+M=<Z ==> NM is N+M | X+NM=<Z.
```

In the labeling declaration the extension in syntax is reflected by requiring the first argument to be ground, such that `X+N` can be evaluated. The four CHRs are straightforward extensions of the ones for the simple inequality. Some auxiliary arithmetic computations with `is` are added to compute the distances for the resulting inequalities in the body. It is assumed that `is` delays if its right-hand side is not ground.

If we allow for negative `N` we can express maximal distances as well. The set of CHRs however will be non-terminating. There is no termination order, because there is no bound anymore on the minimal or maximal distances that could be computed. The termination problem is solved by introducing a new constraint `=<*` which stands for *derived* inequalities (resulting from simplification and propagation) as opposed to the initial ones written with `=<`.

```
label_with XN =< Y if ground(XN),ground(Y).
```

```
XN=<Y :- XN ≤ Y.

label_with XN =<* Y if ground(XN),ground(Y).
XN=<*Y :- XN ≤ Y.


X+N=<Y ==> X+N=<*Y.

X+N=<*X <=> N=<0.
X+N=<*Y,X+M=<*Y <=> NM is max(N,M) | X+NM=<*Y.
X+N=<*Y,Y+M=<*X <=> N=0,M=0 | X = Y.
X+N=<*Y,Y+M=< Z ==> NM is N+M | X+NM=<*Z.
```

The derived inequality constraint of course has the same labeling declaration and predicate specification as the original inequality. The original CHRs are turned into CHRs for the derived inequality. However, there is one exception, which is the crucial detail causing termination. In the last CHR performing transitive closure, one constraint must be not a derived but an original constraint. This also eliminates redundant inequalities that have been produced by the transitive closure before. To get the simplifications started, we have to give some initial derived constraints. This is done by the first CHR, which produces a derived inequality for each initial inequality.

In temporal reasoning applications, usually both minimal and maximal distance of two time points are given. Hence it is a good idea to merge the two constraints `X+N=<Y,Y+M=<X` (`N` positive and `M` negative) into a single constraint `N=<Y-X=<(-M)` (by abuse of the relational notation), where `Y` is the *starting point* and `X` is the *end point* of the interval `Y-X`. This is exactly the notation and meaning used in [DMP91].

```
label_with X =< Y =< Z if ground(X),ground(Y),ground(Y).
X =< Y =< Z:- X ≤ Y, Y ≤ Z.

label_with X =<* Y =<* Z if ground(X),ground(Y),ground(Y).
X =<* Y =<* Z:- X ≤ Y, Y ≤ Z.


A=<X-Y=<B ==> A=<*X-Y=<*B.

A=<*X-X=<*B <=> A=<0=<B.
A=<*X-Y=<*B <=> A=0,B=0 | X = Y.
A=<*X-Y=<*B,C=<*X-Y=<*D <=>
                   AC is max(A,C), BD is min(B,D) | AC=<*X-Y=<*BD.
A=<*X-Y=<*B,C=< Y-Z=< D ==> AC is A+C, BD is B+D | AC=<*X-Z=<*BD.
A=<*X-Y=<*B,C=< Z-Y=< D ==> AC is A-D, BD is B-C | AC=<*X-Z=<*BD.
```

Above, the CHRs have been extended correspondingly. The only interesting thing to note is that the last CHR about transitivity had to be split into two cases. The reason is that we rewrote `X+N=<Y, Y+M=<X` into `N=<Y-X=<(-M)` only, but not into

`M=<X-Y=<(-N)`, as the second formulation would have caused redundant computations for all CHRs except the one for transitivity.

The above CHRs will produce derived inequality constraints for every pair of time points (provided they are connected). Again this means redundant information and hence redundant computation, as we can compute all relations when knowing the distances from one given reference point to all other time points. We will specify the reference point `X` with a dummy constraint `start(X)`. For this optimization only the first CHR has to be restricted from

```
A=<X-Y=<B ==> A=<*X-Y=<*B.
```

to

```
A=<X-Y=<B,start(X) ==> A=<*X-Y=<*B.
```

The resulting set of CHRs defines and implements a specialized constraint solver for temporal constraints on time points. Its behaviour has been tailored to temporal constraints starting from inequality constraints. Further optimizations are possible, for example using a dynamic shortest-path algorithm. If further speed-up is needed, once the prototype has been established and "tuned" as required, it can be reworked in a low-level language. For more on temporal reasoning with constraints, see [Fru94].

5 Reasoning

When seen as logical formulae, the logical *correctness* of CHRs with respect to a constraint theory can be established by using techniques from automated theorem proving. It is also useful to view CHRs as conditional rewrite systems. In this way we can establish that they are *canonical*, i.e. terminating and confluent by adopting well-known techniques such as termination proofs and unfailing completion. If we can prove a set of CHRs both canonical and correct we can be sure that the CHRs indeed implement a "well-behaved" constraint solver.

Briefly, *termination* [Der87] is proved by giving an ordering on atoms showing that the body of a rule is always smaller than the head of the rule. Such an ordering in addition introduces an intuitive notion of a "simpler" constraint, so that we also support the intuition that constraints get indeed simplified. When combining constraint solvers that share constraints, nonterminating simplification steps may arise even if each solver is terminating. E.g. one solver defines less-than in terms of greater-than and the other defines greater-than in terms of less-than.

The notion of *confluence* [Kir89] is important for combining constraint solvers as well as for concurrent applications of CHRs. Concurrent CHRs are not applied in a fixed order. As correct CHRs are logical consequences of the program, any result of a simplification or propagation step will have the same meaning, however it is not guaranteed anymore that the result is syntactically the same. In particular, a solver may be complete with one order of applications but incomplete with another one. Syntactically different constraint evaluations may also arise if combined solvers share constraints, depending on which solver comes first.

A set of CHRs is *confluent*, if each possible order of applications starting from any constraint goal leads to the same resulting constraint goal. A set of CHRs is *locally confluent* if any two constraint goals resulting from one application of a CHR to the initial constraint goal can be simplified into the same constraint goal. It is well-known from rewrite systems that local confluence and termination imply confluence. Furthermore, in a confluent set of CHRs, any constraint goal has a unique normal form, provided it exists. This means that the answer to a query will always be the most simple one[3].

6 Related Work

6.1 Constraint Logic Programming Languages

In the constraint logic programming CHIP [VH89], the general technique of *propagation* is employed over finite domains. The idea is to prune large search trees by enforcing local consistency of built-in and user-defined constraints. These techniques are orthogonal to our approach and thus can be integrated. *Demons* are essentially single-headed simplification CHRs without guards. However, labeling routines for a constraint are not possible. One version of CHIP also included *forward rules* [Gr89], which correspond to CHRs without guards. In practice, demons and forward rules have been proven useful in CHIP applications in the boolean domain for circuit design and verification. Their potential to define constraint solvers in general was not realised, maybe because of their limitations. [Gr89] also gives a detailed account of the semantics of forward rules and therefore CHRs without guards. In this sense, CHRs can be seen as an extension of the work on demons and forward rules in CHIP.

6.2 Combined and Extended Languages

In the following we relate our approach to other work on combining deterministic and nondeterministic computations into one logic programming language.

Amalgamating pure Prolog with single headed simplification CHRs results in a language of the family $cc(\downarrow, \rightarrow, \Rightarrow)$[4] of the cc framework proposed by Saraswat [Sar89, Sar93]. A close study of [Sar89] reveals that he proposes a special *Tell* operation called "inform" that could be used to simulate propagation CHRs. CHRs naturally fit the ask-and-tell interpretation of constraint logic programming introduced by Saraswat and applied by [VH91]. The constraint goal is viewed as constraint store for user-defined constraints. They are matched by the heads of CHRs and the guards ask if certain constraints hold in the built-in constraint store.

Guarded Rules [Smo91] correspond to single headed simplification CHRs. However, they are only used as "shortcuts" (lemmas) for predicates, not as definitions for user-written constraints. There are only built-in constraints. Interestingly, Smolka defines the built-in constraint system as a terminating and determinate reduction system. Hence it could be implemented by simplification CHRs.

[3] It can, however, contain redundant constraints and introduce new variables.

[4] $\downarrow$ means *Ask* in addition *Tell* is supported, $\rightarrow$ is the commit operator for don't care nondeterminism used and $\Rightarrow$ is the commit operator for don't know nondeterminism able to describe pure Prolog.

The Andorra Model of D.H.D. Warren for parallel computation has inspired a rapid development of numerous languages and language schemes. The Andorra Kernel Language (AKL) [JaHa91] is a guarded language with built-in constraints based on an instance of the Kernel Andorra Prolog control framework. AKL combines don't care nondeterminism and don't know nondeterministism with the help of different guard operators. There are three kinds of guard operators, namely cut, commit and wait. In our approach, a logic programming language amalgamated with CHRs inherits the the commit operator of the CHRs as well as the guard operators of the host language (e.g. cut in the case of Prolog). Like most logic programming languages, AKL itself does not support two of the essential features for defining simplification of user-defined constraints: propagation rules and multiple head atoms.

6.3 Multiple Head Atoms

According to [Coh88] at the very beginning of the development of Prolog in the early 70's by Colmerauer and Kowalski, experiments were performed with clauses having multiple head atoms. More recently, clauses with multiple head atoms were proposed to model parallelism and distributed processing as well as objects, e.g. [AnPa90]. The similarity with CHRs is merely syntactical. Rules about distribution or objects cannot be regarded as specifying constraint handling. These rules are supposed to model the distribution and change of objects, while CHRs model equivalence and implication of constraints.

In committed choice languages, multiple head atoms have been considered only rarely. In his thesis, Saraswat remarks on multiple head atoms that "the notion seems to be very powerful" and that "extensive further investigations seems warranted" ([Sar89], p. 314). He motivates so-called *joint reductions* of multiple atoms as analogous to production rules of expert system languages like OPS5. The examples given suggest the use of joint reductions to model objects in a spirit similar to what is worked out in [AnPa90].

Multi-headed simplification CHRs are sufficient to simulate the parallel machine for multiset transformation proposed in [BCL88]. This machine is based on the chemical reaction metaphor as means to describe highly parallel computations for a wide spectrum of applications. Following [BCL88], we can implement the sieve of Eratosthenes to compute primes simply as:

```
primes(1) <=> true.
primes(N) <=> N>1  | M is N-1, prime(N),primes(M).
prime(I),prime(J) <=> 0 is J mod I  | prime(I). % J is multiple of I
```

The answer to the query `primes(`n`)` will be a conjunction of `prime(`p_i`)` where each p_i is a prime ($2 \leq p_i \leq n$).

7 Conclusions

Constraint handling rules (CHRs) are a language extension for writing user-defined constraints. Basically, CHRs are multi-headed guarded clauses. CHRs support rapid

prototyping of built-in constraint solvers by providing executable specifications and implementations. They support specialization, modification and combination of constraint solvers.

By amalgamating a logic programming language with CHRs, a flexible, extensible constraint logic programming language results. It merges the advantages of constraints (simplification via CHRs) and predicates (choices via definite clauses). The result is a tight integration of the logic programming component and user-defined constraint solvers. In this way, a logical reconstruction for constraint solving in logic programming is achieved.

CHRs have been implemented as a library of ECLiPSe, ECRC's constraint logic programming platform and as a prototype in LISP at DFKI, Germany. CHRs have been used to encode a wide range of constraint solvers, including new domains such as terminological and temporal reasoning. Although intended as a language for constraint simplification, CHRs could also serve as a powerful programming language on their own.

We believe that our approach has the potential to provide a comprehensive framework for constraints, because CHRs make it possible

- to add constraint solvers for any required domain of computation.
- to build and costumize constraint solvers for particular applications.
- to generate constraint solvers semi-automatically from constraint theories.
- to debug constraint systems.

Acknowledgements

Pascal Brisset implemented the CHRs library of ECliPSe. Thanks to Alex, Jesper, Mark, Thierry and Volker, my colleagues at ECRC, who discussed these ideas with me. Thanks to Francesca Rossi and Gert Smolka as well as anonymous referees, who commented in detail on this paper in its various technical report versions.

References

[AnPa90] Andreoli J.-M. and Pareschi R., Linear Objects: Logical Processes with Built-In Inheritance, Seventh Intl Conf on Logic Programming MIT Press 1990, pp. 495-510.

[B*94] P. Brisset et al., ECLiPSe 3.4 Extensions User Manual, ECRC Munich, Germany, July 1994.

[BCL88] Banatre J.-P., Coutant A. and Le Metayer D., A Parallel Machine for Multiset Transformation and its Programming Style, Future Generation Computer Systems 4:133-144, 1988.

[CAL88] Aiba A. et al, Constraint Logic Programming Language CAL, Int Conf on Fifth Generation Computer Systems, 1988, Ohmsha Publishers, Tokyo, pp. 263-276.

[Coh88] J. Cohen, A View of the Origins and Development of Prolog, CACM 31(1):26-36, Jan. 1988.

[DMP91] R. Dechter, I. Meiri and J. Pearl, Temporal Constraint Networks, Journal of Artificial Intelligence 49:61-95, 1991.

[Der87] N. Dershowitz, Termination of Rewriting, Journal of Symbolic Computation, 3(1+2):69-116, 1987.

[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,

[F*92] T. Frühwirth, A. Herold, V. Küchenhoff, T. Le Provost, P. Lim, E. Monfroy and M. Wallace. *Constraint Logic Programming - An Informal Introduction*, Chapter in Logic Programming in Action, Springer LNCS 636, September 1992. Also Technical Report ECRC-93-05, ECRC Munich, Germany, February 1993.

[Fru93a] T. Frühwirth, *Entailment Simplification and Constraint Constructors for User-Defined Constraints*, Workshop on Constraint Logic Programming, Marseille, France, March 1993.

[Fru93b] T. Frühwirth, *User-Defined Constraint Handling*, Abstract, ICLP 93, Budapest, Hungary, MIT Press, June 1993.

[Fru94] T. Frühwirth, *Temporal Reasoning with Constraint Handling Rules*, Technical Report ECRC-94-05, ECRC Munich, Germany, February 1994 (first published as CORE-93-08, January 1993), available by anonymous ftp from ftp.ecrc.de, directory pub/ECRC_tech_reports/reports, file ECRC-94-05.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, to appear. Revised version of Technical Report ECRC-94-06, ECRC Munich, Germany, February 1994, available by anonymous ftp from ftp.ecrc.de, directory pub/ECRC_tech_reports/reports, file ECRC-94-06.ps.Z.

[Gr89] T. Graf, Raisonnement sur les contraintes en programmation en logique, Ph.D. Thesis, Version of June 1989 Universite de Nice, France, September 1989 (in French).

[HaJa90] S. Haridi and S. Janson, Kernel Andorra Prolog and its Computation Model, Seventh International Conference on Logic Programming, MIT Press, 1990, pp. 31-46.

[Her93] Eine homogene Implementierungsebene fuer einen hybriden Wissensrepraesentationsformalismus, Master Thesis, in German, University of Kaiserslautern, Germany, April 1993.

[J*92] J. Jaffar et al., The CLP(R) Language and System, ACM Transactions on Programming Languages and Systems, Vol.14:3, July 1992, pp. 339-395.

[JaHa91] S. Janson and S. Haradi, Programming Paradigms of the Andorra Kernel Language, Draft of March 13, 1991, accepted at ILPS 91 in San Diego, Swedish Institute of Computer Science, Kista, Sweden.

[JaLa87] J. Jaffar and J.-L. Lassez, Constraint Logic Programming, ACM 14th POPL 87, Munich, Germany, January 1987, pp. 111-119.

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

[Kir89] C. Kirchner and H. Kirchner, Rewriting: Theory and Applications, Working paper for a D.E.A. lecture at the University of Nancy I, France, 1989.

[Sar89] V. A. Saraswat, Concurrent Constraint Programming Languages, Ph.D. Dissertation, Carnegie Mellon Univ., Draft of Jan. 1989.

[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.

[VH89] P. Van Hentenryck, Constraint satisfaction in Logic Programming, MIT Press, Cambridge, 1989.
[VH91] P. van Hentenryck, Constraint Logic Programming, The Knowledge Engineering Review, Vol 6:3, 1991, pp 151-194.

This article was processed using the LaTeX macro package with LLNCS style