Theory and Practice of Constraint Handling Rules

Thom Frühwirth

▷ Constraint Handling Rules (CHR) are our proposal to allow more flexibility and application-oriented customization of constraint systems. CHR are a declarative language extension especially designed for writing user-defined constraints. CHR are essentially a committed-choice language consisting of multi-headed guarded rules that rewrite constraints into simpler ones until they are solved.

In this broad survey we aim at covering all aspects of CHR as they currently present themselves. Going from theory to practice, we will define syntax and semantics for CHR, introduce an important decidable property, confluence, of CHR programs and define a tight integration of CHR with constraint logic programming languages. This survey then describes implementations of the language before we review several constraint solvers - both traditional and non-standard ones - written in the CHR language. Finally we introduce two innovative applications that benefited from using CHR. ◁

1. INTRODUCTION

The advent of constraints in logic programming (LP) is one of the rare cases where theoretical, practical and commercial aspects of a programming language have been improved simultaneously. *Constraint logic programming* [JaLa87, vH89, vH91, Fr*92, JaMa94, FrAb97] (CLP) combines the advantages of logic programming and constraint solving. 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 using chronological backtrack search to explore choices. In constraint solving, efficient special-purpose algorithms are employed to solve subproblems involving distinguished relations referred to as constraints. A constraint solver can thus be seen as inference system. The solver supports some if not all of

Address correspondence to Ludwig-Maximilians-Universität Muenchen (LMU), Institut fuer Informatik, Oettingenstrasse 67, D-80538 Munich, Germany, fruehwir@informatik.uni-muenchen.de, http://www.pst.informatik.uni-muenchen.de/personen/fruehwir/

655 Avenue of the Americas, New York, NY 10010 0743-1066/94/$7.00

the basic operations on constraints: solving (satisfaction), simplification, propagation, normalization, entailment (deciding implication) and optimization (computing "best" solutions).

In the beginning of CLP, constraint solving was "hard-wired" in a built-in constraint solver written in a low-level language. While efficient, this so-called "black-box" approach makes it hard to modify a solver or build a solver over a new domain, let alone debug, reason about and analyze it. This is a problem, since one lesson learned from practical applications is that constraints are often heterogeneous and application-specific.

Actually, it has been demanded from the beginning of CLP that "constraint solvers must be completely changeable by users" (p. 276 in [Ai*88]). By "user" we mean the application programmer. Since then, several proposals have been made to allow more for flexibility and costumization of constraint systems ("glass-box" or even "no-box" approaches):

- Demons, forward rules and conditionals, CHIP [Di*88, vH89], allow defining propagation of constraints in a limited way (Section 3).
- Constraint combinators, cc(FD) [vH91], allow building more complex constraints from simpler constraints (see also Section 8.1).
- Constraints connected to a Boolean variable, BNR-Prolog [BeOl92], "nested constraints" [Sid93], allow expressing any logical formula over primitive constraints.
- Indexicals, clp(FD) [CoDi96], allow implementing constraints over finite domains at a medium level of abstraction.
- Meta- and attributed variables [Hol92], allow attaching constraints to variables (Section 7).

It should be noted that all the approaches but the last can only extend a solver over a given, specific constraint domain, typically finite domains. Application-specific domains can only be implemented directly using the last approach, however this is tedious, a kind of "constraint assembler" programming, which is currently the low-level basis for most delay mechanisms and constraint solver extensions.

Our proposal is a high-level language extension especially designed for writing constraint solvers, called *constraint handling rules* (CHR) [Fru91, FrBr95a, Fru95, FrBr95b, FAM97]. With CHR, one can introduce *user-defined* constraints into a given host language, be it Prolog or Lisp. As *language extension*, CHR themselves are only concerned with constraints, all auxiliary computations are performed directly in the host language. CHR are typically a library containing a compiler and run-time system written in the host language and solvers written in CHR.

CHR are essentially a committed-choice language consisting of guarded rules that rewrite constraints into simpler ones until they are solved. CHR define both *simplification* of and *propagation* over user-defined constraints. Simplification replaces constraints by simpler constraints while preserving logical equivalence. Propagation adds new constraints which are logically redundant but may cause further simplification. CHR can be seen of generalization of the various CHIP constructs for user-defined constraints.

In contrast to the family of the general-purpose concurrent logic programming languages [Sha89], concurrent constraint languages [Sar93] and the ALPS framework [Mah87], CHR are a *special-purpose language* concerned with defining declarative constraints, not procedures in their generality. In another sense, CHR are more general, since they allow "multiple heads", i.e. conjunctions of constraints in the head of a rule. Multiple heads are a feature that is essential in solving conjunctions of constraints. With single-headed CHR alone, unsatisfiability of constraints could not always be detected (e.g `X<Y,Y<X`) and global constraint satisfaction could not be achieved.

Overview of the Survey Paper

In the next section, we introduce CHR by example. Then we talk about related work. On our way from theory to practice, we will first give syntax and semantics as well as soundness and completeness results for CHR. We will then introduce an important property for constraint solvers, confluence, and a decidable, necessary and sufficient test for it. We will next discuss the specifics of extending a CLP language with CHR (like automatic labeling). We will also describe the principles and characteristics of several existing implementations of CHR in Prolog and LISP.

CHR have been used to encode a wide range of constraint solvers, including new domains such as terminological and temporal reasoning. We will give an overview of several solvers, show how they can be extended or modified and we will briefly describe related work that builds on these solvers. Finally, we will mention two applications in non-standard domains, one optimizes the placement of radio cells for transmitters, the other gives rent advice over the internet.

2. CHR BY EXAMPLE

We define a user-defined constraint for less-than-or-equal, `=<`, that can handle variable arguments. The implementation will rely on syntactical equality, `=`, which is assumed to be a predefined (built-in) constraint.

```
reflexivity @ X=<Y <=> X=Y | true.
antisymmetry @ X=<Y,Y=<X <=> X=Y.
transitivity @ X=<Y,Y=<Z ==> X=<Z.
```

The CHR specify how `=<` simplifies and propagates as a constraint. They implement reflexivity, antisymmetry and transitivity in a straightforward way. CHR `reflexivity` states that `X=<Y` is logically true, provided it is the case that `X=Y`. This test forms the (optional) guard of a rule, a precondition on the applicability of the rule. Hence, whenever we see the constraint `X=<X` we can simplify it to `true`. CHR `antisymmetry` means that if we find `X=<Y` as well as `Y=<X` in the current constraint, we can replace it by the logically equivalent `X=Y`. Note the different use of `X=Y` in the two rules: In the `reflexivity` rule the equality is a precondition (test) on the rule, while in the `antisymmetry` rule it is enforced when the rule fires.

The rules `reflexivity` and `antisymmetry` are *simplification CHR*. The rule `transitivity` propagates constraints. It 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 from propagation CHR is useful, as the query `A=<B,C=<A,B=<C` shows: The first two constraints cause CHR `transitivity` to fire and add `C=<B` to the query. This new constraint together with `B=<C` matches the head of CHR `antisymmetry`, `X=<Y,Y=<X`. So the two constraints are replaced by `B=C`. In general, matching takes into account the syntactical equalities that are implied by built-in constraints. Therefore, since the built-in constraint `B=C` was added, CHR `antisymmetry` applies to the constraints `A=<B,C=<A`, resulting in `A=B`. The query 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 transitivity.
% C=<B,B=<C simplifies to B=C by antisymmetry.
% A=<B,C=<A simplifies to A=B by antisymmetry since B=C.
     A=B,B=C.
```

Note that multiple heads of rules are essential in solving these constraints. Also note that this solver implements a (partial) order constraint over any constraint domain, this generality is only possible with CHR.

For the solver to work, we require conjunctions of constraints to be idempotent, so that multiple occurrences of the same constraint are absorbed. This ensures termination of the solver, since given a finite number of variables, there can only be a finite number of different `=<` constraints between them. Then, the solver is confluent, this means that from a given query, the answer will always be the same, regardless of which order we apply the rules. E.g. in the above query we could have started with applying transitivity to `C=<A,B=<C`.

3. RELATED WORK

3.1. Languages for Defining Constraints

CS-Prolog [KOM87] was presumably the first proposal to implement constraint solvers in a LP language itself utilizing a delay mechanism. Conditional rewrite rules were used to describe the behavior of the solver. However, it was years too early to be able to refine this idea and implement it efficiently.

CHIP was the first CLP language to introduce feasible constructs (demons, forward rules, conditionals) [Di*88, vH89] for user-defined constraints. These various constructs have been generalized into and made uniform by CHR. Demons are essentially single-headed simplification CHR without guards. One version of CHIP also included forward rules [Gr89], which correspond to CHR 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 realized, maybe because of their limitations.

The *Guarded Rules* [Smo93] correspond to single headed simplification CHR. However, they are only used as "shortcuts" (lemmata) for predicates, not as definitions for user-written constraints. Interestingly, Smolka defines the built-in constraint system as a terminating and determinate reduction system. Hence it could be implemented by simplification CHR.

We have already mentioned the other approaches towards user-defined constraints in LP in the introduction. There are also other languages outside of the LP paradigm, that aim at defining constraint systems.

The functional language Betrand [Lel88] uses augmented term rewriting, which is standard term rewriting extended by an equality theory, local variables, objects and types. Confluence is preserved. An extension to allow multiple solutions is also discussed, which would allow Betrand retaining the expressive power of LP. The extensions of Bertrand mimic what is already present in LP: the equality theory for unification of Herbrand terms and local variables.

The object-oriented language extension EQUATE [Wil91] simplifies arithmetic constraints into a sequence of procedural solution steps. EQUATE uses rewrite rules, which can be seen as LP rules. The procedural solutions use destructive assignment, thus an ordering has to be imposed on the solutions steps to avoid read-write conflicts. The approach has some capabilities to deal with added and removed constraints.

3.2. 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. 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 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].

Indeed, clauses with multiple head atoms were proposed in the literature to model parallelism and distributed processing as well as objects. The similarity with CHR is merely syntactical. Rules about distribution, objects or agents involve non-monotonicity, e.g. state changes caused by actions or method calls, as opposed to declarative constraint solving. However, CHR can be (ab)used to model objects or agents, e.g. a stack object equipped with a method `push`:

```
push(X), stack(S) <=> stack([X|S])
```

Multi-headed simplification CHR are sufficient to simulate the parallel machine for multiset transformation proposed in [BCL88]. This "chemical abstract machine" is based on the chemical reaction metaphor as a means to describe highly parallel computations. 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).% generate candidates
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 ($2 \leq p_i \leq n$). One should compare this to the standard concurrent program as given in [Sha89] to appreciate the expressive power of multiple heads. It is about three times as long. Programs for computing primes are contained in the solver `primes.chr` of the CHR library [FrBr96].

4. SYNTAX AND SEMANTICS

In this section we give an overview of syntax and semantics as well as soundness and completeness results for constraint handling rules. More detailed presentations can be found in [FAM97, Abd97, Abd98]. We assume some familiarity with (concurrent) constraint (logic) programming [JaLa87, vH91, Fr*92, Sar93, JaMa94].

As a special purpose language, CHR extend a host language with (more) constraint solving capabilities. Auxiliary computations in CHR programs are directly executed as host language statements. To keep this section essential and self-contained, we will not address host language issues here.

A *constraint* is considered to be a distinguished, special first-order predicate (atomic formula). We use two disjoint sorts of predicate symbols for two different classes of constraints: One sort for *built-in (predefined) constraints* and one sort for *CHR (user-defined) constraints*. Built-in constraints are those handled by a predefined constraint solver that already exists in the host language. CHR constraints are those defined by a CHR program. Since host language statements that appear in CHR must be declarative, we can consider them as built-in constraints in this section (with a rather incomplete solver, the host language).

4.1. Syntax

Definition 4.1. A CHR program is a finite set of CHR. There are three kinds of CHR. A *simplification* CHR is of the form

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

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 *simpagation* CHR is of the form

$$H_1, \ldots, H_l \setminus H_{l+1}, \ldots, H_i \texttt{ <=> } G_1, \ldots, G_j \mid B_1, \ldots, B_k,$$

with $i > 0, j \geq 0, k \geq 0, l > 0$ and where the multi-head $H_1, \ldots, H_i$ is a nonempty sequence of CHR constraints, the guard $G_1, \ldots, G_j$ is a sequence of built-in constraints, and the body $B_1, \ldots, B_k$ is a sequence of built-in and CHR constraints.

Empty sequences are represented by the built-in constraint `true`. For simplicity, the empty guard, `true`, can be removed from a rule together with the commit operator |.

Since a propagation rule could likewise be thought of as an abbreviation of a simplification rule

$$H_1, \ldots, H_l, H_{l+1}, \ldots, H_i \texttt{ <=> } G_1, \ldots, G_j \mid H_1, \ldots, H_l, B_1, \ldots, B_k$$

there is no need to discuss them further in this section, but we use them later when we describe implementations and applications of CHR.

4.2. Declarative Semantics

Unlike general committed-choice programs, CHR programs can be a given a declarative semantics since they are only concerned with defining constraints, not procedures in their generality.

The declarative interpretation of a CHR program P is given by a conjunction of universally quantified logical formulas (one for each rule), $\mathcal{P}$, and a consistent built–in *constraint theory* CT which determines the meaning of the built–in constraints appearing in the program. The theory CT is expected to include an equality constraint `=` and the basic constraints `true` and `false`.

Let $\bar{x}$ denote the sequence of (global) variables occurring in the head atoms $H_1, \ldots, H_i$ of a CHR. Then $\bar{y}$ ($\bar{z}$) are the other (local) variables occurring in the guard $G_1, \ldots, G_j$ (body $B_1, \ldots, B_k$) of the rule (they do not occur in the heads). For simplicity we assume that there are no local variables that occur in both the guard and the body of a rule[1].

Definition 4.2. Declaratively, a simplification CHR is a logical equivalence if the guard is satisfied:

$$\forall \bar{x}\ (\exists \bar{y}\ (G_1 \wedge \ldots \wedge G_j)) \rightarrow (H_1 \wedge \ldots \wedge H_i \leftrightarrow \exists \bar{z}\ (B_1 \wedge \ldots \wedge B_k))$$

A propagation CHR is an implication if the guard is satisfied:

$$\forall \bar{x}\ (\exists \bar{y}\ (G_1 \wedge \ldots \wedge G_j)) \rightarrow (H_1 \wedge \ldots \wedge H_i \rightarrow \exists \bar{z}\ (B_1 \wedge \ldots \wedge B_k))$$

Example 4.1. The CHR

```
reflexivity @ X=<Y <=> X=Y | true
```

from the introductory example in Section 2 has the logical reading

$$\forall \mathtt{X}, \mathtt{Y}\ (\mathtt{X} = \mathtt{Y}) \rightarrow (\mathtt{X} =< \mathtt{Y} \leftrightarrow \mathtt{true}).$$

4.3. Operational Semantics

The *operational semantics* of CHR programs is given by a transition system.

Definition 4.3. A *state* is an annotated tuple

$$\langle F, E, D \rangle_{\mathcal{V}},$$

where F is a conjunction of CHR and built-in constraints called goal (store), E is a conjunction of CHR constraints, D is a conjunction of built-in constraints, called (constraint) stores, and the annotation $\mathcal{V}$ is a sequence of variables. Empty conjunctions are represented by the built-in constraint `true`.

We attribute to each state $\langle F, E, D \rangle_{\mathcal{V}}$ the formula

$$\exists \bar{y}\ F \wedge E \wedge D$$

[1] Else use e.g. $\forall \bar{x} \forall \bar{y}\ ((G_1 \wedge \ldots \wedge G_j) \rightarrow (H_1 \wedge \ldots \wedge H_i \leftrightarrow \exists \bar{z}\ (B_1 \wedge \ldots \wedge B_k)))$ [Fru97].

as its logical meaning, where $\bar{y}$ are the variables occurring in the state except the ones appearing in $\mathcal{V}$, which remain free in the formula.

When it is clear from the context, we will confuse a state S and its logical reading. We also will drop the annotation $\mathcal{V}$ from a state if it is not of interest.

Transitions

With *computation steps (transitions, reductions)* one can proceed from one state to the next. Intuitively, in a state $\langle F, E, D\rangle_{\mathcal{V}}$, F are the constraints that remain to be solved, and D and E are the constraints that have been accumulated and simplified so far. The aim of the computation is to incrementally reduce arbitrary states to states that contain no more goals. There will be one transition for solving built-in constraints, one transition that introduces CHR constraints into their store and three transitions for applying each kind of CHR to them. All transitions leave the annotation $\mathcal{V}$ unchanged.

Definition 4.4. Let P be a CHR program for the CHR constraints and CT be a constraint theory for the built-in constraints. The transition relation $\longmapsto$ for CHR is as follows. All variables occurring in states stand for conjunctions of constraints. $\bar{x}$ denotes the program variables occurring in the multi-head H.

Solve
$\langle C \wedge F, E, D\rangle_{\mathcal{V}} \longmapsto \langle F, E, D'\rangle_{\mathcal{V}}$
if C is a built-in constraint and $CT \models (C \wedge D) \leftrightarrow D'$

Introduce
$\langle H \wedge F, E, D\rangle_{\mathcal{V}} \longmapsto \langle F, H \wedge E, D\rangle_{\mathcal{V}}$
if H is a CHR constraint

Simplify
$\langle F, H' \wedge E, D\rangle_{\mathcal{V}} \longmapsto \langle B \wedge F, E, H = H' \wedge D\rangle_{\mathcal{V}}$
if (H `<=>` $G \mid B$) in P and $CT \models D \rightarrow \exists\bar{x}(H = H' \wedge G)$

Propagate
$\langle F, H' \wedge E, D\rangle_{\mathcal{V}} \longmapsto \langle B \wedge F, H' \wedge E, H = H' \wedge D\rangle_{\mathcal{V}}$
if (H `==>` $G \mid B$) in P and $CT \models D \rightarrow \exists\bar{x}(H = H' \wedge G)$

By equating two constraints, $c(t_1, \ldots, t_n) = c(s_1, \ldots, s_n)$, we mean $t_1 = s_1 \wedge \ldots \wedge t_n = s_n$. By $(H_1 \wedge \ldots \wedge H_n) = (H'_1 \wedge \ldots \wedge H'_n)$ we mean $H_1 = H'_1 \wedge \ldots \wedge H_n = H'_n$. Note that the conjuncts can be permuted since conjunction is assumed to be associative and commutative.

In the **Solve** transition, the built-in solver updates the constraint store D with a new constraint C from the goal store. To *update* the constraint store means to deterministically produce a new constraint store D' that is - according to the constraint theory CT - logically equivalent to the conjunction of the new constraint and the old constraint store.

The **Introduce** transition transports a CHR constraint H from the goal store into the CHR constraint store. There it can be handled together with other CHR constraints by applying rules. A CHR is *applicable* to CHR constraints H' whenever these constraints match the head atoms H of the rule[2](taking into account syntactical equalities implied by the built-in constraint store D) and the guard G is implied (entailed) by the store D.

If a simplification rule (H `<=>` G `|` B) appearing[3] in the given CHR program P is applicable to the CHR constraint H', the **Simplify** transition removes H' from the CHR constraints store, adds B to the goal store and adds the equation $H = H'$ expressing the match between H' and the head atoms H to the built-in constraint store.

If a propagation rule (H `==>` G `|` B) is applicable to H', the **Propagte** transition adds B to the goal store and adds the equation $H = H'$ to the built-in constraint store.

We require that the rules are applied fairly, i.e. that every rule that is applicable is applied eventually. Fairness is respected and trivial non-termination is avoided by applying a propagation rule at most once to the same constraints. A more complex operational semantics that addresses these issues can be found in [Abd97, Abd98].

Initial and Final States

Definition 4.5. The *initial state* consists of a goal F and empty constraint stores,

$$\langle F, \mathtt{true}, \mathtt{true}\rangle_{\mathcal{V}},$$

where $\mathcal{V}$ is the sequence of variables occurring in F. A *final state* is either of the form

$$\langle F, E, \mathtt{false}\rangle_{\mathcal{V}}$$

(such a state is called *failed*), or of the form

$$\langle \mathtt{true}, E, D\rangle_{\mathcal{V}}$$

with no fair computation step possible anymore and D not `false` (such a state is called *successful*).
F is also called *query.* A final state is called *(conditional or qualified) answer* for the query F.

Thus the annotation $\mathcal{V}$ allows distinguishing between the query variables and the variables introduced during the computation.

Example 4.2. A computation of the goal $\mathtt{A} \leq \mathtt{B} \wedge \mathtt{C} \leq \mathtt{A} \wedge \mathtt{B} \leq \mathtt{C}$ for the introductory example in Section 2 proceeds as follows:

[2] This is the effect of the existential quantification over the head equalities, e.g. $\exists\bar{x}(H = H')$.
[3] As usual, variables are renamed apart.

$$
\begin{array}{ll}
 & \langle \mathtt{A} \leq \mathtt{B} \wedge \mathtt{C} \leq \mathtt{A} \wedge \mathtt{B} \leq \mathtt{C}, \mathtt{true}, \mathtt{true}\rangle_{[A,B,C]} \\
\mapsto^3_{Introduce} & \langle \mathtt{true}, \mathtt{A} \leq \mathtt{B} \wedge \mathtt{C} \leq \mathtt{A} \wedge \mathtt{B} \leq \mathtt{C}, \mathtt{true}\rangle_{[A,B,C]} \\
\mapsto_{Propagate}\ \mathtt{Transitivity} & \langle \mathtt{C} \leq \mathtt{B}, \mathtt{A} \leq \mathtt{B} \wedge \mathtt{C} \leq \mathtt{A} \wedge \mathtt{B} \leq \mathtt{C}, \mathtt{true}\rangle_{[A,B,C]} \\
\mapsto_{Introduce} & \langle \mathtt{true}, \mathtt{A} \leq \mathtt{B} \wedge \mathtt{C} \leq \mathtt{A} \wedge \mathtt{B} \leq \mathtt{C} \wedge \mathtt{C} \leq \mathtt{B}, \mathtt{true}\rangle_{[A,B,C]} \\
\mapsto_{Simplify}\ \mathtt{Antisymmetry} & \langle \mathtt{B} = \mathtt{C}, \mathtt{A} \leq \mathtt{B} \wedge \mathtt{C} \leq \mathtt{A}, \mathtt{true}\rangle_{[A,B,C]} \\
\mapsto_{Solve} & \langle \mathtt{true}, \mathtt{A} \leq \mathtt{B} \wedge \mathtt{C} \leq \mathtt{A}, \mathtt{B} = \mathtt{C}\rangle_{[A,B,C]} \\
\mapsto_{Simplify}\ \mathtt{Antisymmetry} & \langle \mathtt{A} = \mathtt{B}, \mathtt{true}, \mathtt{B} = \mathtt{C}\rangle_{[A,B,C]} \\
\mapsto_{Solve} & \langle \mathtt{true}, \mathtt{true}, \mathtt{A} = \mathtt{B} \wedge \mathtt{B} = \mathtt{C}\rangle_{[A,B,C]}
\end{array}
$$

4.4. Soundness and Completeness

We now relate the operational and declarative semantics of CHR. These results are based on [JaLa87, Mah87, vH91] and can be found with with proofs in [FAM97, Abd98].

Definition 4.6. A *computation of a goal* G is a sequence $S_0, S_1, \ldots$ of states with $S_i \mapsto S_{i+1}$ beginning with the the initial state $S_0 = \langle G, \mathtt{true}, \mathtt{true}\rangle_{\mathcal{V}}$ and ending in a final state or diverging. A finite computation is *successful* if the final state is successful. It is *failed* otherwise.

Definition 4.7. $S \mapsto^* S'$ holds iff $S = S'$ or $S \mapsto S_1 \mapsto \ldots \mapsto S_n \mapsto S' \quad (n \geq 0)$.

The following results are based on the fact that the transitions for CHR preserve the logical meaning of states. All states in a computation are logically equivalent.

Lemma 4.1. Let P be a CHR program and G be a goal. If C is the logical reading of a state appearing in a computation of G, then

$$\mathcal{P}, CT \models \forall\, (C \leftrightarrow G)$$

where $\forall F$ denotes the universal closure of a formula F.

Proof. By structural induction over the computation steps.

In the soundness and completeness results for CHR, there is no need to distinguish between successful and failed computations.

Theorem 4.1 Soundness. Let P be a CHR program and G be a goal. If G has a computation with answer C then

$$\mathcal{P}, CT \models \forall\, (C \leftrightarrow G).$$

Proof. Immediately from lemma 4.1.

Theorem 4.2 Completeness. Let P be a CHR program, G be a goal with at least one finite computation and C be a conjunction of constraints. If $\mathcal{P}, CT \models \forall\, (C \leftrightarrow G)$, then G has a computation with answer C' such that

$$\mathcal{P}, CT \models \forall\, (C \leftrightarrow C').$$

Proof. Immediately from Theorem 4.1.

The theorem is stronger than the completeness result for CLP languages presented in [Mah87], in the way that we can reduce the disjunction in the strong completeness theorem to a single disjunct (due to lemma 4.1). The following example shows that the completeness theorem does not hold if G has no finite computations.

Example 4.3. Let P be the CHR program:

```
p <=> p.
```

Let G be `p`. It holds that $\mathcal{P}, CT \models \mathtt{p} \leftrightarrow \mathtt{p}$ since $\mathcal{P}$ is $\{\mathtt{p} \leftrightarrow \mathtt{p}\}$. However, G has only infinite computations.

The soundness result, Theorem 4.1, can be specialized to failed computations.

Corollary 4.1. Let P be a CHR program and G be a goal. If G has a finitely failed computation, then $\mathcal{P}, CT \models \neg\exists G$.
PROOF. By Theorem 4.1.

However, an analogous completeness result, that is, the converse of Corollary 4.1, does not hold in general:

Example 4.4. Let P be the CHR program:

```
p <=> q.
p <=> false.
```

$\mathcal{P}, CT \models \neg q$, but q has no finitely failed computation.

Thus the completeness theorem 4.2 is rather weak for failed computations. A stronger completeness result can be given for correct programs and data-sufficient goals[4]. Data-sufficiency was introduced for completeness of deterministic ALPS programs in [Mah87] (see also Section 5.1).

Definition 4.8. A CHR program P is *correct* iff $\mathcal{P} \cup CT$ is consistent.

Definition 4.9. A goal is *data-sufficient* if it has a computation ending in a final state of the form $\langle \mathtt{true}, \mathtt{true}, D\rangle_{\mathcal{V}}$.

Theorem 4.3 Stronger Completeness of Failed Computations. Let P be a correct CHR program and G be a data-sufficient goal. If $\mathcal{P}, CT \models \neg\exists G$ then G has a finitely failed computation.
PROOF. By Theorem 4.1, the definition of correctness and the fact that a final state contains only built-in constraints, because G is data-sufficient.

We will see that the confluence property introduced next will further improve our soundness and completeness results.

5. CONFLUENCE

We have already shown in the previous section (Lemma 4.1) that in a CHR program, the result of a computation from a given goal will always have the same meaning.

[4] Data-sufficiency is missing from Theorem 19 in [FAM96], thus it is stated wrongly.

However it is not guaranteed that the result is syntactically the same. The confluence property of a program guarantees that any computation starting from an arbitrary given initial state, i.e. any possible order of rule applications, results in the same final state. It does not guarantee that the solver will be (satisfaction) complete, i.e. detect all inconsistencies.

Due to space limitations, we can just give an overview on confluence where some definitions are just informal. Detailed confluence results for simplification rules only are published in [FAM97]. Recently, these results have been simplified and extended to all three kinds of CHR [Abd97, Abd98]. The papers adopt and extend the terminology and techniques of conditional term rewriting systems [DOS88, KiKi91] about confluence. The extensions enable handling of global knowledge (the built-in constraint store), local variables and propagation rules. In [Abd98], it was also possible to adapt to CHR the idea of Knuth-Bendix completion, an algorithm that makes a set of rules confluent by introducing additional rules.

We require that states are normalized so that they can be compared syntactically in a meaningful way. Since the formal definition of the normalization function is quite involved, we describe normalized states just informally. Basically, we require that the built-in constraints are in a (unique) normal form where all equalities are made explicit and are propagated to all components of the state. The normalization also has to make identical all failed states.

Furthermore, we require a more refined operational semantics. We augment states with a second annotation. The new annotation $\mathcal{T}$ is a multiset of tokens representing potential applications of propagation rules to constraints. When a propagation rule is applied, the corresponding token is removed so that the rule cannot be reapplied again to the same constraints. When a simplification rule is applied, the appropriate tokens in which the removed constraints occur are removed.

In the rest of this section we assume that states are normalized and annotated.

Definition 5.1. Two states are *variants* if they can be obtained from each other by a variable renaming.
Two states S_1 and S_2 are called *joinable* if there exist states S_1', S_2' such that $S_1 \mapsto^* S_1'$ and $S_2 \mapsto^* S_2'$ and S_1' is a variant of S_2'.

Definition 5.2. A CHR program is called *confluent* if for all states S, S_1, S_2: If $S \mapsto^* S_1, S \mapsto^* S_2$ then S_1 and S_2 are joinable.
A CHR program is called *locally confluent* if for all states S, S_1, S_2: If $S \mapsto S_1, S \mapsto S_2$ then S_1 and S_2 are joinable.

Example 5.1. The CHR program from example 4.4 is not confluent since `p` can either be simplified to `q` or `false`. The corresponding states are final and differ. However the following program is confluent:

```
p <=> q.
p <=> false.
q <=> false.
```

We give a new motivation for critical pairs here based on the notion of nontrivial direct common ancestor states.

To analyze local confluence of a given CHR program we cannot check joinability of all pairs of states that derive from a common ancestor state, because in general there are infinitely many such states. However one can construct a finite number of minimal states where more than one rule is applicable: A direct common ancestor state consists of the heads and guards of the rules. It suffices to construct non-failed states from two rules. It is obvious that there is only a finite number of such states for a given program. Due to the monotonicity property of CHR, these states can be extended to any context, i.e. to all possible ancestor states. Monotonicity states that adding constraints to the components of the state cannot inhibit the application of a rule as long as the built-in constraint store remains consistent.

We now further restrict ourselves to nontrivial direct common ancestor states: Joinability can only be destroyed if one rule inhibits the application of the other rule. The application of a rule may remove CHR constraints from the user-defined store and introduce new constraints. Only the removal of constraints can effect the applicability of another rule, in case the removed constraint is needed by the other rule. To possibly inhibit each other, at least one rule must be a simplification CHR and the two rules must *overlap*, i.e. have at least one head atom in common in the ancestor state. This is achieved by equating head atoms in the state and by removing the resulting identical copies of head atoms.

Definition 5.3. Given a simplification rule R_1 and an arbitrary (not necessarily different) rule R_2 from a CHR program P, whose variables have been renamed apart. Let G_i denote the guard, B_i denote the body of rule R_i $(i = 1, 2)$. Let H_i^c and H_i be a partition of the head of the rule R_i into two conjunctions, where the conjunction of common head atoms H_i^c is nonempty. Then a *nontrivial direct common ancestor state S of R_1 and R_2* is

$$\langle \mathtt{true}, H_1^c \wedge H_1 \wedge H_2, (H_1^c = H_2^c) \wedge G_1 \wedge G_2 \rangle_{\mathcal{V}}^{\mathcal{T}},$$

provided $(H_1^c = H_2^c) \wedge G_1 \wedge G_2$ is consistent. $\mathcal{V}$ is the sequence of variables appearing in $H_1^c \wedge H_1 \wedge H_2$. If R_2 is a simplification rule, $\mathcal{T}$ is the empty set, if R_2 is a propagation rule, $\mathcal{T}$ is $\{\langle R_2, H_1^c \wedge H_2 \rangle\}$.

The choice of $\mathcal{T}$ is motivated by the minimality criterion for the state: It covers the case that all propagation rules (except possibly R_2) have already been applied to the constraints of the user-defined store before the ancestor state S was reached.

The application of R_1 and R_2 respectively to S leads to two states that form the so-called critical pair. In the states of the critical pair, the body B_i of the rule R_i is in the goal store, H_i^c and H_i have been removed from the CHR constraint store in case R_i is a simplification rule, $\mathcal{T}$ will be empty and the built-in constraint store and the $\mathcal{V}$ annotation remain the same.

Definition 5.4. Let S be a nontrivial direct common ancestor state. If $S \mapsto_{R_1} S_1$ and $S \mapsto_{R_2} S_2$ then the tuple (S_1, S_2) is the *critical pair*[5]of S.

A critical pair (S_1, S_2) is *joinable*, if S_1 and S_2 are joinable.

[5]Due to the condensed presentation, this definition differs from the one in [Abd97]. However, the difference is only syntactical, in the way the critical pair is represented.

Example 5.2. Consider the example of Section 2. The following nontrivial ancestor state comes from equating the first head atom of the antisymmetry rule with the first head atom of the transitivity rule: $\langle \mathtt{true},\ \mathtt{X{\leq}Y{\land}Y{\leq}Z{\land}Y{\leq}X},\ \mathtt{true}\rangle_{\mathcal{V}}^{\mathcal{T}}$, where $\mathcal{V}$ is the sequence of variables $\mathtt{X,Y,Z}$ and $\mathcal{T}$ contains just the token $\langle$ $\mathtt{transitivity,}$ $\mathtt{X{\leq}Y{\land}Y{\leq}Z}$ $\rangle$.
The critical pair is $(\langle \mathtt{X} \leq \mathtt{Z}, \mathtt{X} \leq \mathtt{Y} \land \mathtt{Y} \leq \mathtt{Z} \land \mathtt{Y} \leq \mathtt{X}, \mathtt{true}\rangle_{\mathcal{V}}^{\emptyset}, \langle \mathtt{X} = \mathtt{Y}, \mathtt{Y} \leq \mathtt{Z}, \mathtt{true}\rangle_{\mathcal{V}}^{\emptyset})$. The critical pair is joinable, since there are computations from its two states with empty multisets of tokens that result in the same final state $\langle \mathtt{true}, \mathtt{X} \leq \mathtt{Z}, \mathtt{X} = \mathtt{Y}\rangle_{\mathcal{V}}^{\emptyset}$.

We are now able to give the main theorem connecting joinability of critical pairs with local confluence:

Theorem 5.1. A CHR program is locally confluent iff all its critical pairs are joinable.

Proof. *The if-direction:* Assume that we are in state S where there are two or more possibilities for computation steps. We investigate all pairs of possible computation steps and show that they are joinable.
The only-if-direction: By contradiction. We assume that we have a locally confluent CHR program with a critical pair that is not joinable.

The following corollary gives us a decidable, sufficient and necessary test for confluence of a terminating program:

Definition 5.5. A CHR program is called *terminating*, if there are no infinite computations.

Corollary 5.1. A terminating CHR program is confluent iff all its critical pairs are joinable.

Proof. Immediately from Theorem 5.1 and Newman's lemma [New42].

Our notion of confluence subsumes the notion of determinacy as used by Maher [Mah87] and Saraswat [Sar93] for (concurrent) constraint (logic) programs. In a determinate program, guards of rules for the same predicate are mutually exclusive. Thus they are trivially confluent, since no critical pairs exist.

5.1. Soundness and Completeness Revisited

We showed in [FAM97, Abd98] that confluence implies correctness (see Definition 8).

Theorem 5.2. If P is confluent, then $\mathcal{P} \cup CT$ is consistent.

The following theorem shows that we can improve on soundness and completeness if a CHR program is confluent and terminating.

Theorem 5.3 Strong Soundness and Completeness. Let P be a terminating and confluent CHR program and G be a goal. Then the following are equivalent:

a) $\mathcal{P}, CT \models \forall\ (C \leftrightarrow G)$.

b) G has a computation with answer C' such that $\mathcal{P}, CT \models \forall\,(C \leftrightarrow C')$.

c) Every computation of G has an answer C' such that $\mathcal{P}, CT \models \forall\,(C \leftrightarrow C')$.

PROOF. "a) $\Rightarrow$ b)" by Theorem 4.2.
"b) $\Rightarrow$ c)" by confluence and termination.
"c) $\Rightarrow$ a)" by Theorem 4.1.

The following corollary is a soundness and completeness result for finitely failed computations.

Corollary 5.2. (Soundness and Completeness of Finite Failure) Let P be a terminating and confluent CHR program and G be a data-sufficient goal. Then the following are equivalent:

a) $\mathcal{P}, CT \models \neg\exists G$

b) G has a finitely failed computation.

c) Every computation of G is finitely failed.

PROOF. By Theorems 5.3, 5.2 and 4.3.

Maher proved similar soundness and completeness results for deterministic ALPS programs with data-sufficient goals. Our results hold for a substantially larger class of programs, confluent and terminating CHR programs. Note, however, that ALPS in general has a different semantics (based on Clark's completion) and a different operational semantics (rules can commit more often) than CHR.

6. CLP + CHR

We now assume that constraint handling rules extend a given CLP language and extend the definitions from the previous sections accordingly. For CLP, a tight integration is possible: We allow clauses for CHR constraints. These are used for labeling, i.e. introducing choices. The idea is that if no simplification and propagation is possible anymore, a constraint is automatically chosen for labeling.

Conversely, we can regard any predicate as a (labeling routine of a) constraint and add some CHR for it. Seen this way, CHR are lemmata that allow expressing the determinate information contained in a predicate. Predicates and constraints are just alternate views, don't know and don't care nondeterminism are combined in a declarative way. This is also the idea of *Guarded Rules* [Smo93] mentioned in section 3. To see the power of such lemmata consider the rule `append(L1,[],L) <=> L1=L`. The recursion on the list `L1` in the usual definition of `append` is replaced by a simple unification `L1=L`.

Example 6.1. We continue with the example from Section 2. To illustrate automatic labeling with the CHR constraint `=<`, we use successor notation for numbers.

```
label_with X=<Y if ground(X).
label_with X=<Y if ground(Y).
     0=<Y.
```

```
     s(X)=<s(Y) :- X=<Y.
```

The labeling declarations (starting with `label_with`) state that one may label with `X=<Y` if either `X` or `Y` are ground (variable-free terms).

```
     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.
```

Definition 6.1. A CLP+CHR program is a finite set of CLP clauses for predicates and CHR constraints and of CHR rules for CHR constraints. As usual, a *CLP clause* is of the form

H `:-` $B_1, \ldots, B_k.\ \ (k \geq 0)$

where the head H is an atom but not a built-in constraint, the body $B_1, \ldots, B_k$ is a conjunction of atoms. A *labeling declaration* for a CHR constraint H_L is of the form

label_with H_L if $G_1, \ldots, G_j$.

Definition 6.2. Let $(H_1$ `:-` $B_{11}, \ldots, B_{n1}), \ldots, (H_s$ `:-` $B_{1s}, \ldots, B_{ns})$, $(1 \leq s)$ be all the clauses with the same predicate p in the head with all the variables in different clauses renamed apart. Then the logical reading of the predicate p by *Clark's completion* is defined as:

$$\forall \bar{x}\ (H \leftrightarrow \exists \bar{z}\ ((H = H_1 \wedge B_{11} \wedge \ldots \wedge B_{n1}) \vee \ldots \vee (H = H_s \wedge B_{1s} \wedge \ldots \wedge B_{ns})))$$

H is of the form $p(X_1, \ldots, X_r)$ where $X_1, \ldots, X_r$ are new, pairwise different variables. The labeling declaration serves as a precondition in the logical meaning of the clauses for the CHR constraint:

$$\forall \bar{x}\ (\exists \bar{y}\ (H_L = H \wedge G_1 \wedge \ldots \wedge G_j) \rightarrow (H \leftrightarrow \exists \bar{z}\ (B_1 \vee \ldots \vee B_s))).$$

where $(H \leftrightarrow \exists \bar{z}\ (B_1 \vee \ldots \vee B_s)))$ is Clark's completion.

Definition 6.3. The computation steps involving clauses are:

Unfold
$\langle H' \wedge F, E, D\rangle \longmapsto \langle B \wedge F, E, H = H' \ \wedge\ D\rangle$
if $(H$ `:-` $B)$ in P and H is not a CHR constraint

Label
$\langle F, H' \wedge E, D\rangle \longmapsto \langle B \wedge F, E, H = H' \ \wedge\ D\rangle$
if $(H$ `:-` $B)$ in P and (`label_with` H'' `if` G) in P and
$D \rightarrow \exists \bar{x}\ (H' = H'' \wedge G)$

where $\bar{x}$ denotes the program variables occurring in H''.

To *unfold* an atomic goal H' in F means to look for a CLP clause $(H{:}-B)$ and to replace H' by $(H = H')$ and B. Unfolding is nondeterministic and thus a goal can be solved in different ways using different clauses. The clauses for CHR constraints can only be unfolded by the **Label** transition provided the `label_with` declaration is satisfied.

7. IMPLEMENTATIONS

The first implementation of CHR in summer 1991 was an interpreter written in ECL^{i}PS^e Prolog, called Cheer[6][Fru91, Fru92, Fru93b, FrBr95a]. Since then, the CHR language has been implemented in 1993 in Common LISP at the German Research Institute for Artificial Intelligence (DFKI) [Her93] and in 1994 as a library of ECL^{i}PS^e [FrBr95a, FrBr95b, FrBr96]. It is currently implemented in Sicstus Prolog at LMU, Munich, in ECL^{i}PS^e 2 at IC-Parc of Imperial College and in the concurrent logical object-oriented constraint language OZ [SmTr94a].

Cheer [Fru91, Fru92, Fru93b] was a small but fully functional interpreter. By small we mean about 300 clauses, 900 lines, 25KB of code. By fully functional we mean that Cheer included a preprocessor for CHR, delaying conjunction, incremental constraints residuation, a tracing tool for CHR constraints and variable bindings, a simple partial evaluator based on simplifications, and simple statistics (number of rules fired per kind, timings). First solvers were term equality (unification), finite domains, term manipulation, maximum, types and temporal reasoning.

The LISP implementation [Her93] does not provide for simpagation rules, but offers some interesting extensions. First, rules can be given priorities (encoded as integers). Second, nondeterminism is introduced by disjunction in rule bodies. This extension also allows expressing Prolog clauses. Rules with disjunction usually get the lowest priority. The algorithm for executing CHR is somewhat similar to the first implementation of CHR in Prolog. 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 [Her93], constraint solvers 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.

The CHR library, version 2, of ECL^{i}PS^e 3.5.3 [FrBr95a, FrBr95b, FrBr96] includes a compiler, a run-time system with debugger, 25 solvers (see Section 8) with examples as well as a full color demo using geometric constraints in a real-life application for wireless telecommunication (see Section 9). In extension to the definitions given earlier, CHR rules can have deep guards[7]and local variables can be shared between guard and body of a rule. Prolog and CHR statements can be freely combined. With the library, a complete committed-choice language is available as a side-effect. The compiler utilizes the delay-mechanism and the built-in predicates of ECL^{i}PS^e to create, inspect and manipulate constraints as delayed goals based

[6] Ch for constraint handling, ee for extension of ECL^{i}PS^e, and r for rules.

[7] Guards that allow for user-defined predicates in addition to built-in constraints.

on attributed variables. 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 compilers in ECL^{i}PS^{e}and Sicstus Prolog are based on the idea that all three types of CHR can be transformed into multi-headed and further into single-headed simplification rules, i.e. into the guarded rules of a typical concurrent committed-choice language [Sha89, Sar93] - provided the language can access delayed goals and has deep guards. CHR constraint goals are modeled as goals that can delay. Then these guarded rules are further translated into clauses of a CLP language using its delay-mechanism (coroutining) based on attributed variables. A detailed description of the compilation scheme and its actual implementation can be found in [FrBr95a].

Performance

On a 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). The slow-down is often within an order of magnitude. On some examples (e.g. those involving finite domains with the element-constraint or linear polynomial equations over rationals, see Section 8), and in some applications, our approach is faster, since we can exactly define and tune the amount of constraint simplification and propagation as needed. For performance and simplicity the solver can be kept as incomplete as the application allows it.

Besides the well-defined low-level support for manipulating delayed goals (adding, searching for, activating and removing delayed goals) provided through attributed variables, the reason for the good performance are a number of significant optimizations which are the result of many experiments performed with the interpreter Cheer.

For example, based on the observation that usually the head atoms of a rule are connected through common variables, given one constraint, we usually only search for other constraints in those that delay on a common variable. Since in many constraint domains, the number of constraints in the normal form is linear in the number of variables, one can often find the other constraints in constant time.

Moreover, the order in which the rules are tried matters. The ECL^{i}PS^e CHR compiler prefers simplification to propagation rules, single-headed to multi-headed rules. Propagation from a constraint may cause further propagations from the redundant constraints. The compiler first adds all constraints propagated from a constraint before considering the new ones in turn. In simpagation rules, it is preferred to remove the most recent constraint if there is a choice. In the new Sicstus implementation of CHR the user can control the order of the rules.

Last but not least, there are user declarations and rule annotations that enforce idempotence of constraints. One optimization related to idempotence is not to remove a constraint that is generated again in the body of the rule that wants to remove it. This may speed up the computation, improve the complexity of the resulting algorithm and even avoid non-termination.

8. CONSTRAINT SOLVERS

In this section we introduce some of 25 constraint solvers that are part of the CHR library of ECL^{i}PS^e 3.5.3 (see figure 8) [FrBr95b, FrBr96] - among them solvers for finite domains over arbitrary ground terms, including reals and pairs, incremental path consistency, temporal reasoning, for solving linear polynomials over the reals and rationals, and last but not least for terminological reasoning.

Many of the solvers are described here for the first time. The solver may be slightly edited, mainly to make them self-contained, consistent in presentation and more readable. When we know about it, we also mention related work, i.e. how these solvers have been used by other researchers, and related solvers written by other researchers using CHR.

While we cannot - within the space limitations - introduce each constraint domain, we still can give an idea how one implements it using CHR. 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 CHR in a straightforward way. Starting from this executable specification, the rules can be refined and adapted to the specifics of the application.

Note that any solver written with CHR will be *determinate*, *incremental* and *concurrent* by nature. By "determinate" we mean that the user-defined solver commits to every constraint simplification it makes. By "incremental" we mean that constraints can be added to the constraint store one at a time (without affecting computational cost). The rules can be applied concurrently to different constraints, because logically correct CHR can only replace constraints by equivalent ones or add redundant constraints.

Note that many solvers rely on an order on variables and terms (using the built-in predicate `<`).

8.1. Booleans

The domain of Boolean constraints includes the constants `0` for falsity, `1` for truth and the usual logical connectives of propositional logic, e.g. `and, or, neg, imp, exor`, modeled here as relations. We assume that equality `=` is a built-in constraint. The program `bool.chr`[8]is a simple solver mainly based on value propagation using single-headed simplification rules and automatic labeling. For more sophisticated algorithms see [Me*93].

We can define an and-gate with constraint handling rules (assuming that variables can only take Boolean values):

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

For example, the first rule says that the constraint `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`

[8]File names refer to [FrBr96].

Library File	Constraint Domain	`<=>`	`==>`	`\`	`Co`	`|`
arc.chr	Arc consistency	1	3	0	3	1
bool.chr	Booleans	56	0	19	7	3
cft.chr	Feature Trees	2	1	3	3	2
control.chr	Sound control primitives	6	0	0	5	4
domain.chr	Finite domains	65	14	4	8	54
geons.chr	Geometric objects	1	0	0	1	0
kl-one.chr	Terminological reasoning	25	13	4	3	6
leq.chr	Partially ordered variables	2	1	0	1	0
list.chr	Lists with lengths	9	0	0	2	3
math-gauss.chr	Linear polynomial equations	1	0	1	2	1
math-elim.chr	+ Inequations by slacks	26	0	4	8	16
math-fougau.chr	+ Fouriers algorithm	28	5	3	8	16
math-fourier.chr	+ Fouriers algorithm	25	1	2	8	15
math-eager.chr	+ Inequations by slacks	7	0	2	2	7
math-ineq.chr	+ Inequations by slacks	16	0	0	6	10
math-lazy.chr	+ Inequations by slacks	7	0	3	2	9
minmax.chr	Minima and maxima	17	6	24	5	23
osf.chr	Order-Sorted Feature Trees	5	1	1	2	4
path.chr	Path consistency	2	3	4	1	0
primes.chr	Prime numbers	12	0	2	8	9
set.chr	Finite Sets	22	13	6	12	4
term.chr	Prolog term constructors	10	7	8	7	8
time-pc.chr	Temporal reasoning	11	3	0	2	6
time-point.chr	Time-points	4	2	0	2	4
tree.chr	Rational trees + negation	9	1	2	3	8

FIGURE 8.1. The constraint solvers of the CHR library in ECL^{i}PS^e 3.5.3.`<=>` stands for the number of simplification, `==>` propagation, \ simpagation rules; `Co` for the number of CHR constraints, `|` nonempty guards in the rules.

must be 0. Hence the query `and(X,Y,Z),X=0` will result in `X=0, Z=0`.

It is obvious that the above rules terminate, since the CHR constraints `and` is always reduced to the built-in constraint `=`. It is also confluent. The critical pairs are easy to construct, since all the heads are identical. For example, the rules `and(X,Y,Z) <=> Z=1 | X=1,Y=1` and `and(X,Y,Z) <=> X=Y | Y=Z` lead to the critical pair (⟨`true`, `X=1` ∧ `Y=1`, `X=Y` ∧ `Z=1` ⟩, ⟨`true`, `Y=Z`, `X=Y` ∧ `Z=1` ⟩). Both states simplify to `X=1` ∧ `Y=1` ∧ `Z=1`.

Example 8.1. Consider the predicate `add/4` taken from the well-known full-adder circuit. It adds three single digit binary numbers to produce a single number consisting of two digits:

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

The query `add(I1,I2,I3,[O1,O2]),I3=0,O1=1` will reduce to `I3=0,O1=1,I1=1,` I2=1,O2=0. The computation proceeds as follows: Because I3=0, the output A2 of the and-gate with input I3 must be 0. As O1=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 O2 of the second xor-gate must be 0. The query add(1,1,I3,[O1,O2]) reduces to I3=O2,O1=1. This example illustrates the power of this simple but incomplete solver.

Flexibility and Extensions

The cardinality constraint combinator was introduced in the CLP language cc(FD) [vH91, HSD95] for finite domains. Here we adapt it for Boolean variables. The Boolean cardinality constraint #(L,U,BL,N) holds if between L and U Boolean variables in the list BL of length N are equal to 1. In the solver, we assume that for a constraint #(L,U,BL,N), the condition L=<U,0=<U,0=<N,L=<N initially holds, where N is the length of the finite (closed) list BL. We also assume that arithmetic constraints (or at least tests) between integers involving =< and subtraction are built-in. delete/3 is the usual Prolog predicate removing an element from a list.

```
% trivial, positive and negative satisfaction
triv_sat@ #(L,U,BL,N) <=> L=<0,N=<U | true.
pos_sat @ #(L,U,BL,N) <=> L=N | all_true(BL).
neg_sat @ #(L,U,BL,N) <=> U=0 | all_false(BL).
% positive and negative reduction
pos_red @ #(L,U,BL,N) <=> delete(1,BL,BL1) | 0<U,#(L-1,U-1,BL1,N-1).
neg_red @ #(L,U,BL,N) <=> delete(0,BL,BL1) | L<N,#(L,U,BL1,N-1).

% labeling, choice between positive and negative reduction
label_with #(L,U,[X|BL],N) if true.
#(L,U,[1|BL],N):- 0<U, #(L-1,U-1,BL,N-1).
#(L,U,[0|BL],N):- L<N, #(L,U,BL,N-1).
```

When delete/3 is used in the guard, it will only succeed if the element to be removed actually occurs in the list. E.g. delete(1,BL,BL1) will delay if it tries to bind a variable in BL to 1. It can only succeed if there actually is a 1 in the list. It will fail, if all elements of the list are zeros. The predicate all_true (resp. all_false) binds all elements of the list BL to 1 (resp. 0). Note that the call to #/4 in the bodies of the labeling clauses is a call to the cardinality as constraint.

Since the cardinality constraint is either simplified into a built-in constraint or reduced to a cardinality with a shorter list, this implementation terminates. If the list of an initial cardinality constraint were open(-ended), i.e. its length not fixed, there could be contexts in which the cardinality constraint does not terminate. One can also show that the solver maintains the above condition, i.e. that it is an invariant. With the invariant, the implementation is also confluent.

Related Solvers and Work

In [Dum95] experiments were performed in applying resolution and backtracking to solving Boolean constraint satisfaction problems. A limited version of resolution, called ordered resolution, was introduced and compared to that of the Davis Putnam method [DaPu60].

The DP procedure has been extensively used on satisfiability problems, it is a sound procedure that basically restricts resolution to unit clauses. A labeling phase is added that tries truth values using backtracking for the variables one by one, thus retaining completeness. Ordered resolution is a sound and complete restriction of resolution where the literals in the clauses are globally ordered and resolution can only be performed with the leftmost literals of each clause. This method was found to be an improvement over DP when the length of the clauses generated was limited to some small number and then again labeling was used for preserving completeness.

Here is an incremental version of the DP procedure[9], other versions of resolution can also be found in [Dum95]. Boolean CSPs are modeled as conjunctions of clauses, where a clause is a disjunction of literals (positive or negative atomic propositions). A clause is represented as a list of signed Boolean variables. For example, $\neg a \vee b \vee c$ is represented as `cl([-A,+B,+C])`. The variables in the lists are ordered. `member/2` is the usual Prolog predicate about lists.

```
empty_cl @ cl([]) <=> fail.
tautology @ cl(L) <=> member(-X,L),member(+X,L) | true.

unit_instantiation @ cl([+X]) <=> X=1.
unit_instantiation @ cl([-X]) <=> X=0.
unit_propagation @ cl(L) <=> delete(+0,L,L1) | cl(L1).
unit_propagation @ cl(L) <=> delete(-1,L,L1) | cl(L1).
unit_subsumption @ cl(L) <=> member(+1,L) | true.
unit_subsumption @ cl(L) <=> member(-0,L) | true.

% labeling only necessary if list has at least two elements
label_with cl([_,_|_]) if true.
% X is either 0 or 1 and we already applied the unit_* rules
cl([+X|L]):- X=1 ; X=0, cl(L).
cl([-X|L]):- X=0 ; X=1, cl(L).
```

Note the similarity with the cardinality constraint. The argument for termination is the same. Confluence can be proven.

8.2. Terminological Reasoning

Terminological formalisms are used to represent the terminological knowledge of a particular problem domain on an abstract logical level. To describe this kind of knowledge, one starts with atomic concepts and roles, and then defines new concepts and their relationship in terms of existing concepts and roles. Although there is an established notation for terminologies, we use a more verbose syntax to help readers not familiar with the topic.

Concepts can be considered as unary relations which intensionally define sets of objects (similar to types). Roles correspond to binary relations over objects (not necessarily of the same kind - properties like `color` can be roles as well).

[9] "Pure literal deletion" is not implemented, because it is based on a global condition which is not sound anymore when constraints can be added incrementally as is the case in CHR.

Definition 8.1. Concept terms are defined inductively: Every concept name C is a concept term. If s and t are concept terms and R is a role name then the following expressions are concept terms:

s `and` t (conjunction),
s `or` t (disjunction),
`nota` s (complement),
`every` R `is` s (value restriction),
`some` R `is` s (exists-in restriction).

Objects are constants or variables. Let a, b be objects, R a role, and C a concept term. Then $b : C$ is a *membership assertion* and $(a, b) : R$ is a *role-filler assertion.* An *A-box* is a collection of membership and role-filler assertions.

Definition 8.2. A *terminology* (T-box) consists of a finite set of *concept definitions*

C `isa` s,

where C is the newly introduced concept name and s is a concept term.

Since the concept C is new, it cannot be defined in terms of itself, i.e. concept definitions are acyclic. This also implies that there are concepts without definition, they are called primitive.

We will represent the T-box as CLP predicates and the A-box as CHR constraints, since we want to solve problems over a given terminology.

Example 8.2. The domain of a configuration application comprises at least devices, interfaces, and configurations. The concept definitions express that these concepts are disjoint:

```
interface isa nota device.
configuration isa nota (interface or device).
```

Assume that a simple device has at least one interface. We introduce a role `connector` which relates devices to interfaces and employ the exists-in restriction.

```
simple_device isa device and some connector is interface.
```

We introduce instances of devices and interfaces as constraints:

```
pc:device, rs231:interface, (pc,rs231):connector
```

Solver

Terminological formalisms have a straightforward embedding in first-order logic. However, the limited expressiveness of terminological formalisms allows decision procedures for a number of interesting reasoning problems. These problems include consistency of assertions and classification of concepts. The key idea of [ScSm91, BDS93]) for constructing such inference algorithms is to reduce all reasoning services to consistency checking. The unfolding and completion rules in

[ScSm91] and the propagation rules in [BDS93] for the consistency test translate almost directly to CHR (library solver file `kl-one.chr`). However, the former work does not provide an incremental algorithm and the latter does not simplify constraints.

Roughly, the consistency test of A-boxes simplifies and propagates the assertions in the A-box to make the knowledge more explicit and looks for obvious contradictions ("clashes") such as "`X:device`, `X:nota device`". We need only a single clash rule, one may need more for extensions of the formalism.

```
I:nota S, I:S  <=> false.
```

The following simplification CHR show how the complement operator `nota` can be pushed towards to the leaves of a concept term, e.g.:

```
I:nota (S or T)  <=>  I:nota S and nota T.
I:nota every R is S  <=>  I:some R is nota S.
```

An exists-in restriction generates a variable that serves as a "witness" for the restriction:

```
I:some R is S  <=>  (I,J):R, J:S.
```

A value restriction has to be propagated to all role fillers:

```
I:every R is S, (I,J):R  ==>  J:S.
```

The unfolding rules replaces concept names by their definitions:

```
I:C  <=>  C isa S, I:S.
I:nota C  <=>  C isa S, I:nota S.
```

The conjunction rule generates two new, smaller assertions:

```
I:S and T  <=>  I:S,I:T.
```

Disjunction is treated lazily by a CLP clause using automatic labeling. This is where the exponential complexity of the consistency test for terminologies surfaces.

```
label_with I:S or T if true.
I:S or T :- (I:S ; I:T).
```

The rules simplify terminological constraints until a normal form is reached. In the normal form, the only constraints are `I:C, I:nota C, I:S or T, I:every R is S, (I,J):R`, where `C` is a primitive concept name.

To show termination we show that in each rule, all membership assertions in the body are strictly smaller than the one in the head. We prove this by mapping concept terms into numbers called ranks as follows:

$rank(nota\ T) = 2 * rank(T)$
$rank(T) = 1 + rank(S)$ if (`T isa S`) exists
$rank(f(T_1, \ldots, T_n)) = 1 + rank(T_1) + \ldots + rank(T_n) \quad (n \geq 0)$ otherwise

Note that by definition, concept terms are ground (variable-free) and finite and concept definitions are acyclic and finite.

The solver detects all inconsistencies through the clash rule independently of the order in which constraints are added and CHR are applied, because it is confluent. Since all CHR except the clash rule have pairwise disjoint heads at run-time, critical pairs can only exist with the clash rule. For example, the inconsistent constraints

```
I:nota every R is S, I:every R is S
```

can be simplified by pushing `nota` down in the first constraint

`I:some R is nota S, I:every R is S` $\longmapsto$ (some-rule)
`(I,J):R, J:nota S, I:every R is S` $\longmapsto$ (every-rule)
`(I,J):R, J:nota S, I:every R is S, J:S`

and now the clash rule can still be applied, to `J:nota S, J:S`.

Flexibility and Extensions

Attributes (also called features) are functional roles, i.e. their interpretation is a partial function. Assuming a declaration of an attribute F by a unary predicate `attribute` F, we just have to extend our implementation by

```
(I,J1):F, (I,J2):F  ==> attribute F | J1=J2.
```

Example 8.3. Now we are ready to define a simple configuration which consists of two distinguished simple devices:

```
attribute component_1.
attribute component_2.
simple_config isa configuration and
   some component_1 is simple_device and
   some component_2 is simple_device.
```

Then from the constraints

```
config1:simple_config, (config1,dev1):component_1,
   (config1,dev2):component_2,
```

the solver can derive that `dev1` and `dev2` are simple devices. The reason is that the attribute-rule constrains the witness for `some component_1 is simple_device` and the second argument of the role `(config1,dev1):component_1` to be equal (analogously for `dev2`).

In [FrHa95] we illustrate that other extensions to the basic terminological formalism proposed in the literature carry over to the implementation with CHR in a painless manner. One such extension allows parameterizing terminologies with *concrete domains*, e.g. linear constraints over rational numbers [BaHa91].

Related Solvers and Work

Related solvers where implemented [FrBr96] for various forms of feature trees, namely order sorted feature trees (OSF) [APG93], `osf.chr`, including the arity constraint [SmTr94b], `cft.chr`, as well as rational trees, `tree.chr`, including disequality.

ConTeS is a prototype implementation of an interactive, graphical tool supporting the configuration process of technical systems like process control systems developed by A. Wolf et al. at GMD FIRST, Berlin. ConTeS includes a knowledge base represented by an executable specification language, called TRLC. It is a generalization of the terminological reasoning language and its implementation described before. The first version of ConTeS was presented at the Leipziger Innovationsmesse in September 1996.

Other work looked at theorem proving with constraints where terminological reasoning was one domain of constraints considered. In CLP, proof procedures for Horn clauses are enhanced with efficient constraint solvers. The question arises whether it is possible to incorporate constraint processing into general, non-Horn theorem proving calculi. In the paper [StBa94], a positive answer is given. A new calculus is introduced which combines model elimination with constraint solving. A prototype system has been implemented rapidly by combining a Prolog technology implementation of model elimination with constraint solvers. Some example studies, e.g. terminological reasoning, show the advantages and some problems with this procedure. Using an extension of the terminological solver, the authors were able to solve the lion and unicorn puzzle in about 0.1s on a Sun4, which the authors consider to be quite fast.

8.3. Path Consistency

In this section we introduce a constraint solver that implements the classical Artificial Intelligence algorithm of path consistency and backtracking to solve constraint satisfaction problems.

Definition 8.3. A *binary constraint network* consists of a set of variables and a set of binary constraints between them. The network can be represented by a *directed constraint graph*, where the nodes denote variables and the arcs are labeled by binary constraints.

Definition 8.4. A *disjunctive binary constraint* c_{xy} between two variables X and Y, also written $X\ [r_1, \ldots, r_n]\ Y$, is a finite disjunction $(X\ r_1\ Y) \vee \ldots \vee (X\ r_n\ Y)$, where each r_i is a relation that is applicable to X and Y. The r_i are also called *primitive constraints.* The *converse* of a primitive constraint r between X and Y is the primitive constraint s that holds between Y and X as a consequence.

Usually, the number of primitive constraints is finite and they are pairwise disjoint. For simplicity, unary (domain) constraints are modeled as binary constraints where one variable is fixed.

For example, $A\ [<]\ B, A\ [<,>]\ B, A\ [<,=,>]\ B$ are disjunctive binary constraints c_{AB} between A and B. $A\ [<,>]\ B$ is the same as $A \neq B$, $A\ [<,=,>]\ B$ does not impose any restrictions on A and B, the constraint is redundant.

Definition 8.5. A *solution* of a constraint network is an assignment of values to the variables that satisfies all the constraints. Such an assignment is called *valid.* A constraint network is *consistent* if there exists a solution. A constraint network is *minimal* if each primitive constraint is satisfied in a solution of the network; i.e. there are no primitive constraints that do not participate in at least one solution.

Definition 8.6. A network is *path consistent* if for pairs of nodes (i, j) and all paths $i - i_1 - i_2 \ldots i_n - j$ between them, the direct constraint c_{ij} is tighter (or the same) than the indirect constraint along the path, i.e. the composition of constraints $c_{ii_1} \otimes \ldots \otimes c_{i_n j}$ along the path. A disjunctive constraint is *tighter* if it has less disjuncts.

Path consistency can be used to approximate the minimal network. It follows from the definition of path consistency that we can intersect the direct and indirect constraint to arrive at a tighter direct constraint. Let intersection be denoted by the operator $\oplus$. A graph is *complete* if there is an edge or a pair of arcs, one in each direction, between every pair of nodes. If the graph underlying the network is complete it suffices to repeatedly consider paths of length 2 at most: For each triple of nodes (i, k, j) we repeatedly compute $c_{ij} := c_{ij} \oplus c_{ik} \otimes c_{kj}$ until a fixpoint is reached. The complexity of such an algorithm is $O(n^3)$, where n is the number of nodes in the network [MaFr85].

For example, given $I[<,=]K \wedge K[<,=]J \wedge I[=,>]J$, and taking the triple (i, j, k), $c_{ik} \otimes c_{kj}$ results in $I[<,=]J$, the result of intersecting with c_{ij} is $I[=]J$. From (j, i, k) we get $J[=]K$ (we compute c_{ji} as the converse of c_{ij}). From (k, j, i) we get $K[=]I$. Another round of computation causes no more change, so the fixpoint is reached with $J[=]K, K[=]I$ (which is also minimal). Compare this result with the one using the solver in Section 2.

Solver

Let the constraint c_{ij} be represented by the predicate `c(I,J,C)` where `C` is the disjunction of primitive constraints forming the disjunctive constraint. The basic operation of path consistency, $c_{ij} := c_{ij} \oplus c_{ik} \otimes c_{kj}$, can be implemented by one rule performing the composition and another rule performing the intersection.

```
c(I,K,C1),c(K,J,C2) ==> composition(C1,C2,C3), c(I,J,C3).
c(I,J,C1),c(I,J,C2) <=> intersection(C1,C2,C3), c(I,J,C3).
```

As we will see, splitting into the two operations using two rules offers a high degree of flexibility. These two rules suffice to implement an incremental concurrent path consistency algorithm for complete networks. The rules are confluent for all properly defined (i.e. logically correct) composition and intersection operations.

Although for a given problem, there is only a finite number of variables and possible disjunctive binary constraints, the solver above is too generic to terminate under our operational semantics. The propagation rule can generate the same constraint(s) all over again, if intermediate constraints are not absorbed early enough by the simplification rule, as the following trace shows (new constraints are added to the right):

```
(1) c(X,Y,A), c(Y,X,B)                        % propagate with A and B
(2) c(X,Y,A), c(Y,X,B), c(X,X,C)              % propagate with B and C
(3) c(X,Y,A), c(Y,X,B), c(X,X,C), c(Y,X,D) % simplify B and D
```

```
(4) c(X,Y,A), c(X,X,C), c(Y,X,B)          % propagate with A and B
(5) c(X,Y,A), c(X,X,C), c(Y,X,B), c(X,X,C) % simplify C and C
(6) c(X,Y,A), c(Y,X,B), c(X,X,C)          % same as state (2)
```

In most CHR implementations, however, even this solver will terminate when the rules are applied fairly and idempotence is enforced (e.g. the new `c(X,X,C)` would be absorbed in state `(5)`). Fairness means here that simplification by intersection is applied to constraints over the same variable pair before too much propagation is caused by them. Then, any solver derived from this generic path consistency solver will terminate as well.

Generic path consistency solvers can be found in `path.chr` and `time-pc.chr`. The solver below takes the optimizations of algorithm PC-2 [Mac77] into account, but in addition is incremental, works with incomplete networks, removes redundant constraints and implements equality by the built-in constraint `=/2`. More optimizations are discussed in detail in [Fru94]. The solver maintains the invariant that `I<J` holds for each constraint `c(I,J,C)`, since in PC-2 converses of a constraint are no longer explicit.

```
% Special Cases
c(I,J,C) <=> ground(I),ground(J) | choose(B,C),check_c(I,J,B).
c(I,J,C) <=> empty(C) | false.
c(I,J,C) <=> redundant(C) | true.
c(I,J,C) <=> equality(C) | I=J.
c(I,I,C) <=> choose(B,C),equality(B).
% Intersection
c(I,J,C1),c(I,J,C2) <=> intersection(C1,C2,C3), c(I,J,C3).
% Composition
c(I,K,C1),c(K,J,C2) ==> I<J | composition(C1,C2,C3), c(I,J,C3).
c(K,I,C1),c(K,J,C2) ==> I<J | composition(C1,C3,C2), c(I,J,C3).
c(I,K,C1),c(J,K,C2) ==> I<J | composition(C3,C2,C1), c(I,J,C3).
% Labeling
label_with c(I,J,C) if not singleton(C).
c(I,J,C) :- choose(B,C), c(I,J,B).
```

The special cases are simplification CHR. The first checks the satisfiability of the constraint by trying the primitive constraints in the disjunction until one is found for which the assignment of the variables is valid. The next one detects inconsistent constraints (those having empty disjunctions), one replaces the equality constraint by the built-in constraint `=/2`, and one replaces a constraint between the same nodes by a test if equality was present in the disjunction[10]. The definitions of the auxiliary predicates `check_c, empty, singleton, redundant, equality, choose, intersection, composition` comes with the instance of the path consistency solver (see Section 8.4).

Another simplification CHR performs the intersection, three propagation CHR the composition. In the absence of explicit converses, the composition CHR have to cover all possible orientations of constraints while keeping the nodes `I,J` ordered.

[10]If there are primitive relations which properly contain equality, the rule has to be weakened into a propagation rule.

The computation of the converse is implicit in how the composition predicate is used, if necessary "computing backwards".

The labeling implements backtrack search to make complete the path consistency algorithm. If a disjunctive constraint C is not a singleton, one nondeterministically chooses a primitive constraint B from C and enforces B.

Flexibility and Extensions

The solver for path consistency can be specialized to one for arc consistency by restricting exactly one of the binary constraints involved in the propagation CHR to be actually unary. This is achieved by fixing one variable to a reference point, which is smaller than any variable (e.g. zero). For such a unary constraint `c(0,J,C)` we use the more common notation `dom(J,C)` (`C` is usually called the domain of `J`):

```
% Special Cases
dom(J,C) <=> ground(J) | choose(B,C),check_dom(J,B).
dom(J,C) <=> empty(C) | false.
dom(J,C) <=> redundant(C) | true.
% Intersection
dom(J,C1),dom(J,C2) <=> intersection(C1,C2,C3), dom(J,C3).
% Composition
dom(K,C1),c(K,J,C2) ==> composition(C1,C2,C3), dom(J,C3).
dom(K,C1),c(J,K,C2) ==> composition(C3,C2,C1), dom(J,C3).
% Labeling...
```

A related solver for arc consistency is `arc.chr`. We will use a further specialization of this solver for finite domains in Section 8.5. An instance of path consistency for temporal reasoning is introduced in the Section 8.4. More modifications are discussed in [Fru94].

Related Solvers and Work

An application of the path consistency and backtracking algorithm in CHR to qualitative spatial reasoning is described in [EsTo96]. The framework of Freksa and Zimmermann is implemented and extended by the treatment of 2-dimensional objects with non-zero dimensions. In this framework, space is qualitatively divided into several regions which are defined by means of a reference system. There are 15 primitive relations, which basically denote relative directions (e.g. left-front, behind). An important aspect of the work was that CLP extended with CHR provides a level of abstraction suited for integrating different aspects of space. The results of this research have been applied to toy examples and robot path planning. Current work by the same authors extends the solver (and framework) further to handle distances between objects.

8.4. Temporal Reasoning

Following the framework of Meiri [Mei91], temporal reasoning is viewed as a constraint satisfaction problem about the location of temporal variables along the time line using path consistency and backtrack search. The framework integrates most forms of temporal relations - qualitative and quantitative (metric) over time points and intervals - by considering them as disjunctive binary constraints. We quickly introduce the temporal constraints available.

Qualitative Point Constraints [ViKa86]. Variables represent time points and there are three primitive constraints `<, =, >`. Composition of a constraint with itself or equality yields the constraint again, any other composition yields the redundant constraint.

Quantitative Point Constraints [DMP91]. The primitive constraints restrict the distance of two time points X and Y to be in an interval $a:b$, i.e. $a \leq (Y-X) \leq b$[11], where a and b are signed numbers or ∞. Note that there is an infinite number of primitive quantitative constraints and that they can overlap. The composition of the intervals $a:b$ with $c:d$ results in $(a+c):(b+d)$, and the intersection in $max(a,c):min(b,d)$.

Interval Constraints [All83]. There are 13 primitive constraints possible between two intervals, equality and 6 other relations with their converses. These constraints can be defined in terms of the end-points of the intervals. Let `I=[X,Y]`, `J=[U,V]`. Notationally, we abbreviate chains of (in)equalities between variables.

```
I equals J if X=U<Y=V.        I before J if X<Y<U<V.
I during J if U<X<Y<V.        I overlaps J if X<U<Y<V.
I meets J if X<Y=U<V.         I starts J if X=U<Y<V.
I finishes J if U<X<Y=V.
```

Converses are`equals`,`after`,`contains`,`overlapped_by`,`started_by`,`finished_by`.

Point - Interval Constraints [Mei91]. There are 5 possible primitive constraints between a point and an interval. Let `X` be a point, `J = [U,V]` an interval.

```
X pbefore J if X<U<V.
X pafter J if U<V<X.          X pduring J if U<X<V.
X pstarts J if X=U<V.         X pfinishes J if U<X=V.
```

The converses express interval-point constraints.

Relating Constraints of Different Types [KaLa91]. Qualitative time point constraints can be mapped into quantitative point constraints, while quantitative constraints can only be approximated by qualitative constraints. Points can be represented by end-points of intervals and interval constraints can be approximated by constraints on their endpoints. These mappings are used to solve heterogeneous constraints over the same variables.

Solver

We instantiate the generic path consistency solver of the previous section by defining the intersection and composition operations. The implementation is described in detail and with variations in [Fru94], the solver is `time.chr` using `time-pc.chr`.

Disjunctive constraints are represented as list of their primitive constraints. Intersection is simply defined as list intersection, while composition is defined in terms

[11] For simplicity of presentation we do not distinguish between open and closed intervals.

of pairwise combining the primitive relations. The check for validity is performed by using the definition of the primitive temporal constraints as CLP clauses.

Since there is an infinite number of primitive quantitative constraints and since they can overlap, these constraints need special treatment: Intersection and composition have to deal with overlapping intervals. Labeling can go beyond single intervals by performing binary search on them: A single interval is split in half as long as its size is above a certain threshold `eps`. `eps` is a lower bound for the size of the smallest nonempty interval possible in the constraint problem at hand. Since such a lower bound always exists, termination is not affected [Fru97].

Example 8.4. The constraints on intervals `X, Y, Z`

```
c(X,Y,[pbefore,pstarts]), c(X,Z,[pstarts,pduring]),
c(Y,Z,[before,contains,after])
```

can be tightened by path consistency to

```
c(X,Y,[before]), c(Z,Y,[before]), c(X,Z,[starts,during]),
```

while the constraints on points `U, V` and on intervals `Y, Z`

```
c(V,U,[0-1,3-4]), c(U,Y,[pbefore,pstarts]),
c(Z,V,[pcontains,pstarted_by]), c(Y,Z,[before,contains])
```

turn out to be inconsistent.

Flexibility and Extensions

We specialize our temporal solver to quantitative time point constraints over single intervals as considered in [DMP91]. Their notation for `c(I,J,[A:B])` is `A=<I-J=<B`, meaning that the distance between `I` and `J` is between `A` and `B`. The solver can be found in `time-point.chr` and another derivation for it by extending the solver for inequality (section 2) is described in [Fru95].

```
% Special Cases
A=<I-J=<B <=> ground(I),ground(J) | A=<J-I, J-I=<B.
A=<I-J=<B <=> A>B | false.
A=<I-J=<B <=> A=-∞,B=∞ | true.
A=<I-J=<B <=> A=0,B=0 | I=J.
A=<I-I=<B <=> A=<0, 0=<B.
% Intersection
A1=<I-J=<B1,A2=<I-J=<B2 <=>
                      A3 is max(A1,A2),B3 is min(B1,B2),A3=<I-J=<B3.
% Composition
A1=<I-K=<B1,A2=<K-J=<B2 ==> I<J| A3 is A1+A2,B3 is B1+B2,A3=<I-J=<B3.
A1=<K-I=<B1,A2=<K-J=<B2 ==> I<J| A3 is A2-B1,B3 is B2-A1,A3=<I-J=<B3.
A1=<I-K=<B1,A2=<J-K=<B2 ==> I<J| A3 is A1-B2,B3 is B1-A2,A3=<I-J=<B3.
```

Labeling can be performed by interval splitting (binary search).

Related Solvers

PMON is one of the logics for modeling of dynamical systems presented in [San94]. Syntactically, a scenario description (a description of a dynamical system) in PMON consists of three parts: Observations (formulas that hold at specific time points), action laws (formulas that defines possible change of values of symbols), schedule statements (statements that describes occurrences of and the temporal relations between actions). Additionally there are nochange axioms that specify when a proposition cannot possibly change (to implement inertia). The basic idea of the implementation [Bja96] was to see formulas as constraints and encode the action laws as rules. Many ideas where borrowed from the CHR implementation of the Meiri framework, such as disjunctions handled by the labeling mechanism. Amongst other cases, various classical Turkey Shooting Problems were investigated.

The European Community funded ESPRIT project no. 2409, "Environment for Qualitative Temporal Reasoning" (EQUATOR), 1989-93, was concerned with modeling process-based systems for industrial applications like aircraft scheduling and urban traffic control. An extension of the event calculus [KoSe86, SaKo95] called GRF including time granularity (different time scales) and continuous processes, was implemented in several versions, one using CHR [Don93]. This version was constraint-based in several ways: It used an extension of a CHR solver for inequalities and finite domains (the interval part) for modeling temporal order. It also used a simplified version of the solver for linear equations for conversion between different time scales. It modeled negation as a CHR constraint to avoid floundering and achieve maximum propagation. Thus the predicates of the event calculus could be called even when the time parameter was unknown.

8.5. Finite domains

Finite domains appeared first in CHIP [vH89], more recent and more advanced CLP languages are clp(FD) [CoDi96] and cc(FD) [HSD95]. Since integers are used as domain, some arithmetic is possible. The theory underlying this constraint domain is Presburgers arithmetic. It axiomatizes the linear fragment of integer arithmetic and is decidable. The constraint `X::Dom` means that the value for the variable `X` must be in the given finite domain `Dom`. More precisely, if `Dom` is an

- *enumeration domain*, `List`, then `X` is a ground term[12] in the list `List`,
- *interval domain*, `Min:Max`, then `X` is a ground term between `Min` and `Max`.

The difference between an interval domain and an enumeration domain is that in the former constraint simplification is performed only on the interval bounds, while in the latter constraint simplification is performed on each element in the enumeration. Thus enumeration domains allow more constraint simplification but on the other hand are only tractable for sufficiently small enumerations.

We will derive our solver, `domain.chr`, as an instance of the arc consistency solver of Section 8.3 and time-point solver of Section 8.4. The latter already gives

[12] With CHR, there is no need for restricting the representation to integers.

us a partial solver for interval domains if we specialize it to `A=<0-J=<B` and write it as `J::A:B`. In this specialization, the treatment of equality changes and we add composition with binary constraints, as in the arc consistency solver:

```
% Special Cases
J::A:B <=> ground(J) | A=<J, J=<B.
J::A:B <=> A>B | false.
J::A:B <=> A=-∞,B=∞ | true.
J::A:B <=> A=B | B=J.
% Intersection
J::A1:B1,J::A2:B2 <=> A3 is max(A1,A2),B3 is min(B1,B2),J::A3:B3.
% Composition
K::C1,c(K,J,C2) ==> composition(C1,C2,C3), J::C3.
K::C1,c(J,K,C2) ==> composition(C3,C2,C1), J::C3.
```

One possible instance of `c(I,J,C)` is the constraint `I=<J`:

```
K::A:B, K=<J ==> J::A:∞.
K::A:B, J=<K ==> J::−∞:B.
```

If an argument is known, the two rules can be strengthened to simplification rules by projection onto the other argument:

```
K=<J <=> ground(K) | J::K:∞.
J=<K <=> ground(K) | J::−∞:K.
```

For example, from `X::1:2.5, Y::2.5:3, Y=<X` we get `X=2.5,Y=2.5` by applying the rules for composition and intersection yielding `X::2.5:2.5`, simplifying it to an equality, projecting the inequality on `Y`, then intersecting and simplifying again.

For enumeration domains, we specialize the arc consistency solver:

```
% Special Cases
X::L <=> ground(X) | member(X,L).
X::[] <=> false.
X::[Y] <=> X=Y.
% Intersection
X::L1,X::L2 <=> L1=[_|_], L2=[_|_] | intersect_list(L1,L2,L),X::L.
% Labeling
label_with X::[Y,Z|L] if true.
X::[Y|L]:- member(X,[Y|L]).
```

Flexibility and Extensions

CHIP finite domains included n-ary arithmetic constraints (linear polynomials) and constraints such as `alldifferent, circuit, atmost, element`. In the solver `domain.chr` we implemented a version of `element` constraint which has lower complexity than in CHIP by introducing path consistency for this constraint. This makes sense, since the constraint is binary. It can be seen as an enumeration domain over pairs, `I-V`. Therefore we simply reused the special cases and intersection of normal unary enumeration domains but also introduce some new special cases.

Sample rules for arithmetic constraints are (see also Section 9.2):

```
addz @ X+Y equal Z, X::MinX:MaxX, Y::MinY:MaxZ ==>
    MinZ is MinX+MinY, MaxZ is MaxX+MaxY, Z::MinZ:MaxZ.
addy @ X+Y equal Z, X::MinX:MaxX, Z::MinZ:MaxZ ==>
    MinY is MinZ-MaxX, MaxY is MaxZ-MinX, Y::MinY:MaxY.
```

For example:

	`A::1:3, B::2:4, C::0:4, A+B equal C`
$\mapsto_{addz}$	`A::1:3, B::2:4, C::0:4, A+B equal C, C::3:7`
$\mapsto_{intersection}$	`A::1:3, B::2:4, A+B equal C, C::3:4`
$\mapsto_{addx}\mapsto_{addy}$	`A::1:3, B::2:4, A+B equal C, A::-1:2, B::0:3, C::3:4`
$\mapsto^2_{intersection}$	`A+B equal C, A::1:2, B::2:3, C::3:4`

Related Solvers

In the work [Due96] structural character descriptions for east Asian ideograms (Kanji) are both analyzed and generated. Sketches of characters can be produced from a symbolic coordinate free description, when the description is interpreted as a system of constraints. However, the constraints are highly underdetermined, as there is no exact geometry information, and sometimes implicit, such as the condition that the final sketch has to fill a square of fixed size. Therefore a special constraint solving algorithm tailored to the problem was developed.

An initial solution was rewritten using the finite domain constraint solver of the CHR library. According to the author, CHR lead to improvements in performance, allowing generating sketches for characters with ten or more equivalence classes in one direction. This was not feasible with the original solution that heavily relied on the generate-and-test approach of LP.

8.6. Linear (and Non-Linear) Polynomials

The initial motivation for introducing constraints in LP was the non-declarative nature of the built-in predicates for arithmetic computations. Therefore, from the very beginning, CLP languages included constraint solving for linear equations and inequations over reals (CLP(R) [Ja*92]) or rationals (Prolog-III [Col90], CHIP [Di*88]) adopting variants of Gaussian elimination and the Simplex algorithm [Imb95]. The theory underlying this constraint system is that of real closed fields, which covers linear and non-linear polynomials and was shown to be decidable by Tarski.

In the CHR solver `math-gauss.chr` a minimalistic but powerful variant of variable elimination is employed. A linear polynomial is represented as `Poly equals Constant` where `Poly` is a list of monomials of the form `Variable * Coefficient` with coefficients different from zero and the list is sorted on the variables in strictly descending order. The two rules below suffice to implement a complete and efficient solver for linear equations over both floating point numbers and rational numbers.

```
empty @ [] equals K <=> zero(K).
```

```
eliminate @ [X*C1|P1] equals K1 \ [X*C2|P2] equals K2 <=>
        multiply_poly_const(P1-K1,C2/C1,P3-K3),
        subtract_poly_poly(P2-K2,P3-K3,P4-K4),
        P4 equals K4.
```

The `empty` rule says that if the polynomial is empty, the constant must be zero. The predicate `zero` tests for zero with a user-definable error margin in case of a floating point number. The `eliminate` rule is the workhorse that performs the variable elimination. It takes two equations that start with the same variable, the first equation is left unchanged, it is used to eliminate the occurrence of the common variable in the second equation. Note that no variable is ever made explicit, i.e. no pivoting is performed: Any two equations with the same first variable can react with each other.

The solver terminates since the polynomial is ordered and a large variable is replaced by several strictly smaller ones. The solver is complete since it results in a normal form where the left-most variable of each equation is the only left-most occurrence of this variable. However, it does not create explicit variable bindings or necessarily make implicit equalities between variables explicit.

Even though the solver is not confluent (any of the two equations in the rule `eliminate` could be chosen for eliminating its variable, resulting in different new equations), it could be easily made so by introducing an order on equations. The rule is more efficient as it is, and the result in terms of satisfiability and variables that are uniquely determined are the same.

Flexibility and Extensions

Bindings of variables are introduced as special cases by the rules:

```
unify @ [X*C] equals K <=> X is K/C.
unified @ P equals K <=> delete(X*C,P,P1), ground(X) |
        K1 is K-X*C,
        P1 equals K1.
```

A more eager variant of the `eliminate` rule is possible, that eliminates a variable no matter where it occurs in the equation.

```
eager @ [X*C1|P1] equals K1 \ P equals K2 <=> delete(X*C2,P,P2) |
        % rule body as in rule eliminate
```

The rule makes all implicit equalities explicit. The remarks about termination and confluence of the solver still apply. On an equation solving benchmark proposed by Van Caneghem[13], using rational numbers the above solvers were slightly faster than the lower level implementation of a rational solver in ECL^{i}PS^{e}3.5.1. It solved a system of 50 variables and 50 equations in less than a minute on a 50MHz SUN SPARC. However our solver does not implement optimization and variable projection.

As in the Simplex algorithm, an inequation is handled by replacing it with an equation with the help of an additional variable, called a slack variable, that is constrained to be positive. Then one has to introduce additional rules that maintain

[13] Solving $Ax = b$ for x where A is a dense matrix with $A[i,j] = i^j \bmod 101$.

a normal form for equations that consist only of slack variables. This normal form is more constrained than the standard one. The slack variables have to be reordered such that the left-most slack variable of an equation has the same sign as the constant. If this is not possible, the equations are inconsistent. Also, if all slack variables have the same sign and the constant is zero, then all slack variables must be zero. The solver can be found in `math-elim.chr`.

Another solver, `math-fougau.chr`, is the result of combining the above solver for equations with a solver performing the classical Fourier algorithm for inequations. The idea is to perform variable elimination as long as at least one equation is involved in the process, otherwise - in the case of two inequations - the transitivity rule (i.e. propagation) as suggested by Fourier is used. The combined solver is more efficient than Fouriers algorithm alone and avoids the introduction of slack variables.

Related Solvers and Work

GroAK [MRS96] is a CLP system over non-linear polynomial constraints which appear e.g. in geometric reasoning. Before, techniques like Groebner Bases over complex numbers (CAL [Ai*88]) and Partial Cylindrical Algebraic Decomposition (RISC-CLP(Real) [Hon92]) have been utilized to tackle non-linear polynomials. Another approach is to use interval arithmetic as in CLP(BNR) [Ben95], `Newton` [BMH94] and Numerica [HMD97]. This approach can basically be seen as a sophisticated extension of interval domains to the reals and to non-linear polynomials.

Instead of using a general and often inefficient decision procedure, **GroAK** handles these constraints by cooperation of specialized solvers. This approach requires the design of a client-server architecture to enable communication between the various components and solvers. CHR are used to introduce the constraints and to plan the distribution of constraints to the solvers.

Each solver works on a special domain, with specific constraints: In order to treat the linear constraints, **GroAK** uses the CHR equation solver `math-elim.chr` with rational numbers. GB [Fau94], a software for fast Gröbner bases computation, yields a canonical form of the non-linear constraints from which the solutions can be extracted. The symbolic computation software Maple [GGL91] is used to compute the roots of univariate polynomials. Maple also simplifies polynomials before they are treated by the other solvers.

9. APPLICATIONS

We present two innovative, non-standard uses of constraint techniques, that characterize a large class of potential applications. The necessary constraint handling was expressed and implemented with ease in CHR. Simplicity, flexibility, efficiency and rapid prototyping were the advantages of using CHR. The applications were done at the European Computer-Industry Research Center (ECRC) with the collaboration from visitors, other research institutions and industry.

9.1. Planning Cordless Business Communication Systems

Mobile communications comes to company sites. Employees can be reached at any time at any place. No cabling is required, but small, local radio transmitters (senders) have to be installed. When planning their locations, the specifics of radio wave propagation have to be taken into account. Since radio waves are absorbed and reflected by walls and floors of a building, the received power at a single point may exhibit discontinuities because of tiny changes in the sender location - for example, a move around the corner.

The advanced industrial prototype POPULAR (Planning of Picocellular Radio) [Mol94, FMB96, FrBr97], one of the first systems of its kind, computes the minimal number of senders and their location, given a blue-print of the building and information about the materials used for walls and ceilings. It does so by simulating the propagation of radio-waves using ray tracing and subsequent constraint-based optimization of the number of senders needed to cover the whole building. POPULAR was developed by ECRC, Siemens Research and Development (ZFE), the Siemens Personal Networks Department (PN), and the Institute of Communication Networks at the Aachen University of Technology.

First, the characteristics of the building are computed using a grid of test points. Each test point represents a possible receiver position. For each test point the space where a sender could be put to cover the test point, the "radio cell", is calculated. The radio cell will usually be a rather odd-shaped object, since the coverage is not a smooth or even differentiable function. If the test grid is sufficiently small (several per square meter), we can expect that if two neighbouring test points are covered, the space inbetween - hence the whole building - can also be covered.

For each radio cell a constraint is set up that there must be (at least) one location of a sender (geometrically speaking, a point) somewhere in that space. Then, we try to find locations that are in as many radio cell planes at the same time as possible. Thus the possible locations are constrained to be in the intersections of the radio cell planes covered. A sender at one of these locations will cover several test points at once. In this way, a first solution is computed. To minimize the number of senders, we use a branch-and-bound method. It consists in repeatedly searching for a solution with a smaller number of senders until the minimal number is found.

Solver

In a first attempt restricted to two dimensions, we approximated the radio cell by a single rectangle. The 2-D coordinates are of the form `X#Y`, rectangles are orthogonal to the coordinate system and are represented by a pair of their left upper and right lower corner coordinates. For each radio cell, a constraint `inside(Sender, Rectangle)` is imposed, where `Sender` is a point that must be inside `Rectangle`.

```
% inside(Sender, LeftLowerCorner - RightUpperCorner)
nonempty @ inside(S,A#B-C#D) ==> A<C,B<D.

intersect @ inside(S,A1#B1-C1#D1),inside(S,A2#B2-C2#D2) <=>
        A is max(A1,A2), B is max(B1,B2),
        C is min(C1,C2), D is min(D1,D2),
        inside(S,A#B-C#D).
```

The first rule (named `nonempty`) says that the constraint `inside(S,A#B-C#D)` is only valid if also the condition `A<C,B<D` is fulfilled, so that the rectangle has a nonempty area. The `intersect` rule says that if a senders location `S` is constrained

by two `inside` constraints to be in two rectangles at once, we can replace these two constraints by a single `inside` constraint whose rectangle is computed as the intersection of the two initial rectangles.

To compute a solution, we try to equate as many senders as possible using the following labeling procedure:

```
equate_senders([]).
equate_senders([S|L]) :-
        (member(S,L) ; true),  % equate S with another sender or not
        equate_senders(L).
```

For each sender `S`, `(member(S,L) ; true)` nondeterministically equates `S` with one of the remaining senders in the list `L` using `member` or does not do so (`true`). Equating senders causes the `intersect` rule to fire with the constraints associated with the senders. As a result of this labeling procedure, a senders location will be constrained more and more and thus the `intersect` rule will be applied again and again until the rectangle becomes very small and finally empty. Then the `nonempty` rule applies, causes failure and so initiates backtracking. A good labeling heuristic is to equate senders from radio cells associated with nearby test points first.

It took just 10 minutes to extend this solver so that it works with union of rectangles, that can describe the radio cell to any desired degree of precision. This corresponds to a disjunctive constraint `inside(S,R1)` ∨ `...` ∨ `inside(S,Rn)` which is more compactly implemented as `inside(S,[R1,...,Rn])`.

```
% inside(Sender, List)
intersect @ inside(S, L1), inside(S, L2) <=>
        intersect(L1, L2, L3),
        L3 = [_|_],          % at least one rectangle left
        inside(S, L3).

intersect(L1, L2, L3) :-
        setof(R, intersect1(L1,L2,R), L3).

intersect1(L1, L2, rect(A#B,C#D)) :-
        member(rect(A1#B1,C1#D1), L1),
        member(rect(A2#B2,C2#D2), L2),
        A is max(A1,A2), B is max(B1,B2),
        C is min(C1,C2), D is min(D1,D2),
        A<C, B<D. % nonempty
```

The above solver can be adapted quickly to work with other geometric objects than rectangles by changing the definition of `intersect1/3`. Also, the lifting to three dimensions just amounted to adding a third coordinate and code analogous to the one for the other dimensions. The simplicity of the solver does not mean primitiveness or triviality, it rather illustrates the power of CHR.

It would be quite hard to implement the functionality in a hard-wired black-box solver. With finite domains coordinates would have to be rounded to integers. Also, we found that for our application the built-in finite domain solver of ECL^{i}PS^e was slightly slower than the CHR implementation. Using linear polynomial constraints would be an overkill and thus inefficient, too. Interval arithmetic can express the required constraints more adequately. Moreover, the disjunctive constraints needed would require recasting using auxiliary variables, which is expensive, error-prone and limits the amount of propagation. The cardinality constraint [HSD95] could be used to express the disjunction, but is only available for finite domains.

Evaluation

For a typical office building, an optimal placement is found by POPULAR within a few minutes. The overall quality of the placements produced is comparable to that of a human expert. The only other comparable tool that was available in 1994 was WISE [FGK*95], which is written in about 7500 lines of C++. For optimization WISE uses an adaptation of the Nelder-Mead direct search method that optimizes the percentage of the building covered. The CLP code for POPULAR is just about 4000 lines with more than half of it for graphics and user interface. The big advantage of the CLP approach is flexibility, e.g. when changing the labeling heuristic or extending the solver.

9.2. The Munich Rent Advisor

The Munich Rent Advisor (MRA) [FrAb96], developed by ECRC and LMU, is the electronic version of the "Mietspiegel"(MS) for Munich. MS are published regularly by German cities. They are basically a written description of an expert system that allows to estimate the maximum fair rent for a flat. These estimates are legally binding.

The calculations are based on size, age and location of the flat and a series of detailed questions about the flat and the house it is in. Some of these questions are hard to answer. However, in order to be able to calculate the rent estimate by hand, all questions must be answered. Usually, the calculation is performed by hand in about half on hour by an expert from the City of Munich or from one of the renter's associations. The MRA that brought the advising time down to a few minutes that the user needs to fill in the form. Using constraints, the user of the MRA need not answer all questions. The user may not want to give information away, or he does not care about the question or know the answer.

The MS is derived from a statistical model compiled from sample data using statistical methods such as regression analysis [Al*94]. Due to the underlying statistical approach, there is the problem of inherent imprecision which is ignored in the paper version of the MS. Using constraints the MRA can account for the statistical imprecision.

The MRA is available on the internet. Using the World-Wide-Web (WWW), there is no need for the user to acquire specific software and computer handling skills. To process the answers from the questionnaire and return its result, we wrote a simple stable special-purpose web-server directly in ECL^{i}PS^e using its C-sockets for internet communication. This approach avoids the overhead of CGI interfaces.

Solver

From a CLP point of view, the MRA application is rather atypical: The computation proceeds deterministically from constrained input variables (the user data) to constrained output variables (the rent estimate), since the original MS has already solved the problem. There is no need for NP-hard constraint solving and labeling, only for constraint propagation in the forward direction: The answer we expect is the smallest interval covering all possible rents, not an enumeration of all possible rents by backtracking.

Our approach was first to implement the tables, rules and formulas of the "Mietspiegel" with high-level and declarative programming in ECL^{i}PS^e, as if the provided data was precise and completely known. Then we added constraints to capture the imprecision due to the statistical approach and incompleteness due to partial user answers. Finally, we considered the formulas of the rent calculation as constraints that refine the rent estimate by propagation from the input variables which are constrained by the partial answers.

In the MRA, dealing with imprecise numerical information involves non-linear arithmetic computations with intervals. We simply modified the existing finite domain solver in CHR, `domain.chr`, described in Section 8.5, so that it can deal with interval constraints over non-linear equations of the form

$$c * X_1 * X_2 * ... * X_n = Y$$

where c is a number and the X_i and Y are different variables and $n >= 0$. In the solver, $c * X_1 * X_2 * ... * X_n = Y$ is represented by `mult(C:C,[X1,X2,...,Xn],Y)`.

```
mult(Min:Max, [], Y) <=> Y::Min:Max.

mult(Min:Max, [X|L], Y) <=> number(X) |
    NewMin is min(Min*X,Max*X),
    NewMax is max(Min*X,Max*X),
    mult(NewMin:NewMax, L, Y).

X::XMin:XMax \ mult(Min:Max, [X|L], Y) <=>
    NewMin is min(min(Min*XMin,Max*XMax),min(Max*XMin,Min*XMax)),
    NewMax is max(max(Min*XMin,Max*XMax),max(Max*XMin,Min*XMax)),
    mult(NewMin:NewMax, L, Y).
```

Since we do not need backpropagation in our application, these three rules suffice.

Evaluation

In the last two years, more than ten thousand people haved used our MRA service on the World-Wide-Web (WWW). It is one of the winners of the best application prize of the JFPLC'96 [FAB96] conference in France and was presented at the Systems'96 Computer Show in Munich.

It took about four man weeks to write the WWW user interface, only two weeks to write the calculation part and one week to debug it. We think that the coding would have dominated the implementation effort if a conventional programming language had been used. We could presumably have used interval arithmetic to express the required constraints. However it would have been quite difficult to tailor the amount and direction of constraint propagation to the needs of the application at hand. Our high-level approach also implies that the program can be easily maintained and modified. This is crucial, since every city and every new version comes with different tables and rules for the "Mietspiegel".

The Munich Rent Advisor represents a class of applications that is rather atypical for constraint logic programming, since it is not concerned with the NP-hard constraint-pruned search for a solution, but executing an existing calculation in the presence of partial information. Nevertheless CLP can deal with imprecise knowledge and partial information in an elegant, correct and efficient way, provided it is possible to adopt the constraints to the application. We think that constraint technology can be applied to many engineering applications where one wants to reason with partial information without compromising correctness.

10. CONCLUSIONS

We gave syntax and semantics as well as soundness and completeness results for CHR. We introduced an important property for constraint solvers, confluence, and a decidable, necessary and sufficient test for it. CHR have been used to encode a wide range of solvers, including new domains such as terminological and temporal reasoning. We gave an overview of several solvers, showed how they can be extended or modified and mentioned related work that builds on these solvers.

While existing solvers are usually about datastructures and their operations (e.g. finite domains, Booleans, numbers), CHR open the way for more generic (e.g. path consistency) and more conceptual constraint solvers (e.g. temporal, spatial and terminological reasoning). CHR have been used successfully in challenging applications, where other existing CLP systems could not be applied with the same results in terms of simplicity, flexibility and efficiency. In most real-life applications, soft and dynamic constraints are required. Work that has just been started [Wol97] indicates that CHR are helpful in implementing general schemes to handle such constraints independent of the constraint domain.

The topics for research mentioned in the first draft paper on CHR in 1991 were:

- Correctness w.r.t. specifications
- Termination and confluence
- Negation and entailment of constraints
- Combination and communication of solvers
- Debugging of constraint solvers
- Soft constraints with priorities
- Automatic labeling
- Dynamic constraints, removable constraints
- Variable projection
- Partial evaluation
- Abstract interpretation

Most of these topics are still an issue today. Clearly the termination property is even more important than confluence and has to be a topic of future research (for a start see the long version of this article, [Fru97]). While CHR solve conjunctions of constraints, other operations typically expected from a constraint solver like variable projection and entailment have not been investigated yet (except [Fru93a]).

We think that this survey illustrated that languages like CHR can fulfill the promise of user-defined constraints as described in [ACM]: "For the theoretician meta-theorems can be proved and analysis techniques invented once and for all; for the implementor different constructs (backward and forward chaining, suspension, compiler optimization, debugging) can be implemented once and for all; for the user only one set of ideas need to be understood, though with rich (albeit disciplined) variations (constraint systems)."

Acknowledgements

I would like to thank my collaborators: P. Brisset, T. Fortin, P. Blenninger, all of them visitors to ECRC; especially S. Abdennadher, also H. Meuss, M. Marte, all colleagues at LMU; and P. Hanschke, R. Mollwitz, Ch. Holzbaur.

Many people have discussed CHR with me, contributed with comments and used them. Too many to thank them all by name. However, I would like to mention my colleagues at ECRC: M. Wallace, T. Le Provost, V. Kuechenhoff, C. Gervet, E. Monfroy, all from the constraint team, and J. Schimpf, A. Herold, J. Traeff and N. Eisinger. While at ECRC from 1991 to 1996, my work on CHR was partially supported by ESPRIT Project 5291 CHIC.

Last but not least, I thank my wife Andrea and my daughter Anna for her ongoing support and patience.

The CHR papers and solvers mentioned in this article are available from URL `http://www.pst.informatik.uni-muenchen.de/personen/fruehwir/`.

REFERENCES

[Abd97.] S. Abdennadher, Operational Semantics and Confluence of Constraint Propagation Rules, 3rd Intl Conf on Principles and Practice of Constraint Programming (CP'97), Linz, Austria, Springer LNCS 1330, pp 252-265, October/November 1997.

[Abd98.] S. Abdennadher, Analyse von regelbasierten Constraintlösern (in German), Ph.D. Thesis, Dept of Computer Science, LMU Munich, to appear February 1998.

[ACM.] The Constraint Programming Working Group, ACM-MIT SDRC Workshop, Report Outline, Draft, September 1996.

[Ai*88.] A. Aiba et al, Constraint Logic Programming Language CAL, Intl Conf on Fifth Generation Computer Systems, 1988, Ohmsha Publishers, Tokyo, pp 263-276.

[AiNa86.] H. Ait-Kaci and R. Nasr, Login: A Logic Programming Language with Built-In Inheritance, Journal of Logic Programming 3:185-215, 1986.

[Al*94.] R. Alles et. al, Gutachten zur Erstellung des Mietspiegels für München '94 (in German), Sozialreferat der Stadt München - Amt für Wohnungswesen et. al, City of Munich, Germany, 1994.

[All83.] J. F. Allen, Maintaining Knowledge about Temporal Intervals, Communications of the ACM, Vol. 26, No. 11, 1983, pp 823-843.

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

[APG93.] H. Ait-Kaci, A. Podelski and S. C. Goldstein, Order-Sorted Feature Theory Unification, DEC PRL Research Report 32, May 1993, DEC Paris Research Laboratory, France.

[BaHa91.] F. Baader and P. Hanschke. A scheme for integrating concrete domains into concept languages. 12^{th} Intl Joint Conf on Artificial Intelligence, 1991.

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

[BDS93.] M. Buchheit, F. M. Donini and A. Schaerf, Decidable Reasoning in Terminological Knowledge Representation Systems, Journal of Artificial Intelligence Research, 1:109-138, 1993.

[BeOl92.] F. Benhamou and W.J. Older, Bell Northern Research, Applying interval arithmetic to Integer and Boolean constraints, Technical Report, June 1992.

[Bja96.] M. Bjareland, Proving Consistency in K-IA Chronicles — An Implementation of PMON, Master Thesis, Dept of Information and Computer Science, Linkoepings Universitet, 1996.

[Ben95.] F. Benhamou, Interval constraint logic programming, Chapter in Constraint Programming: Basics and Trends, (A. Podelski, Ed.), Springer LNCS 910, March 1995.

[BMH94.] F. Benhamou, D. MacAllester, and P. van Hentenryck, CLP(Intervals) Revisited, ILPS'94, MIT Press, Cambridge, Mass., USA, 1994.

[CoDi96.] P. Codognet and D. Diaz, Compiling constraints in `clp(fd)`, Journal of Logic Programming, 27(3), 1996.

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

[Col90.] A. Colmerauer, An Introduction to Prolog III, Communications of the ACM 33(7):69-90, July 1990.

[DaPu60.] M. Davis and H. Putnam, A Computing Procedure for Quantification Theory, Journal of the ACM 7(3):201-215, 1960.

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

[Di*88.] M. Dincbas et al., The Constraint Logic Programming Language CHIP, Fifth Generation Computer Systems, Tokyo, Japan, December 1988.

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

[Don93.] G. Dondosolla, A constraint-based implementation of the GRF, ESPRIT Project no. 2409 EQUATOR Report, June 1993.

[DOS88.] N. Dershowitz, N. Okada, and G. Sivakumar, Confluence of conditional rewrite systems, 1st CTRS, pp 31–44, Springer LNCS 308, 1988.

[Due96.] M. Duerst, Prolog for Structured Character Description and Font Design, Journal of Logic Programming, Special issue on Applications, (Leon Sterling, Ed.), February 1996.

[Dum95.] E. Dumbill, Application of resolution and backtracking to the solution of constraint satisfaction problems, Project Report, Dept of Computer Science, University of York, England, 1995.

[EsTo96.] M. T. Escrig and F. Toledo, Qualitative Spatial Orientation with Constraint Handling Rules, ECAI 96, Budapest, Hungary, (W. Wahlster, Ed.), John Wiley & Sons, August 1996.

[FAB96.] T. Frühwirth, S. Abdennadher and P. Blenninger, Rent Estimates with Constraints over the Internet - System Description, Journees Francophones de Programmation en Logique et programmation par Contraintes (JFPLC'96), Clermont Ferrand, France, June 1996.

[FAM96.] T. Frühwirth, S. Abdennadher and H. Meuss, On Confluence of Constraint Handling Rules, 2nd Intl Conf on Principles and Practice of Constraint Programming (CP'96), Cambridge, USA, Springer LNCS 1118, August 1996.

[FAM97.] T. Frühwirth, S. Abdennadher and H. Meuss, Confluence and Semantics of Constraint Handling Rules, Constraint Journal, Kluwer Academic Publishers, accepted for publication, to appear 1998.

[Fau94.] J-C. Faugere. Résolution des systèmes d'équations algébriques. PhD thesis, Université Paris 6, 1994.

[FGK*95.] S. J. Fortune, D. M. Gay, B. W. Kernighan et al., WISE Design of Indoor Wireless Systems, IEEE Computational Science and Engineering, Vol. 2, No. 1, pp 58-68, Spring, 1995.

[FMB96.] T. Frühwirth, J.-R. Molwitz and P. Brisset, Planning Cordless Business Communication Systems, IEEE Expert Magazine, Special Track on Intelligent Telecommunications, February 1996.

[Fr*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.

[FrAb96.] T. Frühwirth and S. Abdennadher, The Munich Rent Advisor, 1st Workshop on Logic Programming Tools for Internet Applications at JICSLP'96, Bonn, Germany, September 1996.

[FrAb97.] T. Frühwirth and S. Abdennadher, Constraint-Programmierung (in German), Textbook, Springer Verlag, Heidelberg, Germany, September 1997.

[FrBr95a.] T. Frühwirth and P. Brisset, High-Level Implementations of Constraint Handling Rules, Technical Report ECRC-95-20, ECRC Munich, Germany, June 1995.

[FrBr95b.] T. Frühwirth and P. Brisset, Chapter on Constraint Handling Rules, in ECLiPSe 3.5.1 Extensions User Manual, ECRC Munich, Germany, December 1995.

[FrBr96.] T. Frühwirth and P. Brisset, The CHR Library, Version 2, of ECL^{i}PS^e 3.5.3, released by ECRC, Munich, Germany, January 1996.

[FrBr97.] T. Frühwirth and P. Brisset, Optimal Planning of Digital Cordless Telecommunication Systems, 3rd Intl Conf on The Practical Application of Constraint Technology (PACT97), London, U.K., April 1997.

[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, Cambridge, Mass., USA, April 1995.

[Fru91.] T. Frühwirth, Introducing Simplification Rules, Workshop Logisches Programmieren, Goosen/Berlin, Germany, October 1991, Workshop on Rewriting and Constraints, Dagstuhl, Germany, October 1991, also Technical Report ECRC-LP-63, ECRC Munich, Germany, October 1991.

[Fru92.] T. Frühwirth, Constraint Simplification Rules, JICSLP'92 Workshop on Constraint Logic Programming, Washington D.C., USA, November 1992, also Technical Report ECRC-92-18, ECRC Munich, Germany, July 1992.

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

[Fru93b.] T. Frühwirth, CHR Constraint Handling, Abstract and Poster, Intl Conf on Logic Programming, (ICLP'93), Budapest, Hungary, MIT Press, Cambridge, Mass., USA, June 1993.

[Fru94.] T. Frühwirth, Temporal Reasoning with Constraint Handling Rules, Technical Report ECRC-94-05, ECRC Munich, Germany, February 1994.

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

[Fru97.] T. Frühwirth, A Declarative Language for Constraint Systems - Theory and Practice of Constraint Handling Rules, Habilitation, Dept of Computer Science, LMU Munich, December 1997.

[GGL91.] K. O. Geddes, G. H. Gonnet, and B. L. Leong, Maple V : Language Reference Manual, Springer, New York, Berlin, Paris, 1991.

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

[Her93.] B. Herbig, Eine homogene Implementierungsebene für einen hybriden Wissensrepräsentationsformalismus (in German), Master Thesis, University of Kaiserslautern, Germany, April 1993.

[Hol92.] C. Holzbaur, Metastructures vs. Attributed Variables in the Context of Extensible Unification, Intl Symposium on Programming Language Implementation and Logic Programming (PLILP'92), pp 260–268, Springer LNCS 631, August 1992.

[Hon92.] H. Hong, Non-linear Real Constraints in Constraint Logic Programming, Algebraic and Logic Programming Conf (Volterra, Italy), (H. Kirchner and G. Levi, Eds.), Springer LNCS 632, 1992, pp 201-212.

[HMD97.] P. van Hentenryck, L. Michel and Y. Deville, Numerica: a Modeling Language for Global Optimization, MIT Press, Cambridge, Mass., USA, 1997.

[HSD95.] P. van Hentenryck, Vijay A. Saraswat, and Y. Deville, Constraint Processing in cc(FD), Chapter in Constraint Programming: Basics and Trends, (A. Podelski, Ed.), Springer LNCS 910, 1995.

[Imb95.] J.-L. J. Imbert, Linear Constraint Solving in CLP-Languages, Chapter in Constraint Programming: Basics and Trends, (A. Podelski, Ed.), LNCS 910, March 1995.

[Ja*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, ILPS 91, San Diego, USA.

[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 19,20:503-581, 1994.

[KaLa91.] H. A. Kautz and P. B. Ladkin, Integrating Metric and Qualitative Temporal Reasoning, AAAI 91, pp 241-246.

[KiKi91.] C. Kirchner and H. Kirchner, Rewriting: Theory and Applications, North-Holland, 1991.

[KOM87.] T. Kawamura, H. Ohwada and F. Mizoguchi, CS-Prolog: A Generalized Unification Based Constraint Solver, 6th Conf on Logic Programming (Tokyo, Japan, June 1987), (K. Furukawa et al., Eds.), Springer LNCS 319, pp 19-39.

[KoSe86.] R. Kowalski and M. Sergot, A Logic-based Calculus for Events, New Generation Computing 4, 1986.

[Lel88.] W. Leler, Constraint Programming Languages: Their Specification and Generation, Addison-Wesley, Reading, Massachusetts, 1988.

[Mac77.] A. K. Mackworth, Consistency in Networks of Relations, Journal of Artificial Intelligence 8:99-118, 1977.

[MaFr85.] A. K. Mackworth and E. C. Freuder, The Complexity of Some Polynomial Network Consistency Algorithms for Constraint Satisfaction Problems, Journal of Artificial Intelligence 25:65-74, 1985.

[Mah87.] M. J. Maher, Logic Semantics for a Class of Committed-Choice Programs, 4th Intl Conf on Logic Programming, Melbourne, Australia, pp 858-876, MIT Press, Cambridge, Mass., USA, 1987.

[Me*93.] S. Menju et al., A Study on Boolean Constraint Solvers, Constraint Logic Programming: Selected Research, (F. Benhamou and A. Colmerauer, Eds.), MIT Press, Cambridge, Mass., USA, 1993.

[Mei91.] I. Meiri, Combining Qualitative and Quantitative Constraints in Temporal Reasoning, AAAI 91, pp 260-267.

[Mol94.] J.-R. Molwitz, Entwicklung eines Werkzeugs zur Planung picozellularer Funknetze für die drahtlose Kommunikation (in German), Master Thesis, University of Technology, Aachen, Germany, June 1994.

[MRS96.] E. Monfroy, M. Rusinowitch, and R. Schott, Implementing Non-Linear Constraints with Cooperative Solvers, (K. M. George et al., Eds.), ACM Symposium on Applied Computing (SAC'96), Philadelphia, PA, USA, pp 63–72, ACM Press, February 1996.

[Nai85.] L. Naish, Prolog Control Rules, 9th Intl Joint Conf on Artificial Intelligence, Los Angeles, California, September 1985, pp 720-722.

[New42.] M. H. A. Newman, On Theories with a Combinatorial Definition of Equivalence, *Annals of Math*, Vol. 43, pp 223–243, 1942.

[SaKo95.] F. Sadri and R. Kowalski, Variants of the Event Calculus, 12th Intl. Conf on Logic Programming, Tokyo, Japan, pp 67-82, MIT Press, Cambridge, Mass., USA, June 1995.

[San94.] E. Sandewall, Features and Fluents, The Representation of Knowledge about Dynamical Systems, Vol. I, Oxford University Press, 1994.

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

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

[ScSm91.] M. Schmidt-Schauß and G. Smolka, Attributive Concept Descriptions with Complements, Journal of Artificial Intelligence, 47, 1991.

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

[Sid93.] G. A. Sidebottom, A Language for Optimizing Constraint Propagation, Thesis, Simon Fraser University, Canada, 1993.

[Smo93.] G. Smolka, Residuation and Guarded Rules for Constraint Logic Programming, Constraint Logic Programming: Selected Research, (F. Benhamou and A. Colmerauer, Eds.), MIT Press, Cambridge, Mass., USA, 1993.

[SmTr94a.] G. Smolka and R. Treinen (Eds.), DFKI Oz Documentation Series, DFKI, Saarbrücken, Germany, 1994.

[SmTr94b.] G. Smolka and R. Treinen, Records for Logic Programming, Journal of Logic Programming 18:229-258, 1994.

[StBa94.] P. Baumgartner and F. Stolzenburg, Constraint model elimination and a PTTP-implementation, 4th Workshop on Theorem Proving with Analytic Tableaux and Related Methods, pp 201-216, (P. Baumgartner, R. Haehnle, and J. Posegga, Eds.), Springer LNAI 918, 1995.

[vH89.] P. van Hentenryck, Constraint Satisfaction in Logic Programming, MIT Press, Cambridge, Mass., USA, 1989.

[vH91.] P. van Hentenryck, Constraint Logic Programming, The Knowledge Engineering Review, Vol 6:3, 1991, pp 151-194.

[ViKa86.] M. Vilain, H. Kautz, Constraint Propagation Algorithms for Temporal Reasoning, AAAI 86, pp 377-382.

[Wil91.] M. R. Wilk, Equate: An Object-Oriented Constraint Solver, ACM OOPSLA 91, 1991, pp 286-298.

[Wol97.] A. Wolf, Incremental Adaption of Constraint Handling Rule Derivations, CP'97 Workshop on the Theory and Practice of Dynamic Constraint Satisfaction, Linz, Austria, November 1997.