CONSTRAINT SOLVING WITH CONSTRAINT HANDLING RULES

THOM FRÜHWIRTH
Institut für Informatik, Ludwig-Maximilians-Universität (LMU)
Oettingenstrasse 67, D-80538 Munich, Germany
fruehwir@informatik.uni-muenchen.de
www.pst.informatik.uni-muenchen.de/~fruehwir/

We describe how constraints are solved in constraint logic programming. To describe the algorithms at a high, abstract level, we use Constraint Handling Rules (CHR), a declarative language extension especially designed for writing user-defined constraints. CHR consist of multi-headed guarded rules that rewrite constraints into simpler ones until they are solved. In this article, we assume some familiarity with Prolog.

1 Introduction

The advent of constraints in logic programming is one of the rare cases where theoretical, practical and commercial aspects of a programming language have been improved simultaneously. *Constraint logic programming*[1,2,3,4,5] (CLP) combines the advantages of logic programming and constraint solving. In logic programming languages like Prolog, 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 sub-problems 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 the basic operations on constraints: solving (satisfaction), simplification, propagation, normalization, entailment (deciding implication) and optimization (computing "best" solutions).

The idea of CLP is to solve problems by stating constraints (conditions, properties) which must be satisified by a solution of the problem. For example, consider a bicycle number lock. We forgot the first digit, but remember some constraints about it: The digit was an odd number, greater than 1, and not a prime number. Combining the pieces of partial information expressed by these constraints (digit, greater than 1, odd, not prime) we are able to derive that the digit we are looking for is "9".

Since the beginning of the 90ties, constraint-based programming is commercially successful. The world-wide revenue generated by constraint technology for 1996 was estimated to be on the order of 100 Million Dollars.

Constraint handling rules (CHR)[7,6] are a high-level language especially designed for writing constraint solvers. CHR are essentially a committed-choice language consisting of multi-headed guarded rules that rewrite constraints into simpler ones until they are solved. CHR can be seen of generalization of the various CHIP constructs[8] for user-defined constraints.

CHR define both *simplification* of and *propagation* over user-defined constraints. Simplification replaces constraints by simpler constraints while preserving logical equivalence (e.g. `X>Y,Y>X <=> false`). Propagation adds new constraints which are logically redundant but may cause further simplification (e.g. `X>Y,Y>Z ==> X>Z`). Repeatedly applying the rules incrementally solves constraints (e.g. `A>B,B>C,C>A` leads to false). With multiple heads and propagation rules, CHR provide two features which are essential for non-trivial constraint handling. These features are not present in the related general-purpose concurrent logic programming languages[9], concurrent constraint languages[10] and ALPS languages[11].

Besides defining the behaviour of constraints, CHR can be and have been used as

- general purpose concurrent constraint language with ask and tell,
- as fairly efficient production rule system,
- as a special kind of theorem prover,
- in general as system combining forward and backward chaining.

CHR exist currently in 7 implementations in several programming languages (Prolog, LISP, OZ, Java). CHR have been used in dozens of projects worldwide to encode dozens of constraint solvers, including new domains such as terminological, spatial and temporal reasoning.

Overview of the Paper. First we introduce CHR by example. Then we will give syntax and simple semantics for CHR. We will illustrate how to solve constraints by using CHR to implement constraint solvers for them. We will give an overview of several solvers ranging from Boolean and arithmetic to terminological and path-consistent constraints.

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. The equality is applied to the rest of the query, `A=<B,C=<A`, resulting in `A=<B,B=<A` where `B=C`. 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`.

3 Syntax and Semantics

In this section we give syntax and simple semantics for CHR, for more detailed semantics see[12,6].

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, given constraint solver. Here we assume that the syntactic equality constraint `=` and the trivial constraints `true` and `false` are built-in. CHR constraints are those defined by a CHR program.

3.1 Syntax

The syntax of CHR is reminiscent of Prolog and GHC. It is defined by EBNF grammar rules.

Definition 3.1 A CHR program is a finite set of CHR. There are two main kinds of CHR. A *simplification CHR* is of the form

```
[Name '@'] Head '<=>' [Guard '|'] Body.
```

a *propagation CHR* is of the form

```
[Name '@'] Head '==>' [Guard '|'] Body.
```

where the rule has an optional `Name`, the multi-head `Head` is a conjunction of CHR constraints. The optional guard `Guard` is a conjunction of built-in constraints. The body `Body` is a conjunction of built-in and CHR constraints. As in Prolog syntax, a conjunction is a sequence of conjuncts separated by commata.

3.2 Semantics

The *declarative semantics* of a CHR program P is a conjunction of universally quantified logical formulas (one for each rule) and a consistent built–in *constraint theory* which determines the meaning of the built–in constraints appearing in the program. The delarative reading of a rule relates heads and body provided the guard is true. A simplification rule means that the heads are true if and only if the body is satisfied. A propagation rule means that the body is true if the heads are true.

The *operational semantics* can be described by a transition system. Here we describe it informally.

A CHR constraint is implemented as both *code* and *data* in the constraint store, which is a data structure holding constraints. Every time a CHR constraint is posted (executed) or woken (reconsidered, re-executed), it checks itself the applicability of the rules it appears in. Such a constraint is called *(currently) active.*

Heads. For each rule, one of its heads is matched against the constraint. Matching succeeds if the constraint is an instance of the head, i.e. the head serves as a pattern. If matching succeeded and a rule has more than one head, the constraint store is searched for the constraints that match the other heads. If the matching succeeds, the guard is executed. Otherwise the next rule is tried.

Guard. A guard is a precondition on the applicability of a rule. The guard either succeeds or fails. A guard succeeds if the execution succeeds without causing an instantiation error and without *touching* a variable from the heads. A variable is *touched* if gets more constrained by a built-in constraint. If the guard succeeds, the rule applies, one commits to it and it fires. Otherwise it fails and the next rule is tried.

Body. If the firing CHR is a simplification rule, the matched constraints are removed from the store and the body of the CHR is executed. If the firing CHR is a propagation rule the body of the CHR is executed without removing any constraints. It is remembered that the propagation rule fired, so it will not fire again with the same constraints. When the currently active constraint has not been removed, the next rule is tried.

(Re-)Try. If all rules have been tried and the active constraint has not been removed, it suspends (waits, delays) until a variable occurring in the constraint is touched. Here suspension means that the constraint is inserted into the constraint store as data. When a constraint is woken, all its rules are tried again.

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[12].

4 Constraint Solvers

In this section we introduce constraint solvers for Booleans (propositional logic), finite interval domains incremental path consistency, temporal reasoning, for solving linear polynomials and for terminological reasoning. For details on the constraint solvers analysed here see[7] and the CHR web pages: `www.pst.informatik.uni-muenchen.de/~fruehwir/chr-intro.html`

While we cannot - within the space limitations - introduce each constraint domain in detail, 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.

4.1 Boolean Algebra, Propositional Logic

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 instead of functions. We can define an `and` constraint using value propagation, a special case of arc consistency. For more sophisticated algorithms see[13].

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

Example 4.1 Consider the predicate `add` 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 illustrated the power of this simple but incomplete solver. *Incompleteness* means that the solver is too weak detect unsatisfiability in all cases. To achieve completeness, search must be employed. This is done by trying to the values 0 or 1 for a variable, then by employing the constraint solver again. This is repeated till a solution consisting only of syntactic equalities is found or unsatisfiability is detected due to contradicting variable bindings.

4.2 Terminological Reasoning

Terminological formalisms (aka description logics) 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. Concepts can be considered as unary relations similar to types. Roles correspond to binary relations over objects. Although there is an established notation for terminologies, we use a more verbose syntax to help readers not familiar with the formalism.

Definition 4.1 *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 also 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 $a : s$ is a *membership assertion* and $(a, b) : r$ is a *role-filler assertion*. An *A-box* is a conjunction of membership and role-filler assertions.

Definition 4.2 A *terminology (T-box)* consists of a finite set of *concept definitions*

> c `isa` s,

where c is a 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 (non-recursive). This also implies that there are concepts without definition, they are called *primitive*.

Example 4.2 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
```

The CHR constraint solver for terminologies encodes the T-box by rules and the A-box as CHR constraints, since we want to solve problems over a given terminology (T-box). A similar solver is described in[14]. The unfolding and completion rules[16] and the propagation rules[15] for the consistency test translate almost directly to CHRḢowever, the former work does not provide an incremental algorithm and the latter does not simplify constraints.

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. This is expressed by the rule:

```
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:

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

An exists-in restriction generates a new 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 replace concept names by their definitions. For each concept definition `C isa S` two rules are introduced:

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

The conjunction rule generates two new, smaller assertions:

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

The rules simplify terminological constraints until a normal form is reached. The normal form is either `false` (inconsistent) or contains constraints of the form `I : C, I : nota C, I : S or T, I : every R is S` and `(I,J) : R`, where `C` is a primitive concept name. There are no clashes and the value restriction has been propagated to every object. To achieve completeness, search must be employed. This is done by splitting `I : S or T` into two cases, `I : S` and `I: T`.

4.3 Linear Polynomial Equations

The initial motivation for introducing constraints in logic programming languages (Prolog) 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 polynomial equations and inequations over reals (CLP(R)[17]) or rationals (Prolog-III[18], CHIP[8]) adopting variants of variable elimination alogrithms like Gaussian elimination and the Simplex algorithm[19]. 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.

Definition 4.3 A *linear polynomial equation* is of the form $p + b = 0$ where b is a constant and the polynomial p is the sum of monomials of the form $a_i * x_i$ with coefficient $a_i \neq 0$ and x_i is a variable. Constants and coefficients are numbers. Variables are totally ordered. In an equation $a_1 * x_1 + \ldots + a_n * x_n + b = 0$, variables appear in strictly descending order.

In constraint logic programming, constraints are added incrementally. Therefore we cannot eliminate a variable in *all* other equations at once, but rather consider the other equations one by one. A simple solved form can exhibit unsatisfiability: It is enough if the left-most variable of each equation is the only left-most occurrence of this variable. Therefore the two rules below implement a complete and efficient solver for linear equations over both floating point numbers (to approximate real numbers) and rational numbers.

```
empty @ B eq 0 <=> number(B) | B=0.

eliminate @
A1*X+P1 eq 0, A2*X+P2 eq 0 <=>
   compute(P2+P1*A2/A1,P3),
   A1*X+P1 eq 0, P3 eq 0.
```

The `empty` rule says that if the polynomial contains no more variables, the constant `B` must be zero. The `eliminate` rule performs 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. The auxiliary built-in constraint `compute` simplifies a polynomial arithmetic expression into a new polynomial. Note that no variable is made explicit, i.e. no pivoting is performed. Any two equations with the same first variable can react with each other. Therefore, the solver is highly concurrent and distributed.

The solver can be extended by a few rules to create explicit variable bindings, to make implicit equalities between variables explicit, to deal with inequations using slack variables as in the Simplex algorithm or fouriers algorithm.

Non-linear polynomial constraints appear e.g. in modelling physical processes and in geometric reasoning for spatial databases and robot motion planning. To tackle non-linear polynomials, techniques like Groebner Bases over complex numbers (CAL[20]) and Partial Cylindrical Algebraic Decomposition (RISC-CLP(Real)[21]) have been used. Another approach is to use interval arithmetic as in CLP(BNR)[22], Numerica[23]. This approach can basically be seen as a sophisticated extension of finite interval domains (described below) to the reals and to non-linear polynomials.

4.4 Path Consistency

In this section we introduce constraint solvers that implement instances of the classical artificial intelligence algorithm of path consistency to simplify constraint satisfaction problems[24].

Definition 4.4 A *binary constraint network* consists of a set of variables and a set of (disjunctive) 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. Logically, a network is a conjunction of binary constraints.

Definition 4.5 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 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.

For example, $A\ \{<\}\ B, A\ \{<,>\}\ B, A\ \{<,=,>\}\ B$ are disjunctive binary constraints c_{AB} between A and B. $A\ \{<\}\ B$ is the same as $A < B$, $A\ \{<,>\}\ B$ is the same as $A \neq B$. Finally, $A\ \{<,=,>\}\ B$ does not impose any restrictions on A and B, the constraint is redundant. Usually, the number of primitive constraints is finite and they are pairwise disjoint. We will asume this in the following.

Definition 4.6 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 at least as tight 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.

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 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. This is the basic path consistency algorithm.

Example 4.3 Given $I \leq K \ \wedge\ K \leq J \ \wedge\ I \geq J$ and taking the triple (i,j,k), $c_{ik} \otimes c_{kj}$ results in $I \leq J$ and the result of intersecting it with c_{ij} is $I = J$. From (j,i,k) we get $J = K$ (we can 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 $J = K \ \wedge\ K = I$.

Since path consistency is an incomplete algorithm, search must be employed by choosing a primitive constraint from a set of disjunctive constraints.

Let the constraint c_{ij} be represented by the CHR constraint `c(I,J,C)` where `I` and `J` are the variables and `C` is a set of primitive constraints representing c. The basic operation of path consistency, $c_{ij} := c_{ij} \oplus (c_{ik} \otimes c_{kj})$, can be implemented directly by the rule:

```
path_consistency @
c(I,K,C1), c(K,J,C2), c(I,J,C3) <=>
  composition(C1,C2,C12), intersection(C12,C3,C123),
  C123=\=C3 |
  c(I,K,C1), c(K,J,C2), c(I,J,C123).
```

The operations $\otimes$ and $\oplus$ are implemented by the built-in constraints `composition` and `intersection`. Composition of disjunctive constraints can be computed by pairwise composition of its primitive constraints. Intersection for disjunctive constraints can be implemented by set intersection. `C123=C̄3` ensures that the newly produced constraint is different to (and thus smaller than) the previous one.

Temporal Reasoning

Following the framework of Meiri[25], temporal reasoning is 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[26]. 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[27]. The primitive constraints restrict the distance of two time points X and Y to be in an interval $a : b$, i.e. $a \leq (Y - X) \leq b$, 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[28]. 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.
```

`equals,after,contains,overlapped_by,started_by,finished_by` are the converses.

Point - Interval Constraints[25]. 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[29]. 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.

We can instantiate the generic path consistency solver of the previous section by defining the intersection and composition operations for the temporal constraints described above. The implementation is described in detail and with variations in[30].

Example 4.4 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.

4.5 Finite domains

Finite domains appeared first in CHIP[31], more recent and more advanced CLP languages are clp(FD)[32] and cc(FD)[33]. 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 in 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*, `Set`, then `X` is an integer in the set `Set`,
- *interval domain*, `Min:Max`, then `X` is an integer 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.

For space limitations, we only consider interval domains here. The following rules implement an arc consistency algorithm for interval constraints. Arc consistency can be seen as special case of path consistency, where all but one constraint is unary instead of binary. Like path consistency, this algorithm is incomplete. Search can be employed for completeness by choosing values from the intervals or by splitting them.

```
% Intervals
inconsistent @ X in A:B <=> A>B | false.
intersection @ X in A:B, X in C:D <=> X in max(A,C):min(B,D).

% (In)equalities
le @ X le Y, X in A:B, Y in C:D <=> B>D |
     X le Y, X in A:D, Y in C:D.
le @ X le Y, X in A:B, Y in C:D <=> C<A |
     X le Y, X in A:B, Y in A:D.
eq @ X eq Y, X in A:B, Y in C:D <=> A=\=C |
     X eq Y, X in max(A,C):B, Y in max(C,A):D.
eq @ X eq Y, X in A:B, Y in C:D <=> B=\=D |
     X eq Y, X in A:min(B,D), Y in C:min(D,B).

% Addition X+Y=Z
add @ add(X,Y,Z), X in A:B, Y in C:D, Z in E:F <=>
      not (A>=E-D,B=<F-C, C>=E-B,D=<F-A, E>=A+C,F=<B+D) |
      add(X,Y,Z), X in max(A,E-D):min(B,F-C),
                  Y in max(C,E-B):min(D,F-A),
                  Z in max(E,A+C):min(F,B+D).
```

The guards ensure that a rule is only applied if at least one interval gets smaller. For example, given

```
A in 1:3, B in 2:4, C in 0:4, add(A,B,C)
```

the `add` rule adds the interval constraints

```
A in -1:2, B in 0:3, C in 3:7
```

and after some `intersection` we arrive at:

```
add(A,B,C), A in 1:2, B in 2:3, C in 3:4
```

The rules above can be modified to work for intervals of real numbers: To avoid non-termination, intervals that are too small are not considered by the rules anymore.

5 Conclusions

We described how constraints are solved in constraint logic programming. To describe the algorithms at a high, abstract level, we used Constraint Handling Rules (CHR), a declarative language extension especially designed for writing user-defined constraints.

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). Indeed, 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.

References

1. P. van Hentenryck, H. Simonis and M. Dincbas, Constraint Satisfaction Using Constraint Logic Programming, Artificial Intelligence, 58(1-3):113–159, December 1992.
2. 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.
3. J. Jaffar and M. J. Maher, Constraint Logic Programming: A Survey, Journal of Logic Programming 19,20:503-581, 1994.
4. T. Frühwirth and S. Abdennadher, Constraint-Programmierung (in German), Textbook, Springer Verlag, Heidelberg, Germany, September 1997.
5. K. Marriott and P. J. Stuckey, Programming with Constraints, MIT Press, USA, March 1998.
6. S. Abdennadher, T. Frühwirth and H. Meuss, Confluence and Semantics of Constraint Simplification Rules, Journal Constraints, Volume 4, Issue 2, Kluwer Academic Publishers, May 1999.

7. T. Frühwirth, Theory and Practice of Constraint Handling Rules, Special Issue on Constraint Logic Programming (P. J. Stuckey and K. Marriot, Eds.), Journal of Logic Programming, Vol 37(1-3):95-138 Oct-Dec 98.
8. M. Dincbas et al., The Constraint Logic Programming Language CHIP, Fifth Generation Computer Systems, Tokyo, Japan, December 1988.
9. E. Shapiro, The Family of Concurrent Logic Programming Languages, ACM Computing Surveys, 21(3):413-510, September 1989.
10. V. A. Saraswat, Concurrent Constraint Programming, MIT Press, Cambridge, Mass., USA, 1993.
11. 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.
12. 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.
13. 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.
14. 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.
15. M. Buchheit, F. M. Donini and A. Schaerf, Decidable Reasoning in Terminological Knowledge Representation Systems, Journal of Artificial Intelligence Research, 1:109-138, 1993.
16. M. Schmidt-Schauß and G. Smolka, Attributive Concept Descriptions with Complements, Journal of Artificial Intelligence, 47, 1991.
17. J. Jaffar et al., The CLP(R) Language and System, ACM Transactions on Programming Languages and Systems, Vol.14:3, July 1992.
18. A. Colmerauer, An Introduction to Prolog III, Communications of the ACM 33(7):69-90, July 1990.
19. J.-L. J. Imbert, Linear Constraint Solving in CLP-Languages, in Constraint Programming: Basics and Trends, (A. Podelski, Ed.), LNCS 910, March 1995.
20. A. Aiba et al, Constraint Logic Programming Language CAL, Intl Conf on Fifth Generation Computer Systems, Ohmsha Publishers, Tokyo, pp 263-276, 1988.
21. 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.

22. F. Benhamou, Interval constraint logic programming, Chapter in Constraint Programming: Basics and Trends, (A. Podelski, Ed.), Springer LNCS 910, March 1995.
23. P. van Hentenryck, L. Michel and Y. Deville, Numerica: a Modeling Language for Global Optimization, MIT Press, Cambridge, Mass., USA, 1997.
24. 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.
25. I. Meiri, Combining Qualitative and Quantitative Constraints in Temporal Reasoning, AAAI 91, pp 260-267, 1991.
26. M. Vilain, H. Kautz, Constraint Propagation Algorithms for Temporal Reasoning, AAAI 86, pp 377-382, 1986.
27. R. Dechter, I. Meiri and J. Pearl, Temporal Constraint Networks, Journal of Artificial Intelligence 49:61-95, 1991.
28. J. F. Allen, Maintaining Knowledge about Temporal Intervals, Communications of the ACM, Vol. 26, No. 11, 1983.
29. H. A. Kautz and P. B. Ladkin, Integrating Metric and Qualitative Temporal Reasoning, AAAI 91, pp 241-246, 1991.
30. T. Frühwirth, Temporal Reasoning with Constraint Handling Rules, Technical Report ECRC-94-05, ECRC Munich, Germany, February 1994.
31. P. van Hentenryck, Constraint Satisfaction in Logic Programming, MIT Press, Cambridge, Mass., USA, 1989.
32. P. Codognet and D. Diaz, Compiling constraints in `clp(fd)`, Journal of Logic Programming, 27(3), 1996.
33. 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.