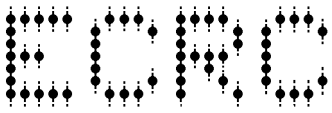

Technical Report ECRC-94-05
February 16, 1994

Temporal Reasoning with Constraint Handling Rules

Thom Frühwirth
thom@ecrc.de

European Computer-Industry
Research Centre GmbH
Arabellastr. 17,
D-81925 München
Germany

Abstract

This paper describes an application of constraint handling rules to temporal reasoning and illustrates the conceptual simplicity and flexibility of the approach. Following the framework of Meiri, temporal reasoning is viewed as a constraint satisfaction problem about location of temporal variables along the time line. Temporal variables may be points or intervals and temporal constraints are disjunctions of qualitative or quantitative primitive binary temporal relations. We use constraint logic programming extended with constraint handling rules to define an incremental, flexible general purpose solver for disjunctive binary constraints based on path consistency and backtrack search. We show how this approach supports rapid prototyping and experimentation with different kinds of temporal reasoning and constraint satisfaction techniques in general.

Keywords: Constraint-Based Reasoning, Temporal Reasoning, Logic Programming.

This work was supported by ESPRIT Project 5291 CHIC. This report was first published in January 1993 as ECRC Technical Report CORE-93-08.

1 Introduction

We have implemented a range of constraint solvers in a constraint logic programming language extended with constraint handling rules[1] and believe that our approach forms an ideal basis for prototyping and implementing new constraints and constraint solvers. The purpose of this paper is to show how our approach is beneficial for understanding and developing temporal reasoning and constraint consistency techniques.

Constraint logic programming (CLP) [JaLa87, Sar89, Coh90, VH91] 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* (LPE) using nondeterministic backtrack search. In constraint solving, efficient special-purpose algorithms are used to solve problems involving distinguished relations referred to as constraints. In current constraint logic programming languages, constraint solving is usually hard-wired in a built-in *constraint solver* (CS). Hard-coding guarantees maximum efficiency for the given set of constraints. However it has the serious drawback that it is hard to extend and specialize the built-in CS, combine it with other CS's and build a CS over a new domain.

Constraint handling rules (CH rules) [Fru92] are a *language extension* providing the application-programmer (user) with a declarative and flexible means to introduce *user-defined* constraints (in addition to *built-in* constraints of the underlying language). CH rules define *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). When repeatedly applied by a CH rule engine (CHE) the constraints may become solved and the resulting functionality is then that of a user-defined constraint solver.

[1] A compiler for CH rules including the constraint solver described here is available as a library of ECLiPSe, ECRC's constraint logic programming platform.

User-defined constraint control is a very active area of research. CHIP was the first constraint logic programming language to introduce the necessary constructs. These constructs have been called "demon constructs" [D*88] because of their event-driven activation whenever new information is available. These various constructs have been generalized into CH rules. CH rules are essentially multi-headed guarded rules and as such have their roots in concurrent logic programming languages [Sha89] and can be related to the Swedish branch of the Andorra family [HaJa90] and Saraswats cc-framework of concurrent constraint programming [Sar89]. However these general purpose programming languages lack features essential to define non-trivial constraint evaluation, namely multi-headed guarded rules and and a way to define constraint propagation by rules, while CH rules form a special purpose language extension for user-defined constraint solvers.

In our approach, constraints are seen as a computationally efficient incarnation of the predicates defined in the underlying host language, as such CH rules have a logical reading and thus preserve the declarative semantics of the underlying logic programming language they extend. This view that provides a tight and sound integration of the host language with user-defined constraint solvers and also motivates the work of Smolka on *Guarded Rules* [Smo91]. The representation of constraint solving in the same formalism as the rest of the program greatly facilitates the prototyping, extension, specialization and combination of constraint solvers as well as reasoning about correctness, termination and confluence of a set of CH rules.

2 Preliminaries

In this section we introduce the framework of Meiri [Mei91]. It integrates many forms of temporal relations - qualitative and quantitative (metric) over time points and intervals - by considering them as disjunctive binary constraints. We give the primitive relations for various forms of temporal constraints. We also describe how different constraints relate to each other - so we are able to reason over heterogeneous constraint networks. Temporal reasoning tasks are formulated as constraint satisfaction problems, and are solved by traditional constraint satisfaction techniques, namely backtrack search and path consistency.

2.1 Temporal Constraints

A *disjunctive binary constraint* c_{xy} between two variables X and Y, also written $X\ \{r_1, \ldots r_n\}\ Y$, is a 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. For simplicity, unary (domain) constraints are modeled as binary constraints where one variable is fixed. Disjunctive binary constraints allow us to model various forms of temporal constraints. $A\ \{<\}\ B, A\ \{<, >\}\ B, A\ \{<, =, >\}\ B$ are disjunctive binary constraints c_{AB} between A and B. $A\ \{<, >\}\ B$ is the same as $A \neq B$.

Qualitative Point Constraints [ViKa86]. Variables represent time points and there are three primitive constraints $<, =, >$. $<$ is the converse of $>$ and equality is the converse of

itself.

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. $a \leq (Y - X) \leq b$[2]. Note that there is an infinite number of constraints. From the meaning it follows that the converse of an interval $a..b$ is $(-b)..(-a)$. The interval 0..0 corresponds to equality.

Relating Point Constraints [KaLa91]. Qualitative can be mapped into quantitative point constraints, quantitative constraints can only be approximated by qualitative constraints. These mappings are used to solve heterogeneous constraints over the same variables. Let $QUANT$ ($QUAL$) be a quantitative (qualitative) temporal constraint.

If $a..b \in QUANT, a \leq 0 \leq b$ then `eq` $\in QUAL$.
If $a..b \in QUANT, a < 0$ then `ge` $\in QUAL$.
If $a..b \in QUANT, b > 0$ then `le` $\in QUAL$.

and conversely

If `eq` $\in QUAL$ then $0..0 \in QUANT$.
If `ge` $\in QUAL$ then $(-\infty)..0 \in QUANT$.
If `le` $\in QUAL$ then $0..\infty \in QUANT$.

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=$[U, V]$.

`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`.

Relating Point and Interval Constraints [KaLa91]. Points can be represented by end-points of intervals. Interval constraints can be approximated by constraints on their endpoints. Given two points X, Y. Let I=$[X, A]$, J=$[Y, B]$ be intervals.

If `X>Y` then `I {during,finishes,overlapped_by,met_by,after} J`.
If `X=Y` then `I {equals,starts,started_by} J`.
If `X<Y` then `I {before,meets,finished_by,contains,overlaps} J`.

Given two intervals I=$[X, Y]$, J=$[U, V]$. The approximation is computed by translating the disjunctive interval constraints into disjunctions of their definitions in terms of their end-points. The disjunction is then approximated by a conjunction of point inequalities. E.g., `I {equals,before} J` $= (X = U < Y = V) \vee (X < Y < U < V) \approx (X < Y \leq V, U \neq Y, X \leq U < V)$. With `I {before,after} J` $\approx (X < Y, U < V)$ we loose all information between I, J.

Point - Interval Constraints [Mei91]. There are 5 possible primitive constraints between a point and an interval. Note that there cannot be equality. Again, the primitive constraints can be defined in terms of the end-points of the interval. Let X be a point, J=$[U, V]$ an interval.

`X before J` if $X < U < V$.
`X after J` if $U < V < X$.
`X during J` if $U < X < V$.
`X starts J` if $X = U < V$.
`X finishes J` if $U < X = V$.

The converses express interval-point constraints. Translations to intervals and points are

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

possible analogous to the point-interval translations shown before.

2.2 Constraint Satisfaction

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.

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. Interesting problems (typically NP-hard) are therefore to

- determine consistency
- compute the minimal network
- compute one, all or "best" solution(s)
- check validity of an assignment

For the special case of *disjunctive* binary constraint networks and temporal constraints in particular we are not so much interested in a particular assignment for temporal variables, but in choosing primitive constraints from the disjunctions such that the resulting disjunction-free binary network is consistent. As there is a clear analogy between choosing values for a variable and primitive constraints between two variables, we will use the established terminology further on.

In the following we will solve the above mentioned constraint satisfaction problems except optimization ("best" solution) by a single algorithm that combines backtrack search and path consistency [LaRe92].

Backtrack search. Backtrack search can be used to compute solutions, but has exponential complexity in the number of choices. Search will be therefore more efficient if we minimize the number of potential choices by computing the minimal network before making a choice. The overall idea is to interleave computing the minimal network and making a single choice.

Algorithm to compute one or more solutions of a given network

Compute minimal network
If inconsistent **then** fail
repeat
  Take a new pair of nodes with its disjunctive constraint
  Choose a primitive constraint from the disjunction
  If no more choice **then** backtrack
  Compute minimal network
  If inconsistent **then** backtrack
until success or failure
If success and more solutions wanted **then** backtrack.

Path consistency. Unfortunately, in general it is as hard to find the minimal network as to compute solutions by backtrack search. The good news however is that it is possible

to compute an approximation of the minimal network (one that is "almost" minimal with some superfluous primitive constraints) in polynomial time by applying local consistency techniques. The basic idea is to split the problem into overlapping subproblems of bounded size, solve them, and to repeat computation over all possible subproblems until a fixpoint is reached. The following operations on disjunctive binary constraints (considered as a set of primitive constraints) will come in handy for computing path consistency. The operations are mostly defined using the corresponding operations on the primitive constraints. All binary operations are associative.

Converse. The converse primitive constraint of r_i is written $\ominus r_i$.
$\ominus c_{ij} := I\ \{r_1, \ldots r_n\}\ J = J\ \{\ominus\ r_1, \ldots \ominus\ r_n\}\ I = J\ \{s_1, \ldots s_n\}\ I\ =\ c_{ji}$. (if $\ominus\ r_i = s_i$).
Note that $\ominus \ominus r_i = r_i$.

Union. By definition of disjunctive binary constraints:
$c_{ij} \uplus c'_{ij} := I\ \{r_1, \ldots r_n\}\ J\ \vee I\ \{s_1, \ldots s_m\}\ J = I\ (\{r_1, \ldots r_n\} \cup \{s_1, \ldots s_m\})\ J$.

Intersection. If the primitive constraints are pairwise disjoint[3]:
$c_{ij} \oplus c'_{ij} := I\ \{r_1, \ldots r_n\}\ J\ \wedge I\ \{s_1, \ldots s_m\}\ J = I\ (\{r_1, \ldots r_n\} \cap \{s_1, \ldots s_m\})\ J$.
Otherwise intersection can be based on pairwise intersection of primitive constraints:
$c_{ij} \oplus c'_{ij} := I\ \{r_1, \ldots r_n\}\ J\ \wedge I\ \{s_1, \ldots s_m\}\ J = I\ \{r \oplus s \mid r \oplus s$ exists$, r \in \{r_1, \ldots r_n\}, s \in \{s_1, \ldots s_m\}\}\ J$, where $r \oplus s$ denotes the relation defined by $r(X,Y) \wedge s(X,Y)$.

Composition. Based on pairwise composition of primitive constraints:
$c_{ik}\ \otimes\ c_{kj}\ :=\ I\ \{r_1, \ldots r_n\}\ K\ \wedge K\ \{s_1, \ldots s_m\}\ J\ =\ I\ \{r \otimes s\ \mid\ r\ \in\ \{r_1, \ldots r_n\}, s\ \in\ \{s_1, \ldots s_m\}\}\ J = c_{ij}$, where $r \oplus s$ is the relation between X and Z in $r(X,Y) \wedge s(Y,Z)$.
Note that composition is not necessarily commutative.

Path consistency can be used to approximate the minimal network. A network is *path consistent*[4] 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 than the indirect constraint along the path, i.e. the composition of constraints $c_{ii_1} \otimes \ldots c_{i_nj}$ along the path. A disjunctive constraint is *tighter* if it has less disjuncts. It follows from the definition of path consistency that we can intersect the direct and indirect constraint to arrive a tighter direct constraint. If the graph underlying the network is complete it suffices to repeatedly compute paths of length 2 at most. A graph is *complete* if there is an edge or a pair of arcs, one in each direction, between every pair of nodes. This means 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), then $c_{ik} \otimes c_{kj}$ 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).

One path consistency algorithm was given by [All83]. for use with his temporal interval constraints. A queue of arcs whose constraint got tighter is used to keep track of the triangles that have to be reconsidered.

[3] Holds for temporal constraints except quantitative constraints.

[4] We can ignore unary constraints in the definition, since we have modeled them as binary constraints.

$Q := \{(i,j) \mid i \leq n, j \leq n\}$
while $Q \neq \{\}$ **do**
    delete a tuple (i,j) from Q
    for each node k **do**
        $Temp := c_{ik} \oplus c_{ij} \otimes c_{jk}$
        if $Temp = \{\}$ **then** inconsistent
        if $c_{ik} \neq Temp$ **then**
            $Q := Q \ \cup \ \{(i,k)\}$
            $c_{ik} := Temp$
        $Temp := c_{kj} \oplus c_{ki} \otimes c_{ij}$
        if $Temp = \{\}$ **then** inconsistent
        if $c_{kj} \neq Temp$ **then**
            $Q := Q \ \cup \ \{(k,j)\}$
            $c_{kj} := Temp$
    endfor
endwhile

Another classical algorithm named PC-2 was given by [Mac77]. It is an optimization based on the idea that if needed, c_{ij} can be computed as the converse of c_{ji}, which saves half of the computation, and that c_{ii} can only be equality. The queue this time keeps track of all node triples that have to be reconsidered.
$Q := \{(i,k,j) \mid i < j, i \neq k, k \neq j\}$
while $Q \neq \{\}$ **do**
    delete a triplet (i,k,j) from Q
    $Temp := c_{ij} \oplus c_{ik} \otimes c_{kj}$
    if $Temp = \{\}$ **then** inconsistent
    if $c_{ij} \neq Temp$ **then**
        $Q := Q \cup AFFECTED((i,j,k))$
        $c_{ij} := Temp$
endwhile
where $AFFECTED((i,j,k)) := \{(u,v,w) \mid u < w, u \neq v, v \neq w\} \cap (\{(i,j,m) \mid m \in N\} \cup \{(j,i,m) \mid m \in N\} \cup \{(m,i,j) \mid m \in N\} \cup \{(m,j,i) \mid m \in N\})$.

3 CLP+CH Languages

3.1 Syntax

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

will say constraint instead of constraint atom.

A *CLP clause* is of the form

$H \leftarrow B_1, \ldots B_n. \quad (n \geq 0)$

where the head H is an atom but not a built-in constraint, the body $B_1, \ldots B_n$ is a conjunction of atoms called *goals*. The empty body ($n = 0$) of a CLP clause may be denoted by the constraint `true`, which is always satisfied. `false` is the constraint representing inconsistency. A CLP clause with an empty body is called fact, any other clause rule. A *query* is a CLP clause without head.

CH rules are essentially multi-headed guarded rules forming a committed-choice (sub)language. There are two kinds of CH rules and a declaration (to be explained later).

Simplification CH rules are of the form

$H_1, \ldots H_i \Leftrightarrow \; G_1, \ldots G_j \mid B_1, \ldots B_k,$

Propagation CH rules are of the form

$H_1, \ldots H_i \Rightarrow \; G_1, \ldots G_j \mid B_1, \ldots B_k,$

Call declarations for a user-defined constraint H are of the form

`callable` H `if` $G_1, \ldots G_j,$

where $(i > 0, j \geq 0, k \geq 0)$ and the multi-head $H_1, \ldots H_i$ is a conjunction of user-defined constraints, the guard $G_1, \ldots G_j$ is a conjunction of atoms which neither are, nor depend on, user-defined constraints, and the body $B_1, \ldots B_k$ is a conjunction of atoms.

Given a clause CL and an atom A, we say that CL is a *clause of* A if the head of CL contains an atom with the same predicate symbol p as A. We say that the predicate p is defined by all its clauses. Note that predicates are defined by CLP clauses and user-defined constraints by CH rules *and* CLP clauses.

3.2 Declarative Semantics

Declaratively, CLP languages are interpreted as formulas in first order logic. A CLP+CH program P is a conjunction of universally quantified clauses. A CLP clause is an implication

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

The meaning of a CH rule is dependent on its guard. A simplification CH rule is a logical equivalence between head and body

$(C_1 \wedge \ldots C_j) \rightarrow (A_1 \wedge \ldots A_i \leftrightarrow B_1 \wedge \ldots B_k).$

A propagation CH rule is an implication from the head to the body

$(C_1 \wedge \ldots C_j) \rightarrow (A_1 \wedge \ldots A_i \rightarrow B_1 \wedge \ldots B_k).$

Call declarations have no declarative reading and do not change the declarative semantics. Extending a CLP language with CH rules preserves its declarative semantics, as *correct* CH rules are logically redundant with regard to the underlying CLP program. CH rules are not supposed to change the meaning of a program, but the way it is executed.

3.3 Operational Semantics

The operational semantics of CLP+CH can be described by a transition system. In the following we do not distinguish between sets and conjunctions of atoms. A *constraint store* represents a set of constraints. Let C_U and C_B be two constraint stores for user-defined and built-in constraints respectively. Let Gs be a set of goals. A *computation state* is a tuple

$< Gs, C_U, C_B >$.

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

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

A *final state* is either *successful* (no goals left to solve),

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

or *failed* (due to an inconsistent constraint store),

$< Gs, \texttt{false}, C_B >$ or $< Gs, C_U, \texttt{false} >$.

The union of the constraint stores in a successful final state is called *conditional answer* for the query Gs, written $answer(Gs)$.

The built-in CS works on built-in constraints in C_B and Gs, the user-defined CS on user-defined constraints in C_U and Gs using CH rules and the LPE on goals in Gs and C_U using CLP clauses.

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

Solve - Built-In CS

$< \{C\} \cup Gs, C_U, C_B > \longmapsto < Gs, C_U, C'_B >$

if $(C \wedge C_B) \leftrightarrow C'_B$

Simplify - CHE with simplification CH rules

$< H' \cup Gs, H'' \cup C_U, C_B > \longmapsto < Gs \cup B, C_U, C_B >$

if $(H \Leftrightarrow G \mid B) \in P$, $(C_B \rightarrow H = (H' \cup H'') \wedge answer(G))$

Propagate - CHE with propagation CH rules

$< H' \cup Gs, H'' \cup C_U, C_B > \longmapsto < Gs \cup B, H' \cup H'' \cup C_U, C_B >$

if $(H \Rightarrow G \mid B) \in P$, $(C_B \rightarrow H = (H' \cup H'') \wedge answer(G))$

Nondeterministic Unfold - LPE with CLP clause

$< \{H'\} \cup Gs, C_U, C_B > \longmapsto < Gs \cup B, C_U, \{H = H'\} \cup C_B >$

if $(H \leftarrow B) \in P$ and H is not a user-defined constraint

$< Gs, \{H'\} \cup C_U, C_B > \longmapsto < Gs \cup B, C_U, \{H = H'\} \cup C_B >$

if $(H \leftarrow B) \in P (H \leftarrow B), (callable\ H''\ if\ G) \in P \wedge (C_B \rightarrow \{H' = H''\} \wedge answer(G))$

During computation either the built-in CS updates the constraint store BC if a new constraint was found in G, or the user-defined CS simplifies and propagates from user-defined constraints in UC if a new user-defined constraint was found in G or the built-in constraint store had been updated, or the LPE unfolds non-constraint goals in G and callable user-defined constraints in UC. Constraint solving has priority over goal unfolding, and built-in constraint solving over user-defined constraint solving.

CS. A CS is a determinate and incremental procedure that updates a persistent constraint store. To *update* the constraint store means to produce a new constraint store that is logically equivalent to the conjunction of the new constraint and the old constraint store. Determinate means that exactly one new constraint store is produced (even if there is a

don't care choice between equivalent stores).

CHE . Similarly to a CS, the CHE is a determinate and incremental procedure with a persistent constraint store. There is don't care indeterminism in choosing a CH rule to apply, but if application is possible, the CHE commits to the choice. To *simplify* user-defined constraints C_U means to replace it by B if C_U match the head H of a simplification CH rule $H \Leftrightarrow G \mid B$ and G is satisfied. To *propagate from* user-defined constraint C_U means to add B if C_U match the head H of a propagation CH rule $H \Rightarrow G \mid B$ and G is satisfied. A guard G is *satisfied* if its execution does not involve user-defined constraints and results in a successful local computation state where the local built-in constraints are entailed by the global built-in constraint store. According to the *ask and tell* language framework of (Saraswat 1989) we may also say that we *ask* if C holds in the current constraint store, while posing a constraint means to *tell* (add) it to the constraint store.

LPE. The LPE is a indeterminate procedure with a persistent goal store.To *unfold* a goal Gs means to looks for a clause $H \leftarrow B$ with a head with the same predicate symbol as Gs, to replace the Gs by $(G = H), B$. As there are usually several clauses for a goal, unfolding is nondeterministic and thus a goal can be solved in different ways. Chronological backtracking over clause choices is used in the LPE to search (depth first) for successful computations. A user-defined constraint goal H is *callable* if there is a call declaration $H \; if \; G$ and no CH rule applies and G is satisfied.

3.4 Example

In the following we illustrate the behavior of Prolog extended with CH rules with an example that defines an inequality constraint.

```
% Constraint Declaration
(1a) constraint X≤Y.
(1b) callable X≤Y if ground(X).
(1b) callable X≤Y if ground(Y).

% Constraint Definition
(2a) X≤Y ← leq(X,Y).
(2b) leq(0,Y).
(2c) leq(s(X),s(Y)) ← leq(X,Y).

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

In clause (2a), $\leq$ is defined by a predicate `leq` which is defined by the two CLP clauses (2b) and (2c). The CH rules of (3) specify how $\leq$ simplifies and propagates as a constraint. CH rule (3a) states that `X≤X` is logically true. Hence, whenever we see the goal `X≤X` we can simplify it to `true`. Similarly, CH rule (3b) means that if we find `X≤Y` as well as `X≤Y` in the current goal, we can replace it by the logically equivalent `X=Y`. CH rule (3a) detects satisfiability of a constraint, and CH rule (3b) solves a conjunction of constraints returning a substitution. CH rule (3c) states that the conjunction `X≤Y,Y≤Z` implies `X≤Z`. The call

declaration (1) states that we may call X≤Y as a predicate if both X and Y are bound.

```
←  X≤Y, X=Y.
% by CH rule 3a
     true.

←  X≤Y, Y≤X.
% by CH rule 3b
     X=Y.

←  A≤B,C≤A,B≤C.
% C≤A,A≤B propagates C≤B by 3c.
% C≤B,B≤C simplifies to B=C by 3b.
% B≤A,A≤B simplifies to A=B by 3b.
     A=B,B=C.

←  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))) by 3c.
% s(s(0))≤s(s(s(0))) is callable and succeeds.
% s(s(0))≤A is callable and succeeds with A=s(s(X)).
% A≤s(s(s(0))) is callable and succeeds with X=0.
     A=s(s(0)).
% On backtracking A≤s(s(s(0))) succeeds with X=s(0).
     A=s(s(s(0))).
% On backtracking A≤s(s(s(0))) fails.
     false.
```

4 Temporal Reasoning in CLP+CH

We use Prolog+CH to define a constraint satisfaction algorithm for complete disjunctive binary constraint networks. Prolog provides for backtracking, while CH rules are used to implement path consistency at a high level of abstraction.

4.1 Incremental Constraint Satisfaction

Let the constraint c_{ij} be represented by the predicate `ctr(I,J,C)` where `C` is the list of primitive relations forming the disjunctive constraint. The basic operation of path consistency is $c_{ij} := c_{ij} \oplus c_{ik} \otimes c_{kj}$, which can be implemented by one CH rule performing the composition yielding an *indirect* constraint `newc` and another CH rule performing the intersection.

```
ctr(I,K,C1),ctr(K,J,C2) ⇒ composition(C1,C2,C3) | newc(I,J,C3).

newc(I,J,C1),ctr(I,J,C2) ⇔ intersection(C1,C2,C3) | ctr(I,J,C3).
```

We will see later on that the splitting into the two operations offers a high degree of flexibility. It separates the propagation step (first CH rule) and the simplification step (second CH rule). Furthermore, this formulation supports fine-grained concurrency in

that the second rule need not be executed *directly* after producing `newc(I,J,C3)`. Indeed, in the meantime the "old" `ctr(I,J,C2)` may be used by the first CH rule to produce a `newc` for another pair of nodes. Termination is guaranteed because the simplification CH rule replaces `C2` of `ctr` by the result `C3` of intersecting it with `C1` of `newc`. `C3` is thus either smaller or the same as `C2`. There is only a bounded number of times `C2` can get smaller. The latter case also poses no problem, as the implementation of CH rules will not add a new constraint but keep the old one in case of identity. Finally, propagation CH rules are never repeated for the same constraint goals.

A simple modification suffices to arrive at a incremental algorithm for incomplete networks. By *incremental* we mean that with every new constraint that arrives, the algorithm does the same amount of path consistency as in the complete(d) network where missing constraints are modeled as redundant constraints. Above, the intersection in CH rule (3) cannot be performed, because the `ctr` is missing. Furthermore, subsequent propagations from the resulting `ctr` are inhibited. The solution is to safely approximate the `ctr` as the redundant constraint, and thus the resulting new `ctr` would be identical to `newc`. So we can replace `newc` by `ctr` in the above implementation. If the simplification step is performed (sufficiently often) before a propagation step, then termination is guaranteed.

Algorithm. The complete algorithm below takes the optimizations of PC-2 [Mac77] into account. The optimization is based on the idea that if needed, c_{ij} can be computed as the converse of c_{ji}, which saves about half of the computation, and that c_{ii} can only be equality. Note that unlike PC-2, there is no need for a queue of modified constraints, as the new constraint goal itself will trigger new applications of the propagation CH rule.

```
% Constraint Declaration and Definition
(1a) constraint ctr(I,J,C).
(1b) callable ctr(I,J,C) if not singleton(C).
(1c) ctr(I,J,C) ← choose(B,C),ctr(I,J,B).

% Special Cases
(2a) ctr(I,J,C) ⇔ bound(I),bound(J) | check_ctr(I,J,K).
(2b) ctr(I,J,C) ⇔ empty(C) | false.
(2c) ctr(I,I,C) ⇔ contains_equality(C).

% Intersection
(3) ctr(I,J,C1),ctr(I,J,C2) ⇔ intersection(C1,C2,C3) | ctr(I,J,C3).

% Composition
(4a) ctr(I,K,C1),ctr(K,J,C2) ⇒ I<J,composition(C1,C2,C3) | ctr(I,J,C3).
(4b) ctr(K,I,C1),ctr(K,J,C2) ⇒ I<J,composition(C1,C3,C2) | ctr(I,J,C3).
(4c) ctr(I,K,C1),ctr(J,K,C2) ⇒ I<J,composition(C3,C2,C1) | ctr(I,J,C3).
```

The predicate `ctr(I,J,C)` is declared as a constraint (1a) and may be called if `C` represents a disjunction of primitive relations (1b). If the corresponding Prolog clause is executed (1c), the predicate `choose(B,C)` nondeterministically chooses one primitive constraint `B` from the disjunctive constraint `C`. This implements the backtrack search part of the algorithm. Special cases are simplification CH rules - (2a) that checks the validity of an assignment to variables, (2b) that detects inconsistent constraints and (2c) that removes constraints between the same nodes. The predicates `empty` and `singleton` are

simply defined as the corresponding list operations, while `contains_equality` depends on a predicate `equality` that gives the right kind of equality relation depending on the type of primitive relations in the list. Simplification CH rule (3) performs the intersection, propagation CH rules (4) the composition. Three CH rules are needed to cover all possible combinations of constraints while keeping the nodes `I,J` ordered.

The converse operation needed for (4b) and (4c) is implicit in how the composition predicate is used, producing an input from the other input and the output. This illustrates the declarative nature of logic programming that naturally supports converses. Intersection is simply defined as list intersection[5], while composition is defined in terms of pairwise combining the primitive relations. as defined by the predicate `comp`. As an example we give the table of CLP facts for composition of primitive qualitative point constraints.

```
comp(le,le,le).  comp(le,eq,le).
comp(le,ge,le).  comp(le,ge,eq).  comp(le,ge,ge).
comp(eq,le,le).  comp(eq,eq,eq).  comp(eq,ge,ge).
comp(ge,le,le).  comp(ge,le,eq).  comp(ge,le,ge).
comp(ge,eq,ge).  comp(ge,ge,ge).
```

The predicate `check_ctr` is implemented by trying the primitive constraints in the disjunction until one is found for which the assignment of the variables is valid. The check for validity is performed by simply using the definition of the primitive temporal constraints as CLP clauses. For example, in the case of point-interval constraints, we have

`check_p_i_c(X,[U,V],before)` $\leftarrow X < U < V.$
`check_p_i_c(X,[U,V],after)` $\leftarrow U < V < X.$
`check_p_i_c(X,[U,V],during)` $\leftarrow U < X < V.$
`check_p_i_c(X,[U,V],starts)` $\leftarrow X = U < V.$
`check_p_i_c(X,[U,V],finishes)` $\leftarrow U < X = V.$

Example. Consider the query `ctr(X,Y,[le,eq]), ctr(Y,Z,[ge,eq]), ctr(X,Z,[le])`. Applying CH rule (4a) to `ctr(X,Y,[le,eq]),ctr(Y,Z,[ge,eq])` adds `ctr(X,Z,[le,eq,ge])`. As the intersection with `ctr(X,Z,[le])` in CH rule (3) yields `[le]` again, only `ctr` is removed, but `ctr` is kept. Applying CH rule (4b) to `ctr(X,Y,[le,eq]), ctr(X,Z,[le])` adds `ctr(Y,Z,[le,eq,ge])`. Again, the corresponding `ctr` is not influenced. Applying CH rule (4c) to `ctr(X,Z,[le]),ctr(Y,Z,[ge,eq])` adds `ctr(X,Y,[le])`. This time CH rule (3) simplifies `ctr(X,Y,[le]),ctr(X,Y,[le,eq])` to `ctr(X,Y,[le]`. The new `ctr` triggers application of CH rules (4a) and (4b) again. Once more, only redundant `ctr` are produced. No new `ctr` is added, no new CH rules are triggered and thus a nondeterministic unfolding step initiated. The only callable goal left is `ctr(Y,Z,[ge,eq])`. It is unfolded and `choose` executed resulting in a new constraint `ctr(Y,Z,[ge])`. The new constraint triggers two propagations, but again the resulting `ctr` do not change the corresponding `ctr`. All constraints have singleton relation lists now, thus none is callable and the computation terminates with `ctr(X,Y,[le]), ctr(Y,Z,[ge]), ctr(X,Z,[le])` . On backtracking, `choose` chooses the other primitive relation `eq` and in an analogical way, the result `ctr(X,Y,[le]), ctr(Y,Z,[eq]), ctr(X,Z,[le])` is produced. No other solution exists. Of course, applying the propagation CH rules in a different order (or in parallel) would have avoided the initial redundant computations involving CH rules (4a) and (4b).

[5] For quantitative constraints, a special implementation has to be used

4.2 Flexibility

In this subsection we illustrate how CH rules support rapid prototyping and easy modification. First we consider some optimizations, then modifications and specializations.

Special Cases. A *redundant* constraint is one that does not impose any restrictions on the relation between its variables, it is logically equivalent to `true`. By definition, intersection with a redundant constraint will just return the other constraint and composition with a redundant constraint will result in another redundant constraint. Therefore we can safely remove redundant constraints by adding the following simplification CH rule to our implementation:

```
ctr(I,J,C) ⇔ redundant(C) | true.
```

If two variables are constrained to be equal, we can actually make them equal by using the built-in CS for equality (the unification algorithm) of Prolog. Again, this is a logically sound simplification that does not influence the result of the constraint solving, but can significantly reduce the number of rule applications. The simplification corresponds to merging the corresponding nodes in the network.

```
ctr(I,J,C) ⇔ equality(C),singleton(C) | I=J.
```

Backtracking. Another optimization, which is also used in [LaRe92], deals with the backtrack search. The idea follows the *first fail principle* in choosing more constraining primitive relations first. This can be achieved by a modification of the `choose` predicate. The most constraining primitive relation is equality. For constraints over intervals, relations that equate end-points of intervals are more constraining (e.g. `starts, finishes, meets`). In quantitative constraints intervals with a smaller length (especially single values) are considered first.

Depending on the class of network, the interaction between path consistency and backtrack search can be modified as well. So far, the implementation strictly prefers consistency checking over backtrack search. By restricting the applicability of the propagation CH rules earlier backtracking can be introduced. In experiments this has proven useful for almost minimal, inconsistent or under-constrained networks. It suffices to restrict one of the constraints involved in a propagation to be disjunction-free, i.e. to add to the guard a check that the relation list is a singleton. This not only reduces the average size of the resulting constraint but also makes composition more efficient. Though the cost is bounded, it is proportional to the product of the size of the involved constraints. So if one constraint is primitive, the cost is linear in the size of the other constraint.

We can also use backtracking to enhance the quality of propagation. For example it is known that for so-called *convex* constraints path consistency produces the minimal network. An important class of convex constraints are qualitative point constraints without inequality. Thus an inequality `[le,ge]` is split. The idea then is to split a constraint into a disjunction of several convex constraints and to backtrack between them. It has also been observed by Reinefeld, that these so-called "pointisable" interval constraints should be preferred.

Propagation. Another very effective way[6] to avoid too much propagation is based on

[6] Our experiments showed a dramatic improvement in almost all cases

the observation that composition is an associative operation. Thus, given e.g. three constraints sharing variables, it does not matter if we first propagate from the first two, and then use the result to propagate from the third, or if we propagate and the last two and use the result to propagate from the first. Because of associativity of composition we can restrict the propagation CH rules to involve at least one direct constraint. We can distinguish the two kinds of constraints by introducing an additional argument that contains the information.

Another optimization avoids *direct back-propagation* which is always redundant. The effect is as follows: If we propagate from two constraints, the new constraint (if it is not simplified) may immediately propagate from one of the direct constraints. Clearly the resulting constraint cannot be more specific than the corresponding direct constraint. This useless propagation can be avoided by remembering for propagation constraints "where they came from", i.e. to keep the third variable that was involved in the propagation and to dissallow any new propagation that involves this variable. This method could be refined by remembering the complete path for indirect constraints and avoiding any nodes therein, however the additional space requirement and time needed to inspect the paths may outweigh the gain of the extension. On the other hand, a path can also serve as an explanation how the constraints were indirect.

Specialization. If a temporal constraint network consists of only disjunction-free (primitive) initial constraints, then we can choose one node as start node k and the precise indirect constraint between any two nodes i, j can be computed from c_{ki} and c_{kj} (by a single propagation). In the general case this propagation would only result in an approximation of the actual constraint. This means that we can restrict the indirect constraints used during constraint satisfaction to include the start node. Given n nodes, instead of n^2 indirect constraints, n suffice, and the number of propagation steps is reduced accordingly.

Based on this observation we give an example of specialization to quantitative constraints over single intervals as considered in [DMP91]. Their notation for `ctr(I,J,[A..B])` is `A=<I-J=<B`. The starting node is specified by the constraint *start*.

```
constraint start(X). % gives starting node
constraint A=<X-Y=<B. % direct constraint
constraint A=<*X-Y=<*B. % indirect constraint

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

A=<*X-Y=<*B <=> A>=B | false.
A=<*X-Y=<*B <=> A=(-inf),B=(+inf) | true.
A=<*X-X=<*B <=> A=<0,0=<B.
A=<*X-Y=<*B <=> A=0,B=0 | X=Y.

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

Finite domains are unary quantitative constraints. They can be modeled by binary con-

straints whose first argument is zero. Because of this, no propagation is necessary. The treatment of equality is different. We use the notation `I::C` of CHIP [D*88] instead of `ctr(0,I,C)`. The choice for backtrack search is modified. Instead of choosing a primitive constraint, a value from the quantitative constraint is chosen.

```
% Constraint Declaration and Definition constraint I::C.
callable I::C if not singleton(C).
I::C ← choose(B,C), I::B.

% Special Cases
I::C ⇔ C=[] | false.
I::C ⇔ C=[J],single_value(J) | I=J.
I::C ⇔ redundant(C) | true.

% Intersection
I::C1,I::C2 ⇔ intersection(C1,C2,C3) | I::C3.
```

Modification. Finally, a modification of the path consistency algorithm to compute the shortest paths between any pair of nodes is presented next. Instead of a disjunctive constraint, the third argument contains the cost.

```
constraint path(I,J,C).
path(I,J,C1),path(I,J,C2) ⇔ minimum(C1,C2,C3) | path(I,J,C3).
path(I,K,C1),path(K,J,C2) ⇒ I<J,add(C1,C2,C3) | path(I,J,C3).
path(K,I,C1),path(K,J,C2) ⇒ I<J,add(C1,C3,C2) | path(I,J,C3).
path(I,K,C1),path(J,K,C2) ⇒ I<J,add(C3,C2,C1) | path(I,J,C3).
```

5 Conclusions

Regarding temporal reasoning as constraint satisfaction problem, we have illustrated that constraint logic programming extended with constraint handling rules provided for

- a level of abstraction suited for integration of different representations
- an incremental path consistency algorithm
- backtracking search through integration with logic programming
- preservation of declarative semantics of logic programming
- flexibility supporting rapid prototyping and ease of modification

A class of modifications we have not considered so far is to take the topology of the constraint graph into account. For example, a tree as temporal constraint network is always path consistent and minimal. We are also looking at extension to take durations of intervals into account [All83].

While our approach cannot match the speed of hard-coded constraint solvers, we think it is still suitable for sparse heterogeneous networks as they are likely appear in real-life (scheduling). If necessary, a constraint solver defined by CH rules once fully tested and found correct, can always be rewritten in a low-level language to improve efficiency. We are experimenting with compilation of CH rules into the host language.

A constraint solver defined by confluent CH rules can run concurrently if the underlying CHE executes the applications of CH rules concurrently. This is a topic of on-going research. Future work will look at applications of temporal constraint logic programming in planning, scheduling and temporal deductive databases.

Acknowledgements: Thanks to Alex, Carmen, Mark, Pascal, Sury and Thierry for discussions and comments on the work presented in this paper.

Bibliography

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

[Coh90] J. Cohen, Constraint Logic Programming Languages, Communications of the ACM 33(7):52-68, July 1990.

[D*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.

[Fru92] T. Frühwirth, Constraint Simplification Rules (old name for CH rules), Technical Report ECRC-92-18, ECRC Munich, Germany, July 1992 (revised version of internal Report ECRC-91-18i, October 1991).

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

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

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

[LaRe92] P. B. Ladkin and A. Reinefeld, Effective Solution of Qualitative Interval Constraint Problems, Journal of Artificial Intelligence 57:105-124, 1992.

[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.

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

[Sar89] V. A. Saraswat, Concurrent Constraint Programming Languages, Ph.D. Dissertation, Carnegie Mellon Univ., also TR CMU-CS-89-108, 1989.

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

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

[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.

Appendix I - Extended Implementation

In this appendix we present an efficient extended implementation and in the next appendix some examples of using it.

The constraint `ctr` is extended by some arguments that carry additional useful information. The initial constraints in addition hold the type of the two temporal variables in the last argument, This is necessary as Prolog is an untyped language. These initial constraints are then replaced by ones with more information. The first argument now gives the size of the disjunctive constraints (the number of primitive constraints, i.e. the size of the list). The last argument holds the length of the shortest path from which the constraint was derived. This is also used to distinguish between direct (path length equals 1) and indirect constraints. This is the basis of an optimization used in the composition. Instead of requiring that the constraint variables are ordered, Intersection is therefore extended by another CH rule to deal with conflicting ordering of variables in the arguments. This variant proved to be faster than the one introduced in the paper. Also various special cases as discussed in the main body of the paper are taken into account.

```
% Constraint Definition
constraint ctr(X,Y,Rels,Type).
constraint ctr(RelsSize,X,Y,Rels,Type,PathL).
callable ctr(N,X,Y,L,T,I) if N>1.
ctr(N,X,Y,L,T,I) ← member(R,L),ctr(1,X,Y,[R],T,I).

% Initialize
ctr(X,Y,L,T) ⇔ length(L,N) | ctr(N,X,Y,L,T,1).

% Special cases
ctr(N,X,Y,L,T,I) ⇔ empty(N,L,T) | false.
ctr(N,X,Y,L,T,I) ⇔ redundant(N,L,T) | true.
ctr(N,X,Y,L,T,I) ⇔ X=Y | contains_equality(L,T).
ctr(N,X,Y,L,T,I) ⇔ singleton(N,L),equality(L,T) | X=Y.

% Intersection
ctr(N1,X,Y,L1,U-V,I),ctr(N2,X,Y,L2,U-V,J) ⇔
inter(L1,L2,L3,U-V),length(L3,N3), K is min(I,J) | ctr(N3,X,Y,L3,U-V,K).
ctr(N1,Y,X,L1,U-V,I),ctr(N2,X,Y,L,V-U,J) ⇔
equality([Eq],V-V),comp(L,L2,[Eq],V-U-V), inter(L1,L2,L3,U-V),length(L3,N3),
K is min(I,J) | ctr(N3,Y,X,L3,U-V,K).
```

```
% Composition
ctr(N1,X,Y,L1,U-V,I),ctr(N2,Y,Z,L2,V-W,J) ⇒ J=1,
comp(L1,L2,L3,U-V-W),length(L3,M), K is I+J | ctr(M,X,Z,L3,U-W,K).
ctr(N1,X,Y,L1,U-V,I),ctr(N2,X,Z,L3,U-W,J) ⇒ J=1,
comp(L1,L2,L3,U-V-W),length(L2,M), K is I+J | ctr(M,Y,Z,L2,V-W,K).
ctr(N1,X,Y,L3,U-V,I),ctr(N2,Z,Y,L2,W-V,J) ⇒ J=1,
comp(L1,L2,L3,U-W-V),length(L1,M), K is I+J | ctr(M,X,Z,L1,U-W,K).
```

Appendix II - Examples

The examples are taken from actual runs of the program in appendix 1. `sirs_only` means that only the CH rules are run (to approximate the minimal network by path consistency), but no choices are made. This execution mode is supported by the underlying CHE. It is shown how many single and multi-headed propagation and simplification CH rules have fired. In the types, `p` (`i`) stands for points (intervals).

Example for Points. Instead of $<, =, >$ (a..b) the notation `le,eq,ge` (`a-b`) is used.

```
←  sirs_only
ctr(X,Y,[le,eq],p-p),
ctr(Y,Z,[eq],p-p),
ctr(X,Z,[le],p-p).
PropS/PropM/SimpS/SimpM = 3 / 0 / 1 / 1
YES Z = Y,
ctr(1, X, Y, [le], p-p, 1).
```

```
←  sirs_only
ctr(X,Y,[0-0,1-2,4-5],p-p),
ctr(Y,Z,[0-0,3-4],p-p),
ctr(X,Z,[1-1],p-p).
PropS/PropM/SimpS/SimpM = 3 / 2 / 1 / 3
YES Z = Y,
ctr(1, X, Y, [1-1], p-p).
```

```
←  sirs_only
ctr(X,Y,[eq,le],p-p),
ctr(Y,Z,[le],p-p),
ctr(Z,W,[1-1,3-7],p-p),
ctr(Y,W,[0-3],p-p).
PropS/PropM/SimpS/SimpM = 4 / 25 / 5 / 18
YES ,
ctr(1, X, W, [1-∞], p-p, 2),
ctr(1, X, Y, [0-∞], p-p, 1),
ctr(1, X, Z, [sup-∞], p-p, 2),
```

```
ctr(1, Y, W, [1-3], p-p, 1),
ctr(1, Z, Y, [-2--sup], p-p, 1),
ctr(2, Z, W, [1-1, 3-3], p-p, 1).
```

Example for Intervals. This example illustrates that path consistency does not necessarily compute the minimal network.

```
← sirs_only
PT=i-i,
L1=[during,contains,overlaps,overlapped_by, meets,finishes,finished_by],
L2=[equals,before,contains,overlaps,starts, started_by,finished_by],
L3=[before,during,overlaps,finishes,finished_by],
L4=[before,during,overlaps,starts],
ctr(D,A,L1,PT),   ctr(B,A,L1,PT),
ctr(B,C,L1,PT),   ctr(A,C,L2,PT),
ctr(D,B,L3,PT),   ctr(D,C,L4,PT). PropS/PropM/SimpS/SimpM = 6 / 20 / 14 / 6
```

No change.

First solution.

```
PropS/PropM/SimpS/SimpM = 6 / 87 / 41 / 46
ctr(1, A, C, [contains], i-i, 1),
ctr(1, B, A, [during], i-i, 1),
ctr(1, B, C, [during], i-i, 1),
ctr(1, D, A, [during], i-i, 1),
ctr(1, D, B, [before], i-i, 1),
ctr(1, D, C, [before], i-i, 1).
```

Add `ctr(B,A,[meets],PT)`, CH rules only

```
PropS/PropM/SimpS/SimpM = 7 / 15 / 2 / 10
false.
```

Example Point-Intervals. The last example uses all types of temporal constraints introduced in this paper.

```
← sirs_only
ctr(X,Y,[before,starts],p-i),
ctr(X,Z,[starts,during],p-i),
ctr(Y,Z,[before,contains,after],i-i).
PropS/PropM/SimpS/SimpM = 3 / 7 / 3 / 4
YES ,
ctr(1,X,Y,[before],p-i,1),
ctr(1,Z,Y,[before],i-i,1),
ctr(2,X,Z,[starts,during],p-i,1).
```

```
← sirs_only
```

```
ctr(V,U,[0-1,3-4],p-p),
ctr(U,Y,[before,starts],p-i),
ctr(Z,V,[contains,started_by],i-p),
ctr(Y,Z,[before,contains],i-i).
PropS/PropM/SimpS/SimpM = 4 / 4 / 1 / 1
false.
```