Constraint Handling Rules
Applications for abduction and diagnosis

Henning Christiansen
Roskilde University, Computer Science Section
KIIS course, 16 September 2008

1. What is constraint logic programming?
2. Intro to CHR + small exercise
3. What is abduction (*à la* Pierce)?
4. Abduction in Prolog + a little help from CHR
5. Diagnosis as abduction + an exercise

1 Constraint Logic Programming & Constraint Handling Rules

Historical background

Constraint logic programming, e.g., CLP($\mathcal{R}$), since 80'es or earlier.
Idea is to treat arithmetic (etc.) in logic programming, with *all* its nice properties:
Reversibility, delayed commitment

Early constraint solvers written as "black-boxes" in C or similar

CHR: an attempt to provide a white-box approach $\approx$ *declarative* language for writing constraint solvers; early 90es; in SICStus from 1998

CHR has proved itself successful for a wide range of other application, e.g.,

- Artificial intelligence and reasoning (e.g., DemoII)
- Language processing (CHR Grammars, CHRG)
- General algorithmic language (?!?!?)
- ... check out CHR's www pages for many other applications

Several implementations; significant improvements and optimizations

1.1 Syntax and semantics of CHR

CHR is an extension of Prolog, inherits its nomenclature, and extends with new sorts of rules:

$$\text{Simplification rules: } c_1, \ldots, c_n \texttt{ <=> } \mathit{Guard} \mid c_{n+1}, \ldots, c_m$$
$$\text{Propagation rules: } c_1, \ldots, c_n \texttt{ ==> } \mathit{Guard} \mid c_{n+1}, \ldots, c_m$$

Operational semantics: Transformations on a *constraint store*.
Simp's replace constraints, Prop's add new ones.
Committed choice
NB: Uses one-way matching in the head and not unification!

Declarative semantics: Indicated by arrow, bi-implic. & implic.

In practice: Body can be any Prolog code;
– op.sem. is try from top when constraint is called
– left-to-right, depth first

One more kind of rules:

$$\text{Simpagation rules: } c_1, \ldots, c_i \setminus c_{i+1}, \ldots c_n \texttt{ <=> } \mathit{Guard} \mid c_{n+1}, \ldots, c_m$$
$$\text{which can be though of as: } c_1, \ldots, c_n \texttt{ <=> } \mathit{Guard} \mid c_1, \ldots, c_i, c_{n+1}, \ldots, c_m$$

But operationally a bit smarter.

1.2 Example: Greatest common divisor

```
:- use_module( library(chr)).
handler gcd.
constraints gcd/1.

gcd(0) <=> true.
gcd(N) \ gcd(M) <=> N=<M | L is M-N, gcd(L).
```

Here are a few test queries.

```
?- gcd(2),gcd(3).
?- X is 37*11*11*7*3, Y is 11*7*5*3, Z is 37*11*5, gcd(X), gcd(Y), gcd(Z).
```

1.3 Another example: Prime numbers

```
:- use_module(library(chr)).
handler primes.
constraints primes/1, prime/1.

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

1.4 Exercise about CHR

This exercise concerns the reconstruction of a constraint solver applied in a system used for generating UML from use case text written in natural language. See separate document about this.

2 Abductive logic programming in Prolog and CHR

2.1 Deduction, abduction, and induction in logic programming

C.S.Peirce (1839–1914)

Three principles for human reasoning ("scientific discovery") as *the* fundamental ones:

- **Deduction**, reasoning within the knowledge we have already, i.e., from those facts we know and those rules and regularities of the world that we are familiar with. E.g., reasoning from causes to effects:
  "If you make a fire here, you will burn down the house."

- **Induction**, finding general rules from the regularities that we have experienced in the facts that we know; these rules can be used later for prediction:
  "Every time I made a fire in my living room, the house burnt down, aha, ... the next time I make a fire in my living room, the house will burn down too".

- **Abduction**, reasoning from observed results to the basic facts from which they follow, quite often it means from an observed effect to produce a qualified guess for a possible cause:
  "The house burnt down, perhaps my cousin has made a fire in the living room again."

2.2 Simplistic-formalistic example of abduction

Given knowledge base $\{a \rightarrow c, b \rightarrow c\}$.
Observing c.
Concluding a.

NB: Abduction is not sound! (explanation follows)

2.3 Applications of abduction/abductive logic programming

- Fault diagnosis
- Planning
- Natural language analysis, discourse analysis

2.4 Abduction implemented in Prolog with a little help from CHR

Abductive logic programming: A usual Prolog program; the system is allowed to invent new facts of *abducible* predicates; *integrity constraints* to exclude nonsense sets of abducibles.

Analogy to CHR constraints and rules observed by Slim Abdennadher & your teacher (2000), and elaborated in later works.

```
:- use_module(library(chr)).
handler abduction.
constraints a/1.
a(1), a(2) ==> fail.
```

2.5 Example: Database view update considered as abduction

Tables: `father`, `mother`
View: `grandfather`
Integrity constraint: "you can only have one father and only one mother"

```
:- use_module(library(chr)).
handler view_update.

constraints father/2, mother/2.

father(X,Y), father(Z,Y) ==> Z=X.
mother(X,Y), mother(Z,Y) ==> Z=X.

grandfather(X,Z):- father(X,Y), father(Y,Z).
grandfather(X,Z):- father(X,Y), mother(Y,Z).

current_db:- father(peter, paul), father(paul,jens), mother(marie,jens).
```

A query that corresponds to updating through a view:

```
?- current_db, grandfather(X,jens).
```

Intuitively: In what ways can facts be added so the "observation" becomes true.

```
?- current_db, grandfather(X,jens).
X = peter,
father(peter,paul),
father(paul,jens),
mother(marie,jens),
father(peter,paul),
father(paul,jens) ? ;

father(peter,paul),
father(paul,jens),
mother(marie,jens),
father(X,marie),
mother(marie,jens) ? ;
no
```

2.6 Simple diagnosis problems formulated as abduction

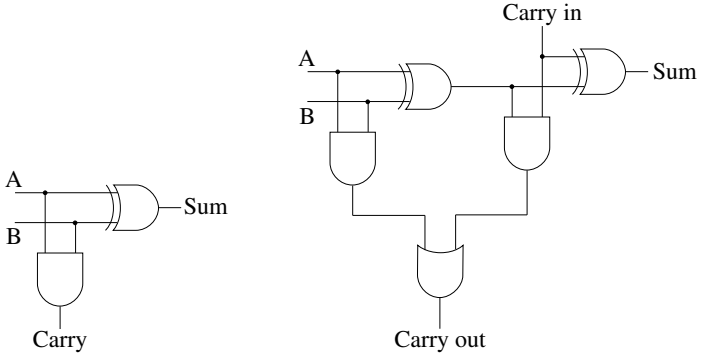
"Definition": A complex system shows faulty behaviour; which component(s) is responsible for the fault?

Different assumptions are possible:

- Periodic faults
- Consistent faults $\Leftarrow$
- — " — " — " — with correct behaviour produced in correct way $\Leftarrow$

2.7 Example: Faults in Logical circuits

Recall:



Extend familiar Prolog program with identifiers for each component:

```
halfadder(A, B, Carry, Sum):-
        and(A, B, Carry, and0),
        xor(A, B, Sum, xor0).

fulladder(Carryin, A, B, Carryout, Sum):-
        xor(A, B, X, xor1),
        and(A, B, Y, and1),
        and(X, Carryin, Z, and2),
        xor(Carryin, X, Sum, xor2),
        or(Y, Z, Carryout, or1).
```

Extend rules for logical components accordingly:

```
and(A,B,X,ComponentId):-
   and(A,B,X),  % -- the usual truth table pred.
   perfect(ComponentId).

and(A,B,X,ComponentId):-
   and(A,B,Z), disturbe(Z,X),
   defect(ComponentId).

disturbe(0,1).
disturbe(1,0).
```

With this Prolog program and *given* facts for `perfect` and `defect` we can predict the behaviour!

But diagnosis is the other way round!!

2.8 Diagnosis based on the assumption of consistent faults

Abducibles: `perfect`, `defect` extended with detailed information of for which input values it is de/per-fect

```
:- use_module(library(chr)).
handler diagnosis.
constraints perfect/3, defect/3.

defect(X,In1,In2)  \ defect(X,In1,In2)  <=> true.
perfect(X,In1,In2) \ perfect(X,In1,In2) <=> true.
defect(X,In1,In2)  , perfect(X,In1,In2) <=> fail.
```

Rules for components adjusted accordingly:

```
and(A,B,X,ComponentId):-
   and(A,B,X),
   perfect(ComponentId,A,B).
```

```
and(A,B,X,ComponentId):-
   and(A,B,Z), disturbe(Z,X),
   defect(ComponentId,A,B).
```

The following is an example of a query (≈ observation) that consists of one wrong and one correct observation.

```
?- halfadder(1,1,0,1), halfadder(1,0,0,1).
defect(and0,1,1),
defect(xor0,1,1),
perfect(and0,1,0),
perfect(xor0,1,0) ? ;
no
```

We see:

- Only one answer is given for *this* example.
- How throughly has the circuit been tested?
  Notice: `and0`'s behaviour on input `0-0` not tested.
  For thorough test, try more samples so that `defect(and0,0,0)` or `perfect(and0,0,0)` may show up.

2.9 Adding correct-results-produced-in-correct-way assumption

This query of three correct and three incorrect observation:

```
?- fulladder(0,1,1,1,0), fulladder(0,1,0,0,1), fulladder(0,0,1,0,1),
   fulladder(1,0,1,1,1), fulladder(1,1,1,0,0), fulladder(0,0,0,0,1).
```

Produces 48 solutions, but

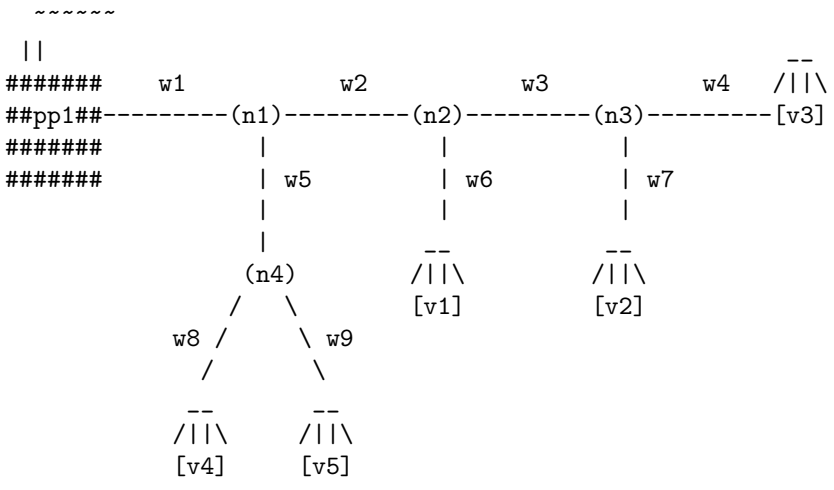
```
?- fulladder(0,1,1,1,0), fulladder(0,1,0,0,1), fulladder(0,0,1,0,1),
   !,
   fulladder(1,0,1,1,1), fulladder(1,1,1,0,0), fulladder(0,0,0,0,1).
```

produces only one:

```
defect(xor2,1,1),
defect(and2,0,1),
defect(xor2,1,0),
defect(or1,1,1),
defect(xor1,0,0)
```

(This is a hack but it works :)

2.10 Exercise of section 9.2: Diagnosis of power supply networks



Observations: Sets of `has_power(`v`)` or `has_no_power(`v`)` where v refers to a village.
In a given situation, we may have such information about all or just some villages.

Diagnosis (abducibles): Sets of `state(`w`,up)` and `state(`w`,down)` where w refers to a wire or the power plant `pp1`.

For simplicity: Nodes always are working (so no `up` or `down` information for those).
However, it is convenient to define the `has_power` and `has_no_power` predicates so they take as arguments both villages and nodes.

These predicates should, then, be defined as Prolog predicates (for the set of nodes and villages) that depend on the `state` of incoming wire and the possible `has_power` or `has_no_power` for the previous node.

In order to make the work with the exercise easier, a Prolog source file that models this power supply network is given, file `deductivePower` which is available from the course schedule web page. However, it does not include the diagnosis part. It is written as a purely deductive program which contains as Prolog facts an arbitrary sample of `state(···,{up,down})` information.

A few test predicates are supplied:

```
observe_all_butV3:-
  has_power(v1), has_power(v2), has_no_power(v3),
  has_power(v4), has_power(v5).

observe_all_butV1V3:- .... .

observe_V4V5:- .... .

observe_total_darkness:- .... .
```