The DLV ASP System

An implementation of a deductive database system

Computes the answer sets (intended models), and stable models for non-extended programs, in particular, of (extended) disjunctive normal programs

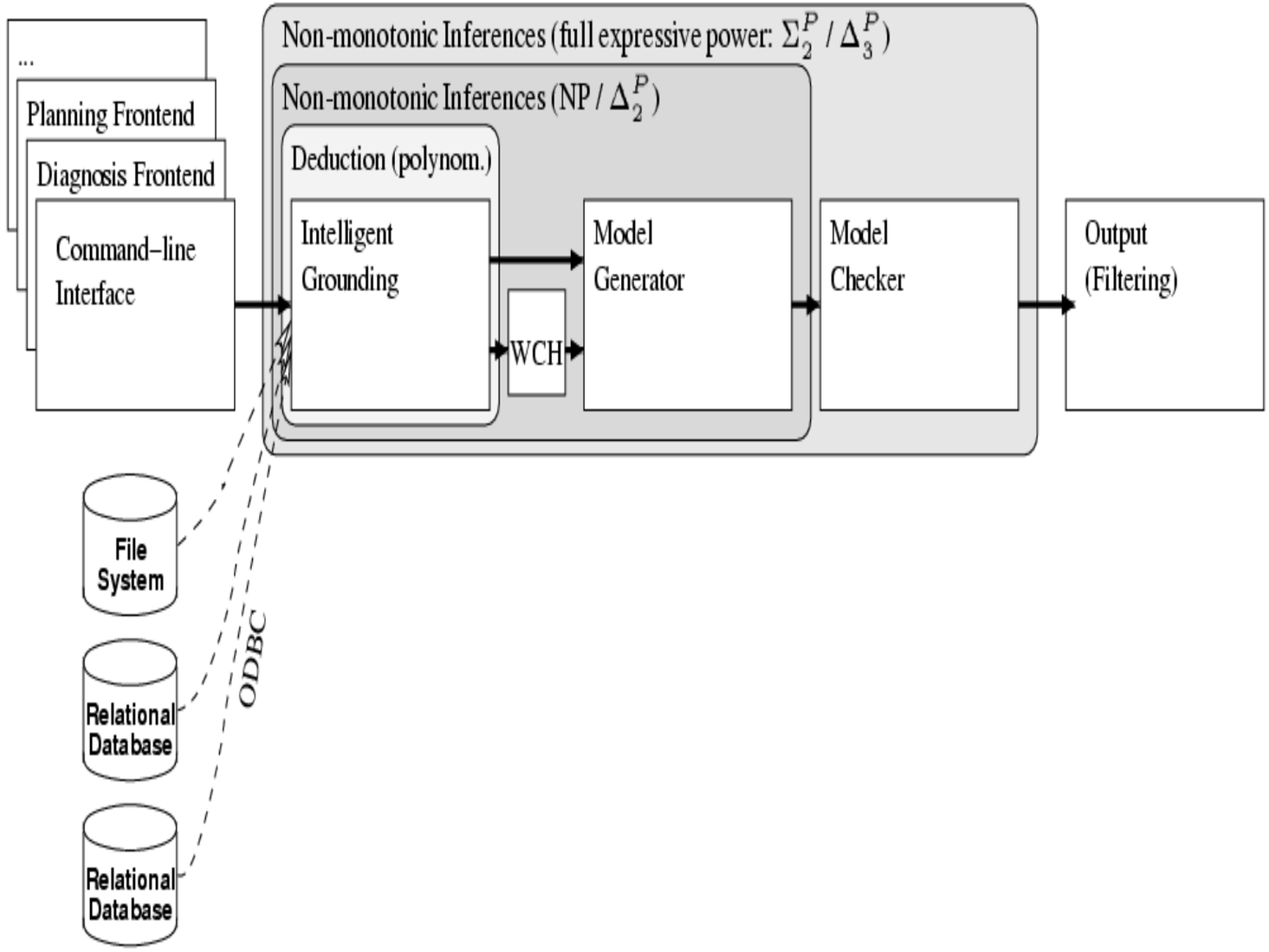
There may be head disjunctions and classical negation coexisting with weak negation

```
p(X) v Q(a) :- s(X,Y), not -r(Y).
```

It can do query answering under the answer-set (stable-model) semantics

Programs may include disjunction, weak negation ("`not`" above) and strong (or classical) negation ("$-$" above)

It offers nice and useful interfaces to commercial relational DBMSs


...
Planning Frontend
Diagnosis Frontend
Command–line Interface
Non-monotonic Inferences (full expressive power: Σ_2^P / Δ_3^P)
Non-monotonic Inferences (NP / Δ_2^P)
Deduction (polynom.)
Intelligent Grounding
WCH
Model Generator
Model Checker
Output (Filtering)
File System
Relational Database
Relational Database
ODBC

- Naively grounding the original program will create a huge ground program, and processing it will be extremely costly

  Better ground as needed, ”intelligently”

- Guesses for models are not chosen randomly, and models are not built from scratch

  It is possible to prove that all the stable models have a large intersection that can be computed in polynomial time

  Individual models are guessed (generated) pivoting from that core, and then put to the test (checked)

To run DLV write the following command in a command window:

$>$ DLV -stats *input_file* $>$*output_file*

Option -stats prints on screen the statistics of the run

In output file one gets the answer sets of the program given in the input file

- `:-` stands for $\longleftarrow$
- Every rule ends with a dot
- A constant is a string of letters, numbers and underscores, beginning with a lowercase letter or number:

  `a1, 1, 9862, aBc1, c__`

- A variable is a string of letter, numbers and underscores that begins with an uppercase:

  `A, V2f, Vi_X3`

- Predicate symbols begin with a letter and may contain letters, underscores and digits: `ord, oBp, r2D2, E_mc2`

- not is weak negation; and "-" or "$\sim$" are strong (classical) negation
- The literals in the body are separated by commas; and in the head, by `v`

Example: Input file for the program $p(a) \leftarrow not\ p(b)$ is:

```
%File: ex1.dlv
p(a) :- not p(b).
```

The program `ex1.dlv` is run using the command:

```
> DLV -stats ex1.dlv >ex1.out
```

The output is automatically stored in the text file `ex1.out`:

```
DLV [build BEN/Jun 26 2003   gcc 2.95.3-10 (cygwin special)]
{p(a)}
```

The statistics of the run are shown in the command window:

```
Residual Instantiation
 Rules                       : 1
 Constraints                 : 0
 Weak Constraints            : 0
 Structural Size             : 2
 Maximum recursion level     : 0
 Choice Points               : 0
 Answer sets printed         : 1
 Time for first answer set : 0.016000
                           (including instantiation)
 Time for all answer sets  : 0.016000
                           (including instantiation)
 Instantiation time          : 0.000000
 Model Generator time        : 0.015000
 Model Checker time          : 0.000000
 Total Model Check time      : 0.000000
 Partial Model Check time    : 0.000000
```

Example: The Senators are organizing a party, and they want to invite all their friends, and the friends of their friends, and the friends of the friends of the friends, etc. However, they want to make sure no Maple Leaf player is invited to the party

```
player(mats,mapleLeaf).
player(bryan,mapleLeaf).
player(owen,mapleLeaf).
player(gary,mapleLeaf).
player(joe,mapleLeaf).
player(darcy,mapleLeaf).
player(marian,senators).
player(daniel,senators).
player(martin,senators).
player(jason,senators).
player(radek,senators).
player(wade,senators).
```

```
friend(wade,kevin).
friend(bryan,carolina).
friend(daniel,carolina).
friend(cathy,bryan).
friend(daniel,monica).
friend(radek,natalia).
friend(radek,cristal).
friend(cristal,isabel).
friend(radek,mary).
friend(mary,jason).
friend(ann,radek).
friend(radek,cristina).
```

```
friend(X,Y) :- friend(Y,X).

transitive_friend(X,Y) :- friend(X,Y).
transitive_friend(X,Y) :- transitive_friend(X,Z),
                          friend(Z,Y).

list(X) :- player(X,senators).
list(X) :- player(Y,senators),
                       transitive_friend(Y,X).

-welcome(X) :- player(X,mapleLeaf).

invited(X) :- list(X), not -welcome(X).
```

Since the program is stratified, with no disjunction, we get only one stable model:

```
{player(mats,mapleLeaf), ... , player(marian,senators),
..., friend(bryan,carolina), ..., friend(carolina,daniel),
... , transitive_friend(bryan,daniel), ... ,
transitive_friend(natalia,jason), ... , list(bryan),
list(marian), list(daniel), list(martin), list(jason),
list(radek), list(wade), list(kevin), list(carolina),
list(cathy), list(monica), list(natalia), list(cristal),
list(isabel), list(mary), list(ann), list(cristina),
-welcome(mats),-welcome(bryan), -welcome(owen),
-welcome(gary), -welcome(joe), -welcome(darcy), invited(marian),
invited(daniel),invited(martin), invited(jason), invited(radek),
invited(wade), invited(kevin), invited(carolina), invited(cathy),
invited(monica),invited(natalia),invited(cristal),invited(isabel),
invited(mary),invited(ann), invited(cristina)}
```

Constraints in DLV:

DLV can handle denial constraints (rules with no head)

$$\leftarrow P(X), not\ Q(X).$$

This restriction says that for all the values of X we cannot have $P(X)$ and not have $Q(X)$

Only stable models that satisfy those program constraints are returned by the program

Program constraints, like integrity constraints in databases, capture more of the semantics of the application

By pruning out undesired models (so as ICs exclude certain DB states)

Example: Is the map of Alberta (A), Saskatchewan (S),
British Columbia (B) and Northwest Territories (N) three colorable?

An instance of a NP-complete combinatorial problem ...

Input File:

```
[commandchars=\\\{\}]
 % Extensional DB (EDB)
 edge(a,s). edge(a,b). edge(s,n). edge(a,n).
 edge(b,n).

 % Intensional DB (IDB)
 node(X) :- edge(X,Y).
 node(Y) :- edge(X,Y).

 colored(X,red) v colored(X,green) v colored(X,blue)
                              :- node(X).          (*)
         :- edge(X,Y), colored(X,C), colored(Y,C).
```

- In general, a disjunctive rule, like (*), unless forced otherwise by the other program rules, interprets a disjunction in an exclusive manner (by minimality)

- There are 5 possible ways to paint the map as shown by the following stable models

  One stable model per solution of the 3-coloring problem:

```
%[fontsize=\large]
{edge(a,s), edge(a,b), edge(a,n), edge(s,n), edge(b,n), node(a),
node(s), node(b), node(n), colored(a,red), colored(s,green),
colored(b,green), colored(n,blue)}

{edge(a,s), edge(a,b), edge(a,n), edge(s,n), edge(b,n), node(a),
node(s), node(b), node(n), colored(a,green), colored(s,red),
colored(b,red), colored(n,blue)}

{edge(a,s), edge(a,b), edge(a,n), edge(s,n), edge(b,n), node(a),
node(s), node(b), node(n), colored(a,red), colored(s,blue),
colored(b,blue), colored(n,green)}

{edge(a,s), edge(a,b), edge(a,n), edge(s,n), edge(b,n), node(a),
node(s), node(b), node(n), colored(a,blue), colored(s,red),
colored(b,red), colored(n,green)}

{edge(a,s), edge(a,b), edge(a,n), edge(s,n), edge(b,n), node(a),
node(s), node(b), node(n), colored(a,blue), colored(s,green),
colored(b,green), colored(n,red)}

{edge(a,s), edge(a,b), edge(a,n), edge(s,n), edge(b,n), node(a),
node(s), node(b), node(n), colored(a,green), colored(s,blue),
colored(b,blue), colored(n,red)}
```

- This program does not seem to contain weak negation

  Actually, there is one hidden in the program constraint (PC)

- If we want to transform a PC, say: $\leftarrow B(\bar{x})$ into a regular headed-rule, we need something like: $q \leftarrow B(\bar{x}),\ not\ q$

  Here, q a fresh, propositional atom

  If in a Herbrand structure $B(\bar{a})$ becomes true for some $\bar{a}$, then in it $q \leftarrow not\ q$ will hold

  No model can make this hold, so the structure cannot be a model (the same effect as discarding it via the original PC)

- This particular program falls in a nice syntactic class of disjunctive programs with negation (head-cycle-free programs)

  For it, the rule (*) can be replaced -in a standard manner- by three non-disjunctive rules (cf. slide 27)

```
colored(X,red)  :- node(X), not colored(X,green),
                           not colored(X,blue).
ETC.
```

The new program has the same stable models as the original one

Not every program allows this transformation

- If we try to paint the map with only two colors, we would get no stable model

  The meaning of not having stable models is that the set of rules of the program is inconsistent, and then any formula is a consequence of it

- What is the meaning of an empty stable model?

  Example: Program `p(X) :- q(X).`

  DLV will return one empty stable model: $\{\}$, which means that no atom is a consequence of the program

Safe Rules in DLV: Each rule in a DLV program has to be safe in order to avoid operations on infinite predicates:

- Variable in the head of a rule has to be in a positive literal in the body of the same rule

  $path(X, X) \leftarrow$ is not safe
  $path(X, X) \leftarrow dom(X)$ is safe

- Variables in a not-negated literal, must also be in a positive literal in the body

  $flies(X) \leftarrow bird(X), not\ ping(X)$ is safe
  $\neg P(X) \leftarrow R(X), not\ T(X, Y)$ is not safe

  However, the second one can be written as:

  $\neg P(X) \leftarrow R(X), not\ S(X).$
  $S(X) \leftarrow T(X, Y).$

- Variables in a built-in comparison predicate, must also appear in a positive literal in the body

$$P(X,Y) \leftarrow R(X), X < Y \qquad \text{is not safe}$$
$$P(Z) \leftarrow R(Z,X), S(Z,Y), X < Y \qquad \text{is safe}$$

Example: Consider a student database with two tables
$Student(ID, Name)$ and $Courses(ID, Course, Status)$

The status can be pass or fail

We want to retrieve the names of the students that haven't failed courses

Consider the following rules? Do they give the expected answers? Are they safe? (test them with DLV)

1. $Ans(N) \leftarrow St(ID, N), not\ C(ID, C, failed).$

   It doesn't give the right semantic nor is safe

2. $Ans(N) \leftarrow St(ID, N), not\ aux(ID, failed).$
   $aux(ID, S) \leftarrow C(ID, C, S).$

   It is safe, but it doesn't give the right semantics

3. $Ans(N) \leftarrow St(ID, N), not\ aux(ID).$
   $aux(ID) \leftarrow C(ID, C, failed).$

   It is safe and gives the right semantics

Queries in DLV: We have seen how to generate the stable models (answer sets)

To determine if an atom is a logical consequence of a program we may consider two different semantics: cautious and brave

You can use them explicitly in DLV by running one of the following commands:

```
> DLV -brave test1
> DLV -cautious test1
```

The query is written at end of input file; it can be a conjunction of ground literals (no variables, but see later)

```
not P(a) ?     Q(a,b), -S(b) ?    not ~a, not b ?
```

The predicates in the query have to be IDB predicates

Example: A non-stratified program:

```
% EDB
   s(a,b).
   s(e,e).
% IDB
   p(X) :- s(X,Y), not t(X).
   t(X) :- s(X,Y), not p(Y).
```

The stable models obtained from DLV are:

```
{s(a,b), s(e,e), t(a), p(e)}
{s(a,b), s(e,e), t(a), t(e)}
```

The following table has the answers for different queries and semantics given by DLV

Query	Semantic	Answer
t(a)?	brave	t(a) is bravely true, evidenced by {s(a,b), s(e,e), t(a), p(e)}
t(a)?	cautious	t(a) is cautiously true
t(e)?	brave	t(e) is bravely true, evidenced by {s(a,b), s(e,e), t(a), t(e)}
t(e)?	cautious	t(e) is cautiously false, evidenced by {s(a,b), s(e,e), t(a), p(e)}
-t(f)?	brave	-t(f) is bravely false
not t(f)?	brave	not t(f) is bravely true, evidenced by {s(a,b), s(e,e), t(a), p(e)}

If we want to pose complex queries? For example, give all the possible values of the first attribute of predicate s

We can use an auxiliary predicate Ans, and add a new rule to the program:

$$Ans(X) \leftarrow s(X, Y).$$

After that, we run DLV to generate all the stable models, that now include extensions for the Ans predicate

If we added that rule to the program in the previous example and ran it in DLV, we would get:

```
{s(a,e), s(e,e), Ans(a), Ans(e), p(e), p(a)}
{s(a,e), s(e,e), Ans(a), Ans(e), t(a), t(e)}
```

The the cautions and brave answers to the query are a and e

Latest versions of DLV accept some forms of SQL3 queries ...