P-log manual

August 26, 2019

Contents

1 System installation

For the latest instructions on system installation, please refer to `https://github.com/iensen/plog2.0/wiki/Installation-Instructions`.

2 System usage

The system requires programs to be stored in ASCII files. The file can be of any extension (we recommend .pl or, not to be confused with Perl, .plog).

A P-log file acceptable by the system should consist of:

1. P-log program, consisting of:
   - sorts definitions,
   - attribute declarations, and
   - program rules.
2. Query.

For example, consider the following program:

```
sorts

#dice={d1,d2}.
#score={1,2,3,4,5,6}.
#person={mike,john}.
#bool = {true,false}.


attributes

roll:#dice->#score.
owns:#dice,#person->#bool.

statements

owns(d1,mike).
owns(d2,john).

random(roll(D)).

%probability information
```

```
pr(roll(D)=6|owns(D,mike))=1/4.

? roll(d1)=1.
```

The program, originally introduced in **??**, describes a scenario with two dice being rolled, belonging to Mike and John respectively. The second dice is more likely to produce '6' as an outcome, as defined by the pr-atom

$$pr(roll(D) = 6|owns(D, mike)) = 1/4.$$

By the so called *principle of indifference* used by P-log, the probabilities of the remaining outcomes equal to (1 - 1/4)/5 = 3/20. Thus, the answer to the query $roll(d1) = 1$ is 1/8. Note that all outcomes of the second die are equally likely, so the answer to the query $roll(d2) = 1$ is equal to 1/6.

To compute the query probability using the system, we need to store it in a file and run the command:

$$plog2\ [path\_to\_file]$$

where $plog2$ is the name of the p-log executable. For example, the file is stored in $plogapp/tests/paper/dice1.plog$ in our system, and we get the following output:

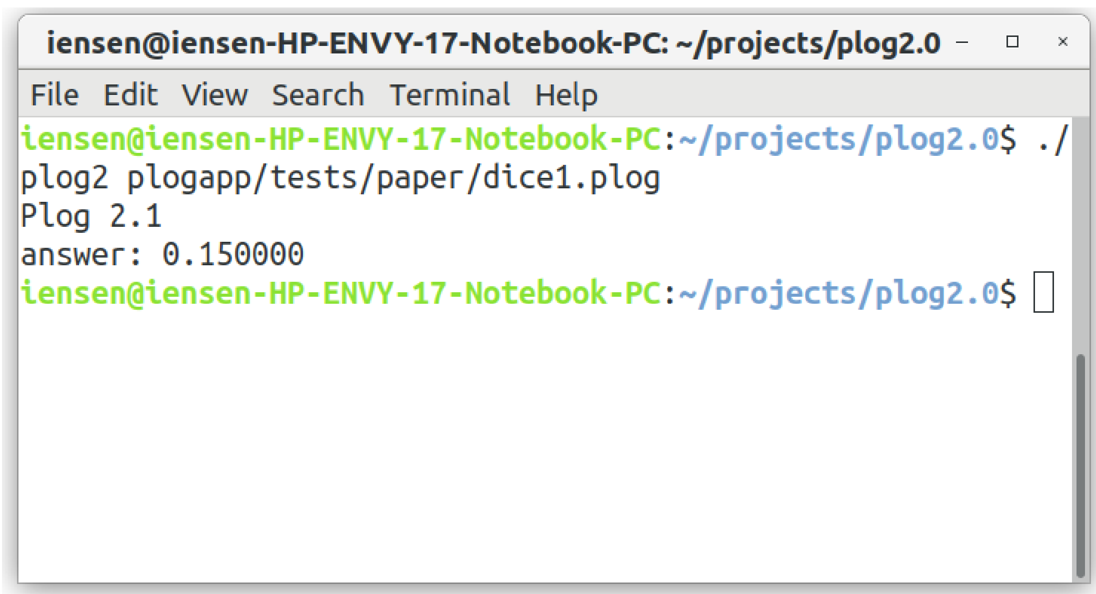

Figure 1: P-log output

The details of the syntax of the language can be found in [] and in the following sections.

3 Command Line Options

In this section we will describe the meanings of command line options supported by P-log. as of right now, the system only

- **-A**

  Compute answer sets of the loaded program.

- **-wcon**

  Show warnings determined by CLP-based algorithm. See section 6.2.2

- **-wasp** [1] Show warnings determined by ASP-based algorithm. See section 6.2.1

- **-solver arg** Specify the solver which will be used for computing answer sets. *arg* can have two possible values: *dlv* and *clingo*.

- **(new!) -n [number]**

  Specify how many answer sets need to be displayed. This option can only be used with option -A.

  Examples:

  - `-n 2` will display exactly one answer sets.
  - `-n 0` will display all the answer sets.

  For the complete list of dlv options, see `http://www.dlvsystem.com/html/DLV_User_Manual.html`

  For the complete list of clingo options, see `http://sourceforge.net/projects/potassco/files/potassco_guide/` Note that the option "0" is passed to clingo solver by default to compute all the answer sets of the program. Also, for programs containing CR-rules, the options `"--opt-mode=optN --quiet=1"` are passed to clingo to ensure correct output.

- **-Help, -H, -help, –Help, –help, -h**

  Show help message.

- **-o arg**

  Specify the output file where the translated ASP program will be written. *arg* is the path to the output file. Note that if the option is not specified, the translated ASP program will not be stored anywhere.

- **input_file**

  Specify the file where the sparc program is located.

[1] This option is temporary not working, use -wcon instead

4 Syntax Description

4.1 Sort definitions

This section starts with a keyword $sorts$ followed by a collection of sort definitions of the form:

$$sort\_name = sort\_expression.$$

sort_name is an identifier preceeded by the pound sign (#). *sort_expression* on the right hand side denotes a collection of strings called As of right now, the system only supports a basic sort definition of the form $sort\_name = \{t_1, \ldots, t_n\}$, where $t_1, \ldots, t_n$ is a collection of ground terms. The remainder of the section is to be implemented in future..

$a\ sort$. We divide all the sorts into *basic sorts* and *non-basic sorts*.

Basic sorts are defined as named collections of numbers and *identifiers*, i.e, strings consisting of

- letters: $\{a, b, c, d, ..., z, A, B, C, D, ..., Z\}$
- digits: $\{0, 1, 2, ..., 9\}$
- underscore: _

and starting with a lowercase letter.

A *non-basic sort* also contains at least one *record* of the form $id(\alpha_1, \ldots, \alpha_n)$ where id is an identifier and

$\alpha_1, \ldots, \alpha_n$ are either identifiers, numbers or records.

We define sorts by means of expressions (in what follows sometimes referred to as statements) of six types:

1. **numeric range** is of the form:

   $$number_1..number_2$$

   where $number_1$ and $number_2$ are non-negative numbers such that $number_1 \leq number_2$. The expression defines the set

   of sequential numbers
   $\{number_1, number_1 + 1, \ldots, number_2\}$.

   Example:

   ```
   #sort1=1..3.
   ```

   `#sort1` consists of numbers $\{1, 2, 3\}$.

2. **identifier range** is of the form:

$$id_1..id_2$$

where id_1 and id_2 are identifiers both starting with a lowercase letter.

id_1 should be lexicographically [2] smaller than or equal to id_2, and the length of id_1 must be less than or equal to the length of id_2. That is, $id_1 \leq id_2$ and $|id_1| \leq |id_2|$.

The expression defines the set of strings $\{s : id_1 \leq s \leq id_2 \wedge |id_1| \leq |s| \leq |id_2|\}$. *Example:*

```
#sort1=a..f.
```

`#sort1` consists of letters $\{a, b, c, d, e, f\}$.

3. **set of ground terms** is of the form:

$$\{t_1, .., , t_n\}$$

The expression denotes a set of *ground terms* $\{t_1, ..., t_n\}$, defined as follows:

- numbers and identifiers are ground terms;
- If f is an identifier and $\alpha_1, \ldots, \alpha_n$ are ground terms, then $f(\alpha_1, \ldots, \alpha_n)$ is a ground term.

Example:

```
#sort1={f(a),a,b,2}.
```

4. **set of records** is of the form:

$$f(sort\_name_1(var_1), ..., sort\_name_n(var_n)) : condition(var_1, ..., var_n)$$

where f is an identifier, for $1 \leq i \leq m$ $sort\_name_i$ occurs in one of the preceeding sort definitions and the condition on variables $var_1, ..., var_n$ (written as $condition(var_1, ..., var_n)$) is defined as follows:

- if var_i and var_j occur in the sequence $var_1, ..., var_n$ and $\odot$ is an element of $\{>, <, \leq, \geq\}$, then $var_i \odot var_j$ is a condition on $var_1, ..., var_n$.
- if $\mathcal{C}_1$ and $\mathcal{C}_2$ are both conditions on $var_1, ..., var_n$, and $\oplus$ is an element of $\{\cup, \cap\}$, then $(\mathcal{C}_1 \oplus \mathcal{C}_2)$ is a condition on $var_1, ..., var_n$.
- if $\mathcal{C}$ is a condition on $var_1, ..., var_n$, then $not(\mathcal{C})$ is also a condition on $var_1, ..., var_n$.

[2] The system default encoding is used for ordering of individual characters

Variables $var_1, ..., var_n$ occurring in parenthesis after sort names are optional as well as the condition :$condition(var_1, ..., var_n)$.

If a condition contains a subcondition $var_i \odot var_j$, then the sorts $sortname_i$ and $sortname_j$

must be defined by basic statements (the definition of a basic statement is given below after the definition of a concatenation statement).

The expression defines a collection of ground terms
$\{f(t_1, \ldots, t_n) : t_1 \in s_i \wedge \cdots \wedge t_n \in s_n \wedge (condition(X_1, \ldots, X_n)|_{X_1=t_1,\ldots,X_n=t_n})\}$

Example

```
#s=1..2.
#sf=f(s(X),s(Y),s(Z)): (X=Y or Y=Z).
```

The sort `#sf` consists of records $\{f(1,1,2), f(1,1,1), f(2,1,1)\}$

5. **set-theoretic expression** can be in one of the following forms:

   - $\#sort\_name$
   - an expression of the form (3), denoting a set of ground terms
   - an expression of the form (4), denoting a set of records
   - $(S_1 \bigtriangledown S_2)$, where $\bigtriangledown \in \{+, -, *\}$ and both S_1 and S_2 are set theoretic expressions

   $\#sort\_name$ must be a name of a sort occurring in one of the preceeding sort definitions. The operations $+$ $*$ and $-$ stand for union, intersection and difference correspondingly.

   Example :

```
#sort1={a,b,2}.
#sort2={1,2,3} + {a,b,f(c)} + f(#sort1).
```

   `#sort2` consists of ground terms $\{1, 2, 3, a, b, f(c), f(a), f(b), f(2)\}$.

6. **concatenation** is of the form

$$[b\_stmt_1]...[b\_stmt_n]$$

   $b\_stmt_1, \ldots, b\_stmt_n$ must be *basic statements*, defined as follows:

   - statements of the forms (1)-(3) are basic
   - statement S of the form (5) is basic if:
     - it does not contain sort expressions of the form (4), denoting sets of records
     - none of curly brackets occurring in S contains a record

– all sorts occurring in S are defined by basic statements

Note that basic statement can only define a basic sort.

Example[3]*.:*

```
#sort1=[b][1..100].
```

`sort1` consists of identifiers $\{b1, b2, \ldots, b100\}$.

4.2 Attribute Declarations

The second part of a P-log program starts with the keyword $attributes$

and is followed by statements of the form

$$attr\_symbol(\#sortName_1, \ldots, \#sortName_n)$$

Where $pred\_symbol$ is an identifier (in what follows referred to as a predicate symbol) and $\#sortName_1$,...,$\#sortName_n$ are sorts defined in sort definitions section of the program.

Multiple declarations containing the same predicate symbol are not allowed. 0-arity predicates must be declared as $pred\_symbol()$. For any sort name $\#s$, the system includes declaration $\#s(\#s)$ automatically.

4.3 Program Rules

The third part of a $\mathcal{SPARC}$ program starts with the keyword *rules* followed by standard ASP rules(supported by the specified ASP solver [4]) , possibly enchanced by arithmetic expressions of arbitrary depth (e.g, p(X*X*X*X+1).) and/or consistency restoring (cr)-rules. CR-rules are of the following form:

$$[label :]l_0 \stackrel{+}{\leftarrow} l_1, \ldots, l_k, not\ l_{k+1} \ldots not\ l_n. \tag{1}$$

where l's are literals. Literals occurring in the heads of the rules must not be formed by predicate symbols occurring as sort names in sort definitions. In addition, rules must not contain *unrestricted variables*.

[3]We allow a shorthand 'b' for singleton set {b}

[4]Currently, only DLV solver is fully supported(excluding #import directives). Clingo's choice rules and minimize statements will be added later

Definition 1 *(Unrestricted Variable) A variable occurrung in a rule of a $\mathcal{SPARC}$ program is called unrestriced if all its occurrences in the rule either belong to some relational atoms of the form $term1$ **rel** $term2$ (where **rel** $\in \{>, >=, <, <=, =, !=\}$) and/or some term appearing in a head of a choice or aggregate element.*

Example 1 Consider the following $\mathcal{SPARC}$ program:

```
sorts
#s={f(a),b}.
predicates
p(#s).
rules
p(f(X)):-Y<2,2=Z,F>3,#count{Q:Q<W,p(W),T<2},p(Y).
```

Variables F,T,Z,Q are unrestricted.

4.4 Display *(New!)*

The last (optional) section of the program starts from the keyword `display` and is followed by a collection of literals of the program. Every literal is followed by a dot symbol ('.').

The section defines which literals are included into the output of answer sets computed in answer set mode (section 2.2). A ground literal is included into the output if and only if it is unifiable with one of the literals from the display section of the program.

If the display section is not present, the output contains all the literals formed by all the predicates of the program.

For example, consider the program:

```
sorts
#s = {a,b,c,f(a),f(b)}.
predicates
p(#s).
q().
s(#s).
rules
s(a):- #s(b).
s(a) :- #s(b).
-q:- #s(a).
p(a) :- -q.
-p(b).
p(f(a)).
-p(f(b)).
display
```

```
-q.
-p(f(X)).
p(X).
#s.
```

The program has one answer set, and the following literals are shown in the output:

```
{-q, -p(f(b)), p(a), p(f(a)), #s(a), #s(b), #s(c),
 #s(f(a)), #s(f(b))}
```

Note that, for example, `p(b)` is not shown because it is not unifiable with any of the literals in the display section.

If the display section is removed from the program, the output is as follows:

```
{s(a), -q, p(a), p(f(a)), -p(b), -p(f(b))}
```

Note that, when compared to the previous scenario, the literals formed by sort names are not included into the output.

5 Answer Sets

A set of ground literals S is an *answer set* of a $\mathcal{SPARC}$ program Π with regular rules only if S is an answer set of an ASP program consisting of the same rules.

To define the semantics of a general $\mathcal{SPARC}$ program, we need notation for abductive support. By $\alpha(r)$ we denote a regular rule obtained from a consistency restoring rule r by replacing $\stackrel{+}{\leftarrow}$ by $\leftarrow$; α is expanded in the standard way to a set X of CR-rules, i.e., $\alpha(A) = \{\alpha(r) \ : \ r \in A\}$. A collection A of CR-rules of Π such that

1. $R \cup \alpha(X)$ is consistent (i.e., has an answer set), and
2. any R_0 satisfying the above condition has cardinality which is greater than or equal to that of R

is called an *abductive support* of Π. A set of ground literals S is an *answer set* of a $\mathcal{SPARC}$

program Π if S is an answer set of $R \cup \alpha(A)$, where R is the set of regular rules of Π, for some abductive support A of Π.

Example

```
sorts
#s1={a}.  % term "a" has sort "s1"

predicates
p(#s1).  %predicate  "p" accepts terms of sort s1
q(#s1).  %predicate  "q" accepts terms of sort s1

rules
p(a) :- not q(a).
-p(a).
q(a):+.  % this is a CR-RULE.
```

Result:

```
username@machine:~$ java -jar sparc.jar program -A
SPARC  V2.25
program translated
DLV [build BEN/Dec 16 2012   gcc 4.6.1]

Best model: {-p(a), appl(r_0), q(a)}
Cost ([Weight:Level]): <[1:1]>
```

Additional literal $appl(r_0)$ was added to the answer set, which means that the first cr-rule from the program was applied.

6 Typechecking

If no syntax errors are found, a static check of the program is performed. Any type-related problems found during this check are classified into type errors and type warnings.

6.1 Type errors

Type errors are considered as serious issues which make it impossible to compile and execute the program. Type errors can occur in all four sections of a $\mathcal{SPARC}$ program.

6.1.1 Sort definition errors

The following are possible causes of a sort definition error that will result in a type error message from $\mathcal{SPARC}$:

1. A set-theoretic expression (statement 5 in section 4.1) containing a sort name that has not been defined.

   Example:

```
sorts
#s={a}.
#s2=#s1-#s.
```

2. Declaring a sort more than once.

   Example:

```
sorts
#s={a}.
#s={b}.
```

3. An identifier range $id_1..id_2$ (statement 2 in section 4.1) where id_1 is greater than id_2.

   Example:

```
sorts
#s=zbc..cbz.
```

4. A numeric range $n_1..n_2$ (statement 1 in section 4.1) where n_1 is greater than n_2.

   Example:

```
sorts
#s=100500..1.
```

5. A numeric range (statement 2 in section 4.1) $n_1..n_2$ that contains an undefined constant.

   Example:

```
#const n1=5.
sorts
#s=n1..n2.
```

6. An identifier range $id_1..id_2$ (statement 3 in section 4.1) where the length of id_1 is greater than the length of id_2.

   Example:

```
sorts
#s=abc..a.
```

7. A concatenation (statement 4 in section 4.1) that contains a non-basic sort.

   Example:

```
sorts
#s={f(a)}.
#sc=[a][#s].
```

8. A record definition (statement 5 in section 4.1) that contains an undefined sort.

   Example:

   ```
   sorts
   #s=1..2.
   #fs=f(s,s2).
   ```

9. A record definition (statement 5 in section 4.1) that contains a condition with relation $>, <, \geq, \leq$ such that the corresponding sorts are not basic.

   Example:

   ```
   #s={a,b}.
   #s1=f(#s).
   #s2=g(s1(X),s2(Y)):X>Y.
   ```

10. A variable that is used more than once in a record definition (statement 5 in section 4.1).

    Example:

    ```
    sorts
    #s1={a}.
    #s=f(#s1(X),#s1(X)):(X!=X).
    ```

11. A sort that contains an empty collection of ground terms.

    Example

    ```
    sorts
    #s1={a,b,c}
    #s=#s1-{a,b,c}.
    ```

6.1.2 Predicate declarations errors

1. A predicate with the same name is defined more than once. *Example:*

   ```
   sorts
   #s={a}.
   predicates
   p(#s).
   p(#s,#s).
   ```

2. A predicate declaration contains an undefined sort. *Example:*

```
sorts
#s={a}.
predicates
p(#ss).
```

6.1.3 Program rules errors

In program rules we first check each atom of the form $p(t_1, \ldots, t_n)$ and each term occurring in the program Π for satisfying the definitions of program atom and program term correspondingly[1]. Moreover, we check that no sort occurs in a head of a rule of Π.

6.2 Type warnings

During this phase each rule in input $\mathcal{SPARC}$ program is checked for having at least one ground instance. Warnings are reported if no ground instance for a $\mathcal{SPARC}$ rule was found. Two options are available:

- `-wcon`: find warnings using constraint solver algorithm described in [1].
- `-wasp`: find warnings using ASP-based algorithm.

While both algorithms are intended to produce same results, their execution time may vary. We recommend using constraint solver based option for programs involving many arithmetic terms and numeric sorts and ASP-based checker for programs with many deeply-nested records and symbolic terms.

6.2.1 ASP based warning checking

The option `-wasp` should be passed to the system to detect and display warnings using a simple ASP based algorithm. For example, consider the $\mathcal{SPARC}$ program below.

```
sorts
#s1={a}.
#s2=f(#s1).
#s3={b}.

predicates
p(#s2).
q(#s3).

rules
p(f(X)):-q(X).
```

The only rule of the program has no ground instances with respect to defined sorts. The execution trace is provided below

```
username@machine:~$ java -jar sparc.jar program.sp -A -wasp
                             -solveropts "-pfilter=warning"
SPARC  V2.29.5
program translated
DLV [build BEN/Dec 16 2012   gcc 4.6.1]
{ warning("p(f(X)):-q(X). ( line: 11, column: 1)")}
```

The atom `warning("p(f(X)):-q(X). ( line:  11, column:  1)")` is included into the answer set as an indicator of potential problem.

In general, when the `-wasp` is passed to $\mathcal{SPARC}$ system, each answer set will contain

```
warning("rule description")
```

for each rule which has no ground instances[5] and

```
has_ground_instance("rule description")
```

for all other rules of the input program.

6.2.2 Constraint solver based warning checking

The option `-wcon` must be passed to the system in order to detect and display warnings using the algorithm described in [1]. Consider the following $\mathcal{SPARC}$ program:

```
#maxint = 1000.
sorts
#s = 1..1000.
predicates
p(#s).
q(#s).
rules
p(X-600):- q(X+600).
```

The only rule of the program has no ground instances with respect to defined sorts. The execution trace is provided below

```
username@machine:~$ java -jar sparc.jar program.sp -A -wcon
                                    -solveropts "-pfilter=p"
%WARNING: Rule p(X-600):-q(X+600). at line 8, column 1
is an empty rule
program translated
DLV [build BEN/Dec 16 2012   gcc 4.6.1]
{}
```

[5] in current version, aggregates are skipped by this algorithm

The message

```
WARNING: Rule p(f(X)):-q(X). at line 8, column 1 is an empty rule
```

is an indicator of a potential problem.

7 $\mathcal{SPARC}$ and ASPIDE

7.1 Installation

For using $\mathcal{SPARC}$ in ASPIDE, you will need to install ASPIDE(version 1.42 or greater). The installer is available from `https://www.mat.unical.it/ricca/aspide/download.html`. See the instructions here: `https://www.mat.unical.it/ricca/aspide/documentation.html`. Once ASPIDE is installed, go to *File ->Plug-ins ->Available plugins* menu, and press install button in the row containing $\mathcal{SPARC}$ plug-in (see Fig.**??**).

7.2 Creating Projects and Adding $\mathcal{SPARC}$ source files

ASPIDE uses *workspaces* to store projects. Workspace is a folder that can contain multiple projects. ASPIDE can have only one workspace opened, that is selected by a user when ASPIDE starts. Source files should belong to a project to be used by ASPIDE query engine and answer set computation tools.

- To create a new project, go to the menu *File ->New* and select *New Project* submenu. Specify the project name in the pop-up window and click on **Finish** button. You should see a new project appeared in **workspace explorer**.
- To add a new SPARC file, right click on the project to display context menu and select *New ->File ->SPARC File* as it is shown on Figure **??** . Choose the file name in the pop-up window. You should see a new file added under the project in workspace explorer and displayed in ASPIDE editor window.

7.3 Executing queries and computing Answer sets

You can execute queries and compute answer sets as for usual ASP file. To execute a query, open a sparc file in the ASPIDE editor and click on the button with a question mark in the toolbar:

A window will appear where you can input and run queries. To run a query,

- mark **Epistemic Mode** checkbox (this is to follow the definition of query given in the class)
- input your query into editbox named **Query** or select one from history

The results will appear in the listview named **Results**. See fig **??** for details.

To compute answer sets of the program, press the button with green arrow marked on figure **??**.

In the appeared **Run Configurations** window:

- make sure a correct path to dlv in selected in **Executable** listbox.
- press **Run** button to see the answer sets

In the displayed window, answer sets are grouped by predicate symbols in their literals. On figure **??**, two answer sets are shown. The first one contains two literals $p(a, b)$ and $p(e, f)$ and some literals with predicate symbol q.

7.4 Warnings Checking

To see allow ASPIDE to show warnings (section 6.2), you need to install swi-prolog on your system. Swi-prolog is available from `http://www.swi-prolog.org/Download.html`

After swi-prolog is installed, go to the ASPIDE menu *File ->Preferences*. In the appeared window select the tab **Executables/Solvers** and add a new *executable* named *swipl* with a path pointing to the swi-prolog executable. Usually, it is named *swipl* in Unix/MacOS operating system and *swipl.exe* in Windows. Click on **Save** button to close the window. See the details on figure **??**. After the executable is added, you need to specify a flag property for the $\mathcal{SPARC}$ plug-in to make it check warnings. Go to ASPIDE menu *File ->Plug-ins ->Manage Plug-ins*. In the appeared window click on the cell Properties in SPARC plug-in line and add a new property `CHECK_WARNINGS=TRUE` as it is shown on figure **??**. Click on **Close** button to save the results. **RESTART ASPIDE FOR THE NEW CHANGES TO TAKE EFFECT**.

After the restart, you should be able to see the warnings in the left lower corner of aspide interface (**Error Console**).

References

[1] Evgenii Balai, Michael Gelfond, and Yuanlin Zhang. Towards answer set programming with sorts. In *Logic Programming and Nonmonotonic Reasoning*, pages 135–147. Springer, 2013.