

SPARC manual

January 29, 2014

Contents

1	System installation	2
2	System usage	2
2.1	Querying mode	4
2.2	Answer Set Mode	5
3	Command Line Options	5
4	Syntax Description	6
4.1	Directives	6
4.2	Sort definitions	7
4.3	Predicate Declarations	9
4.4	Program Rules	10
5	Answer Sets	10
6	Typechecking	11
6.1	Type errors	11
6.1.1	Sort definition errors	12
6.1.2	Predicate declarations errors	13
6.1.3	Program rules errors	14
6.2	Type warnings	14
6.2.1	ASP based warning checking	14
6.2.2	Constraint solver based warning checking	15
7	<i>SPARC</i> and ASPIDE	16

1 System installation

For using the system, you need to have the following installed:

1. Java Runtime Environment (JRE) can be found at <http://www.oracle.com/technetwork/java/javase/downloads/index.html>.
The system was tested on Java versions 1.6.0_37 and 1.7.0_25.
2. The SPARC to ASP translator. It can be downloaded at <https://github.com/iensen/sparc/blob/master/sparc.jar?raw=true>.
3. An ASP solver. It can be one of the following:
 - (a) DLV (recommended). <http://www.dlvsystem.com/dlv/#1>. You need to download the *static* version of the executable file.
 - (b) Clingo <http://sourceforge.net/projects/potassco/files/clingo/4.2.1>.
4. (optional) Swi-Prolog. <http://www.swi-prolog.org/>. This item is only required if option *-wcon* is used for type warning detection. (See sections 3 and 6.2.2).

If you are using the dlv solver, rename the solver executable file to *dlv* (*dlv.exe* for windows).

Be sure the PATH system variable includes the directory where the solver executable is located. For instructions on how to view / modify the PATH system variable, see either of the following links:

<http://www.java.com/en/download/help/path.xml>

<http://www.cyberciti.biz/faq/appleosx-bash-unix-change-set-path-environment-variable/>

To check if the solver is installed correctly, run the command `dlv -v` (for dlv) or `clingo -v` (for clingo). See figures 1 for dlv and 2 for clingo.

2 System usage

To demonstrate the usage of the system we will use the program Π below.

```
sorts
#person={bob,tim,andy}.
predicates
teacher(#person).
rules
teacher(bob).
```

The system can work in one of the two modes: *querying mode* and *answer set mode*.

```
Terminal
username@machine:~$ dlv -v
DLV [build BEN/Dec 16 2012    gcc 4.6.1]

Copyright 1996-2011 Nicola Leone, Gerald Pfeifer, and Wolfgang Faber.
This software is free for academic and non-commercial educational use,
as well as for use by non-profit organisations.
For further information (including commercial use and evaluation licenses)
please contact leone@unical.it, gerald@pfeifer.com, and wf@wfaber.com.

username@machine:~$
```

Figure 1: Checking the version of DLV solver

```
iensen@iensen-OptiPlex-755: ~/sparc_project
username@machine:~$ clingo -v
clingo version 4.2.1
Address model: 32-bit

libgringo version 4.2.1
Copyright (C) Roland Kaminski
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
Gringo is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.

libclasp version 2.2-TP (Rev. 38341)
Configuration: WITH_THREADS=0
Copyright (C) Benjamin Kaufmann
License GPLv2+: GNU GPL version 2 or later <http://gnu.org/licenses/gpl.html>
clasp is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
username@machine:~$
```

Figure 2: Checking the version of Clingo solver

2.1 Querying mode

In this mode we can ask queries about a *SPARC* program loaded into the system. The general command line syntax for this mode is `java -jar spararc.jar program_file`. Queries in *SPARC* are positive or negative literals of the forms $p(t_1, t_2, \dots, t_n)$ or $\neg p(t_1, t_2, \dots, t_n)$ correspondingly, where $p(t_1, t_2, \dots, t_n)$ is an atom of the loaded program Π (note that n can be equal to zero, in this case the query will be of the form p or $\neg p$).

The queries are answered as follows:

- The answer to a query l not containing variables is *yes*, if l (with all arithmetic expressions evaluated) belongs to all answer sets of Π .
- The answer to a query l not containing variables is *no*, if $\neg l$ (with double classical negation removed and all arithmetic expressions evaluated) belongs to all answer sets of Π .
- The answer to a query l not containing variables is *unknown*, if it is not *yes* or *no*.
- The answer to a query of the form l (l is an atom of the form $p(t_1, \dots, t_n)$ possibly preceded by a negation sign) is a collection of assignments $X_1 = t_1, \dots, X_n = t_n$, where X_1, \dots, X_n are all variables in $p(t_1, \dots, t_n)$, t_1, \dots, t_n are ground terms, and the answer to the query $p(t_1', \dots, t_n')$, obtained from $p(t_1, \dots, t_n)$ by replacing each variable X_i by a ground term t_i , is *yes*.

To run *SPARC* on the program above, we change current directory to a directory having the file `program.sp` with the program written in it, and the downloaded file `sparc.jar`. Then, we run the command:

```
username@machine:~$ java -jar spararc.jar program.sp
SPARC V2.25
program translated
?- teacher(bob).
yes
?- teacher(tim).
unknown
?- teacher(X).
X = bob
?- teacher(john).
program.sp: argument number 1 of predicate teacher/1, "john",
violates definition of sort "person"
?-exit.
```

The answer to the first query `?- teacher(bob)` is *yes*, because the atom *teacher(bob)* belongs to the only answer set of Π .

The answer to the second query `?- teacher(tim)` is *unknown*, because neither the atom *teacher(bob)* nor its negation belongs to the answer set of Π .

The answer to the query `?- teacher(X)` is $X = bob$, because there is only one replacement (bob) for X , such that $teacher(X)$ belongs to the answer set of Π .

For the fourth query, we see an error, because $teacher(john)$ is not an atom of Π .

To quit the querying engine, use **exit** command.

2.2 Answer Set Mode

In this mode we can see the computed answer sets of the loaded program. The general command line syntax for this mode is `java -jar sparc.jar program_file -A`.

For the program Π , the answer set may be computed as it is shown below:

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

3 Command Line Options

In this section we describe the meanings of command line options supported by *SPARC*. Some options(flags) do not take an argument and have the form *-option*, while others require arguments and can be written in the form *-option arg*. For each command line option, we indicate whether it requires an argument, and if so, we describe its meaning.

- **-A**
Compute answer sets of the loaded program.
- **-wcon**
Show warnings determined by CLP-based algorithm. See section 6.2.2
- **-wasp**
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*.
- **-solveropts arg**
Pass command line arguments to the ASP solver (DLV or Clingo).
Example: `-solveropts "-pfilter=p"`.
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 options "0" is passed to clingo solver by default.

- **-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 sparac program is located.

4 Syntax Description

4.1 Directives

Directives should be written before sort definitions, at the very beginning of a program. *SPARC* allows two types of directives:

#maxint

Directive **#maxint** specifies maximal nonnegative number which could be used in arithmetic calculations. For example,

```
#maxint=15.
```

limits integers to [0,15].

#const

Directive **#const** allows one to define constant values. The syntax is:

```
#const constantName = constantValue.
```

where *constantName* must begin with a lowercase letter and may be composed of letters, underscores and digits, and *constantValue* is either a nonnegative number or the name of another constant defined above.

4.2 Sort definitions

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

```
sort_name=sort_expression.
```

The sort expression on the right hand side denotes collection of strings called *sorts*. We divide all the sorts into *basic* and *non-basic*.

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

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

and either starting from a letter or containing only digits.

Non-basic sorts also contain *records* of the form $id(\alpha_1, \dots, \alpha_n)$, where *id* is an identifier and $\alpha_1, \dots, \alpha_n$ are either identifiers or records.

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

1. set-theoretic expression.

```
set_expression := #sort_name | {ground_term_list}
set_expression := (set_expression)
                  | (set_expression + set_expression )
                  | (set_expression * set_expression )
                  | (set_expression - set_expression )
```

The operations $+$, $*$ and $-$ stand for union, intersection and difference correspondingly. *ground_term_list* is set of *ground terms*, defined as follows:

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

Example :

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

According to the definition, *sort1* consists of ground terms $\{f(a), a, b, 2\}$, and *sort2* is $\{1, 2, 3, f(c)\}$

2. numeric range.

`numeric_range := number1 .. number2`

number1 should be smaller or equal than *number2*. The expression defines the set of subsequent numbers $\{number1, number1 + 1, \dots, number2\}$

Example:

`sort1=1..3`

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

3. identifier range

`id_range := id1 .. id2`

id1 should be lexicographically smaller or equal than *id2*. *id1* and *id2* should both consist of digits and letters and start from a lowercase letter. The expression defines the set of all strings

$S = \{s : id1 \leq s \leq id2 \wedge |id1| \leq |s| \leq |id2|\}$

Example:

`sort1=a..f.`

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

4. concatenation

`concatenation := [b_stmt_1] ... [b_stmt_n]`

`b_stmt_1, ..., b_stmt_n` must be *basic statements*, defined as follows:

- statements of the forms (2)-(4) are basic
- statement *S* of the form (1) is basic if:
 - all curly brackets occurring in *S* contain only constants consisting of latin letters and digits
 - all sorts occurring in *S* are defined by basic statements

Note that basic statement can only define a basic sort not containing records.

*Example*¹..:

¹We allow a shorthand 'b' for singleton set {b}

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

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

5. record

```
functional_term := f(sort_name1(var_1), ..., sort_namen(var_n)) :
                                     condition(var_1, ..., var_n)
condition(var_1, ..., var_n) := var_i REL var_j
condition(var_1, ..., var_n) := condition and condition
                               | condition or condition
                               | not(condition)
                               | (condition)
```

Variables var_1, \dots, var_n are optional as well as the condition. Condition can only contain variables from the list var_1, \dots, var_n . If there is a subcondition $var_i \text{ REL } var_j$, where REL is either $\{>, \geq, <, \leq\}$ then `sortname_i` and then `sortname_j` must be defined by basic statements.

The expression defines a collection of ground terms

$\{f(t_1, \dots, t_n) : \text{condition}(t_1, \dots, t_n) \text{ is true} \wedge t_1 \in s_i \wedge \dots \wedge t_n \in s_n\}$

Example

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

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

4.3 Predicate Declarations

The second part of a *SPARC* program starts with the keyword *predicates*

and is followed by statements of the form

pred_symbol(sortName, ..., sortName)

Multiple declarations containing the same predicate symbol are not allowed. 0-arity predicates must be declared as *pred_symbol()*. For any sort name *SN*, the system includes declaration *SN(SN)* automatically.

4.4 Program Rules

The third part of a *SPARC* program starts with the keyword *rules* followed by standard ASP rules(supported by the specified ASP solver ²), possibly enhanced 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:

$$[label:]l_0 \stackrel{+}{\leftarrow} l_1, \dots, l_k, not\ l_{k+1} \dots not\ l_n \quad (1)$$

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*.

Definition 1 (*Unrestricted Variable*) A variable occurring in a rule of a *SPARC* program is called *unrestricted* 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 terms appearing in a head of a choice or aggregate element.

Example 1 Consider the following *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.

5 Answer Sets

A set of ground literals S is an *answer set* of a *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 *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

²Currently, only DLV solver is fully supported(excluding #import directives). Clingo's choice rules and minimize statements will be added later

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 *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 spararc.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 program is performed all found type-related problems, classified into type errors and type errors.

6.1 Type errors

Type errors are considered as serious issues which make it impossible to compiled and execute the program. Type errors can occur in all four section of a *SPARC* program.

6.1.1 Sort definition errors

1. Set-theoretic expression (statement (2) in section 4.2) contains a name of undefined sort.

Example:

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

2. Sort with the same name is defined more than once. *Example:*

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

3. In an identifier range $id1..id2$ (statement (2) in section 4.2) the first identifier($id1$) is lexicographically greater than $id2$. *Example*

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

4. In a numeric range $n1..n2$ (statement (2) in section 4.2) $n1$ is greater than $n2$. *Example:*

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

5. Numeric range (statement (2) in section 4.2) $n1..n2$ contains an undefined constant.

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

6. In an identifier range $id1..id2$ (statement (3) in section 4.2) the length of the first identifier($id1$) is greater than length of the second.

Example:

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

7. Concatenation (statement (4) in section 4.2) contains a non-basic sort.

Example:

```

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

```

8. Record definition (statement (5) in section 4.2) contains an undefined sort.

Example:

```

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

```

9. Definition of record (statement (5) in section 4.2) 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. Variable is used more than once in record definition(statement (5) in section 4.2).

Example:

```

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

```

11. Sort 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 $p(t_1, \dots, t_n)$ 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 *SPARC* program is checked for having at least one ground instance. Warnings are reported if no ground instance for a *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 *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 *SPARC* system, each answer set will contain

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

for each rule which has no ground instances³ 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 *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]
{}
```

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.

³in current version, aggregates are skipped by this algorithm

7 *SPARC* and ASPIDE

For using *SPARC* in ASPIDE, you will need to install ASPIDE version 1.37.1 or greater. Once it is installed, go to *File -> Plug-ins -> Available plugins* menu, and press install button in the row containing *SPARC* plug-in (see Fig.3).

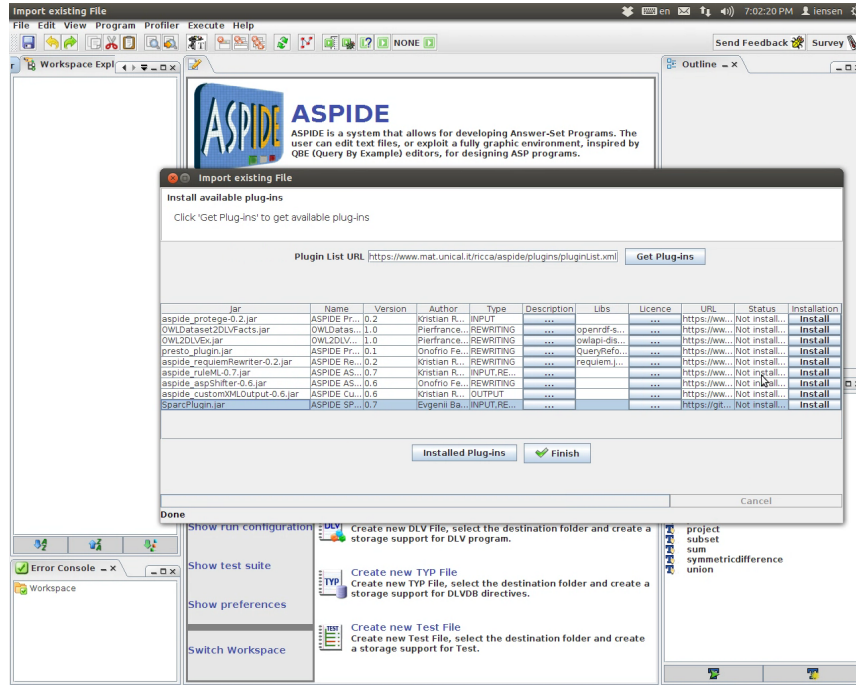


Figure 3: Installing *SPARC* plugin

Once the plugin is installed, you can create a source file and start coding (see Fig.4). You can execute queries and compute answer sets as for usual ASP file.

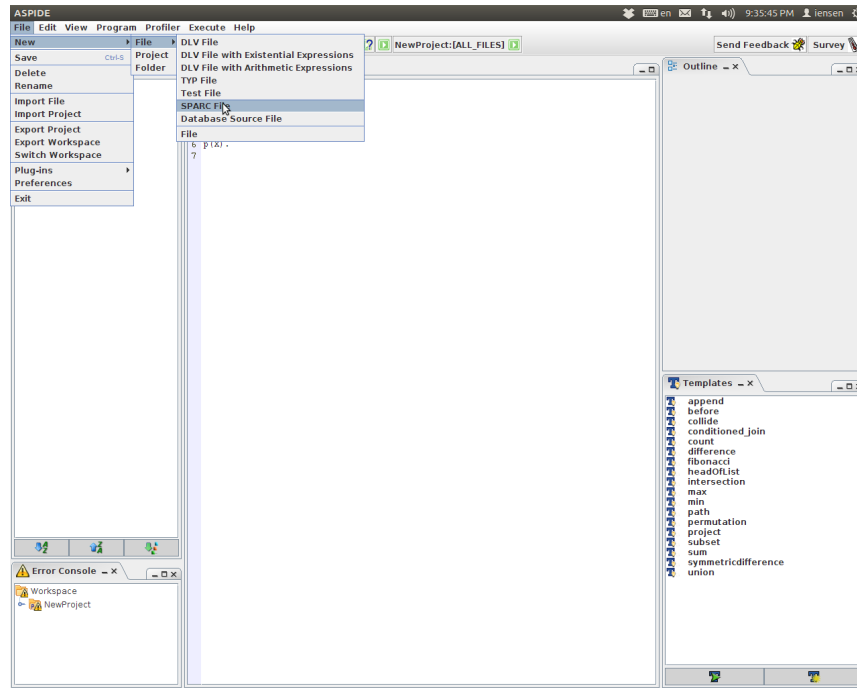


Figure 4: Editing *SPARC* source file

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.