SPARC manual

June 21, 2013

Contents

1 Introduction

A good knowledge representation methodology should allow one to:

- Identify and describe *sorts* (types, kinds, categories) of objects populating a given domain.
- Identify and precisely define important *properties* of these objects and *relationships* between them.

An *abstract model* of the domain will consist of sorts and definitions produced by this process. To deal with a particular problem one needs to describe relevant objects and their sorts and properties and use the corresponding inference engine to solve the problem. ASP based knowledge representation languages have powerful means for describing non-trivial properties of objects but lack the means of conveniently specifying objects and their sorts as well as sorts of parameters of the domain relations and functions. There were some recent attempts to remedy the problem but we do not believe that a sufficiently simple and powerful solution has been found so far. In this manual we describe a knowledge representation language $\mathcal{SPARC}$ which attempts to solve this problem and provide some examples of using the system we implemented.

2 System installation

$\mathcal{SPARC}$ system works in two phases:

1. Translation of $\mathcal{SPARC}$ code to Answer Set Prolog (ASP) code.
2. Running ASP code on existing ASP solvers.

So, to use the system you need two things:

1. The SPARC to ASP translator. It can be downloaded here: `https://github.com/iensen/sparc_paper/blob/master/sparc.jar?raw=true`.
2. An ASP solver. We recommend using DLV for best compatibility with out translator. DLV can be downloaded here: `http://www.dlvsystem.com/dlv/#1` You need to download *static* version of the executable file.

To demonstrate usage of the system, let's start with simple example.

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

This is a SPARC program. It consists of three sections:

- Sort definitions. The section starts with the keyword *sorts* followed by definitions of the sorts in the program.
- Predicate declarations. The section starts with the keyword *predicates* followed by declarations of predicates of the program.
- Program rules. The sections starts with the keyword *rules* and consists of a collection of rules satisfying ASP syntax.

To translate the program, run sparc.jar with arguments specifying the SPARC program and the file where the translation will put its result:

```
$ java -jar sparc.jar example.sp -o example.asp
SPARC to DLV translator V2.21
program translated
```

To get the answer sets of the translated program, run DLV:

```
$ dlv example.asp
DLV [build BEN/Dec 21 2011   gcc 4.6.1]

{teacher(bob)}
```

You should see that the answer set appeared. For a detailed description of $\mathcal{SPARC}$ system options, see `https://github.com/iensen/sparc_paper/wiki/System-usage`.

For Linux, MacOS or Cygwin (Windows) users we prepared a bash script which combines translation and solver execution into one step. It is available here: `https://github.com/iensen/sparc/raw/master/sparc`. Before using the script, make it executable by running command `chmod +x sparc` from the command line.

The general syntax is :

```
sparc input_file [dlv_options]
```

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

Here is an example of using the script:

```
$ sparc example.sp -filter=teacher
...
{teacher(bob)}
```

To see more examples of $\mathcal{SPARC}$ programs, visit `https://github.com/iensen/sparc_paper/wiki/Program-Examples`.

3 Detailed System Description

3.1 Sort definitions

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

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

and either starting from a letter or containing only digits.

Non-basic sorts also contain *records* of the form $id(\alpha_1, \ldots \alpha_2)$, where id is an identifier and $\alpha_1, \ldots \alpha_n$ 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 $\alpha_1, \ldots \alpha_n$ are ground terms, then $f(\alpha_1, \ldots \alpha_n)$ 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 $\{number1, number1 + 1, \ldots number2\}$

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. 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:

```
sort1=[b][1..100].\footnote{We allow a shorthand `b` for singleton  s
```

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

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,..var_n` are optional. Condition can only contain variables from the list `var_1,..var_n`. If there is a subcondion `var_i 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
$\{f(t_1, \ldots, t_n) : condition(t_1, \ldots t_n)\ is\ true\}$

Example

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

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

3.2 Predicate Declarations

The second part of a $\mathcal{SPARC}$ program starts with the keyword $predicate\ declarations$

and is followed by statements of the form

$pred\_symbol(sortName, \ldots, sortName)$

Multiple declarations for one predicate symbol are not allowed. For any sort name SN, the system includes declaration $SN(SN)$ automatically.

3.3 Program Rules

The third part of a $\mathcal{SPARC}$ program consists of standard ASP rules and/or consistency restoring (cr)-rules. CR-rules are of the following form:

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

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.

4 Directives

Directives should be written before sort definitions, at the very beginning of a program. $\mathcal{SPARC}$ allows two types of directives:

4.1 #maxint

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

```
#maxint=15.
```

limits integers to [0,15].

4.2 #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.

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:

```
  ./sparc example2.sp
DLV [build BEN/Dec 21 2011   gcc 4.6.1]

SPARC to DLV translator V2.02
program translated
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 Syntax checking

During this phase the program is checking for satisfying $\mathcal{SPARC}$ lexical and syntactical specifications. All found syntax errors are reported.

7 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.

7.1 Type errors

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

7.1.1 Sort definition errors

1. Set-theoretic expression (statement (2) in section 3.1) 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 3.1) 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 3.1) n1 is greater than n2. *Example:*

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

5. Numeric range (statement (2) in section 3.1) $n1..n2$ contains an undefined constant.

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

6. In an identifier range $id1..id2$ (statement (3) in section 3.1) 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 3.1) contains a non-basic sort.

   Example:

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

8. Record definition (statement (5) in section 3.1) contains an undefined sort.
   Example:

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

9. Definition of record (statement (5) in section 3.1) 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 3.1).
    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}.
```

7.1.2 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).
```

7.1.3 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(!!add reference as soon as it is available). Moreover, we check that no sort occurs in a head of a rule of Π.

7.2 Type warnings

During this phase each rule in input $\mathcal{SPARC}$ program is checked for having at least one ground instance. This is done by applying a standard constraint satisfaction algorithm to a constraint formula over finite domains[9] produced by algorithms from (!!! add link as soon as it is available). Warnings are reported for the rules which have no ground instances.

8 $\mathcal{SPARC}$ and Unsafe Rules

$\mathcal{SPARC}$ helps to avoid *unsafe rule* errors in most cases. **Example.**

```
sort definitions
s1=1..5.
predicate declarations
p(s1).
program rules
p(X).
```

The only rule in the program is unsafe (variable X does not occur in the body). However, by translating the program and running ASP solver, we will be able to avoid this unsafety (with addition of auxiliary predicates). Here is the execution trace:

```
$ ./sparc example3.sp -pfilter=p
DLV [build BEN/Dec 21 2011   gcc 4.6.1]

SPARC to DLV translator V2.02
program translated
{p(1), p(2), p(3), p(4), p(5)}
```

9 $\mathcal{SPARC}$ and ASPIDE

TODO.