

SPARC manual

February 15, 2013

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 *SPARC* which attempts to solve this problem and provide some examples of using the system we implemented.

2 First Steps

SPARC system works in two phases:

1. Translation of *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/blob/master/sparc.jar?raw=true>.

2. An ASP solver. We recommend using DLV for best compatibility with our 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.

```
sort definitions
person=$bob|tim|andy.
predicate declarations
teacher(person) .
program rules
teacher(bob) .
```

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

- Sort definitions. The section starts with the keywords *sort definition* followed by definitions of the sorts in the program.
- Predicate declarations. The section starts with the keywords *predicate declarations* followed by declarations of predicates of the program.
- Program rules. The section starts with the keywords *program 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 V2.02
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 *SPARC* system options, see <https://github.com/iensen/sparc/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 *SPARC* programs, visit <https://github.com/iensen/sparc/wiki/Examples>.

3 Detailed System Description

3.1 Sort definitions

We define sorts as named collections of strings over a fixed alphabet Γ consisting of:

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

We define string collections (languages) by means of expressions: regular expressions and set-theoretic operations, like *union*, *intersection* and *difference*. We denote the set of strings of language L by $\mathcal{D}(L)$. In the following description, α, β and γ denote characters from Γ , s denotes a string, n, n_1 and n_2 denote non-negative numbers, *regexp*, *regexp*₁ and *regexp*₂ denote regular expressions.

In the table 1 we describe the syntax of *regular expressions* used in the system.

Next we define expressions. In table 2, *sort names* (denoted by s, s_1, s_2, \dots) and *functional symbols* (f, f_1, f_2, \dots) are all identifiers starting with a lowercase letter. *Variable names* (denoted by V_1, V_2, \dots) are identifiers starting with a capital letter. $\$regexp$ denotes a language of all strings satisfying regular expression *regexp*. *rel* is a binary relation, one of $\geq, <, >, \leq, =$. u, u_1, u_2, \dots, u_n are strings over alphabet Γ . e, e_1, \dots, e_n are expressions.

The \$ sign before a regular expression is used to avoid grammar ambiguity, i.e., to make regular expression which are expressed by identifiers distinguishable from sort names.

Table 1: Regular Expressions

Syntax	Informal Description	Described language	Example
α ($\alpha \in \Gamma$)	character	$\{a\}$	a
$regex_1 regex_2$	concatenation	$\{s_1 s_2 : s_1 \in \mathcal{D}(regex_1), s_2 \in \mathcal{D}(regex_2)\}$	ab
$regex_1 regex_2$	disjunction	$\mathcal{D}(regex_1) \cup \mathcal{D}(regex_2)$	$a b$
$\sim regex$	complement	$\{s : s \notin \mathcal{D}(regex)\}$	$\sim a$
$regex\{n\}$	n-repetition	$\{\underbrace{ss \dots s}_{n \text{ times}} : s \in \mathcal{D}(regex)\}$	$a\{10\}$
$regex\{n, m\}$	counting	$\mathcal{D}(regex\{n\}) \cup \dots \cup \mathcal{D}(regex\{m\})$	$a\{1, 10\}$
$[\alpha - \beta]$	range	$\{\gamma : \alpha \leq \gamma \leq \beta\}$	$[a - z]$
$(regex)$	parenthesis	$\mathcal{D}(regex)$	(a)

Table 2: Expressions.

Syntax	Informal Description	Described language	Example
$\$regex$	regular expression	$\mathcal{D}(regex)$	$\$[a - z]$
$n_1..n_2$	integer range	$\{n : n_1 \leq n \leq n_2\}$	$1..4$
s	sort name	$\{\mathcal{D}(e) : s \text{ was defined as } s = e\}$	$s1$
$f(e_1[V_1]^{a\ b}, \dots, e_n[V_n])$ [: $\{V_i \text{ op } V_j\}$]	functional term	$\{f(u_1, \dots, u_n) : u_1 \in \mathcal{D}(e_1), \dots, u_n \in \mathcal{D}(e_n) [: u_i \text{ op } u_j]\}^c$	$put(block(X), block(Y)) : \{X! = Y\}$
$e_1 + e_2$	union	$\mathcal{D}(e_1) + \mathcal{D}(e_2)$	$\$a + f(s)$
$e_1 * e_2$	intersection	$\mathcal{D}(e_1) * \mathcal{D}(e_2)$	$\$a * f(s)$
$e_1 - e_2$	difference	$\mathcal{D}(e_1) \setminus \mathcal{D}(e_2)$	$\$a - f(s)$
(e)	parenthesis	$\mathcal{D}(e)$	$(\$a + \$b) * s$

^aSquare brackets denote the optional part.

^bThe Current implementation allows variables only after expressions consisting of single sort name.

^cThe last condition $u_i \text{ op } u_j$ is present if and only if there is a condition of the form $V_i \text{ op } V_j$ in expression.

Example Consider the following sort definitions:

$s = \$a | b.$
 $s1 = s.$
 $s2 = \$s.$

Here sort $s1$ consists of string 'a' and 'b', but $s2$ consist of only one string 's'.

The definition of sort named s is an assignment of the form $s = e$, where e is an expression not containing sort names which were not defined in former definitions.

4 Predicate Declarations

The second part of a *SPARC* program starts with the keyword *predicate declarations* and is followed by statements of the form

pred_symbol(sortName, ..., sortName)

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

5 Program Rules

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

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

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.

6 Directives

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

6.1 #maxint

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

```
#maxint=15.
```

limits integers to [0,15].

6.2 #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.

7 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 $\overset{+}{\leftarrow}$ 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 *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

```
sort definitions
s1=$a. % term "a" has sort "s1"
predicate declarations
p(s1). %predicate "p" accepts terms of sort s1
q(s1). %predicate "q" accepts terms of sort s1
program 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.

8 Typechecking

During typechecking we perform a static check of each predicate from the program rules for satisfying type definitions provided in the first section.

8.1 Type errors

Definition 1 (*Matching*) A ground term t matches a set of ground literals S if $t \in S$. A term with variables tv matches a ground literal l if one of the following conditions holds:

1. tv is a variable.
2. tv is an arithmetic term (i.e, it consists of numbers, variables and arithmetic operations), and l is a number.
3. tv is of the form $f_1(t_1, t_2, \dots, t_n)$, l is of the form $f_2(u_1, u_2, \dots, u_n)$, $f_1 = f_2$, and for all $1 \leq i \leq n$ t_i matches u_i .

A term with variables tv matches a set of ground literals S if it matches at least one of elements of S .

We say that atom $p(t_1, t_2, \dots, t_n)$ follows sort definitions if the following conditions hold:

1. predicate p was declared with exactly n arguments, i.e., in predicate declarations' section of the program there is a declaration $p(s_1, \dots, s_n)$, where s_1, \dots, s_n are sort names.
2. for each term $1 \leq i \leq n$ t_i matches $\mathcal{D}(s_i)$.

For each atom $p(s_1, \dots, s_n)$ type error is produced if it does not follow sort definitions.

8.2 Type warnings

We say that sort s is assigned to term t , if there is an atom $p(t_1, \dots, t_n)$, which was declared as $p(s_1, \dots, s_n)$, and for some i $t = t_i$, $s = s_i$. Type warning is produced for term t , if $\{s_1, \dots, s_n\}$ is a set of sorts assigned to t , and $\mathcal{D}(s_1) \cap \dots \mathcal{D}(s_n)$ is empty.

Example

Consider the following SPARC program:

```
sort definitions
s1=$1..4$ and s2=$5..15$
predicate declarations
p(s1).
q(s2).
program rules
p(X) :- q(X)
```

For term X (which is a variable in this case), sorts s_1 and s_2 are derived. A warning is produced, because $\mathcal{D}(s_1) \cap \dots \mathcal{D}(s_n) = \emptyset$.

%WARNING: Term X occurring in the rule p(X):-q(X).(line 8, column 2) has a

9 *SPARC* and Unsafe Rules

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)}
```

10 *SPARC* and ASPIDE

TODO.