$\mathcal{SPARC}$ manual

June 16, 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 $\mathcal{SPARC}$ which attempts to solve this problem and provide some examples of using the system we implemented.

2 First Steps

$\mathcal{SPARC}$ 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

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

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

We define string collections (sorts) by means of expressions of five types:

1. **set-theoretic expressions**.

```
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 $+ * -$ 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 $\{a, b, c, d, e, f\}$.

Next we define expressions. In table 2, *sort names* (denoted by $s, s_1, s_2, \ldots$) and *functional symbols*($f, f_1, f_2, \ldots$) are all identifiers starting with a lowercase letter. *Variable names*(denoted by $V_1, V_2, \ldots$) 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 $>=, <, >, <=, =$. $u, u_1, u_2, \ldots, u_n$ are strings over alphabet Γ. $e, e_1, \ldots, e_n$ 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: Expressions.

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

[a] Square brackets denote the optional part.
[b] The Current implementation allows variables only after expressions consisting of single sort name.
[c] The last condition u_i op u_j is present if and only if there is a condition of the form V_i 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 $\mathcal{SPARC}$ 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.

5 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.

6 Directives

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

```
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 $f_1(t_1, t_2, \ldots t_n)$, l is of the form $f_2(u_1, u_2, \ldots u_n)$, $f_1 = f_2$, and for all $1 <= i <= n$ 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 $p(t_1, t_2, \ldots, t_n)$ *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 $p(s_1, \ldots, s_n)$, where $s_1, \ldots s_n$ are sort names.
2. for each term $1 <= i <= n$ t_i matches $\mathcal{D}(s_i)$.

For each atom $p(s_1, \ldots, s_n)$ 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 $p(t_1, \ldots t_n)$, which was declared as $p(s_1, \ldots s_n)$,and for some i $t = t_i$, $s = s_i$. Type warning is produced for term t, if $\{s_1, \ldots s_n\}$ is a set of sorts assigned to t, and $\mathcal{D}(s_1) \cap \ldots \mathcal{D}(s_n)$ 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 $s1$ and $s2$ are derived. A warning is produced, because $\mathcal{D}(s_1) \cap \ldots \mathcal{D}(s_n) = \emptyset$.

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

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.