SPASS Input Syntax
Version 1.5

Christoph Weidenbach
Max-Planck-Institut für Informatik
Stuhlsatzenhausweg 85
66123 Saarbrücken
weidenb@mpi-sb.mpg.de

Abstract

This document introduces the SPASS input syntax. It came out of the DFG syntax format that was thought to be a format that can easily be parsed such that it forms a compromise between the needs of the different groups.

1 Introduction

The language proposed in the following is intended to be a common exchange format for logic problem settings. It is thought to be a format that can easily be parsed such that it forms a compromise between the needs of the different groups. Therefore, it is kept *as simple as possible*, in particular, the grammar of the language can be easily processed by some automatic parser-generator.

In any case it will be necessary to provide tools that transform files from the present syntax into other standard formats (e.g., Otter [6] or TPTP [9]) and vice versa. Currently we can (partly) transform Otter input files to DFG-Syntax files and vice versa.

2 Notation

For the grammar defining the syntax, terminals are always underlined while non-terminals and meta-symbols are not. Braces come in different variants and have the following meaning:

$\{\ \}$	optional
$\{\ \}^*$	arbitrarily often
$\{\ \}^+$	at least once

3 Problems

The unit of information we can describe are problems. A problem may not only contain formulae or clauses but also information on parameter settings.

```
problem  ::=  begin_problem(identifier).
              description
              logical_part
              {settings}*
              end_problem.
```

Note that the description part as well as the logical part are mandatory.

4 Descriptions

The description part should help to understand what the problem is about. In particular, the logic part is mandatory, if non-standard quantifiers or operators are used.

```
description  ::=  list_of_descriptions.
                  name( {* text *} ).
                  author( {* text *} ).
                  {version( {* text *} ).}
                  {logic( {* text *} ).}
                  status(log_state).
                  description( {* text *} ).
                  {date( {* text *} ).}
                  end_of_list.
  log_state  ::=  satisfiable | unsatisfiable | unknown
```

5 The Logical Parts

Any non-predefined signature symbol used in a problem has to be defined in the declaration part. Then the logical part may provide a formulation of the problem by formulae as well as by some clause normal forms. In addition, proofs for the conjecture stated by the formulae (clauses) may be contained.

```
logical_part  ::=  {symbol_list}
                   {declaration_list}
                   {formula_list}*
                   {clause_list}*
                   {proof_list}*
```

As mentioned before, non-predefined signature symbols have to be declared in advance. Since the current scope of the syntax only covers first-order logic, we are concerned with function and predicate symbols. The usual first-order operators and quantifiers are predefined. In addition, there is a unique symbol for equality, see below.

```
symbol_list  ::=  list_of_symbols.
                  {functions[fun_sym | (fun_sym,arity)
                      {, fun_sym | (fun_sym,arity)}*].}
                  {predicates[pred_sym | (pred_sym,arity)
                      {, pred_sym | (pred_sym,arity)}*].}
                  {sorts[sort_sym {,sort_sym}*].}
                  end_of_list.
```

All declared symbols have to be different from each other and from all terminal and predefined symbols.

We support a rich sort language that may be introduced by a declaration part. We do not allow free variables in term declarations, but polymorphic sorts.

```
declaration_list ::= list_of_declarations.
                     {declaration}*
                     end_of_list.
     declaration ::= subsort_decl | term_decl | pred_decl | gen_decl
        gen_decl ::= sort sort_sym {freely} generated by func_list.
       func_list ::= [ fun_sym {,fun_sym}*]
    subsort_decl ::= subsort(sort_sym,sort_sym).
       term_decl ::= forall(term_list,term). | term.
       pred_decl ::= predicate(pred_sym{,sort_sym}+).
        sort_sym ::= identifier
        pred_sym ::= identifier
         fun_sym ::= identifier
```

Concerning the term declarations, we assume that all terms in `term_list` are variables or expressions of the form `sort_sym(variable)`.

Now there are two types of formulae: Axiom formulae and conjecture formulae. If the status of the problem (see below) states "unsatisfiable" it refers to the clause normal form resulting from the conjunction of all axiom formulae and the negation of the disjunction of all conjecture formulae. Of course, "satisfiable" means that the overall formula has a model.

```
formula_list ::= list_of_formulae(origin_type).
                 {formula({term}{,label}).}*
                 end_of_list.
 origin_type ::= axioms | conjectures
       label ::= identifier
```

We assume that all formulae are closed, so we do not allow free variables inside a formula expression.

Quantifiers always have two arguments: A term list and the subformulae. The term list is assumed to be a variable list (or a list of variables annotated with a sort) for the usual first-order quantifiers, however, one could easily imagine non-classical quantifiers, where "quantification" over real terms makes sense.

```
     term ::= quant_sym(term_list,term) | symbol |
              symbol(term{,term}*)
term_list ::= [term{,term}*]
quant_sym ::= forall | exists | identifier
   symbol ::= equal | true | false | or | and | not | implies |
              implied | equiv | identifier
```

We support disjunctive normal form as well as clause normal form. Even clauses have to be written as their corresponding formulae, in particular all variables have to be bound by the leading quantifier. Our experience with problems stated by a set of clauses shows that this helps to detect flaws, e.g., if accidentally it was forgotten to declare some constant that would then be considered as a variable. Since free variables are not allowed, this case is detected in our syntax.

```
    clause_list ::= list_of_clauses(origin_type,clause_type).
                    {clause({cnf_clause | dnf_clause}{,label}).}*
                    end_of_list.
    clause_type ::= cnf | dnf
     cnf_clause ::= forall(term_list,cnf_clause_body) | cnf_clause_body
     dnf_clause ::= exists(term_list,dnf_clause_body) | dnf_clause_body
cnf_clause_body ::= or(term{,term}*)
dnf_clause_body ::= and(term{,term}*)
```

In case of `cnf_clause_body` and `dnf_clause_body` we assume all subterms generated for `term` to be literals.

The alphabet allowed to compose identifiers is restricted to letters, digits and the underscore symbol.

```
begin_problem(Pelletier57).

list_of_descriptions.
name({* Pelletier's Problem No. 57 *}).
author({* Christoph Weidenbach *}).
status(unsatisfiable).
description({* Problem taken in revised form from the "Pelletier Collection",
      Journal of Automated Reasoning, Vol. 2, No. 2, pages 191-216 *}).
end_of_list.

list_of_symbols.
functions[(f,2), (a,0), (b,0), (c,0)].
predicates[(F,2)].
end_of_list.

list_of_formulae(axioms).
formula(F(f(a,b),f(b,c))).
formula(F(f(b,c),f(a,c))).
formula(forall([U,V,W],implies(and(F(U,V),F(V,W)),F(U,W)))).
end_of_list.

list_of_formulae(conjectures).
formula(F(f(a,c),f(a,c))).
end_of_list.

end_problem.
```

Figure 1: Pelletier's Problem No. 57

$$
\begin{array}{rcl}
\texttt{identifier} & ::= & \{\texttt{letter} \mid \texttt{digit} \mid \texttt{special\_symbol}\}^+ \\
\texttt{letter} & ::= & \underline{\texttt{a}}\text{–}\underline{\texttt{z}} \mid \underline{\texttt{A}}\text{–}\underline{\texttt{Z}} \\
\texttt{arity} & ::= & \underline{\texttt{-1}} \mid \texttt{number} \\
\texttt{number} & ::= & \{\texttt{digit}\}^+ \\
\texttt{digit} & ::= & \underline{\texttt{0}} \mid \underline{\texttt{1}} \mid \underline{\texttt{2}} \mid \underline{\texttt{3}} \mid \underline{\texttt{4}} \mid \underline{\texttt{5}} \mid \underline{\texttt{6}} \mid \underline{\texttt{7}} \mid \underline{\texttt{8}} \mid \underline{\texttt{9}} \\
\texttt{special\_symbol} & ::= & \underline{\texttt{\_}}
\end{array}
$$

5.1 Examples

We start with a complete description of Pelletier's [7] problem No. 57 that can be found in Figure 1. The syntax for the description part is explained in Section 4.

Our second example, Figure 2, uses the language features provided for the declaration of sorts.

6 Proofs

We also define a first, simple proof format. Basically a proof consists of a sequence of "simple" steps. The semantics of step is that the introduced formula is a logical consequence of the formulae pointed to by the list of parents.

We already have implemented some scripts that can be used to automatically check resolution proofs. Here, the idea is to be able to check complicated, tedious, long proofs found by some prover automatically by using a different prover.

```
begin_problem(Sorts).

list_of_descriptions.
name({* Sorts and Plus *}).
author({* Christoph Weidenbach *}).
status(satisfiable).
description({* Defines plus over successor and zero.  *}).
end_of_list.

list_of_symbols.
functions[plus,s,zero].
sorts[even,nat].
end_of_list.

list_of_declarations.
subsort(even,nat).
even(zero).
forall([nat(x)],nat(s(x))).
forall([nat(x),nat(y)],nat(plus(x,y))).
forall([even(x),even(y)],even(plus(x,y))).
forall([even(x)],even(s(s(x)))).
forall([nat(y)],even(plus(y,y))).
end_of_list.

list_of_formulae(axioms).
formula(forall([nat(y)],equal(plus(y,zero),y))).
formula(forall([nat(y),nat(z)],equal(plus(y,s(z)),s(plus(y,z))))).
end_of_list.

end_problem.
```

Figure 2: Example with Sort Declarations

```
     proof_list  ::=  list_of_proof{(proof_type{,assoc_list})}.
                      {step(reference,result,rule_appl,parent_list{,assoc_list}).}*
                      end_of_list.
      reference  ::=  term | identifier | user_reference
         result  ::=  term | user_result
      rule_appl  ::=  term | identifier | user_rule_appl
    parent_list  ::=  [parent{,parent}*]
         parent  ::=  term | identifier | user_parent
     assoc_list  ::=  [key:value{,key:value}*]
            key  ::=  term | identifier | user_key
          value  ::=  term | identifier | user_value
     proof_type  ::=  identifier | user_proof_type
```

All `user_` non-terminals of the grammar must be compatible with the already defined non-terminals. For example, a `user_key` must be a `term` or an `identifier`.

6.1 SPASS Proofs

Here is the instantiation of the general proof schema for SPASS style proofs that are supported by our proof checker.

```
 user_reference  ::=  number
    user_result  ::=  cnf_clause
 user_rule_appl  ::=  GeR | SpL | SpR | EqF | Rew | Obv | EmS | SoR | EqR
                      |
                      MPm | SPm | OPm | SHy | OHy | URR | Fac | Spt | Inp
                      |
                      Con | RRE | SSi | ClR | UnC | Ter
    user_parent  ::=  number
user_proof_type  ::=  SPASS
       user_key  ::=  splitlevel
     user_value  ::=  number
```

The association list as well as the key/value list is not used. Figure 3 shows an example for a DFG-problem together with a SPASS style resolution proof. The rule application identifiers name the SPASS inference/simplification/reduction rules general resolution (`GeR`), superposition left (`SpL`), superposition right (`SpR`), equality factoring (`EqF`), rewriting (`Rew`), obvious reduction (`Obv`) and clause reduction (`ClR`). Clauses are labelled with numbers and references inside of proof steps refer to these numbers.

7 Settings

The idea to include settings into the problem file format is to enable people to reproduce specific proofs that depend on particular input settings of the respective prover.

```
      settings  ::=  list_of_general_settings {setting_entry}+ end_of_list.
                     |
                     list_of_settings(setting_label). {* text *} end_of_list.
 setting_entry  ::=  hypothesis[label {,label}*].
 setting_label  ::=  KIV | LEM | OTTER | PROTEIN | SATURATE | 3TAP |
                     SETHEO | SPASS
```

The labels name the following systems: KIV [8], LEM [4], OTTER [6], PROTEIN [1], SATURATE [3], $_3T^AP$ [2], SETHEO [5], SPASS [10]. For example, to specify the precedence for SPASS and to direct SPASS to print a proof, we include the following settings:

```
begin_problem(ProofDemo).

list_of_descriptions.
name(*test.dfg*).
author(*SPASS*).
status(unsatisfiable).
description(*File generated by SPASS containing a proof.*).
end_of_list.

list_of_symbols.
functions[(skf1, 1)].
predicates[(P, 2)].
end_of_list.

list_of_clauses(conjectures, cnf).
clause(forall([U],or(P(U,skf1(U)))),1).
clause(forall([U],or(not(P(skf1(U),U)))),2).
clause(forall([V,U,W],or(equal(U,V),equal(V,W),equal(W,U))),3).
end_of_list.


list_of_proof(SPASS).
step(10,forall([V,U,W],or(equal(U,V),equal(V,skf1(W)),P(W,U))),SpR,[3,1]).
step(36,forall([V,U],or(equal(U,V),equal(V,skf1(skf1(U))))),GeR,[10,2]).
step(43,forall([V,U],or(equal(U,V),P(skf1(U),V))),SpR,[36,1]).
step(58,forall([V,U],or(not(P(U,skf1(V))),equal(V,U))),SpL,[36,2]).
step(86,forall([V,U],or(equal(U,skf1(V)),equal(V,skf1(U)))),GeR,[43,58]).
step(87,forall([U],or(not(equal(U,U)),equal(skf1(U),U))),EqF,[86,86]).
step(124,forall([U],or(equal(skf1(U),U))),Obv,[87]).
step(129,forall([U],or(P(U,U))),Rew,[124,1]).
step(130,forall([U],or(not(P(U,U)))),Rew,[124,2]).
step(213,or(false),ClR,[129,130]).
end_of_list.

end_problem.
```

Figure 3: A SPASS Style Resolution Proof

```
list_of_settings(SPASS).
{*
  set_flag(DocProof,1).
  set_precedence(a,b,c,f,F).
*}
end_of_list.
```

8 Miscellaneous

8.1 Comments

After the % symbol the rest of line is ignored. The comment symbols {* and *} are only allowed at the places defined above.

8.2 Conventions

We suggest the following conventions concerning suffixes of file names:

`.dfg`	For general problem files, including formulae, clauses, proofs at the same time.
`.frm`	For problem files containing at least lists of formulae.
`.cnf`	For problem files containing at least lists of clauses in conjunctive normal form.
`.dnf`	For problem files containing at least lists of clauses in disjunctive normal form.
`.prf`	For problem files containing at least lists of proofs.

Acknowledgements

We would like to thank all members of the German "Schwerpunkt Deduktion" group who contributed to previous versions of this paper. Special thanks to Michael Christen, Enno Keen, Andreas Nonnengart and Dalibor Topić who proof-read several versions of this paper.

References

[1] Peter Baumgartner and Ulrich Furbach. Protein: A *pro*ver with a *t*heory *e*xtension *in*terface. In A. Bundy, editor, *12th International Conference on Automated Deduction,CADE-12*, volume 814 of *LNAI*, pages 769–773. Springer, 1994. Available in the WWW, URL: `http://www.uni-koblenz.de/ag-ki/Systems/PROTEIN/`.

[2] Bernhard Beckert, Reiner Hähnle, Peter Oel, and Martin Sulzmann. The tableau-based theorem prover 3tap, version 4.0. In M.A. McRobbie and J.K. Slaney, editors, *13th International Conference on Automated Deduction, CADE-13*, volume 1104 of *LNCS*, pages 303–307. Springer, 1996.

[3] Harald Ganzinger and Robert Nieuwenhuis. The saturate system 1994. http://www.mpi-sb.mpg.de/SATURATE/Saturate.html, 1994.

[4] Birgit Heinz. *Anti-Unifikation modulo Gleichungstheorie und deren Anwendung zur Lemmagenerierung*. Dissertation, TU Berlin, Dec 1995.

[5] Reinhold Letz, Johann Schumann, S. Bayerl, and Wolfgang Bibel. Setheo: A high-performance theorem prover. *Journal of Automated Reasoning*, 8(2):183–212, 1992.

[6] William McCune. Otter 3.0 reference manual and guide. Technical Report ANL-94/6, Argonne National Laboratory, 1994.

[7] Francis Jeffry Pelletier. Seventy-five problems for testing automatic theorem provers. *Journal of Automated Reasoning*, 2(2):191–216, 1986. Errata: *Journal of Automated Reasoning*, 4(2):235–236,1988.

[8] Wolfgang Reif. The kiv-approach to software verification. In Manfred Broy and Stefan Jähnichen, editors, *KORSO: Methods, Languages, and Tools for the Construction of Correct Software – Final Report*, volume 1009 of *LNCS*, pages 339–368. Springer, 1995.

[9] Geoff Sutcliffe, Christian B. Suttner, and Theodor Yemenis. The TPTP problem library. In Alan Bundy, editor, *Twelfth International Conference on Automated Deduction, CADE-12*, volume 814 of *Lecture Notes in Artificial Intelligence, LNAI*, pages 252–266, Nancy, France, June 1994. Springer.

[10] Christoph Weidenbach, Uwe Brahm, Thomas Hillenbrand, Enno Keen, Christian Theobald, and Dalibor Topic. SPASS version 2.0. In Andrei Voronkov, editor, *Proceedings of the 18th International Conference on Automated Deduction (CADE-18)*, volume 2392 of *Lecture Notes in Artificial Intelligence*, pages 275–279, Kopenhagen, Denmark, 2002. Springer.