Common Syntax of the DFG-Schwerpunktprogramm Deduktion Version 1.5

Reiner Hähnle
Fakultät für Informatik
Universität Karlsruhe
D-76128 Karlsruhe
reiner@ira.uka.de

Manfred Kerber
School of Computer Science
The University of Birmingham
Birmingham, B15 2TT, England
M.Kerber@cs.bham.ac.uk

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

Renate A. Schmidt
Department of Computer Science
University of Manchester
Manchester, M13 9PL, England
schmidt@cs.man.ac.uk

Abstract

A common exchange format for logic problems to be used by members of the DFG-Schwerpunktprogramm Deduktion is introduced. 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. The language is partly more general than other popular exchange formats such as Otter or TPTP in allowing non-clausal and sorted formulae, several proof formats as well as user-defined operators. The latter feature makes it also useful for non-classical logics.

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 [8] or TPTP [12]) and vice versa. Currently we can (partly) transform Otter input files to DFG-Syntax files and vice versa.

In December 2002 the modal logic and description logic extension of SPASS implemented in MSPASS [5, 6, 11], was integrated into SPASS.

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 for various provers or some specific proofs.

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

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

4 The Logical Parts

Any non-predefined signature symbol used in a problem has to be defined in a 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}*
                   {special_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 mainly covers first-order logic, we are concerned with function and predicate symbols and non-standard operators and quantifiers. The usual first-order operators and quantifiers are predefined. In addition, there is a unique symbol for equality, see below.

For modal logic and description problems it is sometimes necessary to to link the propositional symbols to their corresponding first-order predicate symbols. This can be done with a translpairs declaration. The first symbol in a pair must always be a nullary predicate symbol, while the second is usually a unary or binary predicate symbol.

```
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}*].}
                  {operators[ op_sym | (op_sym,arity)
                      {, op_sym | (op_sym,arity)}*].}
                  {quantifiers[quant_sym | (quant_sym,arity)
                      {, quant_sym | (quant_sym,arity)}*].}
                  {translpairs[(pred_sym , pred_sym)
                      {, (pred_sym , pred_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 conjunction 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.

4.1 Special types of formulae

Modal logic or description logic problems are specified with special types of formulae, which include first-order formulae, propositional (or Boolean) type formulae and relational type formulae.

```
special_formula_list ::= list_of_special_formulae(origin_type,special_type).
                         {labelled_formula}*
                         end_of_list.
    labelled_formula ::= formula({term}{,label}) |
                         prop_formula_name({prop_term}{,label}) |
                         rel_formula_name({rel_term}{,label})
   prop_formula_name ::= prop_formula | concept_formula
    rel_formula_name ::= rel_formula | role_formula
        special_type ::= eml | dl
```

Propositional and relational type formulae can be constructed using familiar modal logic and description logic operators. The pre-defined logical operators include:

- the standard Boolean operators on propositional type and relational type formulae: true, false, not, and, or, implies (subsumed by), implied (subsumes), equiv,
- multi-modal operators with complex relational arguments: dia and box (synonyms are some and all), as well as domain and range,
- additional relational operators: comp (composition), sum (relative sum), conv (converse), id (the identity relation), div (the diversity relation), and
- test (test), domrestr (domain restriction) and ranrestr (range restriction).

```
            prop_term ::= prop_symbol | prop_symbol(prop_term{,prop_term}*)
                          | prop_quant_sym(rel_term,prop_term) |
                          prop_quant_sym_unary(rel_term)
          prop_symbol ::= true | false | or | and | not | implies |
                          implied | equiv | identifier
       prop_quant_sym ::= box | dia | all | some
 prop_quant_sym_unary ::= domain | range

             rel_term ::= rel_symbol | rel_symbol(rel_term{,rel_term}*)
                          | rel_prop_sym(rel_term,prop_term) |
                          rel_prop_sym_unary(prop_term)
           rel_symbol ::= true | false | id | div | or | and | not |
                          implies | implied | equiv | comp | sum | conv
                          | identifier
         rel_prop_sym ::= domrestr | ranrestr
   rel_prop_sym_unary ::= test
```

Note the symbols true and false have multiple interpretations. Apart from their usual interpretation in propositional logic and first-order logic, true and false may also be used as Boolean or relational formulae. true used as a Boolean type, represents the universal set, and used as a relational type it represents the universal relation. Similarly, false can be used as the bottom Boolean and relational type, representing the empty set and the empty relation.

```
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,b),f(a,c))).
end_of_list.

end_problem.
```

Figure 1: Pelletier's Problem No. 57

5 Alphabet

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

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

6 Examples

We start with a complete description of Pelletier's [9] problem No. 57 that can be found in Figure 1. The syntax for the description part is explained in Section 8.

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

Figures 3 to 5 give examples from modal logic, description logic and Tarski's relational calculus.

```
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

```
begin_problem(Halpern_Moses_branching_formula_1).
list_of_descriptions.
name({* Halpern and Moses (1992), Proposition 6.5 *}).
author({* Renate Schmidt *}).
status(satisfiable).
description({* Branching formulae of size O(m^2) satisfiable in a
     K-model with at least 2^m states.  From the proof of
     Proposition 6.5 of Halpern and Moses (1992).  m = 1.  *}).
end_of_list.

list_of_symbols.
predicates[ (R,2), (r,0), (p0,0), (p1,0), (d0,0), (d1,0), (d2,0) ].
end_of_list.

list_of_special_formulae(conjectures, EML).
prop_formula(
not( and( d0, not(d1),
     implies( d1, d0 ),
     implies( d2, d1 ),
     implies( d0, and( implies( p0, box(r, implies( d0, p0 ))),
     implies( not(p0), box(r, implies( d0, not(p0)))))),
     implies( d1, and( implies( p1, box(r, implies( d1, p1 ))),
     implies( not(p1), box(r, implies( d1, not(p1) ))))),
     implies( and( d0, not(d1) ),
     and( dia(r, and( d1, not(d2), p1 )),
     dia(r, and( d1, not(d2), not(p1) )))),
     box(r, and(
     implies( d1, d0 ),
     implies( d2, d1 ),
     implies( d0, and( implies( p0, box(r, implies( d0, p0 ))),
     implies( not(p0), box(r, implies( d0, not(p0)))))),
     implies( d1, and( implies( p1, box(r, implies( d1, p1 ))),
     implies( not(p1), box(r, implies( d1, not(p1)))))),
     implies( and( d0, not(d1) ),
     and( dia(r, and( d1, not(d2), p1 )),
     dia(r, and( d1, not(d2), not(p1) ))))))))).
end_of_list.

end_problem.
```

Figure 3: Modal logic example

```
begin_problem(Cheese_lovers_example).
list_of_descriptions.
name({* Cheese lovers example *}).
author({* Renate Schmidt*}).
status(unknown).
description({* A description logic example.  *}).
end_of_list.

list_of_symbols.
functions[ (adam,0), (bob,0), (cauliflower,0), (cheddar,0) ].
predicates[ (plant,0), (Plant,1), (cheese,0), (Cheese,1),
(food,0), (Food,1), (person,0), (Person,1),
(vegetarian,0), (Vegetarian,1), (cheese_lover,0), (Cheese_lover,1),
(eat,0), (Eat,2), (sibling_of,0), (Sibling_of,2) ].
translpairs[ (plant,Plant), (cheese,Cheese), (food,Food),
(vegetarian,Vegetarian), (cheese_lover,Cheese_lover),
(eat,Eat), (sibling_of,Sibling_of)].
end_of_list.

list_of_special_formulae(axioms, DL).
% TBox
% Plants and cheese are food
concept_formula( implies( or( plant, cheese ), food ) ).
% Persons eat food
concept_formula( implies( person, some( eat, food ) ) ).
% Vegetarians eat only plants
concept_formula( implies( vegetarian,
     and( some( eat, plant ), all( eat, plant ) ) ) ).
% Cheese lovers eat every cheese
concept_formula( implies( cheese_lover, not( some( not(eat), cheese ) ) ) ).
% sibling_of is a symmetric relation
role_formula( implies( sibling_of, conv( sibling_of ) ) ).
% sibling_of is a transitive relation
formula( forall( [x,y,z],
     implies( and( Sibling_of(x,y), Sibling_of(y,z) ), Sibling_of(x,z)))).

% ABox
formula( Person( adam ) ).
formula( Person( bob ) ).
formula( Sibling_of( adam, bob ) ).
formula( Plant( cauliflower ) ).
formula( Cheese( cheddar ) ).
formula( Eat( bob, cauliflower ) ).
formula( not( Eat( bob, cheddar ) ) ).
end_of_list.

list_of_special_formulae(conjectures, DL).
formula( not( Cheese_lover( bob ) ) ).  % valid
% formula( Vegetarian( bob ) ).  % not valid
end_of_list.

end_problem.
```

Figure 4: Description logic example

```
begin_problem(Relational_calculus_example).
list_of_descriptions.
name({* Relational calculus example *}).
author({* Renate *}).
status(unknown).
description({* Demonstrating the syntax of relational formulae *}).
end_of_list.

list_of_symbols.
predicates[(r,0), (s,0)].
end_of_list.

list_of_special_formulae(axioms, EML).
% r is a subrelation of s
rel_formula(implies(r,s)).

% r is transitive
rel_formula(implies(comp(r,r),r)).

% r is reflexive
rel_formula(implies(id,r)).

% r is symmetric
rel_formula(implies(r,conv(r))).
end_of_list.

list_of_special_formulae(conjectures, EML).
rel_formula(implies(id,s)).     % valid

%rel_formula(implies(comp(s,s),s)).     % not valid
end_of_list.

end_problem.
```

Figure 5: Relational calculus example

7 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.

```
  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`.

7.1 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 6 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.

```
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 6: A SPASS Style Resolution Proof

8 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
```

9 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 [10], LEM [4], OTTER [8], PROTEIN [1], SATURATE [3], $_3T^AP$ [2], SETHEO [7], SPASS [13]. For example, to specify the precedence for SPASS and to direct SPASS to print a proof, we include the following settings:

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

10 Miscellaneous

10.1 Comments

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

10.2 Includes

Another file might be included by

`include(filename)`

Includes are a necessary means to structure problems. However, they should be used with some care because with "includes", problem files are no longer stand alone documents. In order to avoid the problem of different pathname conventions on different machines, only files being in the same directory can be included. It is not allowed to use the include command inside formulae, clauses or declaration lists.

10.3 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 this paper. Special thanks to Michael Christen, Enno Keen and Andreas Nonnengart who proof-read several preliminary versions of this paper. Bernd Fischer generated the first yacc-based parser for the syntax.

Renate Schmidt gratefully acknowledges the support of the UK Engineering and Physical Sciences Research Council via research grants GR/M36700, GR/M88761 and GR/R92035.

References

[1] Peter Baumgartner and Ulrich Furbach. Protein: A *pro*ver with a *t*heory *e*xtension *i*nterface. In A. Bundy, editor, *12th International Conference on Automated Deduction,CADE-12*, volume 814 of *LNAI*, pages 769–773. Springer, 1994. `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] U. Hustadt and R. A. Schmidt. MSPASS: Modal reasoning by translation and first-order resolution. In R. Dyckhoff, editor, *Automated Reasoning with Analytic Tableaux and Related Methods, International Conference (TABLEAUX'2000)*, volume 1847 of *Lecture Notes in Artificial Intelligence*, pages 67–71. Springer, 2000.

[6] U. Hustadt, R. A. Schmidt, and C. Weidenbach. MSPASS: Subsumption testing with SPASS. In P. Lambrix, A. Borgida, M. Lenzerini, R. Möller, and P. Patel-Schneider, editors, *Proc. of Intern. Workshop on Description Logics'99*, pages 136–137. Linköping University, July 30–31 1999.

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

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

[9] 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.

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

[11] R. A. Schmidt. MSPASS. `http://www.cs.man.ac.uk/~schmidt/mspass/`.

[12] 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.

[13] Christoph Weidenbach, Bernd Gaede, and Georg Rock. Spass & flotter, version 0.42. In M.A. McRobbie and J.K. Slaney, editors, *13th International Conference on Automated Deduction, CADE-13*, volume 1104 of *LNAI*, pages 141–145. Springer, 1996.