Designing and Prototyping a PSOA2TPTP Translator from PSOA RuleML to TPTP format

Gen Zou, Reuben Peter-Paul
Supervisor: Harold Boley
Advisor: Alexandre Riazanov

February 5, 2012

1 Introduction

Knowledge representation is at the foundation of Semantic Web applications, using rule and ontology languages as the major kinds of formal languages. PSOA RuleML [1] is a recently developed rule language which combines the ideas of relational (predicate-based) and object-oriented (frame-based) modeling. In order to realize the PSOA RuleML semantics we have implemented a translator to map PSOA RuleML knowledge bases and queries in RIF-like presentation syntax to TPTP[1] full first order format (FOF). Our translations to TPTP are then executed in a full first order theorem prover: the VampirePrime prover[2].

We have based the following ANTLR lexer and grammar rules on initial work done by Dr. Alexandre Riazanov [7]. In our implementation, we translate PSOA RuleML via Abstract Syntax Trees (ASTs) to TPTP-FOF sentences. We use the ANTLR[3] language tool to specify an $LL(1)$ grammar (with rewrite rules) to recognize PSOA RuleML syntax specified in [1] (as well as some of the suggestions for "syntactic sugar"). The ANTLR framework is then invoked to generate a parser (implemented in Java). The "rewrite rules" mentioned above, are used to construct a *simplified Abstract Syntax Tree* for the *translation phase*.

We then specified a *tree grammar* in ANTLR with special *actions* to produce TPTP sentences. The ANTLR framework is invoked again to generate a parser

[1] http://www.cs.miami.edu/~tptp/

[2] http://riazanov.webs.com/software.htm

[3] ANother Tool for Language Recognition, a language framework for constructing recognizers, interpreters, compilers and translators from grammatical descriptions containing actions in a variety of target languages. http://www.antlr.org/

(also implemented in Java), which is then used to parse the above *simplified AST* and produce TPTP-FOF sentences.

Finally, we have implemented a RESTful web service API and client to translate PSOA RuleML and execute TPTP sentences against the VampirePrime prover. Complementing our work on translationg PSOA presentation syntax, Sadnan Manir has been developing and testing a parser for PSOA XML syntax. This will be integrated into the RuleML API and used for translating PSOA XML documents to TPTP documents.

2 Preliminaries

2.1 PSOA RuleML

PSOA RuleML is a rule language which generalizes the POSL [4] as well as the F-Logic and W3C RIF-BLD languages [2][3]. In PSOA RuleML, the positional-slotted, object-applicative (psoa) term is introduced as a generalization of: (1) the positional-slotted term in POSL; (2) the frame term and the class membership term in RIF-BLD. A psoa term has the following general form:

$$\texttt{o\#f([t}_{1,1}\ \ldots\ \texttt{t}_{1,n_1}\texttt{]}\ \ldots\ \texttt{[t}_{m,1}\ \ldots\ \texttt{t}_{m,n_m}\texttt{]}\ \texttt{p}_1\texttt{->v}_1\ \ldots\ \texttt{p}_k\texttt{->v}_k\texttt{)}$$

In the psoa term, `o` is the object identifier (OID) which gives a unique identity to the object described by the term. A psoa term combines three types of information: (1) the class membership information `o#f`, meaning `o` is an instance of type `f`; (2) each positional argument $\texttt{[t}_{i,1}\ \ldots\ \texttt{t}_{i,n_i}\texttt{]}$ represents a tuple of terms associated with the object; (3) each slotted argument $\texttt{p}_i\texttt{->v}_i$ denotes that the object has an attribute $\texttt{p}_i$ and the corresponding value is $\texttt{v}_i$.

A psoa term can be used as an atomic formula. Moreover, formulas in PSOA RuleML can be combined into other well-formed formulas using constructors from first-order logic: conjunction, disjunction, existential and universal quantifiers. Finally, formulas can also be composed to rules through the use of the rule implication connective, `':-'`.

2.2 TPTP-FOF and VampirePrime

TPTP (Thousands of Problems for Theorem Provers) is a library of test problems for automated theorem proving (ATP) systems. A TPTP problem is a list of formulaes of the form:

$$language(name, role, formula, source, useful\ info).$$

Symbol	Logic Meaning
~	not
&	and
|	or
![v1, v2, ...]	universal quantifier
?[v1, v2, ...]	existential quantifier
=>	imply
=	equal
!=	unequal

Table 1: TPTP-FOF Constructors

There are four types of languages in TPTP, within which the FOF is the language representing the full first-order form. *name* is a name given to the formulae. *role* is the type of the formulae. *axiom*, *hypothesis* and *conjecture* are the most widely used type names. *formula* is a formulae written in first order logic. *source* and *useful info* are optional and are neglected in our transformation. Some of the elementary constructs of TPTP-FOF are shown in table 1. Following is an example of a TPTP-FOF formulae.

```
fof(first_order,axiom,(
        ![X]: ( (p(X) | ~q(a)) => ?[Y,Z] : (r(f(Y),Z) & ~s) )
   ).
```

This formulae represents the first order formula

$$\forall \texttt{X} : ((\texttt{p(X)} \vee \neg \texttt{q(a)}) \rightarrow \exists \texttt{Y}\, \exists \texttt{Z} : \texttt{r(f(Y), Z)} \wedge \neg \texttt{s})$$

Vampire[4] is a world-leading automatic theorem prover for first-order logic. It has been developed in the Computer Science Department of the University of Manchester by Prof. Andrei Voronkov together with Kryštof Hoder and previously with Dr. Alexandre Riazanov. It has won the "world cup for theorem provers" in the most prestigious CNF (MIX) division eleven times (1999, 2001-2010). Vampire supports TPTP languages FOF and CNF. VampirePrime is an open source reasoner derived from the Sigma KEE[5] edition of Vampire.

3 Overall Architecture

The overall architecture of the PSOA RuleML implementations is shown in figure 1.

[4]http://en.wikipedia.org/wiki/Vampire_(theorem_prover)
[5]http://en.wikipedia.org/wiki/Sigma_knowledge_engineering_environment

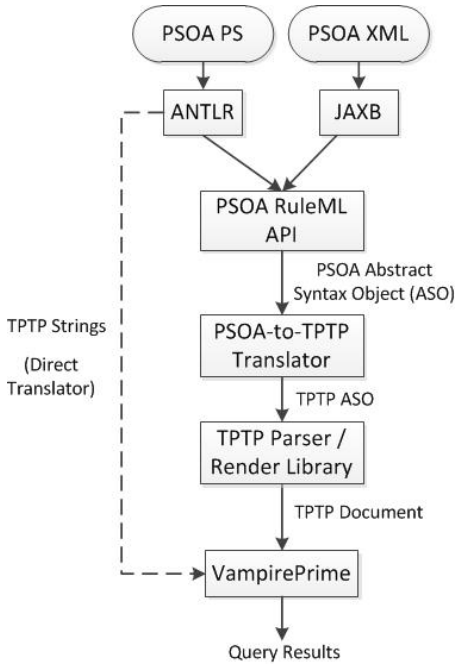

Figure 1: Overall workflow of PSOA RuleML implementations

There are two types of PSOA RuleML inputs: documents conforming to the PSOA presentation syntax (PS) and PSOA XML documents. For these two types of documents, respectively, the ANTLR parser generator and JAXB are used to generate the corresponding parsing codes, which are part of the RuleML API. After parsing, a PSOA Abstract Syntax Object (ASO) is created using the RuleML API factory methods. After that, the PSOA-to-TPTP translator is used to translate the PSOA ASO into a TPTP ASO using the TPTP parser/render library. Finally, TPTP strings are generated from a TPTP ASO, and fed into the VampirePrime theorem prover to get the query results. In this architecture, the parser of PSOA XML documents and the PSOA RuleML API have been implemented by Sadnan Manir, and the translation from PSOA ASO to TPTP ASO have not been implemented yet. Another unimplemented part is the generation of PSOA ASO using the ANTLR tree parser.

A direct translation indicated with a dashed line in figure 1 generates TPTP strings directly using the ANTLR tree parser. This solution is a shortcut used in our current project because of time constraints. The pros of such a solution are: (1) The direct translation is relatively easier and faster to implement; (2) it is more efficient since there are no auxiliary representations and fewer steps are

needed. The cons of this solution are: (1) It cannot be reused for the translation from PSOA XML documents to TPTP documents since it does not create and translate intermediate PSOA ASOs; (2) future developers of the translator need to acquire some knowledge on the ANTLR AST and the tree parser, as shown in figure 2; (3) compared to using the tested TPTP parser library to generate TPTP documents, a direct generation of TPTP strings is more error-prone.

Figure 2 shows the more concrete workflow of ANTLR-generated parsing classes, which indicated by the "ANTLR" box in figure 1. Firstly, the input PSOA PS document is broken into token streams by the lexer. After this step, a parser generated by ANTLR from the grammar file is used to parse the syntactic structure of the token stream and create an ANTLR AST. Finally, the AST is processed by tree parsers. The tree parser in the left side generates TPTP strings directly, which is part of our current implementation. The one in the right side creates PSOA ASOs using the RuleML API, which will be implemented in the future. The tree grammars used in the two tree parsers are identical and the parsing code is different. In the project we have implemented the lexer rules, grammar rules and tree grammar rules based on previous work done by Dr. Alexandre Riazanov. We did some slight changes to the lexer rules. The grammar rules and tree grammar rules are mainly implemented by us.

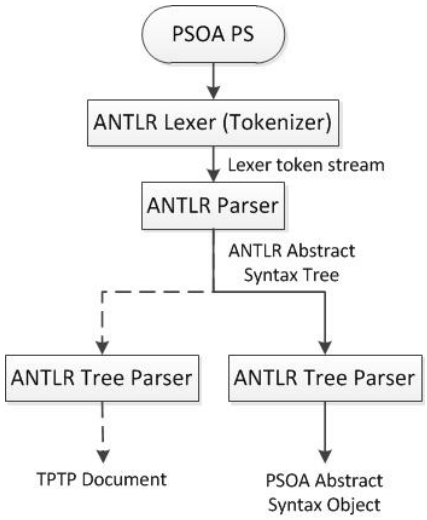

Figure 2: Workflow of ANTLR-generated parsing classes

4 Parser Implementation for PSOA Presentation Syntax

4.1 EBNF Grammar for PSOA RuleML

EBNF (Extended Backus-Naur Form) is a formal way of representing context-free grammars. An EBNF grammar of a language consists of a set of terminal symbols and a set of productions for non-terminals which shows the way terminal symbols are combined into a legal sequence. The EBNF grammar for PSOA RuleML presentation syntax is as follows [1] (shown in the format of ANTLR v3 grammar rules):

Rule Language:

```
document :  'Document''(' base?  prefix* import* group?')'  ;
base     :  'Base''(' ANGLEBRACKIRI ')'  ;
prefix   :  'Prefix''(' name ANGLEBRACKIRI ')'  ;
import   :  'Import''(' ANGLEBRACKIRI profile?  ')'  ;
group    :  'Group''(' (rule | group)* ')'  ;
rule     :  ('Forall' var+ '(' clause ')' ) | clause ;
clause   :  implies | atomic ;
implies  :  (head | 'And' '(' head* ')' ) ':-' formula ;
head     :  atomic | 'Exists' var+ '(' atomic ')'  ;
profile  :  ANGLEBRACKIRI ;
```

Condition Language:

```
formula  :  'And' '('  formula* ')'
         |  'Or' '('  formula* ')'
         |  'Exists' var+ '('  formula ')'
         |  atomic
         |  'External' '('  atom ')'  ;
atomic   :  atom | equal | subclass ;
atom     :  psoa ;
equal    :  term '=' term ;
subclass :  term '##' term ;
psoa     :  term '#' term '(' tuple* (term '->' term)* ')'  ;
tuple    :  '[' term* ']' ;
term     :  const | var | expr | 'External' '('  expr ')'  ;
expr     :  psoa ;
const    :  '"' UNICODESTRING '"^^' SYMSPACE | CONSTSHORT ;
var      :  '?'  UNICODESTRING? ;
symspace :  ANGLEBRACKIRI | CURIE ;
```

The EBNF for PSOA condition language represents the formulas that can be used as queries or premises of PSOA RuleML rules. On top of it, the EBNF for PSOA rule language shows the grammar specification for PSOA rules and documents.

Besides the EBNF grammar shown above, there is some syntactic sugar introduced in [1]:

- In the deobjectified version of a psoa term, the OID and the hash symbol `'#'` are omitted.
- For psoa terms without tuples or slots, the empty pair of parentheses can be omitted. For example, the psoa term `o#f()`, which represents just a membership relation, can be abridged to `o#f`.
- For psoa terms with only one tuple, the square brackets `'['` and `']'` enclosing the tuple's term sequence can be omitted. For example, `o#f([`t_1 t_2`] p->v)` can be abridged to `o#f(`t_1 t_2 `p->v)`.

4.2 Restricted PSOA RuleML Language

In order to parse PSOA RuleML unambiguously and efficiently using ANTLR, we impose the following restrictions on the use of the PSOA language, in which the phrase "simple term" is defined as a term which is either a constant or a variable.

1. The `'-'` character is not allowed to be used in a simple term.
2. The type term in a psoa term must be a simple term.
3. The use of external terms is only allowed in the following ways:
   - On the right side of an equality formula or the superclass side of a subclass formula
     Example: `?result = External(func:compare(?str1,?str2))`
   - as the OID of an atomic psoa formula
     Example: `External(modf:get_pat_by_WID(?WID))#haio:Patient`
   - as an external formula returning boolean values
     Example: `External(func:not(?isMale))`
   - as tuple values, slot names or slot values
     Example: `?o1#married(hasChild->External(eg:child(?o1)))`

The first restriction is introduced for simplified tokenization by the lexer. The `'-'` is disallowed because it is also a part of the slot symbol `'->'`. Without the restriction, the parsing of the slot `'hus->?Wu'` will produce an error. The lexer consumes the `'-'` as a part of the constant `'hus-'` and then the slot arrow token `'->'` cannot be produced by the lexer correctly. Since the ANTLR lexer does not seem to support backtracking, we have to disallow the use of `'-'` in a simple term. Such a restriction may lead to non-acceptance of some built-in external function names such as `pred:numeric-not-equal` and `func:string-length`. This problem can be solved in future by treating these

function names as reserved keywords, which has not been implemented in the current version of the PSOA parser.

The second and third restrictions are brought in when we transform the PSOA EBNF grammar into an $LL(1)$ grammar, which will be elaborated in the following section.

4.3 Grammar Rewriting for Efficient Parsing

ANTLR v3 implements the parser using an $LL(*)$ parsing mechanism. It constructs a DFA (Deterministic Finite Automaton) which looks ahead an arbitrary number of lexer tokens and decides which way to go when there are multiple alternatives in a grammar rule. In order to parse the PSOA document efficiently, we transform the PSOA EBNF grammar into an $LL(1)$ grammar, which means the parser only needs to look ahead one token in the stream to make a decision between multiple alternatives. In the following we follow the formal process in compiler theory to do the transformation [6]: (1) ambiguity resolution; (2) elimination of left recursion; (3) left-factoring.

4.3.1 Ambiguity Resolution

The ambiguity in the PSOA RuleML grammar arises when the use of syntactic sugar `o#f` and `f()` is allowed. The simplest example is the psoa term `o#f()`. It can be accepted as a psoa term in two ways: (1) `o` is accepted as the OID and `f` as the type term; (2) `o#f` is accepted as a single type term and `o#f()` as a deobjectified psoa term. Another example is `a#b#c`, in which the type term can be either `c` or `b#c`.

The method we employ to resolve such ambiguity is to restrict the type term to be only a simple term, which can be either a constant or a variable. The most important reason that we choose this restriction is that in real applications a complex type term is rarely needed. Even if it is useful in certain scenarios where a higher-order logic representation is needed or the type term is retrieved via an external term, we can use separate psoa terms `a#b` and `b#c` to express the meaning instead of a single term `a#b#c`. Another advantage of such a restriction is that it also eliminates some other sources of nondeterminism while parsing psoa terms, even if the syntactic sugar is disallowed, according to our experiment in ANTLR.

4.3.2 Elimination of Left Recursion

Left recursion is one of the main causes of a grammar to become a non-$LL(*)$ grammar. A simple example of a left-recursive ANTLR grammar with a single terminal `A` is:

```
p :  p A |  ;
```

This grammar of non-terminal `p` accepts a string of the form `A*`. However, an LL-based top down parser is incapable of parsing such a production since it does not consume any tokens when it goes to the alternative `p : p A`.

In PSOA EBNF grammar, the production for `psoa` is implicitly left-recursive since its first non-terminal `term` can also be a `psoa`. In order to remove such a left recursion, we employ a transformation in the following steps:

1. Separate `psoa` from the production of `term`.

   ```
   term :  psoa | non_psoa_term ;
   non_psoa_term :  const | var | external_term ;
   ```

2. Separate the left-recursive part from the `psoa` production. The remaining part except for the first term are grouped using a new non-terminal `psoa_rest`.

   ```
   psoa :  non_psoa_term psoa_rest | psoa psoa_rest ;
   ```

3. Rewrite the `psoa` production to remove left recursion.

   ```
   psoa :  non_psoa_term psoa_rest+ ;
   ```

The results of the rewritten productions are:

```
term :  psoa | non_psoa_term ;
non_psoa_term :  const | var | external_term ;
psoa :  non_psoa_term psoa_rest+ ;
```

4.3.3 Left Factoring

After removing the left recursion, a third step of rewriting the grammar is left factoring, which makes the grammar an $LL(1)$ grammar. Left factoring means to combine two or more alternatives into a single alternative by merging their common prefix. A simple example illustrating the problem of a grammar rule before left factoring is shown in the following. The example has two terminals `A, B` and two rules.

```
p :  q A | q ;
q :  B+ ;
```

While implementing the grammar rule for `p` in an LL-based parser, the parser is unable to decide on the alternatives `p : q A` and `p : q` until it reaches the end of the token stream to see whether there is a token `A` or not. However,

since the number of tokens in `q` is unlimited, the parser needs to look ahead an infinite number of tokens. This makes the grammar a non-$LL(*)$ grammar. To transform it to an $LL(*)$ grammar, we need to combine the common prefix `q` of the two alternatives `p : q A` and `p : q` and rewrite the grammar into `p : q A?`.

In the PSOA EBNF grammar there are many occurrences of common prefixes in the productions. The first example is the `‘[’...‘]’`-omitting syntactic sugar for single-tuple psoa terms:

```
tuples_and_slots : tuple* slot*
                 | term+ slot*
                 ;
slot  :  term '->' term ;
tuple :  '[' term* ']' ;
```

The non-terminal `tuples_and_slots` is introduced to match zero or more tuples or slots in psoa terms. The second alternative of the `tuples_and_slots` production is used for matching the syntactic sugar of single-tuple psoa terms in which `'['` and `']'` are omitted. With this syntactic sugar added, the first and second alternatives have a common prefix `term`. This is because when the first alternative matches a token stream where there are one or more slots occurring without tuples, `term` is the start of the first slot. Since `term` can match an arbitrary number of tokens, the grammar becomes non-$LL(*)$. Another problem of the second alternative is that the parser is unable to predict the end of `term+` since `slot*` can also begin with the non-terminal `term`. Only when the slot arrow `'->'` is seen by the parser will it know the processing of `term+` should have been finished one `term` earlier.

To transform the grammar into an $LL(1)$ grammar, we follow the steps shown below:

1. Separate the scenario which has the prefix `term` from the first alternative. That is the case where `tuples_and_slots` matches one or more slots.

   ```
   tuples_and_slots : tuple+ slot*
                    | term+ slot*
                    | slot+
                    |
                    ;
   ```

2. Rewrite the second and third alternatives to separate the prefix `'term'` from `slot*` and `slot+`.

   ```
   tuples_and_slots : tuple+ slot*
                    | term+ (term '->' term slot*)?
   ```

```
                    | term '->' term slot*
                    |
                    ;
```

3. Merge the second and the third alternatives into a single alternative.

```
tuples_and_slots : tuple+ slot*
                 | term+ ('->' term slot*)?
                 |
                 ;
```

   After merging, we can see that the common prefixes between different alternatives are eliminated and the grammar becomes an $LL(1)$ grammar.

Besides the one example shown above, other occurrences of a common prefix in the PSOA grammar include `clause`, `formula`, `atomic` and `const`. Some of them relate to multiple productions and rewriting is more difficult. One method we use to make rewriting relatively simpler is adding restrictions to one alternative so that the separation of common prefix scenarios becomes easier. For example, in the production `formula` the alternatives `formula : atomic` and `formula : external '(' atom ')'` have a common prefix `external_term`. To simplify the left factoring for these two alternatives, we carefully analyzed the use cases of external terms in current samples and disallow certain usages of external terms which are less likely to appear. With the restrictions shown in section 4.2, we are able to separate the cases with the prefix `'external_term'` from `'atomic'` and combine it with the alternative `formula : external '(' atom ')'`.

4.4 Constructing and Parsing ANTLR AST

In the previous section we discussed the rewriting of the PSOA grammar. These transformations we did on the grammar make it much easier for the parser; however, the transformed grammar tends to be more difficult for developers to read than the original grammar. In order to make the tree-processing step easier, we did a complete reconstruction on the output ANTLR AST of the parser. After this reconstruction, we managed to hide all the complexities of the parsing in the grammar implementation and keep the tree grammar simple and reusable for future development.

The ANTLR parser creates an ANTLR AST which has nodes of type CommonTree with the grammar options `output = AST` and `ASTLabelType = CommonTree` on. The default ANTLR tree construction operation for a grammar builds a flat tree containing pointers to all the input tokens. In order to make the tree meaningful, tree-rewriting rules need to be added to the grammar. In ANTLR v3, a

large variety of rewriting rules is provided to support nearly all common rewriting tasks. In the following we give some examples of tree rewriting which are used in our implementation:

1. **Omitting Input Elements**

```
document :  DOCUMENT '(' base? prefix* importDecl* group? ')'
         -> ^(DOCUMENT base? prefix* importDecl* group?)
         ;
```

   In this example the tokens `'('` and `')'` are omitted in the output AST.

2. **Making Input Elements the Root of Others**

```
base :  BASE '(' IRI_REF ')' -> ^(BASE IRI_REF) ;
```

   In this example the `BASE` token is made to be the root of the tree.

3. **Adding Imaginary Nodes**

```
tuple :  '[' term+ ']' -> ^(TUPLE term+) ;
```

   In this example an imaginary token `TUPLE` is added to be the root of the output tree.

4. **Referring to Labels in Rewrite Rules**

```
slot :  name=term '->' value=term -> ^(SLOT $name $value);
```

   In this example the two terms are labeled with `name` and `value` and referenced in the rewrite rules by adding a `'$'` ahead of the label.

5. **Deriving Imaginary Nodes from Real Tokens**

```
constant :  NUMBER -> ^(SHORTCONST NUMBER)
         |  ID -> ^(SHORTCONST LOCAL[$ID.text.substring(1)])
         ;
```

   In this example one rewrite rule is used for each alternative of the rule. In the second alternative, an imaginary node `LOCAL` is created with part of the text of token `ID`. Here `ID` is used to match local constants starting with an `'_'`.

After tree rewriting, the tree grammar used for tree parsing is again close to the original PSOA EBNF grammar, which makes it easier for reading and processing.

5 PSOA-to-TPTP Translation

5.1 Semantics-Preserving Translation from PSOA RuleML to TPTP-FOF

The semantics-preserving translation from PSOA RuleML to TPTP has two phases: normalization and translating elementary constructs. In the normalization phase, we need to 'break' some composite formulas into certain types of elementary formulas. And in the second phase we translate the elementary formulas to the corresponding TPTP-FOF forms.

5.1.1 Normalization

In the normalization phase, we transform the original knowledge base into a semantically equivalent one which only consists of elementary formulas. An elementary formula is a formula which cannot be split into equivalent subformulas. The reason for introducing the normalization phase is that a one-to-one transformation on elementary formulas avoids adding additional rules to derive subformulas. An example is the psoa term `o#f(`t_1`...`t_k`)`. This psoa term is equivalent to a conjunction of two subformulas: `o#f()` and `o#Top(`t_1`...`t_k`)`. If we translate `o#f(`t_1`...`t_k`)` into a single TPTP-FOF term, an additional derivation rule must be added to get the derivation `o#f(`t_1`...`t_k`) => o#f()`.

There are two major tasks in this phase: (1) decouple nested formulas; (2) transform the psoa term into elementary formulas. For the first task, any psoa terms which are not atomic formulas should be separated from the original formula. For example, `o1#f()=o2` must be separated to `o1#f()` and `o1=o2`. The detailed steps of the decoupling process are not the consideration of this project and need to be further investigated in the future. In the second step, we do transformations on simple psoa formulas in which the OIDs, tuple terms, slot name terms and slot values are all simple terms, which will be discussed next.

According to [1], an atomic psoa formula consists of an object-typing membership, a bag of tuples representing a conjunction of all the object-centered tuples (tupribution) and a bag of slots representing a conjunction of all the object-centered slots (slotribution). Its truth value is defined as

- $TVal_{\mathcal{I}}$`(o#f([`$t_{1,1}$ ... `$t_{1,n_1}$`] ... [`$t_{m,1}$ ... `t_{m,n_m}`] `p_1`->`v_1 ... p_k`->`v_k`))` $= \mathbf{t}$ if and only if
  $TVal_{\mathcal{I}}$`(o#f)` $=$
  $TVal_{\mathcal{I}}$`(o#Top([`$t_{1,1}$ ... `$t_{1,n_1}$`]))` $= \ldots =$ $TVal_{\mathcal{I}}$`(o#Top([`$t_{m,1}$ ... `t_{m,n_m}`]))` $=$
  $TVal_{\mathcal{I}}$`(o#Top(`p_1`->`v_1`))` $= \ldots =$ $TVal_{\mathcal{I}}$`(o#Top(`p_k`->`v_k`))` $= \mathbf{t}$.

According to the definition, the psoa formula is equivalent to a conjunction of

1+m+k subformulas, including 1 class membership formula, m single-tuple formulas and k single-slot formulas. These subformulas are the elementary formulas which we will use in later processing.

5.1.2 Translation of Elementary PSOA Constructs

In PSOA RuleML, the elementary formulas and the corresponding TPTP-FOF translations are as follows:

- **Constants**
  In PSOA RuleML, constants have the form "literal"^^symspace, where literal is a sequence of Unicode characters and symspace is an identifier for a symbol space. There are also 6 types of shortcuts for constants, as shown in the production of CONSTSHORT [5]:

  ```
  CONSTSHORT  ::= ANGLEBRACKIRI
                | CURIE
                | '"' UNICODESTRING '"'
                | NumericLiteral
                | '_' NCName
                | '"' UNICODESTRING '"' '@' langtag
  ```

  In TPTP, a constant can be either an identifier starting with a lowercase letter, a single quoted string or a numeric constant. In the current project, we translate constants of the form _NCName into a TPTP identifier by removing '_', and the first character of NCName is converted to lower case. Constants of type NumericLiteral are kept without any change. In future development we may consider using single-quoted full URIs for all constants.

- **Variables**
  In PSOA RuleML a variable starts with a question mark '?' and its name can be any Unicode strings. In contrast, TPTP-FOF variables always start with an upper case letter. This means that some of the PSOA RuleML variables may not be translatable to TPTP-FOF with their names preserved. However, since consistently changing the names of variables does not affect the semantics of rules, we can simply use internally generated variable names in the translation of PSOA knowledge base. For PSOA queries where variable names need to be kept for creating query results, we require that each name of the variable must start with an upper-case letter like in TPTP-FOF.

- **Tuple Terms**
  Tuple terms in PSOA are of the form `o#Top(t1...tk)`. The meaning of the term is that the tuple `[t1...tk]` has an associated object identifier `o`. To translate a PSOA tuple term into TPTP-FOF, we use a reserved predicate, `tupterm`, and move the OID `o` to the first argument of the predicate. The `k` components of the tuple become the second to `(k+1)`th arguments, correspondingly. The result is a `(k+1)`-ary term `tupterm(o,t1...tk)`. Note that the predicate `tupterm` is polyadic – i.e., a predicate of varying arity, which is allowed in TPTP-FOF.

- **Slot Terms**
  Slot terms in PSOA have the form `o#Top(pi->vi)`. The meaning of the slot term is that the object with an OID `o` has a property `pi` and the property value is `vi`. We use another reserved predicate, `sloterm`, to represent the relationship in TPTP-FOF. The result is a ternary term `sloterm(o,pi,vi)`, like an RDF triple.

- **Membership Terms**
  Class membership terms in PSOA are like `o#f()` (abridged `o#f`), meaning `o` is of type `f`. In the translation we use a third reserved predicate, `member`, and the result is a binary term `member(o,f)`. An alternative translation of `o#f()` in TPTP-FOF would be `f(o)`, where the type `f` is treated as a unary predicate. We choose the first translation because it uses a reserved predicate so that the PSOA-generated TPTP can be used along with TPTP from other sources.

- **Subclass Formulas**
  Subclass formulas `c1##c2` in PSOA are reused unchanged from RIF, meaning all the instances of type `c1` are also instances of type `c2`. To translate the subclass formula, a fourth reserved predicate `subclass` is used to represent the subsumption relationship between `c1` and `c2`. The translation of the formula in TPTP-FOF is `subclass(c1,c2)`. Note that solely with such a translation, we are not able to infer the inheritance `o#c2` just from the translation of `o#c1` and `c1##c2`. In order to make that inference, this extra inference rule for inheritance is needed in TPTP-FOF:

  ```
  ![X,Y,Z] (member(X,Y) & subclass(Y,Z) => member(X,Z))
  ```

  The rule means for any `X,Y,Z`, if `X` is of type `Y` and `Y` is a subclass of `Z`, then `X` is of type `Z`. With this single rule added, all the inheritances can be correctly deduced by the reasoner.

Table 2: Mapping from PSOA constructs to TPTP-FOF constructs

PSOA Constructs	TPTP-FOF Constructs
o#Top(t_1...t_k)	tupterm(o,t_1...t_k)
o#Top(p->v)	sloterm(o,p,v)
o#f()	member(o,f)
a##b	subclass(a, b)
a = b	a = b

- **Equality Formulas**
  In PSOA syntax an equality formula a=b means the individuals a and b are equal. This formula can be simply kept unchanged in TPTP-FOF.

Table 1 summarizes the mapping from elementary PSOA constructs to TPTP-FOF constructs.

5.2 Translator Implementation

In the current version of PSOA2TPTP translator, we have implemented the translation of a PSOA subset which has only ground facts in the form of unnested psoa terms, and the acceptable constants are numerals and short-form RIF local constants starting with '_'. The translator is able to translate both PSOA knowledge bases and queries. Ground facts in knowledge bases are translated to hypotheses and ground queries are translated to conjectures in TPTP-FOF. In the following we show some translation samples:

- _Joe#_Male()
  Output: fof(fact1, hypothesis, member(joe, male)).
- _f1#_family(_child->_Pete)
  Output: fof(fact2, hypothesis,
  member(f1, family)& sloterm(f1, child, pete)).
- _m2#_married(_John _Mary _child->_Tom)
  Output: fof(fact3, hypothesis,
  member(m2, married)
  & tupterm(m2, john, mary)
  & sloterm(m2, child, tom)).
- _m3#_married([_Tim _Jane1] [1980])
  Output: fof(fact4, hypothesis,

```
  member(m3, married)
& tupterm(m3, tim, jane1)
& tupterm(m3, 1980)).
```

- `_f4#_family([_Mike _Jessie] [1991] _child->_Tomas _child->_Jane2)`
  Output:
```
fof( fact5, hypothesis,
  member(f4, family)
& tupterm(f4, mike, jessie)
& tupterm(f4, 1991)
& sloterm(f4, child, tomas)
& sloterm(f4, child, jane2)).
```

The examples shown above include psoa terms with single and multiple tuples and slots, as well as the syntactic sugar for single-tuple psoa term without square brackets. More examples can be found in the project repository[6].

6 RESTful Web Service API

To demonstrate the above implementation we have implemented two RESTful web services[7]: a service to provide translation from PSOA RuleML presentation syntax to TPTP-FOF sentences, and a service to execute TPTP-FOF sentences on the VampirePrime theorem prover.

Table 3: RESTful Resources

Resource	Methods	Description
Translate	POST	This resource contains the PSOA2TPTP translator. *Media types: application/json*
Execute	POST	This resource contains the VampirePrime theorem prover. *Media types: application/json*

As listed above, in Table 3, the HTTP POST method is allowed for the listed resources. This means that to translate PSOA RuleML into TPTP-FOF, an HTTP POST is invoked against the Translate resource's URI (seeTable 4). Note also in Table 3, the available media type is application/json which means that JSON is used for data interchange.

[6] https://github.com/reubenPeterPaul/PSOA-to-TPTP/tree/master/ruleml-api/src/test/resources/PSOAFactsTest

[7] RESTful web services are simple web services implemented using HTTP and the principles of REST [8].

Table 4: RESTful Resource URI's

Resource	URI	Description
Translate PSOA RuleML	http://example.ws/translate	{"document": "query":"..."}
Execute TPTP-FOF	http://example.ws/execute	{"document"="[...]", "query"="..."}

To translate a PSOA RuleML document, first, the document must be URL-encoded for HTTP requests and placed in a JSON string, e.g.:[8].

```
{
    "document":"PSOA RuleML document...",
    "query":"PSOA RuleML query..."
}
```

Listing 1: JSON String containing PSOA RuleML document and query.

Then, the HTTP client sends a `POST`-based request to the *Translate* uri:

```
# Request
POST /translate HTTP/1.1
Host: example.ws
Content-Type: application/json

{
    "document":"...", PSOA-RuleML document
    "query":"..." PSOA-RuleML query
}
```

Listing 2: POST-based request to /translate resource

The server should then respond with something like the following:

```
# Response
HTTP/1.1 200 OK
Content-Type: application/json
Date: Thu Dec 22 16:17:13 AST 2011

{
    "tptp-document":[...], # TPTP-FOF sentences
    "tptp-conjecture":"..." # TPTP-FOF conjecture
}
```

Listing 3: Response to POST-based request to /translate resource

[8]HTTP requests can be made using a client as simple as Linux / Unix *curl* or as complex as a browser executing scripts (e.g. html web page with javascript embedded). See our *Online Demo* http://reubenpeterpaul.github.com/PSOA-to-TPTP for an example of such a client.

To execute the generated TPTP-FOF sentences in the VampirePrime, the *Execute* resources listed in Table 3 should be used by sending a `POST`-based response the URI listed in Table 4:

```
# Request
POST /execute HTTP/1.1
Host: example.ws
Content-Type: application/json

{
    "document":[...], # TPTP-FOF sentences
    "query":... # TPTP-FOF conjecture
}
```

Listing 4: POST-based request to the /execute resource

The server should then respond with something like the following:

```
# Response
HTTP/1.1 200 OK
Content-Type: text/plain
Date: Thu Dec 22 16:17:13 AST 2011

... # VampirePrime raw output stream
```

Listing 5: Response to POST-based request to the /execute resource

7 Conclusion

In this project we implement a preliminary translator PSOA2TPTP, which takes PSOA RuleML presentation syntax documents as input and produces semantically equivalent TPTP-FOF documents. Through the translator, PSOA documents with only ground unnested PSOA facts can be successfully translated into TPTP format documents that can be executed by the VampirePrime reasoner. The major work we have done in this project includes: (1) optimization of the complete PSOA RuleML grammar into an $LL(1)$ grammar and implemented it using the ANTLR v3 parser generator framework; (2) we designed and implemented an intermediate AST using ANTLR v3 parse tree rewriting mechanisms; (3) we developed a reusable tree grammar for the AST mentioned in (2); (4) using the tree grammar in (3) we were able to implement the first version of PSOA2TPTP translator. The first two items are subprojects of the RuleML API framework and were accomplished through collaborations with Alexandre Riazanov and Sadnan Manir.

The future work on the PSOA RuleML implementation includes: (1) completing the parser of PSOA RuleML by producing a RuleML API *abstract syntax object (ASO)*; (2) writing a translator from PSOA ASO to TPTP ASO for both

PSOA XML translation and PSOA presentation syntax translation; (3) extending the PSOA2TPTP translator capacity to handle all PSOA constructs; (4) creating a testbed for rigorously testing samples of the PSOA RuleML implementation.

References

[1] H. Boley, A RIF-Style Semantics for RuleML-Integrated Positional-Slotted, Object-Applicative Rules, RuleML Europe 2011, 194-211

[2] H. Boley and M. Kifer, A Guide to the Basic Logic Dialect for Rule Interchange on the Web. *IEEE Transactions on Knowledge and Data Engineering*, 22(11):1593-1608

[3] H. Boley and M. Kifer, RIF Basic Logic Dialect, http://www.w3.org/TR/rif-bld/

[4] H. Boley, Integrating Positional and Slotted Knowledge on the Semantic Web. *Journal of Emerging Technologies in Web Intelligence*, 4(2):343-353, November 2010. http://ojs.academypublisher.com/index.php/jetwi/article/view/0204343353

[5] A. Polleres, H. Boley and M. Kifer, RIF Datatypes and Built-Ins 1.0, W3C Recommendation, http://www.w3.org/TR/2010/REC-rif-dtb-20100622/

[6] A.V. Aho, M.S. Lam, R. Sethi and J.D. Ullman, Compiler Principles, Techniques and Tools (2nd Edition), Prentice Hall, 2006

[7] Alexandre Riazanov. Skype interview. 15 11.2011.

[8] R.T. Fielding, 2000. Architectural styles and the design of network-based software architectures. PhD Dissertation. Dept. of Information and Computer Science, University of California, Irvine