PSOA2TPTP: A Reference Translator for Interoperating PSOA RuleML with TPTP Reasoners

Gen Zou[1], Reuben Peter-Paul[1], Harold Boley[1,2], and Alexandre Riazanov[3]

[1] Faculty of Computer Science, University of New Brunswick, Fredericton, Canada
gen.zou AT unb.ca, reuben.peterpaul AT gmail.com,
[2] National Research Council of Canada
harold.boley AT nrc.gc.ca,
[3] Faculty of Computer Science, University of New Brunswick, Saint John, Canada
alexandre.riazanov AT gmail.com

Abstract. PSOA RuleML is a recently specified rule language combining relational and object-oriented modeling. In order to provide reasoning services for PSOA RuleML, we have implemented a reference translator, PSOA2TPTP, to map knowledge bases and queries in the PSOA RuleML presentation syntax (PSOA/PS) to the popular TPTP format, supported by many first-order logic reasoners. In particular, PSOA RuleML reasoning has become available using the open-source VampirePrime reasoner, enabling query answering and entailment as well as consistency checking. The translator, currently composed of a lexer, a parser, and tree walkers, is generated by the ANTLR v3 parser generator tool from the grammars we developed. We discuss how to rewrite the original PSOA/PS grammar into an $LL(1)$ grammar, thus demonstrating that PSOA/PS can be parsed efficiently. We also present a semantics-preserving mapping from PSOA RuleML to TPTP through a normalization and a translation phase. We wrap the translation and querying code into RESTful Web services for convenient remote access and provide a demo Web site.

1 Introduction

Semantic Web knowledge representations span objects, rules, and ontologies. PSOA RuleML [1] is a positional-slotted object-applicative rule language, including light-weight ontologies, which integrates relations (predicates) and objects (frames). To test the PSOA/PS syntax specification and also illustrate the PSOA RuleML semantics, we have developed a *reference implementation* as a translator named PSOA2TPTP[4] mapping knowledge bases and queries in PSOA RuleML in RIF-RuleML-like Presentation Syntax (PSOA/PS) to the TPTP[5] format – a *de facto* standard supported by many first-order logic reasoners. The translated document can then be executed by the open-source first-order reasoner VampirePrime[6] or other TPTP systems for query answering or simpler

[4] http://psoa2tptp.googlecode.com/
[5] Thousands of Problems for Theorem Provers, http://www.cs.miami.edu/~tptp/
[6] http://riazanov.webs.com/software.htm

reasoning tasks, such as consistency testing and entailment testing (theorem proving).

The main components of our two realizations of PSOA2TPTP include a shared lexer and parser as well as two tree walkers. The lexer breaks the input document up into a stream of tokens. The stream is then transformed by the parser into an Abstract Syntax Tree (AST) which condenses and structures the information of the input. Finally, the AST is traversed by the tree walkers, either for direct TPTP generation or via Abstract Syntax Objects (ASOs). These components are produced by the widely used ANTLR v3 framework[7] from, respectively, the specified grammars, namely a lexer grammar, a parser grammar and two tree grammars.

To prove feasibility of efficient parsing, we rewrite the original PSOA RuleML EBNF grammar into an $LL(1)$ grammar which accepts a slightly restricted subset of the PSOA RuleML language, including some syntactic sugar proposed in [1]. With this rewriting, the grammar is accepted by ANTLR and is more efficient for parsing. However, it becomes less readable and reusable. Thus, we chose to construct a customized AST by embedding additional rewrite rules into the parser grammar, and to develop an understandable and reusable tree grammar for ANTLR to generate the tree walkers.

To combine and deploy the above-mentioned components, we have also developed a RESTful Web API for translating PSOA/PS documents to TPTP and running VampirePrime. We have published a Web site to demonstrate the use of the API, constituting the first PSOA RuleML implementation release.[8]

The rest of the paper is organized as follows. Section 2 explains the translation source and targets of PSOA2TPTP. Section 3 shows the overall translation architecture. Section 4 discusses grammar implementation, especially the rewriting of the parser grammar. Section 5 gives the syntactic translation from PSOA/PS to TPTP-FOF. Section 6 discusses the RESTful Web API implementation. Section 7 concludes the paper and discusses future work.

2 Interoperation Source and Targets

We discuss here the source language, PSOA RuleML, the target language TPTP, and the target reasoner VampirePrime, of our interoperating translator.

2.1 PSOA RuleML

PSOA RuleML is a rule language that generalizes the POSL [2] as well as the F-Logic and W3C RIF-BLD languages [3]. In PSOA RuleML, the notion of a positional-slotted, object-applicative (psoa) term is introduced as a generalization of: (1) the positional-slotted term in POSL and (2) the frame term and the

[7] 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/

[8] http://198.164.40.211:8082/psoa2tptp-trans/

class membership term in RIF-BLD. A psoa term has the following general form:

$$\texttt{o\#f([}t_{1,1} \ldots t_{1,n_1}\texttt{]} \ldots \texttt{[}t_{m,1} \ldots t_{m,n_m}\texttt{]}\ p_1\texttt{->}v_1 \ldots p_k\texttt{->}v_k\texttt{)}$$

Here, o is the object identifier (OID) which gives a unique identity to the object described by the term by connecting three kinds of information: (1) The class membership o # f makes o an instance of class f; (2) each tupled argument $[t_{i,1} \ldots t_{i,n_i}]$ represents a sequence of terms associated with o; (3) each slotted argument p_i->v_i represents a pair of an attribute p_i and its value v_i associated with o.

A psoa term can be used as an atomic formula. Atomic formulas in PSOA RuleML can be combined into more complex formulas using constructors from the Horn-like subset of first-order logic: conjunction, disjunction in premises, as well as certain existential and universal quantifiers. Implication can be used to form rules.

2.2 TPTP and VampirePrime

TPTP (Thousands of Problems for Theorem Provers) is a library of test problems for automated theorem proving systems using a problem format of the same name. A TPTP problem is a list of annotated formulas of the form:

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

language specifies the TPTP dialect used to write the formula. We use the FOF dialect which allows the use of arbitrary first-order formulas. *name* is a name given to the formula; *role* specifies the intended use of the formula. The most important roles are *axiom*, *hypothesis*, *conjecture* and *theorem*. *formula* is the formula body. *source* and *useful info* are optional and irrelevant for us. Some of the constructors of TPTP are shown in Table 1.

Table 1. TPTP Constructors

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

Following is an example of an annotated TPTP formula.

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

This formula represents the first order formula

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

Vampire [4] is a mature high-performace reasoner for first-order logic. VampirePrime is an open source reasoner derived from the Sigma KEE[9] edition of Vampire. In addition to the standard first-order logic theorem proving tasks, such as consistency checking and entailment, VampirePrime supports query answering by implementing incremental query rewriting [5]. It can also be used for semantic querying on relational (SQL) databases modulo arbitrary first-order logic axioms.

3 Translation Architecture

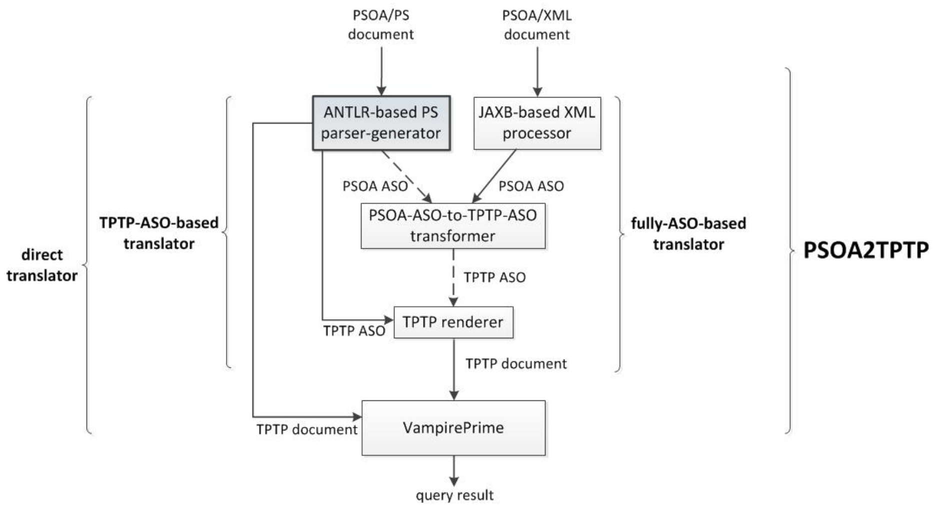

Fig. 1. Components and workflow of the PSOA2TPTP architecture

The overall architecture of PSOA2TPTP shown in Figure 1 includes three realizations: the direct translator, the TPTP-Abstract-Syntax-Object-based (TPTP-ASO-based) translator, and the fully-ASO-based translator. We have completed the first two for a subset of the PSOA RuleML language and the last one will be completed in the future. The input for the direct translator is a PSOA/PS document. We use the ANTLR v3 tool to generate a parser-translator that parses the input rulebase and query and generates a semantics-preserving TPTP document, which can be fed into VampirePrime to compute the query results. The concrete steps will be explained later. In contrast to the direct translator, the fully-ASO-based translator will create and transform PSOA RuleML Abstract Syntax Objects (PSOA ASOs) – simple data structures representing the information of the input document in a straightforward manner. The key component is the PSOA-ASO-to-TPTP-ASO translator, which transforms a PSOA

[9] http://en.wikipedia.org/wiki/Sigma_knowledge_engineering_environment

ASO into a TPTP ASO using the TPTP parser/renderer library.[10] The fully-ASO-based translator will also support documents conforming to a PSOA/XML syntax designed in the companion effort PSOA RuleML API [6], which also has developed a JAXB-based[11] XML parser. The dashed lines in the diagram give the context of components under development. The TPTP-ASO-based translator generates TPTP ASO directly using ANTLR.

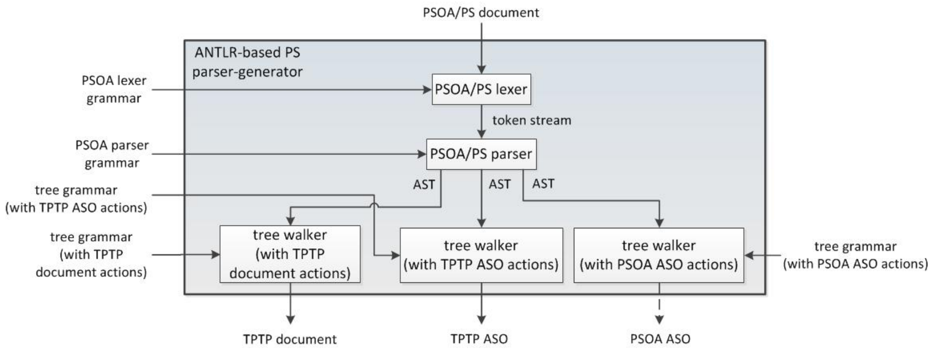
PSOA/PS document
ANTLR-based PS parser-generator
PSOA lexer grammar
PSOA/PS lexer
token stream
PSOA parser grammar
PSOA/PS parser
AST
AST
AST
tree grammar (with TPTP ASO actions)
tree grammar (with TPTP document actions)
tree walker (with TPTP document actions)
tree walker (with TPTP ASO actions)
tree walker (with PSOA ASO actions)
tree grammar (with PSOA ASO actions)
TPTP document
TPTP ASO
PSOA ASO

Fig. 2. Workflow of the parsing code generated from the ANTLR grammars

Figure 2 'explodes' the shaded box of Figure 1, showing the detailed workflow of the ANTLR-based PSOA/PS processor (a parser-generator). Firstly, the input PSOA/PS document is broken up into a token stream by the PSOA/PS lexer. In the stream, every token has an associated regular expression representing all the strings that will be accepted as this token, and the decomposition is done by matching these regular expressions. After this step, the PSOA/PS parser feeds off the token stream, parses its syntactic structure, and creates an Abstract Syntax Tree (AST), which is a condensed version of the input with a tree data structure. Finally, the AST is processed by a tree walker which generates the output. The tree walker on the left-hand side generates the TPTP document directly, which is part of the direct translator in Figure 1. The tree walker in the middle (resp., on the right-hand side) creates a TPTP (resp., PSOA) ASO, which is part of the TPTP-ASO-based (resp., fully-ASO-based) translator. The lexer, parser, and three tree walkers are generated by ANTLR from the corresponding ANTLR grammars. The syntactic specifications of the three tree grammars are identical while the embedded output-creating actions in the tree grammars are different. The tree grammar with PSOA ASO actions has not yet been developed while the two other ones have been completed.

Comparing the direct translator and the fully-ASO-based translator, the main advantages of the direct translator are: (d1) It requires fewer steps in Figure 1; (d2) it is only based on Java and ANTLR; (d3) it is more efficient, since there are no intermediate representations.

[10] http://riazanov.webs.com/tptp-parser.tgz
[11] http://jaxb.java.net/

The advantages of the fully-ASO-based translator are: (f1) It reuses an existing parser/renderer library for TPTP concrete syntax generation; (f2) the TPTP document generated from the TPTP library is more readable; (f3) it can be reused for the translation from other concrete PSOA syntaxes, such as the PSOA/XML serialization, provided that the corresponding parsers are able to generate PSOA ASOs; (f4) it will be easier to evolve the translation procedure, e.g., by implementing different translation styles, because the developers can work with a clean and simple ASO API instead of the complicated ANTLR AST parsing.

The TPTP-ASO-based translator is an intermediate implementation combining the advantages (d1), (d2), (f1), and (f2). To cover the entire interoperation space, we started with the direct translation, are now developing the TPTP-ASO-based translation, and will then proceed to the PSOA-ASO-based translation, thus allowing an overall comparison.

4 Grammar Implementation for PSOA RuleML Presentation Syntax

In this section, we discuss the implementation of the ANTLR grammars, especially the parser grammar. Firstly, we review the original ANTLR parser grammar for PSOA/PS which, in particular, incorporates the syntactic sugar into the EBNF grammar spefication. After that, we propose some additional restrictions on PSOA/PS to make efficient parsing possible. Then we apply some rewriting techniques to the parser grammar to make it acceptable for ANTLR. Finally we discuss the construction of AST and the tree grammar.

4.1 Original Parser Grammar

In [1], the EBNF grammar of PSOA/PS is specified. Besides this core grammar, there is some optional syntactic sugar for psoa terms:

– In the anonymous 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, e.g., `o # f([`t_1 t_2`] p->v)` can be abridged to `o # f(`t_1 t_2 `p->v)`.

In order to incorporate this syntactic sugar into the parser grammar, we need to change the original production (1) for psoa term into productions (2) and (3) shown below, where all the productions are in the ANTLR grammar style:

```
psoa : term '#' term '(' tuple* (term '->' term)* ')';  (1)

psoa : term '#' term ('(' tuples_and_slots ')')?        (2)
     | term '(' tuples_and_slots ')' ;

tuples_and_slots : tuple* (term '->' term)*            (3)
                 | term+ (term '->' term)* ;
```

The productions (2) and (3) will be further rewritten in Section 4.3.

4.2 Restricted PSOA RuleML Language

In order to simplify rewriting of the original PSOA/PS grammar into one which can be accepted by ANTLR, we impose the following restrictions on the use of the PSOA RuleML language.

1. The '`-`' character is not allowed in constant and variable names.
2. The class term in a psoa term must be a constant or variable.
3. A subclass formula, an equality formula or an anonymous psoa term must not start with an external term.

The first restriction is introduced to simplify tokenizing of the lexer. The second and third restrictions are brought in when we rewrite the original parser grammar into an $LL(1)$ grammar, which will be elaborated in the next section.

4.3 Grammar Rewriting

The ANTLR-generated parser uses an LL parsing mechanism. It constructs a DFA (Deterministic Finite Automaton) which can look ahead an arbitrary number of lexer tokens and choose to match one of the candidate patterns in a production. However, the original PSOA RuleML grammar is a non-LL grammar and cannot be used directly by ANTLR to generate the parser. So we rewrite it into an $LL(1)$ grammar, accepted by ANTLR. The grammar is efficient in that a single-token lookahead tells the parser which alternative to consider. We follow the formal process in the compiler theory to do the rewriting [7]: (1) ambiguity resolution; (2) elimination of left recursion; (3) left-factoring.

Ambiguity Resolution In the original parser grammar, the production for psoa term shown in Section 4.1 is ambiguous. The term `o#f()` can be accepted in two ways: (1) `o` is accepted as the OID and `f` as the class term; (2) `o#f` is accepted as a class term and `o#f()` as an anonymous psoa term. To resolve the ambiguity, we restrict the class term to be either a constant or a variable. With this restriction, a higher-order psoa term like `a#b#c` in which `b#c` is the class term needs to be expressed as a conjunction of separate psoa terms `a#b` and `b#c`.

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 grammar with a single terminal `A` is:

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

This grammar of non-terminal `p` accepts a string of the form `A*`. However, no $LL - based$ parser is capable of parsing such a production since it would not be able to consume any token when it applies the alternative `p : p A`.

In PSOA/PS, the production for `psoa` is implicitly left-recursive since its first non-terminal `term` can also be a `psoa`. In order to eliminate left recursion, we employ a rewriting in the following steps, where productions (4), (5) and (8) are the results:

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

   ```
   term : psoa | non_psoa_term ;                        (4)
   non_psoa_term : const | var | external_term ;      (5)
   ```

2. Merge the two alternatives of the `psoa` production in Section 4.1 by combining the common prefix `term`, and group the remaining part using a new non-terminal `psoa_rest`, yielding production (6). Then we separate the left-recursive part from (6) and get (7).

   ```
   psoa : term psoa_rest ;                                  (6)
   psoa : non_psoa_term psoa_rest | psoa psoa_rest ;      (7)
   ```

3. Rewrite (7) to remove left recursion.

   ```
   psoa : non_psoa_term psoa_rest+ ;      (8)
   ```

Left Factoring After removing the left recursion, the third step of rewriting is left factoring, which makes the grammar an $LL(1)$ grammar. Left factoring means to combine multiple alternatives into one by merging their common prefix. Following is an example consisting of non-terminals `p,q` and terminals `A,B`:

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

While parsing a sentence of `p`, the parser needs to reach the end of the string to decide on the two alternatives which have a common prefix `q`. Since `q` can match an arbitrary number of tokens, no $LL(k)$ parser is able to distinguish the alternatives of `p`. By merging the common prefix, we can rewrite the production into `p : q A?` and the grammar becomes an $LL(1)$ grammar.

One of the examples of common prefixes in the original parser grammar is the production for `tuples_and_slots` in Section 4.1 which matches zero or more tuples or slots. The two alternatives have a common prefix `term` which accepts an arbitrary number of lexer tokens. Apart from this, an LL-parser is also incapable of predicting the end of `term+` since the start of a slot is also a `term`.

We follow the steps below to rewrite the production into $LL(1)$:

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+ (term '->' term)*
                    | term+ (term '->' term)*
                    | (term '->' term)+
                    |
                    ;
   ```

2. Rewrite the second and third alternatives to separate the prefix `'term'`.

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

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

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

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

Besides the example shown above, there are many other occurrences of common prefixes in the original parser grammar. Some of them relate to multiple productions and rewriting is more difficult. One method we employ to simplify rewriting is adding restrictions to some alternatives to make it easier to separate a common prefix. For example, the third restriction we introduced in 4.2 prohibits some use cases of external terms in an atomic formula. It allows us to separate the cases with the prefix `external_term` from `atomic` and combine it with the alternative `formula : external '(' atom ')'`.

4.4 Abstract Syntax Tree and Tree Grammar

In the previous section we have illustrated the rewriting of the parser grammar. The resulting $LL(1)$ grammar is easier for generating the parser but tends to be less reusable and more difficult for future developers to read and work on. Thus, we chose to construct an Abstract Syntax Tree (AST) by embedding rewrite rules into the parser grammar and develop a simple and reusable tree grammar for traversing the AST. Examples of rewrite rules used can be found in [8]. The AST retains the meaningful input tokens and encodes them into a tree structure, while some auxiliary tokens like `'('` and `')'` are removed. Some *imaginary tokens*, in the ANTLR terminology, are also added for easy recognition and navigation.

5 PSOA-to-TPTP Translation

5.1 Semantics-Preserving Translation from PSOA RuleML to TPTP

The semantics-preserving translation from PSOA RuleML to TPTP has two phases: (1) Normalization of composite formulas into a conjunction of elementary constructs and (2) translating them into corresponding TPTP forms.

Normalization In the normalization phase, we transform the original knowledge base (KB) into a semantically equivalent one which only uses elementary constructs. An elementary construct is a term or a formula that cannot be split into equivalent subformulas. The reason for normalization is that the subsequent one-to-one translation of elementary constructs into TPTP avoids adding additional axioms to derive subformulas. For example, the psoa formula `o # f(`t_1`...`t_k`)` is equivalent to a conjunction of two translation-ready subformulas: `o # f()` and `o # Top(`t_1`...`t_k`)`, where `Top` is the root class such that `o # Top` is true for any `o`. If we translated `o # f(`t_1`...`t_k`)` into a single TPTP formula, an additional axiom, corresponding to `o # f() :- o # f(`t_1`...`t_k`)`, would need to be added to be able to derive `o # f()` from `o # f(`t_1`...`t_k`)`.

There are two major steps in this phase: flattening nested psoa formulas and splitting flat composite psoa formulas. For the first step, any atomic formula with nested psoa terms (which must be anonymous [1]) will be flattened: The original formula is replaced by a conjunction containing equations pairing fresh variables with the nested psoa terms and containing the atomic formula in which each nested psoa term is replaced by its corresponding variable. Flattening will be applied recursively to equations that contain psoa terms with nested psoa terms until all the psoa terms are flat.

The second step is only needed for flattened psoa formulas that apply a predicate (not for psoa terms that apply a function), where the OID, slot names and values, and tuple components are all constants or variables. The definition of the truth value of a psoa formula in [1] introduces splitting semantically as follows (the meaning of a psoa formula is *defined* via its elementary constructs):

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

The composite psoa formula is split into a conjunction of `1+m+k` subformulas, including `1` class membership formula, `m` single-tuple formulas and `k` (RDF-triple-like) single-slot formulas. Normalization perfoms such splitting syntactically.

Translation of Elementary PSOA RuleML Constructs We define the translation function $\tau_{psoa}(\ldots)$ mapping each PSOA/PS elementary construct to a TPTP construct 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 six kinds of shortcuts for constants, as shown in the production of `CONSTSHORT` [9]:

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

In TPTP, a constant can be either an identifier starting with a lower-case letter, a single-quoted string or a numeric constant. In the current version of the translator, 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 as a configuration option.

– **Variables**
A PSOA/PS variable is a '`?`'-preceded Unicode string. To translate it into a TPTP variable starting with an upper-case letter, we replace '`?`' with '`Q`'. For example, a PSOA variable `?job` is mapped to a TPTP variable `Qjob`.

– **Tuple Terms**
A tuple term in PSOA/PS is of the form $\texttt{o}\,\#\,\texttt{Top}(\texttt{t}_1\texttt{...t}_\texttt{k})$. It associates the tuple $[\texttt{t}_1\texttt{...t}_\texttt{k}]$ with the OID `o` (other tuple terms can use the same OID). To translate it into TPTP, we use a reserved predicate, `tupterm`, and use `o` as the first argument of the predicate. The `k` components of the tuple follow as the sequence of remaining arguments. Since `o # Top` is true for every `o`, `Top` is omitted without affecting the semantics. The result is a `(1+k)`-ary term, $\texttt{tupterm}(\tau_{psoa}(\texttt{o}), \tau_{psoa}(\texttt{t}_1) \ldots \tau_{psoa}(\texttt{t}_\texttt{k}))$. Note that the predicate name `tupterm` is polyadic – i.e., representing predicates of different arities – as allowed by the TPTP syntax.

– **Slot Terms**
A slot term in PSOA/PS has the form $\texttt{o}\,\#\,\texttt{Top}(\texttt{p}_\texttt{i}\texttt{->v}_\texttt{i})$. Its meaning is that the object with an OID `o` has a property $\texttt{p}_\texttt{i}$ and the property value is $\texttt{v}_\texttt{i}$. We use another reserved predicate, `sloterm`, to represent this relationship in TPTP. `Top` is omitted as for tuple terms. The result is a ternary term, $\texttt{sloterm}(\tau_{psoa}(\texttt{o}), \tau_{psoa}(\texttt{p}_\texttt{i}), \tau_{psoa}(\texttt{v}_\texttt{i}))$, corresponding to an RDF triple.

– **Membership Terms**
Class membership terms in PSOA/PS are of the form `o # f()` (abridged `o # f`), meaning `o` is an instance of class `f`. In the translation, we use a third reserved predicate, `member`, so that the result is a binary term $\texttt{member}(\tau_{psoa}(\texttt{o}), \tau_{psoa}(\texttt{f}))$. In future versions, we may optionally use an alternative translation where `o # f()` would be translated as `f(o)`, i.e., treating

the class `f` as a unary predicate, for compatibility with other sources of TPTP formulas, such as the translator from RIF to TPTP.[12]

- **Subclass Formulas**
  Subclass formulas `c1 ## c2` in PSOA/PS are reused unchanged from RIF, meaning all the instances of class `c1` are also instances of class `c2`. To translate the subclass formula, a fourth reserved predicate `subclass` is used to represent the subsumption relation `##` in TPTP. The translation of the formula in TPTP is `subclass(`τ_{psoa}`(c1),` τ_{psoa}`(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 axiom for inheritance is needed in TPTP:

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

- **Equality Formulas**
  In PSOA/PS an equality formula `a = b` means the terms `a` and `b` are equal. This formula can be translated to τ_{psoa}`(a)` $=$ τ_{psoa}`(b)` in TPTP.
- **Rule Implications**
  In PSOA/PS a rule is represented by φ `:-` ψ, meaning formula φ is implied by formula ψ. It can be translated to $\tau_{psoa}(\psi)$ `=>` $\tau_{psoa}(\varphi)$.

Table 2. Mapping from PSOA/PS constructs to TPTP constructs

PSOA/PS Constructs	TPTP Constructs
`o # Top(`$t_1 \ldots t_k$`)`	`tupterm(`τ_{psoa}`(o),` $\tau_{psoa}(t_1) \ldots \tau_{psoa}(t_k)$`)`
`o # Top(p -> v)`	`sloterm(`τ_{psoa}`(o),` τ_{psoa}`(p),` τ_{psoa}`(v))`
`o # f()`	`member(`τ_{psoa}`(o),` τ_{psoa}`(f))`
`a ## b`	`subclass(`τ_{psoa}`(a),` τ_{psoa}`(b))`
`a = b`	τ_{psoa}`(a) =` τ_{psoa}`(b)`
`AND(`$f_1 \ldots f_n$`)`	`(`$\tau_{psoa}(f_1)$ `& ... &` $\tau_{psoa}(f_n)$`)`
φ `:-` ψ	$\tau_{psoa}(\psi)$ `=>` $\tau_{psoa}(\varphi)$

Table 2 summarizes the mapping from elementary PSOA/PS constructs to TPTP constructs, including one extra row for conjunctions, as needed in rule premises. The mapping is sufficient for the translation of a KB but not yet for a query: we expect to get the bindings for any query variables. Since query answering in VampirePrime is done through answer predicates, we introduce a reserved predicate `ans` as the answer predicate and map a PSOA query `q` into the following formula

```
![X1,X2,...] (τpsoa(q) =>
                 ans("?X1 = ", τpsoa(X1), "?X2 = ", τpsoa(X2), ...))
```

where `X1,X2, ...` are free variables in `q`. Answers from VampirePrime are of the form

[12] http://riazanov.webs.com/RIF_BLD_to_TPTP.tgz

```
ans("?X1 = ", v1, "?X2 = ", v2, ...)
```

where *v1, v2, . . .* are bindings for the variables. When the query has no variables, `ans` is used alone as the conclusion. A sample will be given in the next section.

5.2 Translator Implementation

In the current version of PSOA2TPTP, we have implemented the translation for a PSOA RuleML subset where the accepted constants are numerals and short-form RIF-like local constants starting with '_'. Following is a sample translation, where the two conjunctions resulting from normalization are implicit in the enclosing `Group`:

- **Input KB:**

```
Document(
  Group(
    _f1#_family(_Mike _Jessie _child->_Fred _child->_Jane)
    _Amy#_person([_female] [_bcs _mcs _phd] _job->_engineer)
  )
)
```

- **Normalized KB:**

```
Document(
  Group(
    _f1#_family()   _f1#Top(_Mike _Jessie)
         _f1#Top(_child->_Fred)   _f1#Top(_child->_Jane)
    _Amy#_person() & _Amy#Top(_female)
         _Amy#Top(_bcs _mcs _phd)   _Amy#Top(_job->_engineer)
  )
)
```

- **Query:** `_Amy#_person(_job->?job)`
- **Translator Output:**

```
fof( ax1, axiom,
     member(f1, family) & tupterm(f1, mike, jessie) &
     sloterm(f1, child, fred) & sloterm(f1, child, jane)).
fof( ax2, axiom,
     member(amy, person) & tupterm(amy, female) &
     tupterm(amy, bcs, mcs, phd) & sloterm(amy, job, engineer)).
fof( query, theorem,
     ![Qjob]: ((member(amy, person) & sloterm(amy, job, Qjob))
             => ans("?job = ", Qjob) )).
```

- **VampirePrime Output:**

```
Proof found.
...
... | «ans»("?job = ", engineer) ...
```

The sample KB has two psoa formulas as facts. The first fact has one tuple for the family's adults, where `_Mike _Jessie` is equivalent to `[_Mike _Jessie]`, a shortcut allowed only in single-tuple psoa terms; it has two slots for the family's children. The second fact has two tuples, of lengths 1 and 3, and also a slot. The two formulas in the first stage are broken into two conjunctions of elementary constructs, as shown in the normalized KB above. In the second stage, each construct is mapped to a corresponding TPTP term. The above translator output contains three translations, of which the first two are for the KB and the last is for the query.[13] In the VampirePrime output, `«ans»("?job = ",engineer)` indicates one binding, `engineer`, is obtained for the variable `?job`.

6 RESTful Web API

Representational State Transfer (REST) is an architectural style for distributed systems, specified in [11]. A RESTful Web API is an API implemented by using HTTP (operations, URIs, Internet media types, response codes) and by conforming to the *architectural constraints* specified in REST. We have implemented a RESTful Web API consisting of two resources, see URIs in Table 3: a resource representing the PSOA2TPTP translation component, and a resource representing the VampirePrime reasoner component. As shown in Table 4, the HTTP POST method is allowed and the *application/json* Internet media type is supported for the listed resources.

For example, to translate PSOA RuleML into TPTP, an HTTP POST [14] request with a JSON-encoded PSOA document in the body is sent to the *Translate* resource URI. The response will be a JSON encoding of the translated TPTP document.

Table 3. RESTful Resource URIs

Resource	URI
Translate	http://example.ws/translate
Execute	http://example.ws/execute

To execute the translated TPTP sentences in the VampirePrime reasoner, the *Execute* resource mentioned in Table 4 should be used by sending an HTTP POST request containing an *application/json* encoding of the TPTP sentences to the *Execute* resource URI. The server will then return the raw output stream (*text/plain*) of the VampirePrime-generated solutions (see listing 1.5 in [8]).

[13] PSOA2TPTP targets VampirePrime by using TPTP's `axiom` for the KB and `theorem` for queries, while [10] would suggest the use of `conjecture` for queries.

[14] POST is chosen over PUT to reflect idempotency of the translation service: a new translation (instance) is produced for each request.

Table 4. RESTful Resources

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

7 Conclusion

To enable rule/theorem prover interoperation, we implemented a first version of the PSOA2TPTP translator. It takes a document in PSOA/PS as input and generates a semantics-preserving TPTP document. Through the translator, PSOA RuleML documents are translated into the TPTP format and can then be executed by the VampirePrime reasoner or other TPTP systems.

Our work makes heavy use of the ANTLR v3 parser generator framework. We (1) rewrite the complete PSOA/PS EBNF grammar into an $LL(1)$ grammar; (2) construct an intermediate AST using ANTLR's tree rewrite mechanisms embedded in the parser grammar; (3) develop reusable tree grammars for parsing the AST; and (4) embed code into the tree grammars to generate ASOs and TPTP documents.

Future work on the PSOA RuleML implementation includes: (1) Extending the PSOA2TPTP translator capability to handle all PSOA RuleML constructs, introducing another reserved predicate for functional psoa terms; (2) implementing the fully-ASO-based translator; (3) creating a testbed for rigorously testing PSOA RuleML implementations; (4) 'inverting' PSOA2TPTP into a TPTP2PSOA translation for the Horn subset of first-order logic; (5) deploying PSOA2TPTP in the Clinical Intelligence use case [12], where PSOA rules are used to define a semantic mapping for a hospital data warehouse.

References

1. Boley, H.: A RIF-Style Semantics for RuleML-Integrated Positional-Slotted, Object-Applicative Rules. In Bassiliades, N., Governatori, G., Paschke, A., eds.: RuleML Europe. Volume 6826 of LNCS., Springer (2011) 194–211
2. Boley, H.: Integrating Positional and Slotted Knowledge on the Semantic Web. Journal of Emerging Technologies in Web Intelligence **4**(2) (November 2010) 343–353 Academy Publisher, Oulu, Finland, http://ojs.academypublisher.com/index.php/jetwi/article/view/0204343353.
3. Boley, H., Kifer, M.: A Guide to the Basic Logic Dialect for Rule Interchange on the Web. IEEE Transactions on Knowledge and Data Engineering **22**(11) (November 2010) 1593–1608
4. Riazanov, A., Voronkov, A.: The Design and Implementation of Vampire. AI Communications **15**(2-3) (2002) 91–110
5. Riazanov, A., Aragao, M.A.: Incremental Query Rewriting with Resolution. Canadian Semantic Web II (2010)

6. Al Manir, M.S., Riazanov, A., Boley, H., Baker, C.J.O.: PSOA RuleML API: A Tool for Processing Abstract and Concrete Syntaxes. (2012) In these proceedings.
7. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D., eds.: Compilers: Principles, Techniques, and Tools. Second edn. Pearson/Addison Wesley, Boston, MA, USA (2007)
8. Zou, G., Peter-Paul, R.: PSOA2TPTP: Designing and Prototyping a Translator from PSOA RuleML to TPTP Format. Technical report `http://psoa2tptp.googlecode.com/files/PSOA2TPTP_Report_v1.0.pdf`.
9. Polleres, A., Boley, H., Kifer, M.: RIF Datatypes and Built-ins 1.0 (June 2010) W3C Recommendation, `http://www.w3.org/TR/rif-dtb`.
10. Sutcliffe, G.: The TPTP Problem Library. `http://www.cs.miami.edu/~tptp/TPTP/TR/TPTPTR.shtml`
11. Fielding, R.T.: Architectural Styles and the Design of Network-based Software Architectures. PhD thesis, University of California, Irvine, Irvine, California, USA (2000)
12. Riazanov, A., Rose, G.W., Klein, A., Forster, A.J., Baker, C.J.O., Shaban-Nejad, A., Buckeridge, D.L.: Towards Clinical Intelligence with SADI Semantic Web Services: a Case Study with Hospital-Acquired Infections Data. In: Proceedings of the 4th International Workshop on Semantic Web Applications and Tools for the Life Sciences. SWAT4LS '11, New York, NY, USA, ACM (2012) 106–113