PSOATransRun: Translating and Running PSOA RuleML via the TPTP Interchange Language for Theorem Provers

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] Information and Communications Technologies, National Research Council Canada
harold.boley AT nrc.gc.ca,
[3] Department of Computer Science & Applied Statistics, UNB, Saint John, Canada
alexandre.riazanov AT gmail.com

Abstract. PSOA RuleML is an object-relational rule language generalizing POSL, OO RuleML, F-logic, and RIF-BLD. In PSOA RuleML, the notion of positional-slotted, object-applicative (psoa) terms is used as a generalization of: (1) positional-slotted terms in POSL and OO RuleML and (2) frame and class-membership terms in F-logic and RIF-BLD. We demonstrate an online PSOA RuleML reasoning service, PSOATransRun, consisting of a translator and an execution engine. The translator, PSOA2TPTP, maps knowledge bases and queries in the PSOA RuleML presentation syntax to the popular TPTP interchange language, which is supported by many first-order logic theorem provers. The translated documents are then executed by the open-source VampirePrime reasoner to perform query answering. In our implementation, we use the ANTLR v3 parser generator tool to build the translator based on the grammars we developed. We wrap the translator and execution engine as resources into a RESTful Web API for convenient access. The presentation demonstrates PSOATransRun with a suite of examples that also constitute an online-interactive introduction to PSOA RuleML.

1 Introduction

Knowledge representation is at the foundation of Semantic Web applications, using rule and ontology languages as the main kinds of formal languages. PSOA RuleML is a recently developed rule language which combines the ideas of relational (predicate-based) and object-oriented (frame-based) modeling. In order to demonstrate the PSOA RuleML semantics, we have implemented an online PSOA RuleML reasoning service PSOATransRun. It enables PSOA RuleML deduction using the first order open-source VampirePrime reasoner via the interchange language TPTP (Thousands of Problems for Theorem Provers), which is supported by many reasoners, especially theorem provers. PSOATransRun is composed of a translator, PSOA2TPTP, and a run-time environment in the form of a TPTP-aware execution engine. The translator maps knowledge bases

and queries of PSOA RuleML in RIF-like Presentation Syntax (PSOA/PS) into a document in TPTP's First Order Form (FOF), which is then fed into the VampirePrime reasoner to deduce the query results.

Our implementation of PSOA2TPTP is built upon the ANTLR v3 parser generator framework.[4] The main components include a lexer, a parser and a tree walker generated from the input ANTLR grammars. The input document is first broken up, by the lexer, into a token stream; then converted, by the parser, into a structured Abstract Syntax Tree (AST); and finally traversed, by the tree walker, to generate a TPTP document via TPTP Abstract Syntax Objects.

We wrapped the PSOA2TPTP translator and the VampirePrime-based execution engine as resources into a RESTful Web API, and published a Web site demonstrating its use.[5]

2 Preliminaries

2.1 PSOA RuleML

PSOA RuleML [1] is an object-relational rule language generalizing POSL, OO RuleML, F-logic, and RIF-BLD. In PSOA RuleML, the notion of positional-slotted, object-applicative (psoa) terms is introduced:

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

This notion generalizes (1) positional-slotted terms in POSL and OO RuleML and (2) frame and class-membership terms in F-logic and RIF-BLD. In a psoa term, `o` is the object identifier (OID) which uniquely identifies the object represented by the term. A psoa term integrates three types of information: (1) The class membership `o#f` makes `f` the type of instance `o`; (2) every slotted argument p_i`->`v_i associates `o` with an attribute p_i and its value v_i; (3) every tupled argument `[`$t_{i,1} \ldots t_{i,n_i}$`]` associates `o` with a sequence of terms.

2.2 TPTP-FOF and VampirePrime

TPTP is a collection of test problems for automated theorem proving systems using the problem format of the same name. TPTP-FOF is the dialect allowing the use of arbitrary first-order formulas. A TPTP-FOF problem is a list of annotated formulas of the form:

$$\mathit{fof}(\mathit{name}, \mathit{role}, \mathit{formula}, \mathit{source}, \mathit{useful\ info}).$$

Here, *name* is a name given to the formula; *role* specifies the type of intended use of the formula, e.g. *axiom*, *theorem*, *conjecture*, etc. *formula* is the formula body (*source* and *useful info* are optional and irrelevant for our translation). Table 1 shows the most widely used TPTP constructors.

[4] http://www.antlr.org/
[5] http://198.164.40.211:8082/psoa2tptp-trans/index.html

Table 1. TPTP Constructors

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

VampirePrime is an open source reasoner derived from Vampire [2], a mature high-performace reasoner for first-order logic. VampirePrime supports not only standard theorem proving tasks like consistency checking and entailment, but also query answering using the Incremental Query Rewriting Technique [3].

3 System Architecture

In Figure 1, we present an architectural view of the PSOATransRun framework. We use Linux for our host environment, and VampirePrime can be re-compiled for any platform that supports gcc 4.x. We use a Java servlet container to host the PSOATransRun RESTful-Web-API web application, which we depict in Figure 1 as a *Web ARchive* (WAR). The RESTful Web API WAR, basically consists of two JAX-RS[6] resources, and a static HTML page *application.html*. The Web API depends on the PSOA2TPTP-Translator Java application (see Figure 2) and it is also packaged into the WAR. The PSOATransRun application component, *application.html* is a static HTML Web page that accesses (via *XMLHttpRequests*[7]) the PSOATransRun RESTful resources (*Translate* and *Execute*) and composes them to provide an experimental PSOA Presentation Syntax (PSOA/PS) prototype for basic reasoning. The design and implementation of the RESTful Web API is described in more detail in Section 4.2.

The architecture of the PSOA2TPTP translator is depicted in more detail in Figure 2. The translation consists of four phases:

1. The *PSOA/PS lexer* feeds off the input document as a character stream and does lexical analysis, grouping the characters into a stream of tokens.
2. The *PSOA/PS parser* operates on the token stream emanating from the lexer, and parsing the grammatical structure while constructing an intermediate data structure called Abstract Syntax Tree (AST), which is a highly structured and condensed version of the input.
3. The *tree walker* traverses the AST and builds an internal data structure, TPTP Abstract Syntax Objects (TPTP ASOs), representing semantically equivalent TPTP formulas, based on the translation rules.

[6] JAX-RS is a Java API for RESTful Web Services that facilitates the creation of Web services according to the Representational State Transfer (REST) architectural style.

[7] Used to send HTTP requests directly to a Web server.

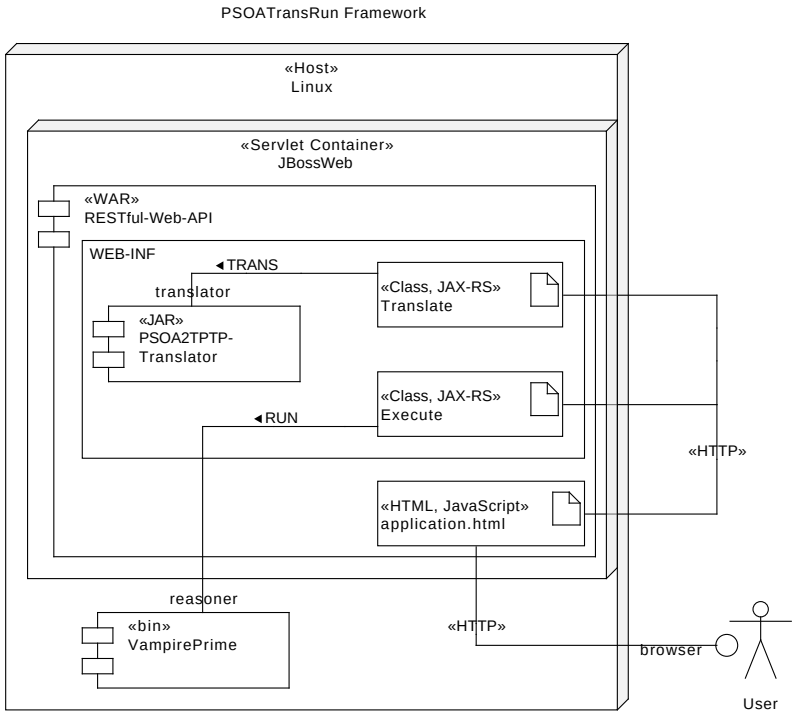

Fig. 1. Architecture of PSOATransRun. The PSOATransRun application, *application.html*, composes the *Translate* and *Execute* (Run) resources for PSOA/PS queries.

4. The *TPTP renderer* reuses an existing parser/renderer library[8] for generating TPTP documents in concrete syntax from TPTP ASOs.

The lexer, parser and tree walker are generated by the ANTLR framework[9] from the provided lexer grammar, parser grammar and tree grammar, respectively.

Our intention was to create an application programming interface (API) and expose our growing set of translation tools and reasoner services over the *World Wide Web* via Web services. We chose to apply the REpresentational State Transfer (REST)[10] architectural style when designing our API for practical

[8] http://riazanov.webs.com/tptp-parser.tgz

[9] ANother Tool for Language Recognition (ANTLR) is a parser generator widely used for building translators and interpreters for domain-specific languages. http://www.antlr.org/

[10] REST is an architectural style for distributed systems such as the World Wide Web. A RESTful Web API is an API that conforms to the RESTful architectural constraints specified in [4]

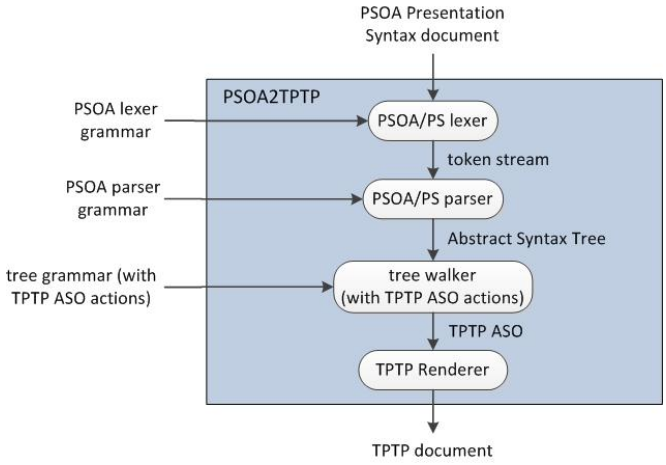

Fig. 2. Detailed architectural view of the PSOA2TPTP translator

reasons. While there are other architectural styles for distributed computing besides REST, RESTful Web APIs tend to be much easier to understand and use (see [5]).

4 Implementation

4.1 Translation

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.

In the first phase, every psoa formula of the form

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

is split into a conjunction of `1` class membership formula `o#f()`, `m` single-tuple formulas $\texttt{o\#Top(t}_{i,1}\ \ldots\ \texttt{t}_{i,n_i}\texttt{)}$ and `k` (RDF-triple-like) single-slot formulas $\texttt{o\#Top(p}_i\texttt{->v}_i\texttt{)}$. The rewriting preserves the semantics since the truth value of a psoa formula is *defined* by the conjunction.

In the second phase, we define the translation function $\tau_{psoa}(\cdot)$ mapping each PSOA/PS elementary construct to a TPTP construct as shown in Table 2.

In the translation, we use `'l'` and `'Q'` as the prefixes for translated local constants and variables in TPTP, respectively.[11] The KB is translated sentence by sentence using $\tau_{psoa}(\cdot)$, while for the query we use a preserved answer predicate `ans` to show the bindings of variables. More explanations can be found in [6].

[11] In TPTP, constants and variables start with lower case and upper case letters, respectively.

Table 2. Mapping function $\tau_{psoa}(\cdot)$ from PSOA/PS constructs to TPTP constructs

PSOA/PS Constructs	TPTP Constructs
`_C`	`lC`
`?v`	`Qv`
`o#Top(`$\mathtt{t}_1 \ldots \mathtt{t}_k$`)`	`tupterm(`$\tau_{psoa}(\mathtt{o}), \tau_{psoa}(\mathtt{t}_1) \ \ldots \ \tau_{psoa}(\mathtt{t}_k)$`)`
`o#Top(p->v)`	`sloterm(`$\tau_{psoa}(\mathtt{o}), \tau_{psoa}(\mathtt{p}), \tau_{psoa}(\mathtt{v})$`)`
`o#f()`	`member(`$\tau_{psoa}(\mathtt{o}), \tau_{psoa}(\mathtt{f})$`)`
`a##b`	`subclass(`$\tau_{psoa}(\mathtt{a}), \tau_{psoa}(\mathtt{b})$`)`
`a=b`	$\tau_{psoa}(\mathtt{a}) = \tau_{psoa}(\mathtt{b})$
`And(`$\mathtt{f}_1 \ \ldots \ \mathtt{f}_n$`)`	`(`$\tau_{psoa}(\mathtt{f}_1)$ `&` $\ldots$ `&` $\tau_{psoa}(\mathtt{f}_n)$`)`
`Or(`$\mathtt{f}_1 \ \ldots \ \mathtt{f}_n$`)`	`(`$\tau_{psoa}(\mathtt{f}_1)$ `|` $\ldots$ `|` $\tau_{psoa}(\mathtt{f}_n)$`)`
`Exists ?`$\mathtt{v}_1 \ldots ?\mathtt{v}_n$ `f`	`? [`$\tau_{psoa}(?\mathtt{v}_1) \ldots \tau_{psoa}(?\mathtt{v}_n)$`] :` $\tau_{psoa}(\mathtt{f})$
`Forall ?`$\mathtt{v}_1 \ldots ?\mathtt{v}_n$ `f`	`! [`$\tau_{psoa}(?\mathtt{v}_1) \ldots \tau_{psoa}(?\mathtt{v}_n)$`] :` $\tau_{psoa}(\mathtt{f})$
φ `:-` ψ	$\tau_{psoa}(\psi)$ `=>` $\tau_{psoa}(\varphi)$

4.2 RESTful Web API

Both the translation and execution operations are exposed as RESTful Web services as shown in Figure 1. This was accomplished by creating two REST resources: *Translate*, a REST resource for representing the PSOA2TPTP translator; *Execute*, a REST resource for representing a reasoner (VampirePrime). [12] Currently POST is the only HTTP operation supported by these resources along with application/json (JSON encoding) and text/plain (plain text) Internet media types.

To translate a PSOA/PS document into a TPTP document, the PSOA/PS document must be JSON-encoded and sent, in an HTTP POST request, to the *Translate* URI; the response is the result of the PSOA2TPTP translator encoded as a JSON array of TPTP-FOF sentences. See [7] for details.

The *Execute* Web service allows an application programmer to execute a reasoner; the reasoner we use is the VampirePrime reasoner, which accepts TPTP-FOF sentences as input. Therefore, to query an input knowledge base using PSOA/PS the application programmer must first request translation and then send the resulting TPTP-FOF sentences in an HTTP POST request to the *E*xecute URI. The result will be the plain text output from the reasoner (see Listings 2-5 in [7]) and the example in the next section.

5 Examples

In this section we demonstrate some examples showing how input knowledge bases (KBs) and queries are translated into TPTP-FOF and executed by VampirePrime to get the query results. We start with a simple example with only ground facts in the KB, followed by an advanced one with rules.

[12] Note that the designation of *resource* is not in and of itself a Web service, which requires the combination of the resource URI, an HTTP operation and an Internet media type.

5.1 Example 1

– **Input KB:**

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

– **Translated KB:**

```
fof( ax01, axiom,
     member(lf1, lfamily) & tupterm(lf1, lMike, lAmy)
     & sloterm(lf1, lchild, lFred) & sloterm(lf1, lchild, lJane)).
fof( ax02, axiom,
     member(lAmy, lperson) & tupterm(lAmy, lbcs, lmcs, lphd)
     & tupterm(lAmy, lmarried) & sloterm(lAmy, ljob, lengineer)).
```

The KB has two psoa formulas as facts. The first fact has one tuple for the family's adults, where `_Mike _Amy` is equivalent to `[_Mike _Amy]`, 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 are first broken into two conjunctions of elementary constructs, and then mapped to two TPTP conjunctions according to the function $\tau_{psoa}(\cdot)$ defined in the last section.

– **Query 1.1:** `_Amy#_person(_job->_engineer)`

– **Translated Query:**

```
fof( query, theorem,
     ((member(lAmy, lperson) & sloterm(lAmy, ljob, lengineer))
       => ans )).
```

– **VampirePrime Output:**

```
Proof found.
...
... | «ans» ...
```

The translated query is combined with the translated KB into a document and executed by VampirePrime. In the output, `«ans»` indicates that the queried fact is true. Note that this query is a ground fact, so that the task here is to prove the fact rather than asking for variable bindings, which we will show next.

– **Query 1.2:** `_Amy#_person(_job->?Job)`

– **Translated Query:**

```
fof( query, theorem,
     ((member(lAmy, lperson) & sloterm(lAmy, ljob, QJob))
       => ans("?Job", QJob) )).
```

– **VampirePrime Output:**

```
Proof found.
...
... | «ans»("?Job = ",lengineer) ...
```

This query asks for the job of `_Amy`, and the answer `«ans»("?Job = ",lengineer)` means `?Job` can unify with `_engineer`.

5.2 Example 2

– **Input KB:**

```
Document(
  Group (
    Forall ?X ?Y ?Z (
        ?X#_person(_descendent->?Z) :-
        And(?X#_person(_child->?Y) ?Y#_person(_descendent->?Z))
    )
    Forall ?X ?Y (
        ?X#_person(_descendent->?Y) :- ?X#_person(_child->?Y)
    )
    _Tom#_person(_child->_Amy _job->_professor)
    _Eva#_person(_child->_Amy)
    _Amy#_person([_married] [_bcs _mcs _phd] _child->_Fred)
    _Fred#_person(_school->_UNB)
  )
)
```

– **Translated KB:**

```
fof(ax01,axiom,(
    ! [QZ,QY,QX] :
      ( ( member(QX,lperson) & sloterm(QX,lchild,QY)
        & member(QY,lperson) & sloterm(QY,ldescendent,QZ))
      => ( member(QX,lperson) & sloterm(QX,ldescendent,QZ) ) ) )).
fof(ax02,axiom,(
    ! [QY,QX] :
      ( ( member(QX,lperson) & sloterm(QX,lchild,QY) )
      => ( member(QX,lperson) & sloterm(QX,ldescendent,QY) ) ) )).
fof(ax03,axiom,
      ( member(lTom,lperson) & sloterm(lTom,lchild,lAmy)
        & sloterm(lTom,ljob,lprofessor) )).
fof(ax04,axiom,
      ( member(lEva,lperson) & sloterm(lEva,lchild,lAmy) )).
fof(ax05, axiom,
      ( member(lAmy, lperson) & tupterm(lAmy, lbcs, lmcs, lphd)
      & tupterm(lAmy, lmarried) & sloterm(lAmy,lchild,lFred) )).
fof(ax06,axiom,
      ( member(lFred,lperson) & sloterm(lFred,lschool,lUNB) )).
```

The KB has two rules and four facts. The facts shows the information of `_Tom`, `_Eva`, `_Amy`, `_Fred`. The rules define the descendent relationship.

– **Query 2.1:** `?Ancestor # _person(_descendent->?Who)`

– **Translated Query:**

```
fof(query,theorem,(
    ! [QWho,QAncestor] :
      ( sloterm(QAncestor,ldescendent,QY)
      => ans("?Ancestor = ",QAncestor,"?Y = ",QWho) ) )).
```

– **VampirePrime Output:**

```
Proof found.
...
... | «ans»("?Ancestor = ",lAmy,"?Who = ",lFred) ...
...
... | «ans»("?Ancestor = ",lEva,"?Who = ",lAmy) ...
...
...
...
... | «ans»("?Ancestor = ",lEva,"?Who = ",lFred) ...
...
```

The query asks for all the descendent pairs `<?Ancestor, ?Who>` in the KB, and the output `«ans»("?Who = ",lMike)` and `«ans»("?Who = ",lTom)` from VampirePrime means gives all the unifications.

– **Query 2.2:**

```
And (?Ancestor1 # _person(_descendent->_Fred)
    ?Ancestor2 # _person(_descendent->_Fred))
```

– **Translated Query:**

```
fof(query,theorem,(
     ! [QAncestor2,QAncestor1] :
      ( ( member(QAncestor1,lperson)
      & sloterm(QAncestor1,ldescendent,lFred)
      & member(QAncestor2,lperson)
      & sloterm(QAncestor2,ldescendent,lFred) )
      => ans("?Ancestor1 = ",QAncestor1,
        "?Ancestor2 = ",QAncestor2) ) )).
```

– **VampirePrime Output:** `Proof found.`

```
...
... | «ans»("?Ancestor1 = ",lAmy,"?Ancestor2 = ",lAmy) ...
...
... | «ans»("?Ancestor1 = ",lAmy,"?Ancestor2 = ",lEva) ...
...
...
... | «ans»("?Ancestor1 = ",lTom,"?Ancestor2 = ",lEva) ...
...
```

– **Query 2.3:**

```
And (?Ancestor1 # _person(_descendent->?Who)
    ?Ancestor2 # _person(_descendent->?Who))
```

– **Translated Query:**

```
fof( query,theorem,(
       ! [QAncestor2,QWho,QAncestor1] :
       ( ( member(QAncestor1,lperson)
       & sloterm(QAncestor1,ldescendent,QWho)
       & member(QAncestor2,lperson)
       & sloterm(QAncestor2,ldescendent,QWho) )
       => ans("?Ancestor1 = ",QAncestor1,
             "?Who = ",QWho,"?Ancestor2 = ",QAncestor2) ) )).
```

– **VampirePrime Output:** `Proof found.`

```
...
... | «ans»("?Ancestor1 = ",lAmy,"?Who = ",lFred,"?Ancestor2 = ",lAmy)
...
......
... | «ans»("?Ancestor1 = ",lTom,"?Who = ",lAmy,"?Ancestor2 = ",lEva)
...
......
... | «ans»("?Ancestor1 = ",lTom,"?Who = ",lFred,"?Ancestor2 = ",lEva)
...
...
```

6 Conclusions and Future Work

PSOATransRun is the first implementation of PSOA RuleML. It translates a PSOA/PS knowledge base and queries into semantically equivalent TPTP documents, and then executes them through the VampirePrime reasoner to obtain the query results. Future work on the project includes: (1) Extend the capability of PSOATransRun to support all PSOA RuleML constructs; (2) build a complete benchmark suite for testing PSOA RuleML reasoners; (3) deploy PSOATransRun in real applications, e.g. the Clinical Intelligence use case [8], where PSOA rules are used to define semantic mappings for a hospital data warehouse.

The wiki page on PSOA RuleML[13] documents the ongoing development of PSOATransRun, gives further examples, and links to the online system. Users of PSOATransRun are encouraged to send their email feedback to the authors.

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

[13] http://wiki.ruleml.org/index.php/PSOA_RuleML

2. Riazanov, A., Voronkov, A.: The Design and Implementation of Vampire. AI Communications **15**(2-3) (2002) 91–110
3. Riazanov, A., Aragao, M.A.: Incremental Query Rewriting with Resolution. Canadian Semantic Web II (2010)
4. Fielding, R.T.: Architectural Styles and the Design of Network-based Software Architectures. PhD thesis, University of California, Irvine (2000)
5. DuVander, A.: New Job Requirement: Experience Building RESTful APIs. `http://blog.programmableweb.com/2010/06/09/new-job-requirement-experience-building-restful-apis/` (July 2010)
6. Zou, G., Peter-Paul, R., Boley, H., Riazanov, A.: PSOA2TPTP: A Reference Translator for Interoperating PSOA RuleML with TPTP Reasoners. In Bikakis, A., Giurca, A., eds.: RuleML 2012. LNCS, Springer, Heidelberg (2012) 264–279
7. 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`.
8. 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