PSOA RuleML API: A Tool for Processing Abstract and Concrete Syntaxes

Mohammad Sadnan Al Manir[1], Alexandre Riazanov[1], Harold Boley[2] and Christopher J.O. Baker[1]

[1] Department of Computer Science and Applied Statistics
University of New Brunswick, Saint John, Canada
{sadnan.almanir,bakerc}[at]unb.ca
alexandre.riazanov[at]gmail.com

[2] Information and Communications Technologies
National Research Council Canada
harold.boley[at]nrc.gc.ca

Abstract. PSOA RuleML is a rule language which introduces positional-slotted, object-applicative terms in generalized rules, permitting relation applications with optional object identifiers and positional or slotted arguments. This paper describes an open-source PSOA RuleML API, whose functionality facilitates factory-based syntactic object creation and manipulation. The API parses an XML-based concrete syntax of PSOA RuleML, creates abstract syntax objects, and uses these objects for translation into a RIF-like presentation syntax. The availability of such an API will benefit PSOA rule-based research and applications.

1 Introduction

F-logic [1] and W3C RIF [2] define objects (frames) separately from functions and predicates. POSL and PSOA RuleML [3, 4] provide an integration of object identifiers with applications of functions or predicates to positional or slotted arguments, called **p**ositional-**s**lotted, **o**bject-**a**pplicative (psoa) term. While RIF requires different kinds of terms for positional and slotted information as well as for frames and class memberships, PSOA RuleML can express them with a single kind of psoa term. As a result, PSOA rules permit a compact way of authoring rule bases, which are as expressive as POSL and semantically defined in the style of RIF-BLD. The constructs of PSOA RuleML are described in [3] in detail. In this paper, 'psoa' in lower-case letters refers to a kind of terms while 'PSOA' in upper-case letters refers to the language.

Here we describe an open-source PSOA RuleML API. The inspiration comes from well-known APIs for Semantic Web languages such as the OWL API [5] and the Jena API [6]. The existence of these APIs facilitates a lot of experimental research and development in Semantic Web technologies and we hope that our API will have a similar effect on the PSOA adoption. Our API allows creation of objects corresponding to PSOA constructs, such as constants, variables, tuples, slots, atoms, formulas, rules, etc., using factory-based method calls, as well as traversal of those objects using simple recursive traversal. Moreover, it supports parsing of XML-based PSOA documents and generation of presentation syntax. Thus, users will be able to employ this API for rule processing, including rule authoring, rule translation into other languages, rule-based applications, and rule engines.

Due to space constraints, here we briefly describe the key features of the PSOA RuleML presentation syntax. The language is best described using conditions and rules built over various terms, centered around psoa terms in particular.

We begin with the disjoint sets of alphabets of the language. The alphabets include a countably infinite set of constant and variable symbols, connective symbols (e.g., `And,Or,:-`), quantifiers (e.g., `Exists,Forall`), and other auxiliary symbols (e.g., `=,#,##,->,External,Group,(,),<,>,^^,_`).

The language contains literal constants and IRI constants, the latter sometimes abbreviated as short constants.

The following examples illustrate double-type and string-type literal constants:

```
"27.98"^^xs:double        "The New York Times"^^xs:string
```

Constants like `family`, `kid` are short constants.

Each variable name is preceded by a '`?`' sign, such as `?1,?Hu,?Wi`, etc.

In a psoa term, the function or predicate symbol is instantiated by an object identifier (OID) and applied to zero or more positional or named arguments. The positional arguments are referred to as tuples while named arguments (attribute-value pairs) are called slots.

For example, a psoa term (an atom), containing `family` relation with the OID `?inst`, tuples husband `?Hu`, and wife `?Wi`, along with a slot `child->?Ch` can be represented as follows:

```
?inst#family(?Hu ?Wi child->?Ch)
```

Terms include psoa terms as well as several different types of logic terms, such as constants and variables, equality, subclass, and external terms.

An atomic formula with `f` as the predicate is defined as `f(...)` in general. PSOA applies a syntactic transformation to incorporate the OID, which results in the objectified atomic formula `o#f(...)`, with `o` as the OID and `f` acting as its class. The OID is represented by a stand-alone '`_`' for a ground (variable-free) fact, an existentially-scoped variable for a non-ground fact or an atomic formula in a rule conclusion, and a stand-alone '`?`' as an anonymous variable for any other atomic formula.

Condition formulas are used as queries or rule premises. Conjunction and disjunction of formulas are denoted by `And` and `Or`, respectively. Formulas with existentially quantified variables are also condition formulas. An example of a condition formula is given below:

```
And(?2#married(?Hu ?Wi) Or(?3#kid(?Hu ?Ch) ?4#kid(?Wi ?Ch)))
```

Aside from the condition formulas, the premise can also contain atomic formulas and external formulas.

A conclusion contains a head or conjunction of heads. A head refers to an atomic formula which can also be existentially quantified. A conclusion example is given below:

```
Exists ?1 (?1#family(husb->?Hu wife->?Wi child->?Ch))
```

An implication contains both conclusion and condition formulas. A clause is either an atomic formula or an implication. A rule is generated by a clause within the scope of the `Forall` quantifier or solely by a clause. Several formulas can be collected into a `Group` formula.

The `Group` formula below contains a universally quantified formula, along with two facts. The `Forall` quantifier declares the original universal argument variables as well as the generated universal OID variables ?2, ?3, ?4. The infix `:-` separates the conclusion from the premise, which derives the existential `family` frame from a `married` relation `And` from a `kid` of the `husb` `Or` `wife`. The following example from [3] shows an objectified form on the right.

```
Group (
 Forall ?Hu ?Wi ?Ch (

  family(husb->?Hu wife->?Wi child->?Ch) :-
   And(married(?Hu ?Wi)
       Or(kid(?Hu ?Ch) kid(?Wi ?Ch))) )
 married(Joe Sue)
 kid(Sue Pete)
      )
```

```
Group (
 Forall ?Hu ?Wi ?Ch ?2 ?3 ?4 (
  Exists ?1 (
   ?1#family(husb->?Hu wife->?Wi child->?Ch)) :-
    And(?2#married(?Hu ?Wi)
        Or(?3#kid(?Hu ?Ch) ?4#kid(?Wi ?Ch))) )
 _1#married(Joe Sue)
 _2#kid(Sue Pete)
      )
```

The objectified `family` term in the rule conclusion is slotted with 3 slots:

```
?1#family(husb->?Hu wife->?Wi child->?Ch)
```

The rules's condition formulas use the relations `married` and `kid`, containing only 2-tuples `Hu`, `Wi`, and `Ch`:

```
?2#married(?Hu ?Wi) ?3#kid(?Hu ?Ch) ?4#kid(?Wi ?Ch)
```

The next section will describe the API components and their uses. We begin by describing the organization of the package, then illustrate the object creation and traversal as well as parsing the PSOA/XML input, and rendering in presentation syntax. For all of these operations, we use the objectified `family` example above. Finally, we conclude by mentioning the scope of using our API with other complementary tools and potential work directions in the future.

2 The API Structure and Functionality

2.1 Package Organization

The API is divided into two main components: one is for the creation and traversal of abstract syntax objects and the other is for parsing and rendering of those objects.

The *AbstractSyntax* is the top level class for factories and contains all Java interfaces for different types of abstract syntax objects. A simple implementation of *AbstractSyntax* interfaces is in the *DefaultAbstractSyntax* class, which is suitable for most purposes. However, more demanding uses may require custom implementations of the interfaces.

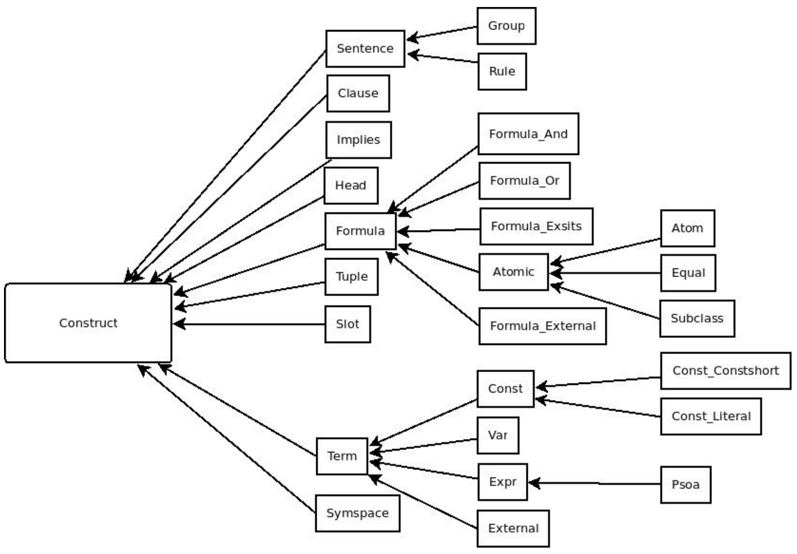

Fig. 1. The API Structure

The package also contains a *Parser* class, which provides PSOA/XML parsing and translation into presentation syntax. Parsing is implemented using the Java Architecture for XML Binding (JAXB) [7], which creates equivalent Java objects based on XML schema files. The schema is a straight-forward encoding of the syntactic construct hierarchy and is available in [8]. The parsed XML is then converted to the abstract syntax by calling factory methods.

Figure 1 presents the most important classes and interfaces implementing different types of syntactic constructs. Each of the names in a rectangular box represents a Java interface, and is kept as close as possible to the presentation syntax construct names. The corresponding implementations of these interfaces use Java inheritance, shown by the solid arrows.

The interface `Construct` sits at the top of the hierarchy. `Group` can be populated with more `Group`s and `Rule`s. Both universal facts and universal rules are represented by the `Rule` class, which encapsulates a `Clause`. The interface `Clause` represents either an implication or an atomic formula. Implication is represented by the interface `Implies` whereas `Atomic` represents atomic formulas. The generalized interface `Atomic` is implemented either by `Atom` (representing a psoa term like `?2#married(?Hu ?Wi)`) or by `Subclass` (representing a subclass term like `student##person`) or by `Equal` (equality term like `?cost = ?spent`).

Implementations of the generalized `Formula` interface represent either disjunction, conjunction, or existential formulas by implementing `Formula_Or` (represented as `Or(?3#kid(?Hu ?Ch) ?4#kid(?Wi ?Ch))`, `Formula_And` (`And` instead of `Or`), and `Formula_Exists`, respectively. In addition to atomic formulas, external formulas (`External(func:numeric-add(?cost1 ?cost2))`) can be represented using the `Formula_External` interface. The generalized interface `Term` is represented by implementing `Const` for different kinds of constants, `Var` for variables like `?Hu`, `?Wi`, `Expr` for expressions denoting psoa terms, and `External` for external `Expressions`. Constants can be either `Const_Literal` (e.g., `"47.5"^^xs:float`, with `Symspace` referring to `xs:float`) or `Const_Constshort` (e.g., `family,kid`). Finally, the interface `Psoa` is implemented to represent objectified functions or predicates with membership symbol '`#`' and tuples and slots as arguments, e.g., `inst#family(Homer Merge child->Bart)`. The internal nodes `Sentence`, `Formula`, `Atomic`, `Term`, `Const`, and `Expr`, are generalized classes and implemented by more specific classes.

2.2 Construction of Abstract Syntax Objects

The abstract syntax objects are constructed by factory-based ***createX*** methods calls, ***X*** being the object type name. The rest of this paper represents each method in ***emphasized*** font. A factory can be created as follows:

```
DefaultAbstractSyntax absSynFactory = new DefaultAbstractSyntax()
```

We are going to illustrate the creation of facts and rules below.

Construction of Facts A fact is of type `Atomic`. Let us look at the first fact that tells us `Joe` and `Sue` are `married` to each other with the OID _1, whereas the second fact says `Pete` is the `kid` of `Sue` with the OID _2, each fact referring to a psoa term.

The creation of fact `_1#married(Joe Sue)` starts by creating the four constants `_1`, `married`, `Joe`, `Sue` as `const_1`, `const_married`, `const_Joe`, `const_Sue`, respectively using the method ***createConst_Constshort***.

```
Const_Constshort const_1 = absSynFactory.createConst_Constshort("_1")
Const_Constshort const_married = absSynFactory
                                        .createConst_Constshort("married")
Const_Constshort const_Joe = absSynFactory
                                        .createConst_Constshort("Joe")
Const_Constshort const_Sue = absSynFactory
                                        .createConst_Constshort("Sue")
```

Tuples `const_Joe` and `const_Sue` are constructed by the method ***createTuple***. The list of such tuples is referred to as a `tuplesList`.

```
Tuple tuples = absSynFactory.createTuple(tuplesList)
```

Method ***createPsoa*** assembles `_1`, `married` and `tuples` into a `psoaTerm`, while ***null*** indicates the absence of slots.

```
Psoa psoaTerm = absSynFactory
                        .createPsoa(const_1, const_married, tuples, null)
```

Here is how we create an atom:

```
Atom atom = absSynFactory.createAtom(psoaTerm)
```

Thus, we use the method ***createAtom*** for creating a fact of type `Atom`, ***createEqual*** for a fact of type `Equal`, and ***createSubclass*** for type `Subclass`. This creation is completed by the ***createClause*** and ***createRule*** method calls. The representation for creating the fact `_2#kid(Sue Pete)` is similar to the method calls described above, hence omitted.

Construction of Rules A rule contains condition and conclusion. We will start with the condition formula, which is a conjunction of the atomic formula `?2#married(?Hu ?Wi)` and disjunction of two atomic formulas, `?3#kid(?Hu ?Ch)` and `?4#kid(?Wi ?Ch)`.

```
Forall ?Hu ?Wi ?Ch ?2 ?3 ?4 (
   Exists ?1 (
           ?1#family(husb->?Hu wife->?Wi child->?Ch)) :-
                          And(?2#married(?Hu ?Wi) Or(?3#kid(?Hu ?Ch) ?4#kid(?Wi ?Ch)))
   )
)
```

The following code snippet creates the disjunction of two atoms. Method ***createFormula_Or*** defines the disjunction of two atomic formulas, `atomOr_1` and `atomOr_2`. The OIDs `?3` and `?4` (`var_4`), as well as tuples `?Hu` (`var_Hu`), `?Wi` (`var_Wi`) and `?Ch` (`var_Ch`), are variables. Only the construction of `?3` (`var_3`) is shown below to avoid repetition (...).

```
Var var_3 = absSynFactory.createVar("3")
...
Tuple tuples = absSynFactory.createTuple(tuplesList_1)
Psoa psoaTerm_1 = absSynFactory
                       .createPsoa(var_3, const_kid, tuples, null)
Atom atomOr_1 = absSynFactory.createAtom(psoaTerm_1)
...
Atom atomOr_2 = absSynFactory.createAtom(psoaTerm_2)
```

Both of the atomic formulas `atomOr_1` and `atomOr_2` are in a list called `formulaOrList`.

```
Formula_Or formula_Or = absSynFactory.createFormula_Or(formulaOrList)
```

The conjunction of the newly created `formula_Or` and another atomic formula `atom_And`, `?2#married(?Hu ?Wi)` is described next. Here `var_2`, `var_Hu` and `var_Wi` denote the variables `?2`, `?Hu` and `?Wi`, respectively. The code below does this using the method ***createFormula_And***. The list `formulaAndList` contains atomic formulas `atom_And` and `formula_Or`. The conjunction formula `formula_And` is the rule premise and created as follows:

```
Formula_And formula_And = absSynFactory
                               .createFormula_And(formulaAndList)
```

We now move on to the rule `Head` creation, which is a psoa term containing the OID `?1` with `family` class name and three slots, `husb->?Hu`, `wife->?Wi`, and `child->?Ch`, in `?1#family(husb->?Hu wife->?Wi child->?Ch)` as arguments. These slots will be called `slot_1`, `slot_2`, and `slot_3`, respectively.

```
Var var_1 = absSynFactory.createVar("1")
...
Slot slot_1 = absSynFactory.createSlot(const_husb, var_Hu)
Slot slot_2 = absSynFactory.createSlot(const_wife, var_Wi
Slot slot_3 = absSynFactory.createSlot(const_child, var_Ch)
```

The list of slots `slot_1`, `slot_2` and `slot_3`, called `slotsList`, is used to create the `psoa` term.

```
Psoa psoa = absSynFactory.createPsoa(var_1, const_family, null,slotsList)
```

The atom created is `atom_head`, and thus the rule head is created by the method ***createHead***, where the variable `var_1` is existentially quantified. Both existentially and universally quantified variables are treated as a list of variables, called `varsList`.

```
Head rule_head = absSynFactory.createHead(varsList, atom_head)
```

The method ***createImplies*** combines the rule head, `rule_head` and the rule premise, `formula_And`, into an `implication`. Method ***createClause*** creates the `implication`. Finally, method ***createRule*** collects all the universally quantified variables, `var_Hu`, `var_Wi`, `var_Ch`, `var_2`, `var_3`, and `var_4` into a `varsList` and creates the `rule` with the `clause`.

```
Implies implication = absSynFactory.createImplies(rule_head,formula_And)
Clause clause = absSynFactory.createClause(implication)
Rule rule = absSynFactory.createRule(varsList, clause)
```

2.3 Abstract Syntax Structure Traversal

Our implementation recursively traverses the object tree generated from the abstract syntax structure and is usually simpler than writing visit methods [9] as used in OWL API.

All components of the abstract syntax structure can be accessed directly by the corresponding accessor methods, which are ***getX*** methods. The generalized classes (see Figure 1) are `Sentence`, `Formula`, `Atomic`, `Term`, `Const`, and `Expr`, containing ***isX*** methods to recognize the specific instance types. Alternatively, specific classes of particular instances have to be identified, e.g., by using `instanceof`.

For an atomic formula, an ***isX*** method in `Atomic` class needs to recognize if the instance is of type `Atom`, `Subclass`, or `Equal` object. This principle applies to each of the generalized classes.

For example, ***isEqual*** method in generalized class `Atomic` recognizes the instance of Equality atom `?cost = "47.5"^^xs:float`. Immediately, a cast is made as the instance type `Equal` and ***getLeft*** and ***getRight*** methods are called, each referring to an instance of another generalized class `Term`. Class `Term` contains appropriate ***isX*** methods, which use similar techniques to find out if the instance is of type `Const`, or `Var`, or an `External` expression.

```
assert this instanceof AbstractSyntax.Equal
return (AbstractSyntax.Equal) this
...
AbstractSyntax.Term getLeft()
AbstractSyntax.Term getRight()
```

Method ***getLeft***, in this case, retrieves the instance of `Var` and thus string variable `?cost` is retrieved by the method ***getName*** as the variable instance. On the other hand, ***getRight*** refers to the instance of type `Const`. Method ***isConstLiteral*** recognizes `Const_Literal` involving the literal and the type float involving the instance of type `Symspace`. The literal object `47.5` is retrieved as string by the method ***getLiteral***. Finally, `xs:float` object is retrieved by the method ***getValue*** as an instance of type `Symspace`.

Thus, the traversal of objects in the API structure follows the same strategy of going down to most specific instances in a recursive manner for both facts and rules.

2.4 Parsing and Rendering

Aside from creating and traversing objects, the API is able to parse PSOA/XML inputs and render them in human readable presentation syntax.

In section 2.1 we discuss an XML schema for PSOA RuleML. We generate the XML parser with the help of JAXB, which creates Java classes from a schema traversal, where the ultimate output of the parser is abstract syntax objects.

The following example shows a transformation of an XML input for a fact and its rendering in presentation syntax.

```
<Atom>                                            inst1#family(Joe Sue Child->Pete)
  <Member>
    <instance>
      <Const type="\&psoa;iri">inst1</Const>
    </instance>
    <class>
      <Const type="\&psoa;iri">family</Const>
    </class>
  </Member>
  <tuple>
    <Const type="\&psoa;iri">Joe</Const>
    <Const type="\&psoa;iri">Sue</Const>
  </tuple>
  <slot>
    <Const type="\&psoa;iri">Child</Const>
    <Const type="\&psoa;iri">Pete</Const>
  </slot>
</Atom>
```

A ***toString*** method in each class implements this pretty-printing, which follows the same traversal procedure described in section 2.3.

3 Conclusion and Future Work

The API is open-source and hosted in [10]. The companion effort PSOA2TPTP [11] has developed a reference translator for PSOA RuleML, which facilitates inferencing using TPTP reasoners (see e.g., [12]). One component of the translator is a parser for the presentation syntax. Our API will greatly benefit from including this presentation syntax parser. The other component of the PSOA2TPTP translator is its mapping from abstract syntax objects to TPTP. Combined with our API, this will also make PSOA/XML executable on the TPTP-aware VampirePrime [13] reasoner.

Currently, the API can render PSOA/XML only into presentation syntax. As an extension, we plan to also include the translation of abstract syntax objects back to PSOA/XML.

We have been using the API in our HAIKU work [14], where PSOA is used to capture semantic modeling of relational data and needed, at least, to support authoring, including syntactic and, to some extent, logical validation (consistency checking). We also plan to use it for automatic generation of Semantic Web services from declarative descriptions.

PSOA RuleML API has become an input to the Object Management Group's API4KB effort [15], which tries to create a universal API for knowledge bases that among other things combines the querying of RDF-style resource descriptions, ODM/OWL2-style ontologies, and RIF RuleML-style rules.

References

1. Michael Kifer, Georg Lausen, and James Wu. Logical Foundations of Object-Oriented and Frame-Based Languages. *J. ACM*, 42(4):741–843, July 1995.
2. Harold Boley and Michael Kifer. A Guide to the Basic Logic Dialect for Rule Interchange on the Web. *IEEE Trans. Knowl. Data Eng.*, 22(11):1593–1608, 2010.
3. Harold Boley. A RIF-Style Semantics for RuleML-Integrated Positional-Slotted, Object-Applicative Rules. In Nick Bassiliades, Guido Governatori, and Adrian Paschke, editors, *RuleML Europe*, volume 6826 of *Lecture Notes in Computer Science*, pages 194–211. Springer, 2011.
4. Harold Boley. Integrating Positional and Slotted Knowledge on the Semantic Web. *Journal of Emerging Technologies in Web Intelligence*, 4(2):343–353, November 2010. Academy Publisher, Oulu, Finland, http://ojs.academypublisher.com/index.php/jetwi/article/view/0204343353.
5. Matthew Horridge and Sean Bechhofer. The OWL API: A Java API for OWL Ontologies. *Semantic Web*, 2(1):11–21, 2011.
6. Jeremy J. Carroll, Ian Dickinson, Chris Dollin, Dave Reynolds, Andy Seaborne, and Kevin Wilkinson. Jena: Implementing the Semantic Web Recommendations. Technical Report HPL-2003-146, Hewlett Packard Laboratories, 2003.
7. Joe Fialli and Sekhar Vajjhala. Java Architecture for XML Binding (JAXB) 2.0. Java Specification Request (JSR) 222, October 2005.
8. `http://code.google.com/p/psoa-ruleml-api/source/browse/trunk/PSOARuleML-API/src/main/resources/`.
9. Erich Gamma, Richard Helm, Ralph E. Johnson, and John M. Vlissides. Design Patterns: Abstraction and Reuse of Object-Oriented Design. In *Proceedings of the 7th European Conference on Object-Oriented Programming*, ECOOP '93, pages 406–431, London, UK, UK, 1993. Springer-Verlag.
10. PSOA RuleML API: A Tool for Processing Abstract and Concrete Syntaxes. `http://code.google.com/p/psoa-ruleml-api/`, 2012.
11. Gen Zou, Reuben Peter-Paul, Harold Boley, and Alexandre Riazanov. PSOA2TPTP: A Reference Translator for Interoperating PSOA RuleML with TPTP Reasoners. 2012. In these proceedings.
12. System on TPTP. `http://www.cs.miami.edu/~tptp/cgi-bin/SystemOnTPTP`.
13. VampirePrime Reasoner. `http://riazanov.webs.com/software.htm`.
14. Alexandre Riazanov, Gregory W. Rose, Artjom Klein, Alan J. Forster, Christopher J. O. Baker, Arash Shaban-Nejad, and David L. Buckeridge. Towards Clinical Intelligence with SADI Semantic Web Services: A Case Study with Hospital-Acquired Infections Data. In Adrian Paschke, Albert Burger, Paolo Romano 0001, M. Scott Marshall, and Andrea Splendiani, editors, *SWAT4LS*, pages 106–113. ACM, 2011.
15. `http://www.omgwiki.org/API4KB/lib/exe/fetch.php?media=api4kb:rfp.pdf`.