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

Mohammad Sadnan Al Manir$^1$ Alexandre Riazanov$^1$
Harold Boley$^2$ Christopher J.O. Baker$^1$

$^1$Department of Computer Science and Applied Statistics
University of New Brunswick, Saint John, Canada

$^2$Information and Communications Technologies
National Research Council Canada

The 6th International Symposium on Rules: Research Based and Industry Focused (RuleML-2012)
Outline

1. Introduction
   - PSOA RuleML and the API
   - Motivation

2. The API Structure and Functionality
   - Package Organization
   - Construction of Abstract Syntax Objects
   - Abstract Syntax Structure Traversal
   - Parsing and Rendering

3. Conclusion and Future Work
Outline

1. Introduction
   - PSOA RuleML and the API
   - Motivation

2. The API Structure and Functionality
   - Package Organization
   - Construction of Abstract Syntax Objects
   - Abstract Syntax Structure Traversal
   - Parsing and Rendering

3. Conclusion and Future Work
PSOA RuleML

- An object-relational Web rule language
- Unlike F-logic and W3C RIF, PSOA RuleML defines
  - Objects (frames) uniformly with relations
- Permits relation applications with
  - Optional Object IDentifiers (OIDs) and
  - Positional and Slotted arguments
- Allows positional-slotted object-applicative (psoa) terms and rules
Example (Rule-defined anonymous family frame)

Group is used to collect a rule and two facts. Forall quantifier declares original universal argument variables and generated universal OID variables ?2, ?3, ?4. Infix :- separates conclusion from premises of rule, which derives anonymous/existential family frame from married relation And from kid relation of husb Or wife (the left-hand side is objectified 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 (  
  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 API at a Glance

- Open-source at
  http://code.google.com/p/psoa-ruleml-api/
- Allows factory-based Abstract Syntax Object (ASO) creation and manipulation
- Parses XML-based concrete PSOA RuleML syntax
- Translates ASOs into RIF-like Presentation Syntax (PS)
Outline

1 Introduction
   - PSOA RuleML and the API
   - Motivation

2 The API Structure and Functionality
   - Package Organization
   - Construction of Abstract Syntax Objects
   - Abstract Syntax Structure Traversal
   - Parsing and Rendering

3 Conclusion and Future Work
Motivation

- Java API facilitates creating software using PSOA
  - Enables PSOA adoption
- Creating Rule-based applications such as
  - Rule authoring
  - Rule engines
- To be used in a Clinical Intelligence project (HAIKU)
  - Semantic modeling of Relational Databases
  - Automatic generation of Semantic Web services
Outline

1 Introduction
   - PSOA RuleML and the API
   - Motivation

2 The API Structure and Functionality
   - Package Organization
   - Construction of Abstract Syntax Objects
   - Abstract Syntax Structure Traversal
   - Parsing and Rendering

3 Conclusion and Future Work
API Components

Two main components

- Classes for creating and traversal of ASOs
  - AbstractSyntax: Top level class containing interfaces for ASOs
  - DefaultAbstractSyntax: Implements these interfaces

- Classes for parsing and rendering
  - Validator: Validates input, calls rendering methods
  - Parser: Parses PSOA/XML input using Java Architecture for XML Binding (JAXB)
The API Structure

Figure: Main Components of the API Structure
Outline

1 Introduction
   - PSOA RuleML and the API
   - Motivation

2 The API Structure and Functionality
   - Package Organization
   - Construction of Abstract Syntax Objects
   - Abstract Syntax Structure Traversal
   - Parsing and Rendering

3 Conclusion and Future Work
How to Create a Factory?

Code Snippet

```java
// Initialize Validator
Validator v = new Validator();
// Initialize Parser
Parser p = new Parser();
// PSOA/XML input file to be parsed
File file = new File("family.psoa");
// Factory creation
DefaultAbstractSyntax absSynFactory = new DefaultAbstractSyntax();
// Parsing of input document using JAXB
AbstractSyntax.Document doc = p.parse(file, absSynFactory);
```
Construction of Facts

- **CreateX** methods are used for ASO construction
- A fact is of type *Atomic* (e.g., Atom, Equal, Subclass)
- Example of an Atom \(_1\text{married}(Joe\ Sue)\)
  - Joe and Sue are in a married relation with the OID \(_1\)

### Creating *Constants* \(_1\), *married*, Joe and Sue

```java
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")
```
Adding positional terms const_Joe and const_Sue in a list of tuples

```java
LinkedList<AbstractSyntax.Term> tuplesList = new LinkedList<AbstractSyntax.Term>();

// iterate through all positional terms
for (Object obj : tuple.getTERM()) {
    // check if the term is a variable
    if (obj instanceof Var) {
        tuplesList.addLast((Var) obj);
    }
    // check if the term is a constant
    else if (obj instanceof Const) {  // as a refinement of Const, the
        tuplesList.addLast((Const) obj); // short constants const_Joe and
    }                                // const_Sue are added to tuplesList
    else if (obj instanceof Expr) {  // as a refinement of Expr
        tuplesList.addLast((Expr) obj);
    }
    ...
}
```
Creating a list of tuples const_Joe and const_Sue

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

Assembling _1, married and tuples into a psoaTerm

```java
// null indicates absence of slots
Psoa psoaTerm = absSynFactory
    .createPsoa(const_1, const_married, tuples, null)
```

Creating a fact of type Atom

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

- `CreateSubclass`, `CreateEqual` methods are used for facts of type Subclass, Equality
Construction of Rules

- A Rule contains *Condition* and *Conclusion*
- The Condition of the following Rule consists of
  - 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)

\f\text{family}(husb->?Hu wife->?Wi child->?Ch)) :- \f\text{And}(?2#married(?Hu ?Wi) Or(?3#kid(?Hu ?Ch) ?4#kid(?Wi ?Ch)))
)
Construction of Rules (Cont’d)

- Creating *Condition* formula

\[
\text{And}(\#married(\text{Hu} \ \text{Wi}) \ \text{Or}(\#kid(\text{Hu} \ \text{Ch}) \ \#kid(\text{Wi} \ \text{Ch})))
\]

**Atom ?3#kid(?Hu ?Ch) as atomOr_1**

```java
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 ?4#kid(?Wi ?Ch) as atomOr_2**

```java
Atom atomOr_2 = absSynFactory.createAtom(psoaTerm_2)
```

**Disjunction of two Atoms as a formulaOrList**

```java
Formula_Or formula_Or = absSynFactory.createFormula_Or(formulaOrList)
```
Construction of Rules (Cont’d)

- Creating *Condition* formula

\[
\text{And}(?2\#\text{married}(\text{Hu} \text{ Wi}) \text{ Or}(?3\#\text{kid}(\text{Hu} \text{ Ch}) \text{ ?4}\#\text{kid}(\text{Wi} \text{ Ch})))
\]

**Atom ?2#married(?Hu ?Wi) as atom\_And**

Atom atom\_And = absSynFactory.createAtom(psoaTerm\_3)

**Conjunction of atom\_And and formula\_Or as a formulaAndList**

Formula\_And formula\_And = absSynFactory.createFormula\_And(formulaAndList)
Construction of Rules (Cont’d)

- Creating **Conclusion** formula


  psoa term from variables, constants, 3 slots as a slotsList

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

  Conclusion with existentially quantified variable ?1 as a varsListExists

  Head rule_head = absSynFactory. createHead(varsListExists, atom_head)
Construction of Rules (Cont’d)

- Creating Rule with the *Condition* and *Conclusion* formula

\[
\text{Forall } ?\text{Hu} ?\text{Wi} ?\text{Ch} ?2 ?3 ?4 ( \\
\quad \text{Exists } ?1 ( \\
\quad \quad ?1\#\text{family}(\text{husb-}\to ?\text{Hu} \text{ wife-}\to ?\text{Wi} \text{ child-}\to ?\text{Ch})) :- \\
\quad \quad \text{And}(?2\#\text{married}(?\text{Hu} ?\text{Wi}) \text{ Or}(?3\#\text{kid}(?\text{Hu} ?\text{Ch}) ?4\#\text{kid}(?\text{Wi} ?\text{Ch})))
\]

**Implication from Condition and Conclusion**

\[
\text{Implies implication = absSynFactory.} \text{createImplies}((\text{rule_head,formula}_{\text{And}})
\]

**Clause (Either an Implication or an Atomic formula)**

\[
\text{Clause clause = absSynFactory.} \text{createClause}(\text{implication})
\]

**Rule with universally quantified variables as a varsListUniv**

\[
\text{Rule rule = absSynFactory.} \text{createRule}((\text{varsListUniv, clause})
\]
Outline

1. Introduction
   - PSOA RuleML and the API
   - Motivation

2. The API Structure and Functionality
   - Package Organization
   - Construction of Abstract Syntax Objects
   - Abstract Syntax Structure Traversal
   - Parsing and Rendering

3. Conclusion and Future Work
Traversing ASOs

- Data structures representing PSOA expressions can be processed by simple recursive traversal.
- All components of the structures can be accessed by the `getX` accessor methods.
- General classes (see earlier Figure):
  - are `Sentence`, `Formula`, `Atomic`, `Term`, `Const`, and `Expr`.
  - contain `isX` methods to recognize the specific instance types.
- Specific classes of particular instances have to be identified by using `instanceof`.
- Both ways are legitimate.
- e.g., for an Atomic formula, an `isX` method in Atomic class needs to recognize if the instance is of type `Atom`, `Subclass`, or `Equal` object.
Traversal of ASOs (Cont’d)

- **isEqual** method in general class Atomic
  - recognizes the instance of Equality Atom
    - `?cost = "47.5"^^xs:float`
  - immediately, makes a cast as the instance type Equal
  - calls **getLeft** and **getRight** methods

```
Traversing Equality Atom ?cost = "47.5"^^xs:float
assert this instanceof AbstractSyntax.Equal
return (AbstractSyntax.Equal) this
```

- The methods **getLeft** (?cost) and **getRight** ("47.5"^^xs:float) refer to instances of other general class Term
  - which uses **isVar**, **isConstLiteral**, **isConstShort** to recognize variables and constants
Outline

1. Introduction
   - PSOA RuleML and the API
   - Motivation

2. The API Structure and Functionality
   - Package Organization
   - Construction of Abstract Syntax Objects
   - Abstract Syntax Structure Traversal
   - Parsing and Rendering

3. Conclusion and Future Work
Parsing and Rendering as Presentation Syntax

- Parsing: JAXB is used to generate the XML Parser
  - which creates Java classes by binding a schema
- Rendering: PSOA/XML Fact transformed into PSOA/PS

```xml
<Atom>
  <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>
```

inst1#family(Joe Sue Child->Pete)
Summary

- Inspired by the OWL API and Jena API
- The API is open-source and hosted in [1]
- Currently the API renders PSOA/XML only as PSOA/PS
- The reference translator PSOA2TPTP [2] is pursued in a companion effort
  - Interoperates PSOA RuleML with TPTP Reasoners
Future Work

- Translation of ASOs back to PSOA/XML
- Merge the API with PSOA2TPTP
- Deploy for semantic modeling of Relational Databases in a Clinical Intelligence project
  - Semantic mapping of Relational Databases to Ontologies
http://code.google.com/p/psoa-ruleml-api/.

*In RuleML-2012 proceedings*