Visualizing SWRL Rules Using Grailog Frame Formulas

Team 8

Bo Yan

Junyan Zhang,

Ismail Akbari

Instructor: Harold Boley

December 9, 2012

Table of Contents

1. Introduction

SWRL (Semantic Web Rule Language) is an implemented Semantic Web rule language, combining the sublanguages Web Ontology Language, Description logic (OWL DL) and Rule Markup Language (Unary/Binary Datalog) RuleML[1]. SWRL has become one of the most popular Web rule languages, possibly because of its use as SWRLTab in the very wide-spread Protégé ontology editor and knowledge-base framework. SWRL employs Unary/Binary Datalog formulas, which can also be seen as slotted formulas in F-logic, W3C RIF-BLD, or PSOA RuleML.

Graph inscribed logic (Grailog) is a proposed cognitively motivated systematic graph standard for visual-logic knowledge. Graphs should make it easier for humans to read and write logic constructs by exploiting a 2-dimensional representation with shorthand & normal forms, from Controlled English to advanced logics [2].

In our project, we visualize SWRL's slotted formulas in Grailog. Every SWRL rule will be converted to its Grailog slotted formula, and then the equivalent visual graph (again based on Grailog format) is generated. From the graphs, people can more easily understand the rules.

2. Background

2.1 Grailog

Although people can have a good starting sight on knowledge representation from directed labeled graphs (DLGs), they cannot straight forwardly figure out the nested structures, non-binary relationships, and relation descriptions. Such problems can be solved by applying encoded constructs with auxiliary nodes and relationships, which also need to be kept separate from straightforward constructs. Hence, various extensions of DLGs have been proposed for knowledge representation, including graph partitionings (possibly interfaced as complex nodes), n-ary relationships as directed labeled hyperarcs, and (hyper)arc labels

used as nodes of other (hyper)arcs. Meanwhile, a lot of AI / Semantic Web research and development on ontologies & rules have gone into extended logics for knowledge representation such as object (frame) logics, description logics, general modal logics, and higher-order logics. And Grailog is one of the outstanding representatives, in which knowledge representation with graphs and logics can be reconciled. It proceeds from simple to extended graphs for logics needed in AI and the Semantic Web. Along with its visual introduction, each graph construct is mapped to its corresponding symbolic logic construct. These graph-logic extensions constitute a systematics defined by orthogonal dimensions, which has led to the Grailog language as part of the Web-rule industry standard RuleML. While Grailog's DLG sublanguage corresponds to binary-associative memories, its hypergraph sublanguage corresponds to n-ary content-addressable memories, and its complex-node modules offer various further opportunities for parallel processing [2].

There are several principals of Grailog: (1) Visual representations (here graphs) is easier for humans to understand (2) Graphs should be natural extensions (e.g. n-ary) of Directed Labeled Graphs (DLGs), often used to represent simple semantic nets (3) Graphs make stepwise refinements possible for all logic constructs: Description Logic constructors, F-logic frames, general PSOA RuleML terms, etc.; (4) Extensions to boxes & links should be orthogonal.

2.2 OWL-DL

OWL DL is a W3C standard based on the description logic *SHOIN (D)* with support of data values, data types and data type properties, but since OWL is also based on RDF(S), the terminology differs slightly. A concept from DL is referred to as a class in OWL and a role from DL is referred to as a property in OWL. The DL syntax can be used for the description of OWL ontologies or knowledge bases [3]. Description Logics can overcome the ambiguities of early semantic networks and frames. OWL DL benefits a lot from many years of DL research.

OWL DL supports those users who want the maximum expressiveness while retaining computational completeness (all conclusions are guaranteed to be computable) and decidability (all computations will finish in finite time). OWL DL includes all OWL language constructs, but they can be used only under certain restrictions (for example, while a class may be a subclass of many classes,

a class cannot be an instance of another class). OWL DL is so named due to its correspondence with description logics, a field of research that has studied the logics that form the formal foundation of OWL [4].

2.3 F-Logic (Frame Logic)

F-logic (frame logic) is a knowledge representation and ontology language. F-logic combines the advantages of conceptual modeling with object-oriented, frame-based languages and offers a declarative, compact and simple syntax, as well as the well-defined semantics of a logic-based language. Features include, among others, object identity, complex objects, inheritance, polymorphism, query methods, encapsulation. F-logic stands in the same relationship to object-oriented programming as classical predicate calculus stands to relational database programming.

In contrast to description logic based ontology formalism the semantics of F-logic are normally that of a closed world assumption as opposed to DL's open world assumption. Also, F-logic is generally undecidable, whereas the SHOIN description logic that OWL DL is based on is decidable. However it is possible to represent more expressive statements in F-logic that are not possible with description logics [5].

2.4 PSOA RuleML

Knowledge representation is at the foundation of Semantic Web applications, using rule and ontology languages as the main formalisms. PSOA RuleML is a rule language that combines the ideas of relational (predicate-based) and object-oriented (frame-based) modeling. 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 class membership term in RIF-BLD. A psoa term has the following general form:

$$o\#f([t_{1,1} \ldots t_{1,n1}] \ldots [t_{m,1} \ldots t_{m,nm}]\ p_1\text{->}v_1 \ldots p_k\text{->}v_k)$$

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 type f; (2) each tupled argument [$t_{i,1}$ ... $t_{i,ni}$] represents a sequence of terms associated with o; (3) each slotted argument pi -> vi represents a pair of an attribute pi and its value vi associated with o.

A psoa term can be used as an atomic formula. Such psoa atoms can be asserted as psoa facts. They can also be combined into more complex formulas using constructors from the Horn-like subset of first-order logic: conjunction, disjunction in negative positions, existential and universal quantifiers. Implication ':-' can be used to form rules. Such psoa rules can be used to derive psoa atoms on demand, through psoa querying [6].

2.5 Unary/Binary Datalog Formulas

2.5.1 Predicates: Unary Relations (Classes, Concepts, Types)

A unary relation can be seen as a class, where the relation's single argument is an instance of the class. The example shows in figure 1 uses a class named "Billionaire", and there is an instance called "Warren Buffett" belonging to this class.

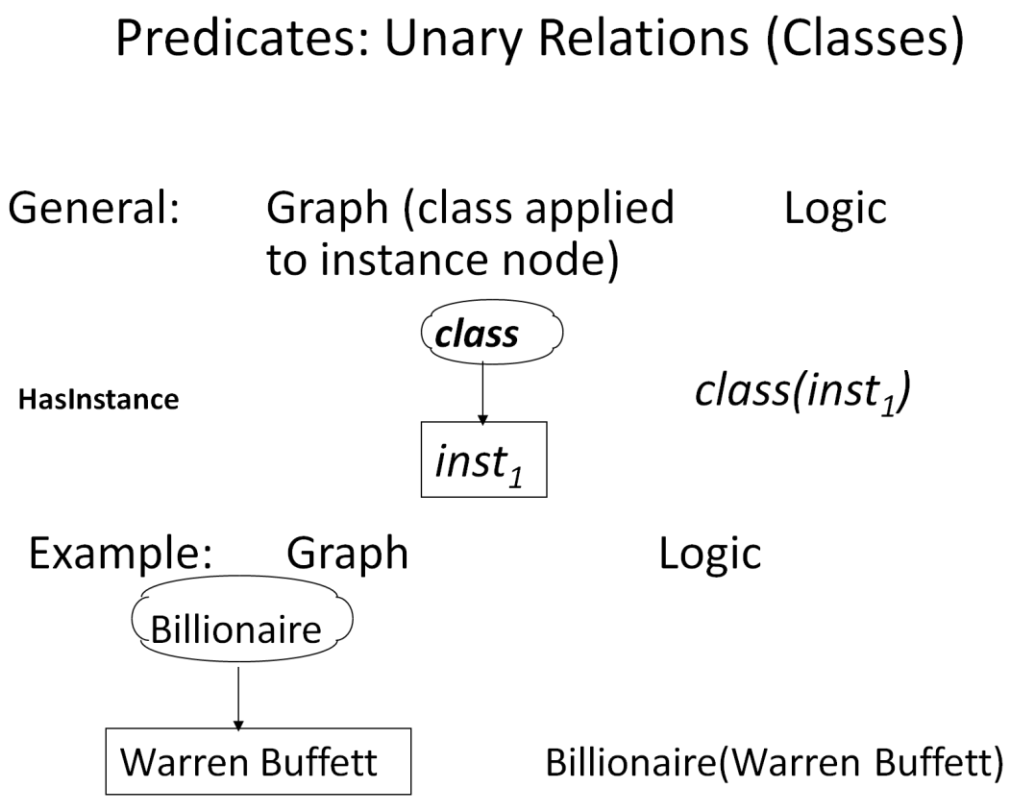

Figure1: Unary relations (adapted from [2])

2.5.2 Predicates: Binary Relations

Binary relation describes relationship between two instances. In figure 2, there is a graph and its related logic description, and below them there is an example illustrated.

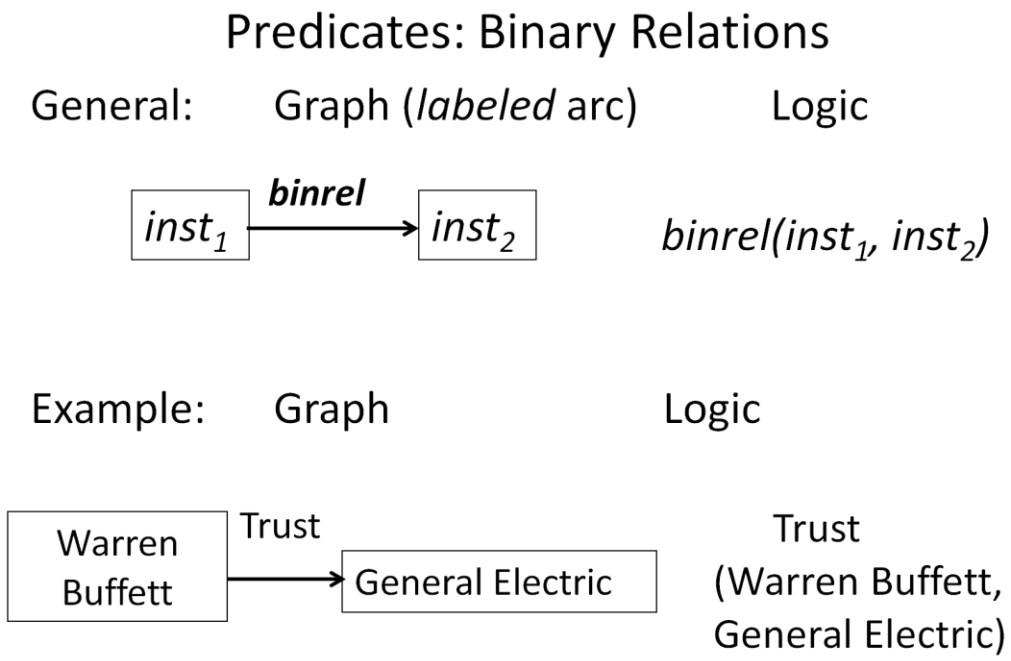

Figure2: Binary relations (adapted from [2])

2.5.3 Object Centering: Grouping Binary Relations Around Instances

After the description of unary and binary relations, we can group all relations around their (first argument) instance. As demonstrated in figure 3, there is one class and one instance ($inst_0$) belonging to it. Besides, there are n binary relationships, $binrel_1$,…, $binrel_n$, leading from $inst_0$ to $inst_1$, …, $inst_n$. And there is also an example to illustrate specifically.

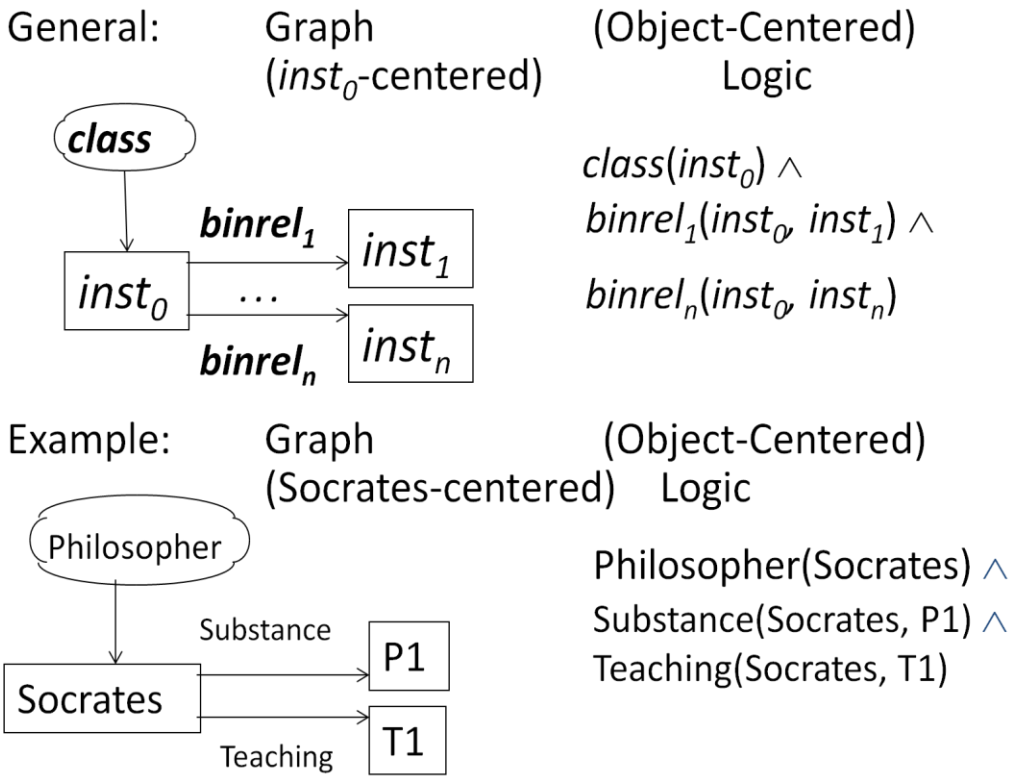

Figure 3: Grouping binary relations around instances (adapted from [2])

2.6 Frame Formulas: Associating Slots with OID-Distinguished Instance

This part is the main basis of our project. We can compare figure 4 with figure 3. We can find they are similar, but also have crucial differences. Beside the rectangle for the instance of the class, there is a dot (Bullet) at the tail of the arrow, and the label "slot" replaces "relation". Moreover, the symbolic form of the logic also changes. In that case, we can see the OID is more distinguished, because it is an F-logic/PSOA-like frame term which combines the advantages of conceptual modeling with object-oriented, frame-based languages and offers a declarative, compact and simple syntax, as well as the well-defined semantics of a logic-based language.

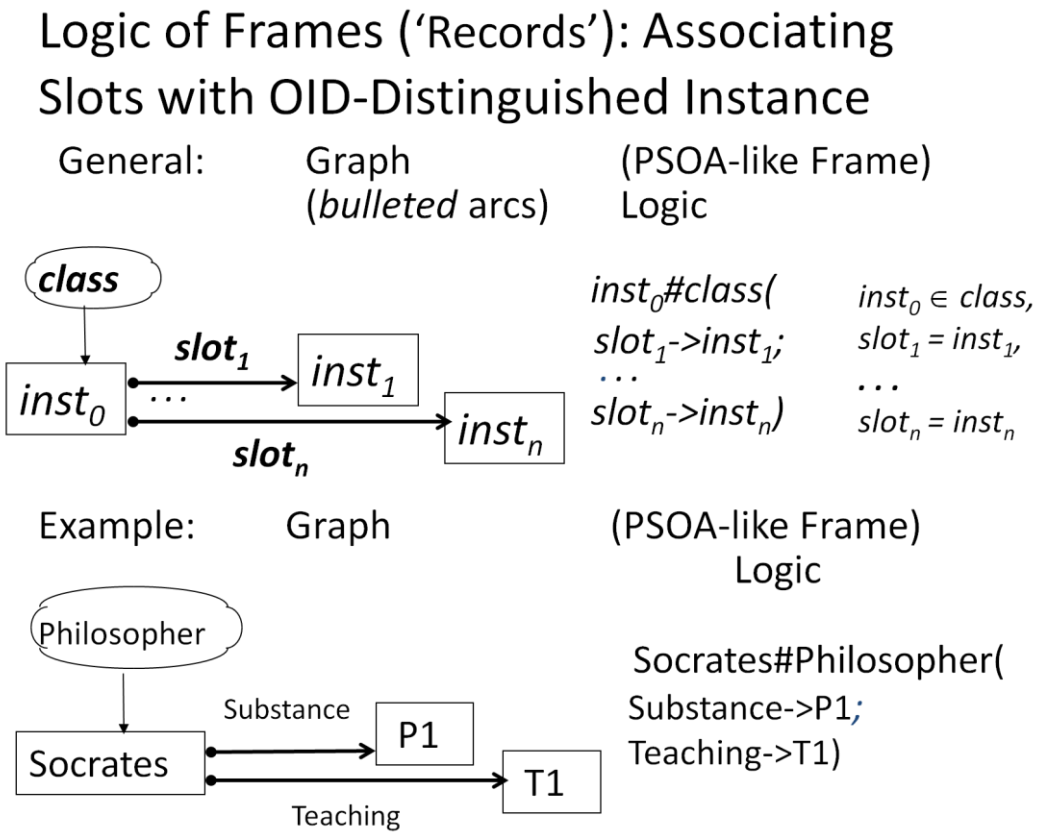

Figure 4: Associating slots with OID-Distinguished instances (adapted from [2])

3. Tools

3.1 Eclipse

Eclipse is a multi-language software development environment comprising a workspace and an extensible plug-in system. It is written mostly in Java. It can be used to develop applications in Java and, by means of various plug-ins, other programming languages including Ada, C, C++, COBOL, Fortran, Haskell, Perl, PHP, Python, R, Ruby (including Ruby on Rails framework), Scala, Clojure, Groovy, and Scheme. It can also be used to develop packages for the software Mathematica. Development environments include the Eclipse Java development tools (JDT) for Java, Eclipse CDT for C/C++ and Eclipse PDT for PHP, among others.

The initial codebase originated from IBM VisualAge. The Eclipse SDK (which includes the Java development tools) is meant for Java developers. Users can extend its abilities by installing plug-ins written for the Eclipse Platform, such as development toolkits for other programming languages, and can write and contribute their own plug-in modules.

Released under the terms of the Eclipse Public License, Eclipse SDK is free and open source software (although it is incompatible with the GNU General Public License). It was one of the first IDEs to run under GNU Classpath and it runs without problems under IcedTea [7].

3.2 Graphviz

Graphviz (short for *Graph Visualization Software*) is a package of open-source tools initiated by AT&T Labs Research for drawing graphs specified in DOT language scripts. It also provides libraries for software applications to use the tools. Graphviz is free software licensed under the Eclipse Public License [8].

The Graphviz layout programs take descriptions of graphs in a simple text language, and make diagrams in useful formats, such as images and SVG for web pages, PDF or Postscript for inclusion in other documents; or display in an interactive graph browser. (Graphviz also supports GXL, an XML dialect.) Graphviz has many useful features for concrete diagrams, such as options for colors, fonts, tabular node layouts, line styles, hyperlinks, rolland custom shapes [9].

3.3 Other Tools

Moreover, other tools, such as SWRLTab and Jambalaya, can also be used to visualize SWRL's slotted formulas in Grailog. The SWRLTab is a development environment for working with SWRL rules in Protege-OWL. It supports the editing and execution of SWRL rules and includes a set of libraries that can be used in rules, including libraries to interoperate with XML documents, and spreadsheets, and libraries with mathematical, string, RDFS, and temporal

operators [10]. Jambalaya is a plug-in created for Protégé which uses Shrimp to visualize the knowledge bases the user has created [11].

4. Structure and Implementation

4.1 Structure and Main Steps

In figure5, we show the structure and main steps of our project to be implemented. First we get some SWRL rules (Unary/Binary Datalog formulas) from the input of our interface, and then we translate the Unary/Binary formulas into slotted formulas. Meanwhile, each unit of the rules will be split into three parts: properties, instances and classes. After this procedure, we generate the result of the split parts into a text file and use it to rewrite the template of the dot file, which can be recognized by Graphviz, an important component of OWLVIZ. At last, we generate the structure of the input rules via delivering the dot file to Graphviz.

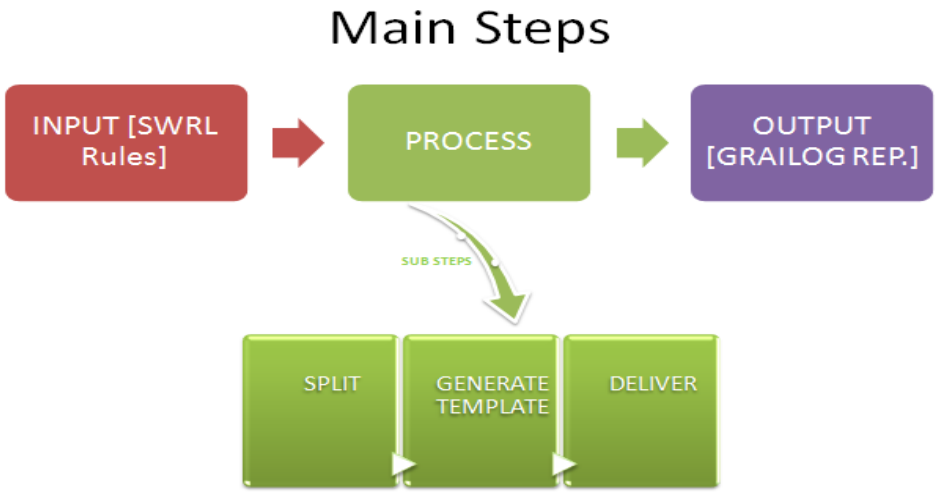

Figure 5: Main Steps

4.2 Split

In our program, we define that the input rules must be unary/binary formulas in SWRL presentations syntax, using symbol '^' to combine each unit and symbol '->' to combine the body part and the head part. Figure 6 shows an example of a SWRL rule.

```
Person(?x) ^ Man(?y) ^ hasSibling(?x,?y) ^
hasAge(?x,?agel) ^ hasAge(?y,?age2)^
swrlb:greaterThan(?age2,?age 1)—>
hasOlderBrother(?x,?y)
```

Figure 6: An example of a SWRL rules in presentation syntax

First we define an n-dimensional array which has four columns: predicate, origin, destination, attribute, and then we use java split to separate each unit from a whole rule by detecting symbol '^', and insert the split result into the first three columns of the n-dimensional array. For the last column, we fill it based on the following clues.

- If the destination column is null for a unit, this unit should be a class (unary relation).

Person(?x)

predicate	origin	destination	attribute
Person	X		Class

Figure 7: An example of the class attribute

- If the first three columns are all filled in, this unit should be a property (binary relation).

has_sibling(?x,?y)

predicate	origin	destination	attribute
Person	X	Y	Property

Figure 8: An example of the property attribute

- If the predicate part starting with the string "swrlb:", this unit should be a swrl-built-in. The set of build-ins for SWRL is motivated by a modular approach that

will allow further extensions in future releases within a (hierarchical) taxonomy. At the same time, it will provide the flexibility for various implementations to select the modules to be supported with each version of SWRL [12].

swrlb:greaterThan(?age2,?age1)

predicate	origin	destination	attribute
swrlb:greaterThan	age2	age1	swrl_built_in

Figure 9: An example of the SWRL-built-in attribute

By doing the split, we can clearly recognize the attribute of each unit and classify the properties of each class's instance, so it's convenient for us to translate the unary/binary formulas into slotted formulas in our interface. What's more, when it comes to write the template of dot file, it's easy to distinguish the classes from the properties, since there are more properties in the rules than classes.

4.3 Visualization

This section describes how to visualize the SWRL rules based on our Grailog visualization method. Before doing the visualization, the slotted logical syntax of each rule (based on the Grailog slotted formula) will also be shown. So in the visualization part two tasks will be done: In the first phase, every SWRL rule will be converted to its Grailog slotted formula and in the second phase, their equivalent visual graph (again based on Grailog format) is generated. A prototype tool was developed for these tasks.

SWRL rules can be input in two ways. They can be entered directly into the GUI of the tool or we can read the rules from an input file, in the current directory named "swrl.txt". In the second way (reading from a file) each rule should be in a separate line. The remainder of this section is as follows: in subsection 4.3.1 we show how to convert SWRL rules to logical slotted formulas using an example, and subsection 4.3.2 describes how to generate the output graph.

4.3.1 SWRL Rules in Presentation Syntax

The following SWRL conjunctive formula shows that the "$inst_0$" object belongs to the class of "class" and it has properties (relations) $slot_1$, $slot_2$,…, $slot_n$ with "$inst_1$" , "$inst_2$", …, "$inst_n$" as their second argument :

$$class(?\,inst_0) \wedge slot_1(?\,inst_0, ?\,inst_1) \wedge slot_2(?\,inst_0, ?\,inst_2) \wedge \ldots \wedge slot_n(?\,inst_0, ?\,inst_n) \qquad Eq.(1)$$

Here the "?" symbol determines the variable and "^" shows denotes conjunction. The following shows a frame formula in PSOA RuleML equivalent to $Eq.(1)$:

$$inst_0 \#class(slot_1 -> inst_1\,;\ slot_2 -> inst_2\,;\ldots;\ slot_n -> inst_n) \qquad Eq.(2)$$

SWRL rules use a conjunctive formula as premise and as conclusion. When we receive a SWRL rule it will be converted to a PSOA RuleML logical frame formula as mentioned above. For an example consider that we have the following SWRL rule:

$$Person(?\,x) \wedge Man(?\,y) \wedge hasSibling(?\,x, ?\,y) \wedge hasAge(?\,x, ?\,age1) \wedge hasAge(?\,y, ?\,age2) \qquad Eq.(3)$$
$$\wedge\ swrlb{:}greaterThan(?\,age2, ?\,age1) -> hasOlderBrother(?\,x, ?\,y)$$

So, it can be converted to the following PSOA RuleML rule:

```
hasOlderBrother(?x ?y)                                    Eq. (4)
    :-
     And(
          ?x#Person(
                    hasAge(?age1)
                   )
          ?y#Man(
                   hasAge(?age2)
                   )
          hasSibling(?x ?y)
          swrlb:greaterThan(?age2 ?ag1)
         )
```

This says that object "?x" is of class "Person" and has a property named "hasAge". Also object "?y" is of class "Man" and has property "hasAge".

4.3.2 Visualize SWRL Rules

After converting SWRL rules to Grailog's frame formula, a visual graph of the rules is generated to show the objects and their properties and relations. This graph is a great way to represent a rule to a human expert. So, he/she will be able to check and understand the rule visually.

Grailog uses special shapes and arrows to visualize a rule as unary/binary formulas or slotted formulas. For more information you can refer to [2].

For drawing the graph we use the Graphviz library. Graphviz is open source graph visualization software. Graph visualization is a way of representing structural information as diagrams of abstract graphs and networks. It has important applications in networking, bioinformatics, software engineering, database and web design, machine learning, and in visual interfaces for other technical domains.

The Graphviz layout programs take descriptions of graphs in a simple text language, and generate diagrams in useful formats, such as images and SVG for web pages, PDF or Postscript for inclusion in other documents; or display in an interactive graph browser. (Graphviz also supports GXL, an XML dialect.) Graphviz has many useful features for concrete diagrams, such as options for colors, fonts, tabular node layouts, line styles, hyperlinks, and custom shapes [9].

This project uses the Graphviz "dot" tool to generate the graph. So, when you want to run this program you should have Graphviz already installed. It takes the following steps to generate the graph and show it in the output:

- Read the rule
- Create the .dot file named "output.dot" as an input to Graphviz
- Ask Graphviz to generate the output graph

Please note that the tool can generate the graph with several different output formats like .png, .gif, .svg, .imap and so on.

In the following, an example will be used to make the context clearer and also to show how the tool works. Consider that we have Eq. (3) as input SWRL rule. The first output of the tool is Eq. (4) and the second output (the output graph) is shown in Figure 10 which is a .png picture based on user selection:

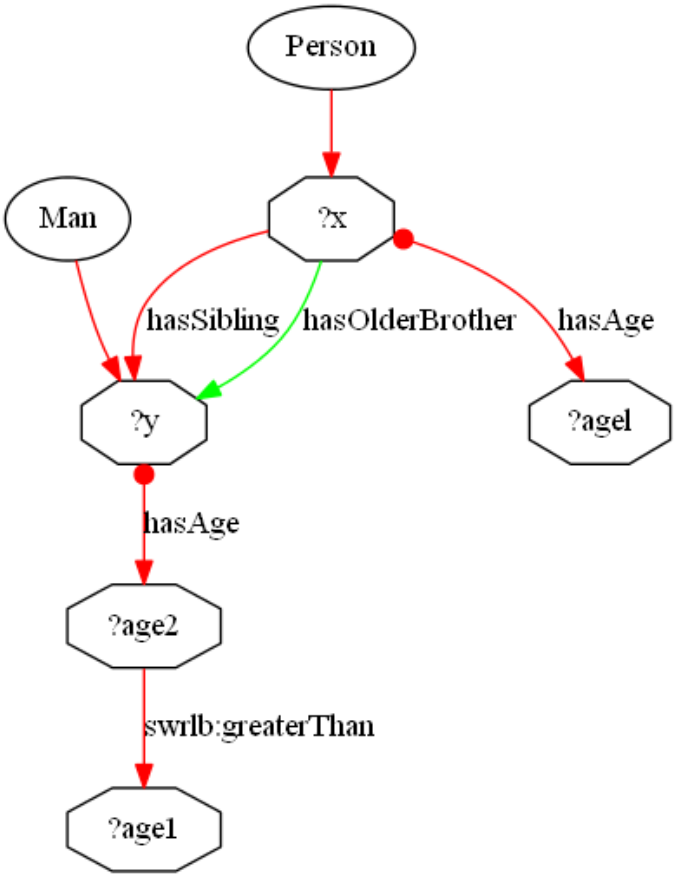

Figure 10: Output graph for Eq. (3)

In Figure 10, the simple red arrows show a connection between an object and its class or between objects. The red dotted arrows show a connection between an object and its property. The green arrows show the conclusion of the rule. An oval shows a class, an octagon shows an object, and a box shows an instance or constant.

As shown in Figure 10, there are two objects “x” and “y”, which both have one property named “hasAge” that is connected to “age1” and “age2”, respectively.

5. Conclusion and Future Work

SWRL can be represented as unary/binary Datalog formulas or (slotted) frame formulas, and both of them have their own characteristics as graphs and logics. Visualizing the slotted formulas lets people easily understand the logic of SWRL rules.

In our project, we visualize SWRL's formulas in Grailog. Every SWRL rule will be converted to its PSOA RuleML frame formula, and then the visual equivalent graph (in Grailog) is generated. We have uploaded our implementation to the website of Githup https://github.com/boliuy/SWRL-RULES-VISUALIZER, and everybody is welcome to the website and comment on our project.

In the future, Grailog can be applied in various fields. Researchers can refine and extend Grailog as an open standard. Grailog structures could be generated and transformed from 2 dimensions to 3 dimensions; in that case, people might be able to understand and write rules even more easily.

References

[1] http://en.wikipedia.org/wiki/Semantic_Web_Rule_Language.

[2] The Grailog Systematics for Visual-Logic Knowledge Representation with Generalized Graphs. Boley Harold, http: //www.cs.unb.ca/~boley/talks/RuleMLGrailog.pdf.

[3] http://www.obitko.com/tutorials/ontologies-semantic-web/owl-dl-semantics.html

[4] http://www.w3.org/TR/owl-features/

[5] http://en.wikipedia.org/wiki/F-logic

[6] http://wiki.ruleml.org/index.php/PSOA_RuleML#Rule_Test_-_Psoa_Terms

[7] http://en.wikipedia.org/wiki/Eclipse_(software)

[8] http://en.wikipedia.org/wiki/Graphviz

[9] http://graphviz.org/

[10] http://protege.cim3.net/cgi-bin/wiki.pl?SWRLTab.

[11] http://thechiselgroup.org/2004/07/06/jambalaya.

[12] http://www.w3.org/Submission/2004/SUBM-SWRL-20040521/

Appendix

In this appendix, we test 3 examples, simple, medium, and difficult, respectively, to test our project. For each rule, we used our program to generate the graph shown.

(1) Rule 1 (simple):

Person(?x) ^ Man(?y) ^ hasSibling(?x,?y) ^

hasAge(?x,?agel) ^ hasAge(?y,?age2) ^

swrlb:greaterThan(?age2,?age1) ->

hasOlderBrother(?x,?y)

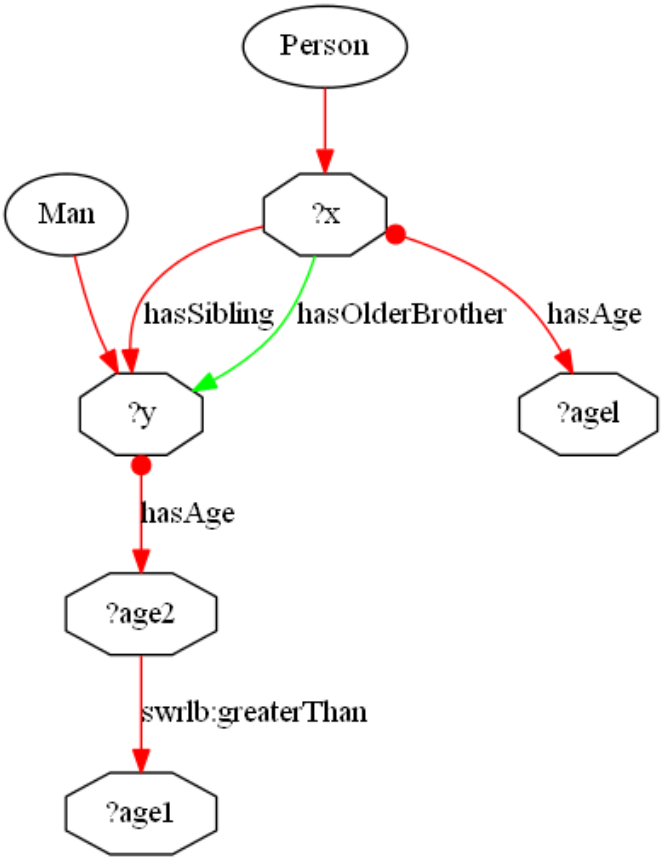

Figure 11: Test Rule 1

(2) Rule 2 (medium):

Individuals under the age of 18 are a potential driver of a vehicle with a weight of less than 26,000 lbs in California if they possess an out-of-state driver license and are visiting the state for less than 10 days.

SWRL:Person(?p) ^ has_Driver_License(?p,?d) ^ issued_in_state_of(?d,?s) ^ swrlb:notEqual(?s,"CA") ^ hasAge(?p,?g) ^ swrlb:lessThan(?g,18) ^ Number_of_Visiting_Days_in_CA(?p,?x) ^ swrlb:lessThan(?x,10) ^ Car(?c)^ has_weight_in_lbs(?c,?w) ^ swrlb:lessThan(?w,26000) -> can_Drive(?p,?c)

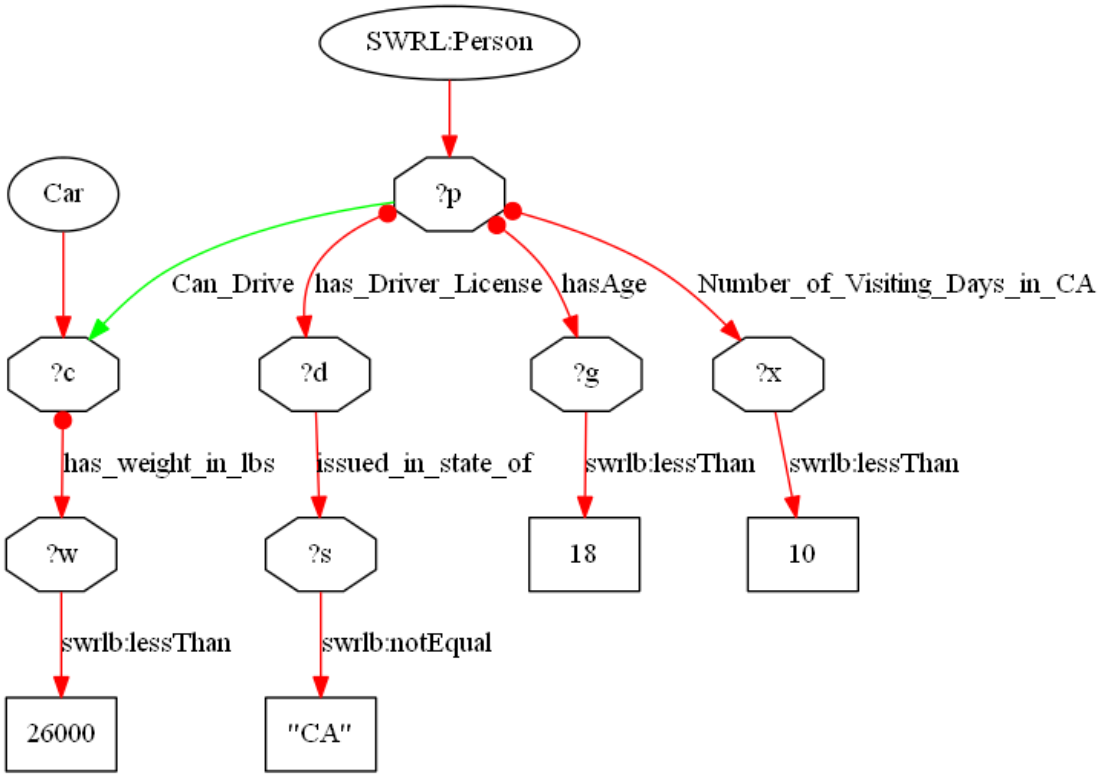

Figure 12: Test Rule 2

(3) Rule 3 (difficult):

An alert will be turned on or not by using the gps to test the speed of the car if it's enough or not to across the road when considered the traffic light.

Car(?car)^hasGPS(?car,?gps1) ^ hasLongitude(?gps1,?gps1Longitude) ^ hasLatitude (?gps1,?gps1Latitude) ^ TrafficLight(?trafficLight) ^ hasSignal(?trafficLight, ?signal) ^ swrlb:equal(?signal, "Green") ^ hasGPS(?trafficLight, ?gps2) ^ hasLongitude (?gps2, ?gps2Longitude) ^ hasLatitude(?gps2, ?gps2Latitude) ^ swrlb:lessThan(?distance, "100") ^ hasSpeed(?car, ?carSpeed) ^ swrlb:divide(?carTime, ?distance, ?carSpeed) ^ hasRemainingTime(?trafficLight, ?remainingTime) ^ swrlb:greaterThan (?carTime, ?remainingTime) ^ swrlx:createOWLThing(?alert, "Alert") -> Alert(?alert) ^ hasVehicle(?alert, ?car) ^ hasTrafficLight(?alert, ?trafficLight)

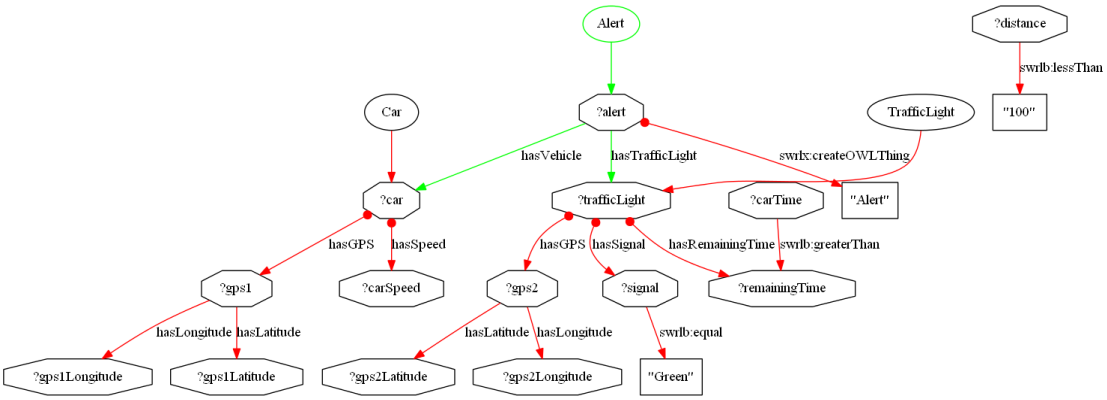

Figure 13: Test Rule 3