Grailog KS Viz: A Grailog Visualizer for Datalog RuleML Using an XSLT Translator to SVG

Martin Koch[1,2], Sven Schmidt[1,2], Harold Boley[1], and Rainer Herpers[1,2]

[1] University of New Brunswick, Faculty of Computer Science, Fredericton, NB, E3B 5A3, Canada

[2] Bonn-Rhein-Sieg University of Applied Sciences, Institute of Visual Computing & Department of Computer Science, Grantham-Allee 20, 53757 Sankt Augustin, NRW, Germany

{martin.koch,sven.schmidt,harold.boley}[AT]unb.ca,
rainer.herpers[AT]h-brs.de

Abstract. Grailog embodies a systematics to visualize knowledge sources by graphical elements. Its main benefit is that the resulting visual presentations are easier to read for humans than the original symbolic source code. In this paper we introduce a methodology to handle the mapping from Datalog RuleML, serialized in XML, to an SVG representation of Grailog, also serialized in XML, via eXtensible Stylesheet Language Transformations (XSLT) 2.0/XML; the SVG is then rendered visually by modern Web browsers. This initial mapping is realized to target Grailog's "fully node copied" normal form. Elements can thus be translated one at a time, separating the fundamental Datalog-to-SVG translation concern from the concern of merging node copies for optimal (hyper)graph layout and avoiding its high computational complexity in this online tool. The resulting open source Grailog Knowledge-Source Visualizer (Grailog KS Viz) supports Datalog RuleML with positional relations of arity n>1. The on-the-fly transformation was shown to run on all recent major Web browsers and should be easy to understand, use, and extend.

Keywords: visualization, graphs, directed hypergraphs, Grailog, computational logic, rules, Datalog, XML, RuleML, XSLT, SVG, JavaScript

1 Introduction

Datalog RuleML [4, 5] is an XML serialization of Datalog and the n-ary core sublanguage of the RuleML family. Because of its interoperation usage in Artificial Intelligence (AI) and the Semantic Web, its normative syntax is in the Extensible Markup Language (XML), which is more suitable for machine processing than for human readability. To make the Datalog RuleML language more readable for humans, one natural approach is to translate its knowledge bases, consisting of facts and rules, in a human-oriented manner.

One method is to create a visualized representation from knowledge bases. Well-developed visualizations, as optional two-dimensional syntaxes, can help people to better understand logical constructs than through symbolic one-dimensional syntaxes alone. This has been explored in an approach of visualizing major (Semantic Web) formalisms in Graph inscribed logic (Grailog) [2,3]. It describes the mapping for several defined graph constructs to corresponding symbolic logic constructs of the considered sublanguage, which leads to better human readability. The current work constitutes a first step to try to automate this transformation as far as possible, so that ultimately each Datalog and other RuleML knowledge base can be easily visualized as a Grailog representation.

1.1 Objectives

The goal of this work has been to handle the task described before, i.e. the translation from Datalog RuleML to a visualized Grailog representation. This has resulted in an initial version of the Grailog Knowledge-Source Visualizer (Grailog KS Viz). The tool is open-source and deployed on the Web for access by users of RuleML, Grailog, Datalog and related systems (http://www2.unb.ca/~mkoch/cs6795swt/index.html).

The main objective of this work is to define a mapping from Datalog RuleML, serialized in XML [6], to the Scalable Vector Graphics (SVG) [8], both being W3C standards. This should be realized using eXtensible Stylesheet Language Transformations 2.0 (XSLT 2.0) [10], another W3C standard, which permits transformations from an XML source document to an (XML) target document. A visualization of the overall process can be seen in Figure 1.

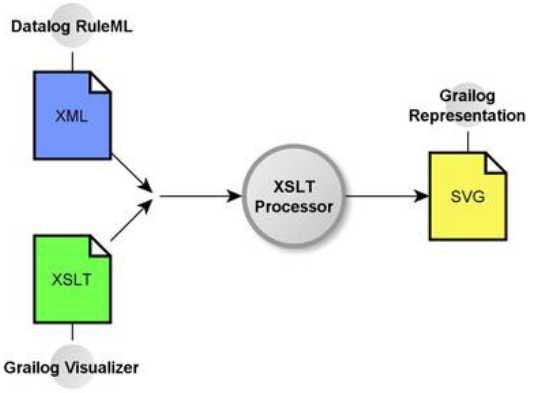

Fig. 1. Visualization of the general transformation process: Datalog RuleML (written in XML) in combination with an XSLT file (Grailog KS Viz) leads to an SVG file (the corresponding Grailog representation)

By implementing different elements in the XSLT 2.0 document, each Datalog RuleML fact and rule in XML should automatically be transformed to its corresponding SVG element in XML. Elements can thus be translated one at a time, separating

the fundamental Datalog-to-SVG translation concern from the concern of merging node copies for optimal (hyper)graph layout and avoiding its high computational complexity in this online tool. A Grailog layout optimizer would deserve an entire R&D effort on its own. Our resulting visualization will instead stay in the "fully node copied" Grailog normal form, which means that the separation of clauses of the symbolic form in the Datalog RuleML source document is also kept in the resulting graphical form of the SVG Grailog representation.

Grailog is a comprehensive aid to visualize logics. The task of this work is to create an initial version of Grailog KS Viz, i.e. to use only a specific subset and functionality of Grailog corresponding to Datalog RuleML. This includes the important capability to translate Datalog's n-ary relationships, for the current work with n>1, to Grailog's directed hyperarc arrows, which connect the argument-box nodes.

For usability, it should be possible to render the resulting SVG representation of the Datalog RuleML source document as a Grailog diagram "on the fly" by using suitable tools, e.g. one of the recent major Web browsers.

1.2 Languages

Several languages have constituted the backbone of this work. Datalog RuleML and Grailog have been important as initial source and final target of the envisioned translation. XML, SVG, XSLT and JavaScript are needed for the implementation. In the following, each language will be briefly described.

Datalog RuleML. RuleML is a family of rule based languages, which are used for sharing rule bases written in XML and publishing them on the Web. Datalog RuleML is one rule sublanguage of RuleML, providing the relational-view-like expressivity of Datalog in RuleML. Datalog [7] is at the foundation of RuleML and can be seen as an intersection of SQL and Prolog [4]. Datalog is used to define facts and rules and is therefore suitable to create function-free Horn logic knowledge bases and queries. XML files that contain stripe-skipped Datalog RuleML are the initial source files of this work. A tool called RuleML Official Compactifier (ROC) [14] can be used for the transformation from the fully expanded normal form to the stripe-skipped form.

Grailog. Grailog embodies a systematics to visualize knowledge representations by graphical elements. Each of the introduced graphical elements is mapped to its corresponding symbolic logic construct. The main benefit of Grailog is that its resulting visual representations of knowledge sources are much easier to read for humans than the original symbolic source code. Moreover, the transformation rules for the graphical-to-symbolic mapping are easy to learn and remember.

Grailog is based on several principles. One principle is that the used graphs should be natural extensions of Directed Labeled Graphs. Another principle mandates that the used graphs should allow stepwise refinements for logic constructs, like Description Logic constructors or general PSOA RuleML terms. Moreover, Grailog also implements the principle of orthogonality, which means that it consists of several inde-

pendent concepts which can be freely combined with each other. All these principles support the main goal of Grailog, namely easy understanding and application of the systematics.

A major part of Grailog's visualization elements are the directed hypergraphs, which are used for dealing with the n-ary relationships Datalog RuleML is capable of expressing.

Scalable Vector Graphics (SVG). SVG is an XML specification for two-dimensional graphics. It supports static and dynamic (i.e., interactive or animated) graphics, with different types of graphical objects and functions. Documents, written is SVG, are generally supported by all recent major Web browsers, but in fact all browsers have advantages and disadvantages. This is why this work has two final Grailog KS Viz versions. The following sections will describe this issue more precisely. To reproduce dynamic graphics, SVG needs the help of JavaScript, which was another important element of this work.

JavaScript. JavaScript is a scripting language and was defined in the ECMAScript language standard, developed by Ecma. Scripts do not need to be preprocessed before running and can make Web pages more dynamic. Due to these advantages, Web pages behave more like traditional software applications. Specific capabilities are interactive contents, the animation of page elements or the loading of new page contents without reloading of the whole page. [9]

Extensible Stylesheet Language Transformations (XSLT). The XSLT language permits transformations of XML documents into XML target documents or several other forms, like e.g. HTML, XHTML or SVG. The XSLT processor takes XML sources and a stylesheet, which describes the rules for the transformation. These template rules associate patterns, which match nodes in the source document, with a sequence constructor. In many cases evaluating the sequence constructor will cause new nodes to be constructed that can be used to produce part of a result tree. The structure of the result trees can be completely different from the structure of the source trees. In constructing a result tree, nodes from the source trees can be filtered and reordered, and any structure can be added.

2 Methodology

The result of this work should be an initial version of Grailog KS Viz, which automatically translates Datalog RuleML to the applicable Grailog representation, both written in XML. The translation should be done via XSLT. The final results should be described by SVG and be presentable through any suitable tool, like one of the recent major Web browsers.

The procedure of this work was divided into three parts. The first part was to determine all necessary elements of Grailog and Datalog RuleML. That means especial-

ly for Grailog the possible relationship elements, and for Datalog RuleML the supported XML declarations. The second part was to create all important Grailog representations in SVG, to have a pattern-like set of graphical elements for the last part of this work. The last part was the implementation of an XSLT file, which transforms a Datalog RuleML XML file into the Grailog representation. The graphical elements of the second step were necessary to lead the transformation to the right results. The following subsections will describe this procedure in more detail.

2.1 Determination of the Required Elements of Grailog and Datalog RuleML

The result of this work could only be an initial version of Grailog KS Viz because of the limited time of this work. Therefore, the first step was to determine the required elements of Grailog and Datalog RuleML.

The chosen subset of Grailog and Datalog RuleML can be seen in Figure 2. This subset contains all elements for an adequate Datalog RuleML knowledge base. As mentioned previously, the resulting visualization should be in a "fully node copied" normal form (in Grailog, all node occurrences with the same unique name remain one node).

To support the Grailog representations seen in Figure 2, Grailog KS Viz had to support the following XML elements of Datalog RuleML: "RuleML" (with and without namespace and schema declaration), "Assert", "Implies", "And", "Atom", "Rel", "Var", "Ind" and "Data".

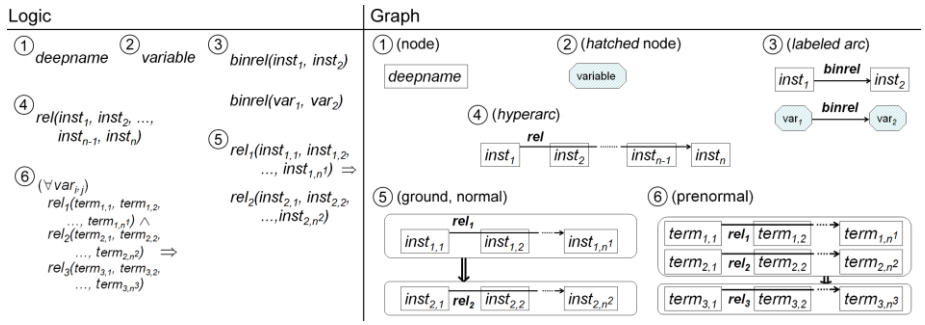

Fig. 2. Chosen Datalog RuleML subset visualized in Grailog: Both logical (left) and graphical (right) representations shown for (1) individual constants, (2) variables, (3) binary relations, (4) (n>1)-ary relations, (5) single-premise rules and (6) multi-premise rules (modified from [2])

2.2 Grailog

The second step of this work was to create the Grailog representations in SVG. SVG allows three different types of graphical objects, the vector graphic shapes (e.g. paths or polygon lines), images and text. For the different graphical objects SVG provides further methods, like grouping objects, assigning different styles per object, and performing further transformations. Moreover, SVG provides a rich feature set which includes nested transformations, clipping paths, alpha masks, filter effects and tem-

plate objects. SVG is completely described through XML and has to be introduced by a “svg” element as the root. The dynamics of graphical elements is possible through JavaScript. JavaScript was especially essential to get the lengths of the different texts, to scale the elements and to position them. The importance of JavaScript was not obvious in the first considerations of this work and was therefore not mentioned in the objectives.

The creation of the Grailog representation started with the simplest element: a single individual constant. An individual constant is described in Grailog through a specific text surrounded by a rectangle. So, the first step was to create an SVG “text” element. This “text” element had to contain the desired text of the individual constant as well as a specific “id” and the coordinates “x” and “y”, as attributes. Every “text” element needs its own ID-number for distinction. The coordinates are relevant to place the text on a specific position in the viewbox which represents the image section on the screen. The very first text element always starts at the coordinates “x = 50” and “y = 50”. All further elements will be arranged to correspond to the first “text” element. Rectangles can be visualized in SVG through the “rect” element. For individual constants the “rect” element has to include the attributes “id”, “x” and “y” (as start coordinates like before), “height” and “width” (as determination for the size) and “style” (to assign stroke color and width). Apart from the “id” and the “style”, all attributes will be assigned through JavaScript code, to be independent of the varying text lengths. First, the JavaScript code computes the width of the “rect” element by calculating the length of the text of the “text” element. Then it determines suitable values for “height” and the coordinates “x” and “y” in consideration of the coordinates of the “text” element. Figure 3 shows the SVG source code for an exemplary individual constant and the resulting Grailog representation.

The second simplest element of the Grailog subset was the representation of a variable. A variable is a specific text surrounded by a hexagonal box, hatched by diagonally arranged blue lines. The hatched lines are a result of a specific “pattern” element. A “pattern” element of SVG can contain different other SVG elements. This self-implemented “pattern” element contains several “path” elements, which are in this case nothing more than diagonally arranged blue lines with specific start and end points. The pattern is allocated to a “polygon” element, which finally represents the hexagonal box of the variable. The “polygon” element of this variable representation contains attributes for “id”, “points” (determinations for the edges) and “style” (to assign the hatched pattern as well as stroke color and width). As seen for the individual constant, most of the work for positioning is done via JavaScript. In this case JavaScript especially computes the edges of the polygon to arrange the box around the text.

The last important elements of the different Grailog representations were the arrow for relations and the double-arrow and rectangles with rounded corners for rules. An arrow for relations consists only of a black “path” element and a “marker” element, which contains the arrow-head. The double-arrow is implemented as a single “path” element, which follows a particular track. To create a rectangle with rounded corners, one only has to assign the attributes “rx” and “ry” to a “rect” element.

The last step of the SVG part was to create different SVG documents out of the above described simple elements of the Grailog representation. The final documents included a representation for a single individual constant, a single variable, any combination of binary and n-ary relationships, as well as single- and multi-premise rules. After creating all these standard Grailog representations, the SVG elements could be used as patterns to transform any Datalog RuleML XML file through XSLT. The next subsection describes this transformation and the procedure of the implementation.

SVG source code

```
<svg version="1.1" xmlns="http://www.w3.org/2000/svg">
    <text id="text1" x="50" y="50">inst1</text>
    <rect id="rect1" style="stroke:#000000; fill: none; stroke-width:1;"/>
    <script>
        document.getElementById("rect1").setAttribute("width",
        parseFloat(document.getElementById("text1").getComputedTextLength()) + 20);
        document.getElementById("rect1").setAttribute("height",40);
        document.getElementById("rect1").setAttribute("x",
            parseFloat(document.getElementById("text1").getAttribute("x")) - 10);
        document.getElementById("rect1").setAttribute("y",
            parseFloat(document.getElementById("text1").getAttribute("y")) - 25);
    </script>
</svg>
```

Grailog representation

Fig. 3. SVG source code (left) and the resulting Grailog representation (right) of an individual constant

2.3 Creating the Translation from Datalog RuleML to Grailog

In this part of this work, the implementation of the transformation will be explained, which is done by using XSLT 2.0. The transformation mainly depends on the part described before, i.e. emulating of the created patterns. In the following, the basic structure of Grailog KS Viz is illustrated. After that, an example of a small part of the actual transformation is given and explained in detail.

Basic Structure of Grailog KS Viz. To work with XSLT in general, the "stylesheet" or the completely synonymous "transform" element have to be used. For Grailog KS Viz, version 2.0 of XSLT is used, which brought along many useful functions, as it builds on XPath 2.0, instead of XPath 1.0 like its predecessor. With the help of the attributes of the "output" element, the output document is set to be of the type "XML" with the encoding "ISO-8859-1", to allow a rich enough character set for the possible facts and rules that come with the source document. Moreover, the doctype of the resulting document is set to the official SVG doctype, to guarantee syntactic correctness of the produced SVG document.

The "apply-templates" element in conjunction with the "template" element is used to structure the transformation. The "template" element contains rules that are applied when a specific node is matched. With the help of "apply-templates" and its "select" attribute, the templates can be matched and the defined rules are executed. For Grailog KS Viz, these elements are used to divide the given knowledge base into

meaningful (sub)sections, like facts or the head of a single-premise rule. Another element which is used in this context is the “for-each” element. This element mostly loops through all terms of an atom. Because of the overall goal of a final SVG document as output, the first step of the transformation creates the SVG root element along with its attributes. Additionally, the definitions of the arrow head and the variable pattern are created.

The next step of the transformation process by Grailog KS Viz is to differentiate between rules and relations. Moreover, it is important to differentiate single-premise rules from multi-premise rules, because they result in different Grailog representations. The differentiation is done by searching the Datalog RuleML document for the elements “Implies”, “And” and “Atom”. The idea is to use parent and child relationships and positions to determine the type of the considered atom. If an atom is a child of an “Implies” element, it means that it is part of a rule. If it is a direct child of “Implies”, it is either the head of a multi-premise rule or the head or premise of a single-premise rule. This can be clarified by checking how many atoms are direct children of a specific “Implies” element. If there are two atoms, it is a single-premise rule, with the first atom as premise and the second atom as head. If there is only one atom as direct child, it has to be the head of a multi-premise rule. If an atom is child of an “And” element, it is part of the premise of a multi-premise rule. The XSLT / XPath functions that are mostly used for the search are among other things “current()”, “count((item, item, ...))”, “position()” and “last()”. These can be used to gain information about positions of specific nodes, about the number of child elements and much more. The last differentiation is done between binary and n-ary relations. Therefore, simply the number of children of an atom can be used. The differentiation is used to determine if a simple arrow or a hyperarc is needed in the resulting SVG representation.

After knowing the kind of the currently processed atom, the suited SVG elements and corresponding JavaScript code will be created. The elements can be “text”, “rect”, “polygon” or “path”, depending on the current node. The “value-of” element offers the possibility to get the value of the “Rel”, “Var” and “Ind” nodes of an atom. Moreover, it is frequently used to create unique variable names for all the different elements of the resulting SVG document. This is a huge part of the transformation process, because the SVG patterns that were achieved in the first part of this work are now dynamically created as part of the transformation. To get the unique variable names, a function of XSLT for concatenation is used to create names based on the type of the relation or rule that is currently processed and on its position in the XML tree. Besides the creation of the SVG elements, also the corresponding JavaScript code is inserted. The “if” element of XSLT is used to alternate the code for different cases, e.g. if a term is the first term of a relationship or the last. The JavaScript code is also used for keeping track of the maximum height and width of the viewbox of the SVG document.

Example for the Translation of an Individual Constant. Figure 4 shows a small part of Grailog KS Viz source code, which for this example results in the Grailog

representation of an individual constant of an n-ary relation, as previously seen in Figure 3 for SVG.

The first line of this extracted source code checks if the considered atom is a direct child of the "Assert" element. In this case it is part of a fact. Then the variable "countRelations" is defined. It holds the number of facts and rules that exist before the actual considered fact plus one, which equals the exact position of the fact in the knowledge base. Then each child element of the atom is processed. If it is an "Ind" or "Data" element, which both lead to the same Grailog representation, an SVG "rect" element and an SVG "text" element are created. For this case, the rectangles name consists of the substrings "rect", "Relation", the value of "countRelations" and the position of the term in the atom. This ensures that it gets a unique name, so that no problems occur in the resulting SVG document, regardless of the number and kinds of individual constants. Beneath the SVG elements, the corresponding JavaScript code is inserted, with its adjusted variable names and some further adjustments.

XSLT source code

```
<xsl:if test="parent::r:Assert">
  <xsl:variable name="countRelations" select="count(preceding-sibling::*) + 1"/>
    <xsl:for-each select="./*">
      <xsl:if test="position()=2"> <!-- First term -->
        <xsl:if test="(ancestor-or-self::r:Ind) or (ancestor-or-self::r:Data)">
          <xsl:element name="rect">
            <xsl:attribute name="id">
              <xsl:value-of select="concat('rect','Relation',$countRelations,position())"/>
            </xsl:attribute>
            <xsl:attribute name="style">
              stroke:#000000; fill: none; stroke-width:1;
            </xsl:attribute>
          </xsl:element>
          <xsl:element name="text">
            <xsl:attribute name="id">
              <xsl:value-of select="concat('text','Relation',$countRelations,position())"/>
            </xsl:attribute>
              <xsl:value-of select="."/>
          </xsl:element>
          <script type="text/javascript" language="JavaScript">
            … (relevant JavaScript code with adjusted variable names etc.) …
          </script>
  … (second term, next terms, last term; each for „Ind“, „Data“, „Var“ and „Rel“) …
</xsl:if>
```

Grailog representation

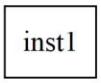

Fig. 4. XSLT source code (left) and its correspondent Grailog representation (right) for an individual constant

3 Results

The final result of the implemented translation process of Grailog KS Viz is an automatically generated SVG document, which can be used to render the Grailog representation of a given Datalog RuleML knowledge base. To demonstrate the reliability

and give a deeper understanding of the transformation and translation process, the following part shows and discusses two kinds of rules via examples.

3.1 Translation of Single-Premise Rules from Datalog RuleML to Grailog

The first kind of rule can be considered via an introductory example, which only consists of a single-premise rule and binary relations. In this case, the rule has no explicit meaning and only uses placeholder names. The rule in Datalog RuleML XML syntax and its Grailog representation, created by Grailog KS Viz, are shown in Figure 5.

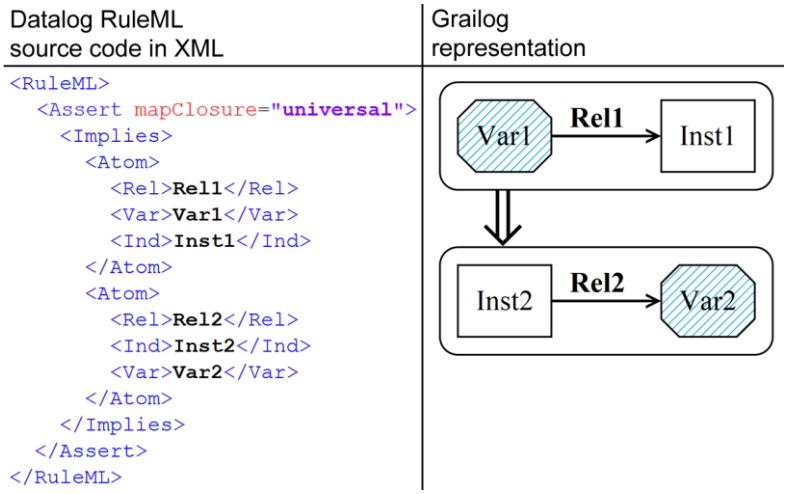

Fig. 5. XML source code (left) and its correspondent Grailog representation (right) for a single-premise rule with binary relations

For the above shown example, Grailog KS Viz first recognizes that there is a rule, because there is an "Implies" element in the XML source document that can be matched. Then it checks if it is a single-premise or a multi-premise rule by looking at the number of atoms that are direct children of the "Implies" node. Because there are exactly two, it has to be a single-premise rule. Therefore, first the upper rounded rectangle is created, followed by its content. The premise of the rule consists of a binary relation "Rel1" and two terms, the variable "Var1" and the individual constant "Inst1". The first term is created, followed by the text element with the value "Rel1". This can be done because the text with the relator name is always at the same relative x-position, directly besides the representation of the first term. Then the second term is transformed. After that, finally the arrow can be created, starting from the first term, either polygon or rectangle and ending at the second. After the content of the rounded rectangle is complete, the width of the rectangle is adjusted and then the double-arrow is created. The creation of the head of the rule is done the same way, only that the vertical positions are adjusted.

3.2 Translation of Multi-Premise Rules from Datalog RuleML to Grailog

The second kind of rule can be considered via an advanced example, which consists of a multi-premise rule containing two binary relations and a 3-ary relation. In general, our subset of Grailog allows (n>1)-ary relations, where the binary relation "be" is used to avoid unary relations (n=1). The rule has the following meaning: If George knows a player and an arena is a hockey rink, then George plays with the player in the arena. Figure 6 shows the XML source code and the resulting Grailog representation.

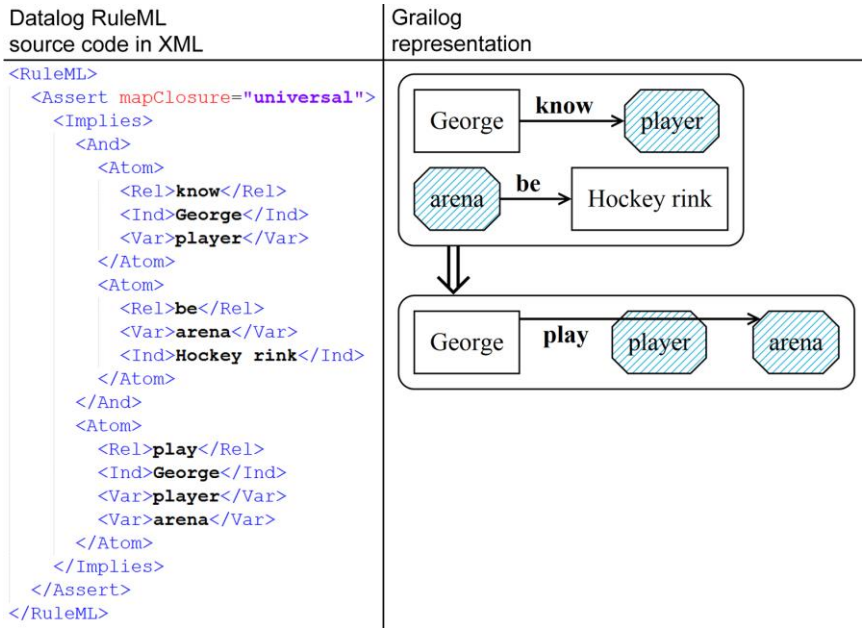

Fig. 6. XML source code (left) and its correspondent Grailog representation (right) for a multi-premise rule with binary and n-ary relations

In this example, the tool detects a multi-premise rule, because the "And" element is the first child element of the "Implies" element. Therefore, first the upper rounded rectangle is created, followed by its content. This affects the width and the height of the rounded rectangle, because a multi-premise rule can have an unbounded number of atoms in the premise. Because the premise of the rule only consists of binary relations, they are created as in the previous example. After the content of the upper rounded rectangle is complete, its width and height are adjusted and the double-arrow is created. The creation of the head of the rule cannot be done as in the previous example because it contains no binary relation, but the (n>2)-ary relation "play". Therefore, there can be an unbounded number of terms, and this leads to the usage of a hyperarc. The hyperarc is created after all terms are processed, and starts at the first term, cuts through intermediate ones and ends at the last. Finally, the lower rounded rectangle is created and then the SVG document is finalized. For this example the resulting SVG document consists of 289 lines of code.

There is no loss of generality with this example since a multi-premise rule with more than two premises can be reduced to multiple rules with two premises. However, Grailog KS Viz handles multiple premises directly.
The authors have started to complement the .ruleml examples (RuleML/XML) in the Datalog section of the public RuleML 1.0 exa library [13] with .grailog visualizations (RuleML/Grailog), including for the classical Datalog RuleML 'own' example (http://ruleml.org/1.0/exa/Datalog/own.ruleml paired with its visualization as http://ruleml.org/1.0/exa/Datalog/own.grailog).

3.3 Issues and Alternatives

Grailog KS Viz was realized by splitting the implementation part of this work into two mostly independent tasks, the creation of the Grailog representations in SVG and the actual transformation from Datalog RuleML to SVG. The independence of these two tasks was essential to be able to implement both tasks in parallel. This is also the reason for the usage of both, XSLT and JavaScript. Another reason that led to this approach was the fact that this was the first time to work extensively with SVG and XSLT for the first two authors, who implemented Grailog KS Viz. In retrospect, it might be possible to replace several parts of the JavaScript code with XSLT code. This would definitely have made the planning phase more complex and probably would have increased the time needed for completing this work. The benefit of such a solution would be much less code for the resulting SVG document.

Because one of the goals of this work was to support different recent major Web browsers, in the end two separate versions of the tool had to be implemented. Both use different methods to compute the length of the text elements, which is needed for determining the width and the position of the graphical elements. The “normal version” uses the JavaScript method “getComputedTextLength()”, which computes the length of the rendered text. This method unfortunately does not work for Firefox or Chrome at the on-the-fly transformation, i.e. by executing the transformation within the browser. After saving the resulting SVG document and opening it again, the method works for all tested browsers. The problem seems to be that at this moment when Firefox or Chrome use the “getComputedTextLength()”, no text is rendered, yet. To overcome this, a so called “monospaced version” of the tool was implemented. It uses the font “Monospace”, the “XMLSerializer()” and the “serializeToString()” method, which does work for Firefox, but not for Chrome or Safari, even after saving as SVG file and opening it again. The detailed support information can be seen in the appendix (Figure 7 and 8).

Currently, Grailog KS Viz consists of nearly 4000 lines of code, which is also based on the need for the support for two different source document versions, Datalog RuleML documents with and without the RuleML namespace and schema. Because this was realized in the last moments of this work, huge parts of the code exist in two forms, with the namespace in the XPath expressions and without it. By implementing a possible XSLT only-version, without the need for JavaScript and with a better implemented distinction between the two source document cases, it might be possible to

ultimately only have one, presumably monospaced and much shorter version of Grailog KS Viz.

Related work in the area of interest has been done for example by another RuleML/Grailog team [15]. This team also used the principles of Grailog for visualization purposes, but instead of building it from scratch, they used the Graphviz framework to visualize SWRL's unary/binary Datalog RuleML in Grailog. The benefit of building it from scratch, however, is that the final results are identical to the given specifications of Grailog.

4 Conclusion

The result of this work is the successful implementation of an initial version of Grailog KS Viz, a Grailog knowledge-source visualizer based on RuleML-to-SVG translation.

The first section gave a short introduction to the topic and stated how the approach of this work relates to previous work. Moreover, a short preview of the actual work was given and each language used throughout this work was explained briefly. The methodology for the successful accomplishment of the objectives was developed in three parts. The first part dealt with the determination of the required elements of Grailog and Datalog RuleML. The result was a mapping from a Datalog RuleML subset to a Grailog subset. The second part was about the creation of the Grailog representations in SVG. It explained the used elements and functions of SVG and JavaScript and illustrated an example for an individual constant. The last part of this section described the creation of the translation from Datalog RuleML to the Grailog representation. Therefore, it explained the basic structure of Grailog KS Viz and also illustrated an example of the transformation of an individual constant. The results are shown in the third section of the report. First, the transformation of a single-premise rule containing binary relations was explained, and then a more complex multi-premise example. Instead of focusing on the actual code as in the previous examples, these two were explained on a higher level. After that, some issues of the current implementation were highlighted and a general picture of this work and its environment was given.

The implemented Grailog KS Viz meets the given requirements and offers a robust and usable functionality with response times suitable for online rendering. The tool is open source and all its versions, documents, source files and many more examples and explanations can be found on the official website of this work. Several links to different parts of the website of this work can be found in the appendix.

4.1 Future Work

As mentioned before, although this work is an overall success, there are different aspects that definitely can be improved in further work. Besides the already mentioned possible improvements in the previous section, the following changes are envisioned:

A simple but effective improvement would be to transform the source document in a way that the resulting SVG document appears in a pretty print layout. Currently, the structure of the resulting code is not pretty printed, which complicates further adjustments, users of the tool could want to do manually. Furthermore, the merging of the individual SVG elements of rules and facts to a "node copy-free" graph would be a big improvement of this initial version of the visualizer. This would make the tool more powerful, because the user could directly see connectivity, although the high computational complexity of merging node copies for optimal (hyper)graph layout might entail response-time issues in online Grailog rendering of large Datalog knowledge sources. Besides these Grailog-target improvements, also the support of more RuleML-source features such as unary relations, (positional-)slotted relations and typed variables is planned. Then, the development should proceed from Datalog RuleML to Hornlog RuleML and gradually cover the remaining Grailog-visualized branches of the RuleML family. Grailog generators for other rule and ontology languages could be similarly implemented as well.
Complementing the approach of this work, inverse translators parsing Grailog SVG/XML diagrams into RuleML/XML trees could be realized. This could result in an authoring tool that allows users to visually design rule bases in the graphically rendered SVG representation, which will then be parsed into the Datalog RuleML/XML representation. In combination with our current Grailog generator, this Grailog parser could ultimately lead to a complete Grailog IDE.

5 Acknowledgments

Financial support of the DAAD in the ISAP program line, project No 54890855, is gratefully acknowledged. Also, NSERC is thanked for its support through Discovery Grants. Finally, we would also like to thank the RuleML reviewers, Kenneth Kent, Diana Kraus and all others who have supported us during this work.

References

1. Boley, H.: CS 6795 Semantic Web Techniques - Fall 2012 Projects. http://www.cs.unb.ca/~boley/cs6795swt/fall2012projects.html, visited on October 19th, 2012
2. Boley, H.: Grailog 1.0: Graph-Logic Visualization of Ontologies and Rules. Preprint: http://www.cs.unb.ca/~boley/papers/GrailogVisOntoRules.pdf, visited on May 9th, 2013. To appear: Proc. RuleML 2013, Springer LNCS 8035, July 2013
3. Boley, H.: Grailog. http://wiki.ruleml.org/index.php/Grailog, visited on May 24th, 2013
4. Boley, H., Athan, T.: RuleML Primer, August 2012. http://ruleml.org/papers/Primer/RuleMLPrimer2012-08-09/RuleMLPrimer-p0-2012-08-09.html, visited on October 19th, 2012
5. Boley, H., Athan, T., Paschke, A., Tabet, S., Grosof, B., Bassiliades, N., Governatori, G., Olken, F., Hirtle, D.: Schema Specification of Deliberation RuleML Version 1.0. http://ruleml.org/1.0/, visited on October 19th, 2012

6. Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., Yergeau, F.: Extensible Markup Language (XML) 1.0 (Fifth Edition) - W3C Recommendation, November 2008. http://www.w3.org/TR/2008/REC-xml-20081126/, visited on October 19th, 2012
7. Ceri, S., Gottlob, G., Tanca, L.: What You Always Wanted to Know About Datalog (And Never Dared to Ask). IEEE Transactions on Knowledge and Data Engineering, vol. 1 (1), pp. 146-166 (1989)
8. Dahlstroem, E., Dengler, P., Grasso, A., Lilley, C., McCormack, C., Schepers, D., Watt, J., Ferraiolo, J., Fujisawa, J., Jackson, D.: Scalable Vector Graphics (SVG) 1.1 (Second Edition) - W3C Recommendation, August 2011. http://www.w3.org/TR/2011/REC-SVG11-20110816/, visited on October 19th, 2012
9. Hazael-Massieux, D.: JavaScript Web APIs. http://www.w3.org/standards/webdesign/script.html, visited on November 15th, 2012
10. Kay, M.: XSL Transformations (XSLT) Version 2.0 - W3C Recommendation, January 2007. http://www.w3.org/TR/2007/REC-xslt20-20070123/, visited on October 19th, 2012
11. Refsnes Data: W3Schools - SVG Tutorial. http://www.w3schools.com/svg/default.asp, visited on October 19th, 2012
12. Refsnes Data: W3Schools - XSLT Tutorial. http://www.w3schools.com/xsl/, visited on October 19th, 2012
13. RuleML: The Rule Markup Initiative - Library of Datalog Examples. http://www.ruleml.org/1.0/exa/Datalog, visited on June 12th, 2013
14. Singh, S., Aayush, B. R., Shah, P.: Testing, Inverting, and Round-Tripping the RON Normalizer for RuleML 1.0 in XSLT 2.0. http://ruleml-roc.yolasite.com/, visited on Januar 9th, 2013
15. Yan, B., Zhang, J., Akbari, I.: Visualizing SWRL's Unary/Binary Datalog RuleML in Grailog. https://github.com/boliuy/SWRL-RULES-VISUALIZER, visited on December 13th, 2012

Appendix

Useful Links to this Work's Website

Index Page and Virtual Handout. Index page with illustrations of the objectives, methodology and the final results of this work.

- http://www2.unb.ca/~mkoch/cs6795swt/index.html

Results. Exemplary overview of different SVG result outputs and downloadable versions of Grailog KS Viz (normal and monospaced font version).

- http://www2.unb.ca/~mkoch/cs6795swt/media/html/project/results.html

Documentation. Overview of this work's documentation. Proposal, final presentation and report are downloadable there.

- `http://www2.unb.ca/~mkoch/cs6795swt/media/html/project/documentation.html`

Supported Web Browsers

Representation in Browser	Firefox (16.0.2)	Google Chrome (23.0.1271.64 m)	Internet Explorer (9.0.8112.16421)	Opera (12.10)	Safari (5.1.7)
After on-the-fly transformation	not supported	not supported	supported	supported	supported
Retrospectively saved as SVG	supported	supported	supported	supported	supported

Fig. 7. Supported (green) and unsupported (red) Web browsers of the normal version of Grailog KS Viz

Representation in Browser	Firefox (16.0.2)	Google Chrome (23.0.1271.64 m)	Internet Explorer (9.0.8112.16421)	Opera (12.10)	Safari (5.1.7)
After on-the-fly transformation	supported	not supported	supported	supported	not supported
Retrospectively saved as SVG	supported	not supported	supported	supported	not supported

Fig. 8. Supported (green) and unsupported (red) Web browsers of the monospaced font version of Grailog KS Viz