Faculty of Computer Science

CS6795 Semantic Web Techniques

December 17, 2012

Project Report

"Testing, Inverting and Round Tripping the RON Normalizer for RuleML 1.0 in XSLT 2.0"

Instructor: Dr. Harold Boley

Advisor: Dr. Tara Athan

Team: Simranjit Singh

Bijiteshwar R Aayush

Pratik Shah

Acknowledgement

*We hereby acknowledge **RON team** for the help which we have got from their documentation and website. We extend our heartiest thanks to our supervisor **Dr. Harold Boley** for his esteemed guidance and motivation. We are thankful to our advisor **Dr. Tara Athan** for her guidance on understanding RON.*

Table of Contents

1. Introduction

RuleML is a markup language for sharing rules in XML. It is serialized as an XML tree whose elements alternate between representing class or type tags (nodes), and representing method or role tags (edges). Alternating node/edge/…/edge/node elements give rise to a layered pattern referred to as 'stripes'. An example of such a striped serialization is RuleML's main rule node <Implies>, which contains <if> and <then> edges, leading to the next node layer, etc. Some edges can be omitted for brevity. In the normalization of a RuleML document we need to reconstruct all edges, transforming such 'stripe-skipped' serializations back into ones that are 'fully striped'. To normalize a document we also assure that the sublements are in proper canonical order, and make all attributes that have default values explicit. Finally, we perform pretty-print formatting.

Because of XML's left-to-right ordering we can often rely on the sub elements' positions to determine the implied roles of, say, the nodes under the <if> and <then> edges; therefore we can remove such edges completely. A document with all reconstructable edges removed is an (extreme) example of **stripe-skipping.**

The new tool which we have developed is a Compactifier for RuleML which will 'invert' the transformation of the existing normalizer. The transformation occurs by removing the stripes which have been added by normalizer or which are already present in the RuleML document. ***Due to its functioning we have named our tool ROC: RuleML Official Compactifier***. ROC will check the canonical order of the stripes in each element. After the order check is performed, ROC will remove the stripes from elements to convert them into a fully skipped form. We have implemented our tool in XSLT 2.0.

The compactified form obtained from ROC can be composed with RON for round tripping, which will lead to improvements of both the test suites obtained from a) testing the RON and b) implementation and testing of ROC.

1.1 Elements and their Role Tags

The role tags which should be added by the normalizer are given in the following list. For each element the name of the role tag it may be wrapped in is given, the RuleML Official Compactifier will skip all these role tags and make the XML element stripe skipped.

- <Retract> is checked for the <oid> tag which is allowed and if present, is skipped. If the name tag of the child is <formula> then the role tag is also skipped.
- <Query> is checked for the <oid> tag which is allowed and if present, is skipped. If the name tag of the child is <formula> then the role tag is also skipped.
- <Entails> is checked for the <oid> tag which is allowed and if present, is skipped. If a child of <Entails>, <if> or <else> is wrapped, then these tags are skipped.
- <Rulebase> is checked for the <oid> tag which is allowed and if present, is skipped. If the name tag of the child is <formula> then the role tag is also skipped.

- <Exists> is checked for the <oid> tag which is allowed and if present, is skipped. If the children are wrapped in <declare> or <formula> then these role tags are also skipped. If the child of <Exists> is <Var>, and <Var> is wrapped in <declare> then <declare> is skipped and rest all the elements which has been wrapped with <formula> are also skipped.

- <Forall> is checked for the <oid> tag which is allowed and if present, is skipped. If the children are wrapped in <declare> or <formula> then these tags are skipped.

- <Implies> is checked for the <oid> tag which is allowed and if present, is skipped. If a children is wrapped in <torso> , <if> or <then> then these tags are skipped.

- <And> is checked for the <oid> tag which is allowed and if present, skipped. If the name tag of the children is <formula> then this role tag is also skipped.

- <Or> is checked for the <oid> tag which is allowed and if present, is skipped. If the name tag of the children is <formula> then the role tag is also skipped.

- <Atom> is checked for the tags <oid>, <op>, <arg>, <degree>, <Reify>, and <Skolem>, which are allowed and if present, are skipped.

- <Assert> is checked for the <oid> tag which is allowed and if present, is skipped. If the name tag of the children is <formula> then the role tag is skipped.

- <Equal> copies foreign elements and <oid> unchanged as long as they are not in the second to last or last position. If either of the child is wrapped in <left> or <right> tag then that tag is skipped.

- <Neg> is checked for the <oid> tag which is allowed and if present, skipped. If the name tag of the children is then the role tag is also skipped.

- <Naf> is checked for the <oid> tag which is allowed and if present, skipped. If the name tag of the children is <weak> then the role tag is also skipped.

- <Expr> is checked for the tags <oid>, <op>, <arg>, <degree>, <Reify>, and <Skolem>, which are allowed and if present, are skipped.

- <Plex> is checked for the tags <oid>, <op>, <arg>, <degree>, <Reify>, and <Skolem>, which are allowed and if present, are skipped.

Note that the role tag <slot> is a special tag for user defined roles and cannot be skipped.

2. Testing RON:

As it is easily understood from RON's name itself, this Normalizer is a tool which is used to transform RuleML/XML into canonical form. Normalization involves re-creating the full expanded shape of the XML tree so that it can be context freely interpreted, implemented. The normalizer place all the stripes in proper canonical order which is the fully expanded form. The current normalizer was developed in XSLT 2.0 by a team in 2011 (http://v37s3b4h7dn47s37hg1br4h7rs7n3du7s8nu.unbf.ca/~lbidlak1/).

We will be testing RON against several test cases. The test design of RON is as follows. We will be testing RON against eight test cases.

Below explained is a test case which will show the workings of RON while the links for the other test cases have been given on the website and also in section 2.1.

2.1 Test Cases:

Test Case :

```
<?xml version="1.0" encoding="UTF-8"?>
<RuleML xmlns="http://ruleml.org/spec"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://ruleml.org/spec http://www.ruleml.org/1.0/xsd/datalog.xsd">
  <Assert mapClosure="universal">
    <Implies>
      <then>
        <Atom>
          <Rel>sells</Rel>
          <Var>vendor</Var>
          <Var>food</Var>
        </Atom>
      </then>
      <if>
        <Atom>
          <Rel>husband</Rel>
          <slot> <Ind>wife</Ind>
                 <Ind>Kathrina</Ind>
          </slot>
           <Ind>Adam</Ind>
```

```
                <slot> <Data>seller</Data>
                        <Var>vendor</Var>
                    </slot>
                </Atom>
            </if>
          </Implies>
        </Assert>
    </RuleML>
```

Result for the Test Case:

```
<?xml version="1.0" encoding="UTF-8"?>

<RuleML xmlns="http://ruleml.org/spec" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://ruleml.org/spec http://www.ruleml.org/1.0/xsd/datalog.xsd">

  <act index="1">
    <Assert mapClosure="universal">
      <formula>
        <Implies material="yes" direction="bidirectional" mapMaterial="yes" mapDirection="bidirectional">
          <if>
            <Atom>
              <op><Rel>husband</Rel></op>
              <arg index="1"><Ind>Adam</Ind></arg>
              <slot><Ind>wife</Ind><Ind>Kathrina</Ind></slot>
              <slot><Data>seller</Data><Var>vendor</Var></slot>
            </Atom>
          </if>
          <then>
            <Atom>
              <op><Rel>sells</Rel></op>
              <arg index="1"><Var>vendor</Var></arg>
              <arg index="2"><Var>food</Var></arg>
            </Atom>
          </then>
        </Implies>
      </formula>
    </Assert>
  </act>
```

</RuleML>

Observations from this test case: Following are four aspects extracted from the observations from the above test case which will give a general idea about the working of RON:

1. The canonical ordering of <if> and <then> child nodes is established.
2. The elements are wrapped into the role tags as defined in the above section. E.g.<op>, <arg>.
3. The default values are made explicit as illustrated in the above output.
4. The pretty printing of the output has been done

Following are the URL's of some more test cases against which RON has been tested extensively. Both the links to test cases and results are provided.

Test Case 1: http://ruleml-roc.yolasite.com/resources/RON%20Test%20Case%201.xml

Result 1: http://ruleml-roc.yolasite.com/resources/Result%201.txt

Test Case 2: http://ruleml-roc.yolasite.com/resources/RON%20Test%20Case%202.xml

Result 2: http://ruleml-roc.yolasite.com/resources/Result%202.txt

Test Case 3: http://ruleml-roc.yolasite.com/resources/RON_TestCase3.xml

Result 3: http://ruleml-roc.yolasite.com/resources/Result3.txt

Test Case 4: http://ruleml-roc.yolasite.com/resources/RON%20TestCase4.xml

Result 4: http://ruleml-roc.yolasite.com/resources/Result4.txt

Test Case 5: http://ruleml-roc.yolasite.com/resources/RON%20TestCase5.txt

Result 5: http://ruleml-roc.yolasite.com/resources/Result5.txt

3. RuleML Official Compactifier (ROC)

What is ROC?

ROC stands for RuleML Official Compactifier inverting aspect two of RON, ROC converts a striped form of an XML document to stripe-skipped form. Hence ROC omits the roles, edges and produces a pure form of XML document where we have only the relevant information. The aspect one and four of the RON need not to be inverted because canonical ordering is done using the same technique in ROC. Also in ROC pretty printing is done using the same technique as in RON so we need not invert phase four of RON also. The aspect three of RON needs to be inverted because we have skipped the default value attributes in ROC output.

3.1 ROC Implementation

The development of this project has been divided into different parts. The output of each part will be used as input for the next, and the parts' composition will give us the stripe-skipped form of the XML document.

Part A: Canonical Ordering (cf. 1)

The subelements of the following elements have been put in canonical order. An explanation is provided for each element, detailing the canonical order of that element's subelements.

1. <Retract> is a performative component. It mainly can contain zero or more formulas. According to the schema it may next contain an entailment or Rulebase declaration. However, in the first phase, each Rulebase and Entails element that is a subelement of the Retract tag gets wrapped with a formula role tag. Therefore the canonical order of the subelements of Retract is: oid, formula subelements (in the order that they appear), followed by all other subelements in the order that they appear.

2. <Query> is a performative component. It mainly contains zero or more formulas. According to the schema it may next contain an entailment or Rulebase declaration. However, in the first phase each Rulebase and Entails elements that is a subelements of the Query tag gets wrapped with a formula role tag. Therefore the canonical order of the subelements of Query is: oid, formula subelements (in the order that they appear), followed by all other subelements in the order that they appear.

3. <Entails> is a connective component. It mainly contains an <if> and a <then> tag. The <if> of an Entails element contains a single Rulebase (and thus the canonical order of its subelements is irrelevant). Similarly the <then> of an Entails element contains a single Rulebase. Therefore the canonical order of the subelements of Entails is: oid, if, then, followed by all other subelements in the order that they appear.

4. <Rulebase> is a connective component. It mainly contains zero or more formulas. Therefore, the canonical order of subelements for Rulebase is: oid, formula subelements (in the order that they appear), followed by all other subelements in the order that they appear.

5. <Exists> is a quantification component. It mainly contains one or more variables (<Var>), which may be surrounded by a <declare> role, followed by a formula. In the first phase of the normalization, each variable will be wrapped with a <declare> role tag except for the last

child which will be wrapped with a <formula> role tag. Therefore, the schema tells us that the canonical ordering for the subelements of Exist is: oid, declare subelements (in the order that they appear), formula.

6. <Forall> is a quantification component. It mainly contains one or more variables (<Var>), which may be surrounded by a <declare> role, followed by a formula. In the first phase of the normalization, each variable will be wrapped with a <declare> role tag except for the last child which will be wrapped with a <formula> role tag. Therefore, the schema tells us that the canonical ordering for the subelements of Forall is: oid, declare subelements (in the order that they appear), formula.

7. <Implies> is a connective component. Implies mainly contains an <if> tag and a <then> tag. Therefore the canonical ordering for the subelements of Implies is: oid, if, then, followed by all other subelements in the order that they appear.

8. <Equivalent> is a connective component. It mainly contains a pair of <torso> roles. Therefore, the canonical ordering of the subelements of Equivalent is: oid, torso (in the order that they appear).

9. <And> is a connective component. It mainly contains zero or more formulas. Therefore, the canonical ordering of the subelements of And is: oid, formula subelements (in the order that they appear).

10. <Or> is a connective component. It mainly contains zero or more formulas. Therefore, the canonical ordering of the subelements of Or is: oid, formula subelements (in the order that they appear).

11. <Atom> is an atomic component. It mainly contains a relation (<Rel>) that can be surrounded by an operator (<op>) role. In the first phase of the normalization each relation gets wrapped by an <op> tag, thus this will be true for the relation of each Atom element. For the subelements to achieve canonical order we must have the operation first, then all positional arguments, followed by all slotted arguments. Also, each <arg> tag has an attribute called index which is a positive-integer value. Therefore the canonical ordering of the subelements of Atom is: oid, op, arg subelements (sorted by index), repo, slot subelements, resl, followed by all other subelements in the order that they appear.

12. <Equal> is an equality component. It mainly consists of two expressions which can be (and will be, because of the added role tags in phase 1) surrounded by a left (<left>) or a right (<right>) role tag. Therefore the canonical ordering of the subelements of Equal is: oid, left, right.

13. <Neg> is a strong negation component. It mainly contains a logical atom (<Atom>) that can be (and will be, because of the added role tags in phase 1) surrounded by a role. Therefore the canonical ordering of the subelements of Neg is: oid, strong.

14. <Naf> is a weak negation component. It mainly contains a logical atom (<Atom>) that can be (and will be, because of the added role tags in phase 1) surrounded by a <weak> role tag. Therefore the canonical ordering for the subelements of Naf is: oid, weak.

15. <Expr> is an expression component. It consists of a function (<Fun>) that will be surrounded by an <op> tag as a result of the first phase of our normalization. The <op> tag will be followed by a sequence of arguments each of which will have an attribute called index that will be the positive-integer index of the argument. There may also be user-defined slots

(<slot>). Therefore, the canonical ordering of the subelements of Expr is: oid, op, arg subelements (sorted by index), repo, slots subelements, resl, followed by all other subelements in the order in which they appear.

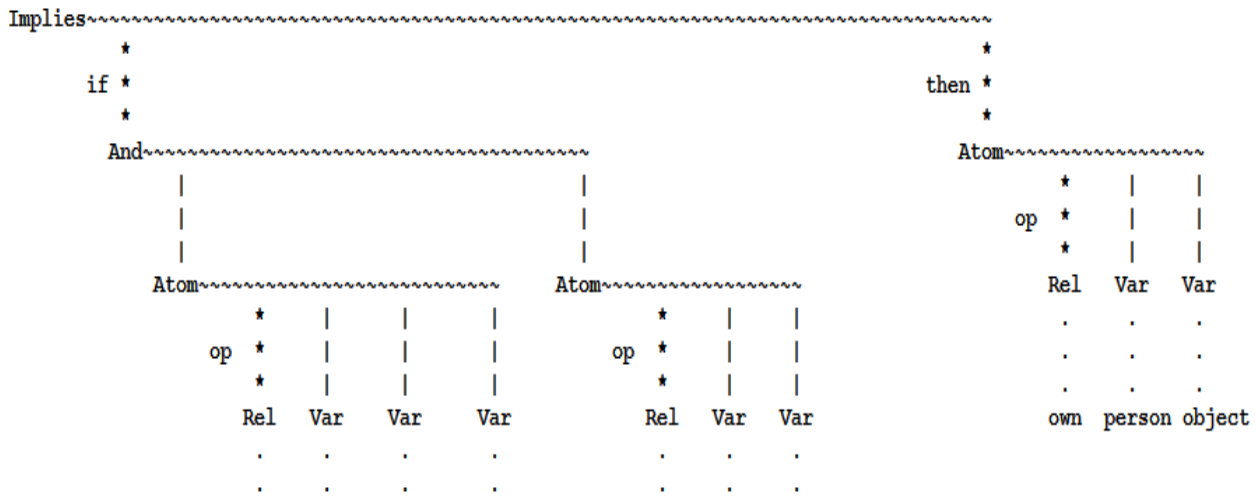
16. <Plex> is a generalized list component. It consists of a collection of ordered arguments (which will be surrounded by ordered argument tags (<arg>) as a result of phase 1), as well as user-defined slots. Each <arg> tag will have a positive-integer attribute called index. Therefore, the canonical ordering of the subelements of Plex is: oid, arg subelements (sorted by index), repo, slot subelements, resl.

Canonical ordering Implementation (cf. 1)

To implement the canonical ordering for each element we need to check the order of each element to determine if we need to put each element in the correct canonical order.

After that we build the canonical order for each element. For example, Implies mainly contains an <if> tag and a <then> tag. Therefore, the canonical ordering for the subelements of Implies is: oid, if, then, followed by all other sub elements in the order that they appear.

```
<!-- Builds canonically-ordered content of Implies. -->
  <xsl:template match="r:Implies" mode="phase-1">
    <xsl:copy>
      <xsl:apply-templates select="@*"/>
      <xsl:apply-templates
        select="*[namespace-uri(.)!='http://ruleml.org/spec']" mode="phase-1"/>
      <xsl:apply-templates select="r:meta" mode="phase-1"/>
      <xsl:apply-templates select="*[namespace-uri(.)='http://ruleml.org/spec' and
        (local-name()!= 'meta' and local-name()!= 'if'
        and local-name()!= 'then')]" mode="phase-1"/>
      <xsl:apply-templates select="r:if" mode="phase-1"/>
      <xsl:apply-templates select="r:then" mode="phase-1"/>
    </xsl:copy>
  </xsl:template>
```



Similarly when we take Atom it has the canonical ordering of an <Atom> element should be <op>, <arg index ="1">, <slot>, <slot>. Also, the <then> element precedes the <if> element which is not the proper canonical ordering so these elements will be switched so that the <if> element precedes the <then> element.

```
<!-- Builds canonically-ordered content of Atom. -->
<xsl:template match="r:Atom" mode="phase-1">
  <xsl:copy>
    <xsl:apply-templates select="@*"/>
    <xsl:apply-templates
      select="*[namespace-uri(.)!='http://ruleml.org/spec']" mode="phase-1"/>
    <xsl:apply-templates select="r:meta" mode="phase-1"/>
    <xsl:apply-templates select="@*|*[
      namespace-uri(.)='http://ruleml.org/spec' and
      local-name()!= 'meta' and
      local-name()!= 'oid' and
      local-name()!= 'degree' and
      local-name()!= 'op' and
      local-name()!= 'arg' and
      local-name()!='repo' and
      local-name()!='slot' and
      local-name()!='resl']" mode="phase-1"/>
    <xsl:apply-templates select="r:oid" mode="phase-1"/>
    <xsl:apply-templates select="r:degree" mode="phase-1"/>
    <xsl:apply-templates select="r:op" mode="phase-1"/>
    <xsl:apply-templates select="r:arg" mode="phase-1"/>
    <xsl:apply-templates select="r:repo" mode="phase-1"/>
    <xsl:apply-templates select="r:slot" mode="phase-1"/>
    <xsl:apply-templates select="r:resl" mode="phase-1"/>
  </xsl:copy>
</xsl:template>
```

Example of **Unordered form** of data:

```
- <Implies>
    - <then>
        - <Atom>
            <Rel>sells</Rel>
            <Var>vendor</Var>
            <Var>food</Var>
          </Atom>
      </then>
    - <if>
        - <Atom>
            <Rel>husband</Rel>
          - <slot>
              <Ind>wife</Ind>
              <Ind>Kathrina</Ind>
            </slot>
            <Ind>Adam</Ind>
          - <slot>
              <Data>seller</Data>
              <Var>vendor</Var>
            </slot>
          </Atom>
      </if>
  </Implies>
```

After Canonical ordering and skipping of role tags: The output will be as follows. Please note that this is not the final output of ROC. This is an intermediate form, just to show the canonical ordering. The final output of this is shown in the end of Part B.

```
- <Implies mapDirection="bidirectional" mapMaterial="yes" direction="bidirectional" material="yes">
  - <if>
    - <Atom>
      - <op>
          <Rel>husband</Rel>
        </op>
      - <arg index="1">
          <Ind>Adam</Ind>
        </arg>
      - <slot>
          <Ind>wife</Ind>
          <Ind>Kathrina</Ind>
        </slot>
      - <slot>
          <Data>seller</Data>
          <Var>vendor</Var>
        </slot>
      </Atom>
    </if>
  - <then>
    - <Atom>
      - <op>
          <Rel>sells</Rel>
        </op>
      - <arg index="1">
          <Var>vendor</Var>
        </arg>
      - <arg index="2">
          <Var>food</Var>
        </arg>
      </Atom>
    </then>
</Implies>
```

Part B: Removal of Stripes

This part of the project requires a canonically ordered form of document so that it can find stripes which we need to skip. After the application of this phase onto the XML document, this project will give us the data but in the most compact form or the stripe-skipped form.

Methodology

In this part we have an already ordered structure "a canonically ordered XML document" the syntax is checked for edge and stripes using the XSLT style sheet. Since the document is already in canonically ordered form, now we have to look for the role tags in which the elements have been wrapped. We need to find the wrapping role tags and skip them to make our elements stripe skipped.

Major Tags

For the use of removing stripes we have created a template "**copy-1**". This template is responsible for copying selected data to output.

```
<xsl:template match="node() | @*" mode="phase-2">
    <xsl:call-template name="copy-1"/>
  </xsl:template>

  <xsl:template name="copy-1">
```

```
    <xsl:copy>
      <xsl:apply-templates select="node() | @*" mode="phase-2"/>
    </xsl:copy>
  </xsl:template>
```

This template matches all the nodes recursively and gets applied to each and every node recursively. The purpose of creating this template is for the modularity and simplicity of the code. With the help of this snippet, whenever we need to remove some tags, we can remove it by calling this template to copy the child nodes to output.

Removal of Each Element's Role Tags

The following elements are checked to ensure that their children are structured as it is defined in RuleML. As soon as ROC finds the elements and the sub elements, it removes the edges and except for the edges it copies everything present inside the tag using the template call of "copy-1".

Technically, when this template is called in the XML document, it starts by selecting the root or the start and matches every element node till the end of the document. After the complete traversal it removes the role, edge stripes and copies all the other data for output.

Functioning of Copy-1 template

The function of copy-1 template is to copy the elements wrapped inside the role tags/ edges to the output. Local test is performed at each node recursively and when the role tag for which the local test has been performed is found then the copy-1 template is called which copies the elements which has been wrapped inside the role tag/ edge under local test condition, so this is how we skip the role tags/ edges using copy-1 template.

1. <RuleML> is checked for the <act> stripe from the start of <RuleML> and till its end. Inside this node we will call **copy-1 template.** If the name stripe <act> is present inside this node, copy template will copy all the children that were wrapped inside this tag to the next phase except for the stripe itself. This will happen for all occurrences of <act> stripe recursively.

```
<xsl:template match="r:RuleML/*[namespace-uri(.)='http://ruleml.org/spec']" mode="phase-2">
    <xsl:if test="local-name()='act'">
      <xsl:for-each select="./*">
        <xsl:call-template name="copy-1" />
      </xsl:for-each>
    </xsl:if>
  </xsl:template>
```

2. <Assert> is checked for the <formula> edge element from the start of <Assert> and till its end. Inside this node we will call **copy-1 template.** If the name stripe <formula> is found inside this node, copy template will copy all the children that were wrapped inside this tag to

the next phase except for the stripe itself. This will happen for all occurrences of <formula> stripe.

```
<xsl:template match="r:Assert/*[namespace-uri(.)='http://ruleml.org/spec']" mode="phase-2">
      <xsl:if test="local-name()='formula'">
        <xsl:for-each select="./*">
          <xsl:call-template name="copy-1" />
        </xsl:for-each>
      </xsl:if>
  </xsl:template>
```

3. <Rulebase> is checked for the <formula> edge element from the start of <Rulebase> and till its end. Inside this node we will call **copy-1 template.** If the name stripe <formula> is found inside this node, copy template will copy all the children that were wrapped inside this tag to the next phase except for the stripe itself. This will happen for all occurrences of the <formula> stripe.

```
<xsl:template match="r:Rulebase/*[namespace-uri(.)='http://ruleml.org/spec']" mode="phase-2">
      <xsl:if test="local-name()='formula'">
        <xsl:for-each select="./*">
          <xsl:call-template name="copy-1" />
        </xsl:for-each>
      </xsl:if>
  </xsl:template>
```

4. <Equivalent> is checked for the <torso> edge element from the start of <Equivalent> and till its end. Inside this node we will call **copy-1 template.** If the name stripe <torso> is found inside this node, copy template will copy all the children that were wrapped inside this tag to the next phase except for the stripe itself. This will happen for all occurrences of the <torso> stripe.

```
<xsl:template match="r:Equivalent/*[namespace-uri(.)='http://ruleml.org/spec']" mode="phase-2">
      <xsl:if test="local-name()='torso'">
        <xsl:for-each select="./*">
          <xsl:call-template name="copy-1" />
        </xsl:for-each>
      </xsl:if>
  </xsl:template>
```

5. <Retract> is checked for the <formula> edge element from the start of <Retract> and till its end. Inside this node we will call **copy-1 template.** If the name stripe <formula> is found inside this node, copy template will copy all the children that were wrapped inside this tag to the next phase except for the stripe itself. This will happen for all occurrences of the <formula> stripe.

```
  <xsl:template match="r:Retract/*[namespace-uri(.)='http://ruleml.org/spec']" mode="phase-2">
      <xsl:if test="local-name()='formula'">
        <xsl:for-each select="./*">
          <xsl:call-template name="copy-1" />
        </xsl:for-each>
        </xsl:if>
  </xsl:template>
```

6. <Query> is checked for the <formula> edge element from the start of <Query> and till its end. Inside this node we will call **copy-1 template.** If the name stripe <formula> is found inside this node, copy template will copy all the children that were wrapped inside this tag to the next phase except for the stripe itself. This will happen for all occurrences of the the <formula> stripe.

```
<xsl:template match="r:Query/*[namespace-uri(.)='http://ruleml.org/spec' ]" mode="phase-2">
      <xsl:if test="local-name()='formula'">
        <xsl:for-each select="./*">
          <xsl:call-template name="copy-1" />
        </xsl:for-each>
      </xsl:if>
  </xsl:template>
```

7. <And> is checked for the <formula> edge element from the start of <And> and till its end. Inside this node we will call **copy-1 template.** If the name stripe <formula> is found inside this node, copy template will copy all the children that were wrapped inside this tag to the next phase except for the stripe itself. This will happen for all occurrences of the <formula> stripe.

```
  <xsl:template match="r:And/*[namespace-uri(.)='http://ruleml.org/spec']"  mode="phase-2">
    <xsl:if test="local-name()='formula'">
      <xsl:for-each select="./*">
        <xsl:call-template name="copy-1" />
      </xsl:for-each>
    </xsl:if>
  </xsl:template>
```

8. <Or> is checked for the <formula> edge element from the start of <Or> and till its end. Inside this node we will call **copy-1 template.** If the name stripe <formula> is found inside this node, copy template will copy all the children that were wrapped inside this tag to the next phase except for the stripe itself. This will happen for all occurrences of the <formula> stripe.

```
<xsl:template match="r:Or/*[namespace-uri(.)='http://ruleml.org/spec']" mode="phase-2">
      <xsl:if test="local-name()='formula'">
        <xsl:for-each select="./*">
          <xsl:call-template name="copy-1" />
        </xsl:for-each>
      </xsl:if>
  </xsl:template>
```

9. <Neg> is checked for the and <formula> edge element from the start of <Neg> and till its end. Inside this node we will call **copy-1 template.** If the name stripe or <formula> is found inside this node, copy template will copy all the children that were wrapped inside this tag to the next phase except for the stripe itself. This will happen for all occurrences of the and the <formula> stripe.

```
<xsl:template match="r:Neg/*[namespace-uri(.)='http://ruleml.org/spec' ]" mode="phase-2">
      <xsl:if test="local-name()='strong' or local-name()='formula'">
        <xsl:for-each select="./*">
```

```
            <xsl:call-template name="copy-1" />
          </xsl:for-each>
        </xsl:if>
    </xsl:template>
```

10. <Naf> is checked for the <weak> and <formula> edge element from the start of <Naf> and till its end. Inside this node we will call **copy-1 template.** If the name stripe <weak> or <formula> is found inside this node, copy template will copy all the children that were wrapped inside this tag to the next phase except for the stripe itself. This will happen for all occurrences of the <weak> and the <formula> stripe.

```
<xsl:template match="r:Naf/*[namespace-uri(.)='http://ruleml.org/spec' ]" mode="phase-2">
      <xsl:if test="local-name()='weak'  or local-name()='formula'">
        <xsl:for-each select="./*">
          <xsl:call-template name="copy-1" />
        </xsl:for-each>
      </xsl:if>
    </xsl:template>
```

11. <Equal> is checked for the <left> and <right> edge element from the start of <Equal> and till its end. Inside this node we will call **copy-1 template.** If the name stripe <left> or <right> is found inside this node, copy template will copy all the children that were wrapped inside this tag to the next phase except for the stripe itself. This will happen for all occurrences of the <left> and the <right> stripe.

```
<xsl:template match="r:Equal/*[namespace-uri(.)='http://ruleml.org/spec']" mode="phase-2">
      <xsl:if test="local-name()='left' or local-name()='right'">
        <xsl:for-each select="./*">
          <xsl:call-template name="copy-1" />
        </xsl:for-each>
      </xsl:if>
    </xsl:template>
```

12. <Atom> is checked for the <op> and <arg> edge element from the start of <Atom> and till its end. Inside this node we will call **copy-1 template.** If the name stripe <op> or <arg> is found inside this node, copy template will copy all the children that were wrapped inside this tag to the next phase except for the stripe itself. This will happen for all occurrences of the <op> and the <arg> stripe.

```
    <xsl:template match="r:Atom/*[namespace-uri(.)='http://ruleml.org/spec' ]" mode="phase-2">
      <xsl:if test="local-name()='op' or local-name()='arg' ">
        <xsl:for-each select="./*">
          <xsl:call-template name="copy-1" />
        </xsl:for-each>
      </xsl:if>
    </xsl:template>
```

13. <Implies> is checked for the <if> and <then> edge element from the start of <Implies> and till its end. Inside this node we will call **copy-1 template.** If the name stripe <if> or <then> is found inside this node, copy template will copy all the children that were wrapped inside this tag to the next phase except for the stripe itself. This will happen for all occurrences of the <if> and <then> stripe.

```
<xsl:template match="r:Implies/*[namespace-uri(.)='http://ruleml.org/spec']" mode="phase-2">
    <xsl:if test="local-name()='if' or local-name()='then'">
      <xsl:for-each select="./*">
        <xsl:call-template name="copy-1" />
      </xsl:for-each>
    </xsl:if>
  </xsl:template>
```

14. <Forall> is checked for the <declare> and <formula> edge element from the start of <Forall> and till its end. Inside this node we will call **copy-1 template.** If the name stripe <declare> or <formula> is found inside this node, copy template will copy all the children that were wrapped inside this tag to the next phase except for the stripe itself. This will happen for all occurrences of the <declare> and <formula> stripe.

```
<xsl:template match="r:Forall/*[namespace-uri(.)='http://ruleml.org/spec']" mode="phase-2">
    <xsl:if test="local-name()='declare' or local-name()='formula'">
      <xsl:for-each select="./*">
        <xsl:call-template name="copy-1" />
      </xsl:for-each>
    </xsl:if>
```

15. <Exists> is checked for the <declare> and <formula> edge element from the start of <Exists> and till its end. Inside this node we will call **copy-1 template.** If the name stripe <declare> or <formula> is found inside this node, copy template will copy all the children that were wrapped inside this tag to the next phase except for the stripe itself. This will happen for all occurrences of the <declare> and <formula> stripe.

```
<xsl:template match="r:Exists/*[namespace-uri(.)='http://ruleml.org/spec']" mode="phase-2">
    <xsl:if test="local-name()='declare' or local-name()='formula'">
          <xsl:for-each select="./*">
            <xsl:call-template name="copy-1" />
          </xsl:for-each>
          </xsl:if>
  </xsl:template>
```

16. <Entails> is checked for the <if> and <then> edge element from the start of <Entails> and till its end. Inside this node we will call **copy-1 template.** If the name stripe <if> or <then> is found inside this node, copy template will copy all the children that were wrapped inside this tag to the next phase except for the stripe itself. This will happen for all occurrences of the <if> and <then> stripe.

```
<xsl:template match="r:Entails/*[namespace-uri(.)='http://ruleml.org/spec' ]" mode="phase-2">

    <xsl:if test="local-name()='if' or local-name()='then'">
      <xsl:for-each select="./*">
        <xsl:call-template name="copy-1" />
      </xsl:for-each>
    </xsl:if>
  </xsl:template>
```

17. <Expr> is checked for the <op> and <arg> edge element from the start of <Expr> and till its end. Inside this node we will call **copy-1 template.** If the name stripe <op> or <arg> is found inside this node, copy template will copy all the children that were wrapped inside this tag to the next phase except for the stripe itself. This will happen for all occurrences of the <op> and <arg> stripe.

```
<xsl:template match="r:Expr/*[namespace-uri(.)='http://ruleml.org/spec']"  mode="phase-2">

    <xsl:if test="local-name()='op' or local-name()='arg'">
      <xsl:for-each select="./*">
        <xsl:call-template name="copy-1" />
      </xsl:for-each>
    </xsl:if>
  </xsl:template>
```

18. <Plex> is checked for the <oid>, <op>, <arg>, <degree>, <repo> and <resl> edge element from the start of <Plex> and till its end. Inside this node we will call **copy-1 template.** If the name stripe <oid>, <op>, <arg>, <degree>, <repo> and <resl> is found inside this node, copy template will copy all the children that were wrapped inside this tag to the next phase except for the stripe itself. This will happen for all occurrences of the <oid>, <op>, <arg>, <degree>, <repo> and <resl> stripe.

```
<xsl:template match="r:Plex/*[namespace-uri(.)='http://ruleml.org/spec']" mode="phase-2">
<xsl:if test="local-name()='oid' or local-name()='op' or local-name()= 'arg' or local-name()='degree' or
local-name()='repo' or local-name()='resl'">
        <xsl:for-each select="./*">
          <xsl:call-template name="copy-1" />
        </xsl:for-each>
    </xsl:if>
  </xsl:template>
```

Final output of the test case described in Part A

```
<?xml version="1.0" encoding="UTF-8"?>
<RuleML xmlns="http://ruleml.org/spec"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" schemaLocation="http://ruleml.org/spec  http://www.ruleml.org/1.0/xsd/datalog.xsd">
 <Assert mapClosure="universal">
 <Implies material="yes"  direction="bidirectional" mapMaterial="yes"
mapDirection="bidirectional">
    <Atom>
     <Rel>husband</Rel>
     <Ind>Adam</Ind>
     <slot><Ind>wife</Ind><Ind>Kathrina</Ind></</slot>
     <slot><Data>seller</Data><Var>vendor</Var></slot>
    </Atom>
    <Atom>
     <Rel>sells</Rel>
     <Var>vendor</Var>
     <Var>food</Var>
    </Atom>
   </Implies>
  </Assert>
</RuleML>
```

Part C: Pretty Print (cf. 1)
The pretty print phase was developed by a team in 2011. The comments in the output were not formatted properly so this phase was coded to get a proper formatting of the output. The comments which were long often run off the page and were not visible in the output. Also there was no new line before or after the comments. Hence in order to address the mentioned issues in formatting this phase was developed.

The third issue was that for some tags the standard formatting for RuleML does not have a new line after the open tag or before the close tag. The default output obtained was hard to understand by humans.

```
<op>
        <Rel>haschild</Rel>
</op>
```

Pretty print created a formatting that was much easier to read, as shown:

```
 <op><Rel>haschild</Rel></op>
```

The pretty printing was obtained using variables and parameters in XSLT. Using the XLST text command, which forces the output to have exactly the text inside it, we defined two variables: one containing a new line and the other containing the amount of space desired for one tag. The pseudo-code for a generic pretty print output is as follows:

```
template:
        parameter tabs
        output newline
        output tabs
        copy:
                for each attribute:
                        output the attribute
                choose:
                        when this node has no children:
                                output value of this node
                        otherwise:
                                apply templates with tabs = tab + tabs
                                output new line
                                output tabs
```

To give an appropriate space to a new element we use tab parameter of XSLT. It's default value is empty which we can give according to our own requirement. When there are children we also output a new line and the tab spacing before outputting the closing tag. We do not do this in the case when there are no children because that would result in output that is undesirable, as shown:

```
<Rel>relation
</Rel>
```

A new line was added before and after each comment by coding a new template which when detects a comment, automatically adds a new line before and after it.

For some tags spacing was done using the following technique. Two different templates were coded namely “new-line” and “no- new-line”. The no-new-line templates takes two parameter tabs , and newlines. The newlines parameter is used to determine whether we want a new-line or not. The pseudo-code for the above mentioned templates is as follows:

```
template match *:
        parameter tabs
        choose:
                when locale-name is 'op', 'arg', or 'slot':
                        call template no-new-line with newlines = yes and
            tabs
                otherwise:
                        call template new-line with tabs

template no-new-line:
        parameter newlines
        parameter tabs
        if newlines = yes:
                output new line
                output tabs
        copy:
                for each attribute:
                        output the attribute
                choose:
                        when no children:
                                output node value
                        otherwise:
                                for each child node:
                                        call template no-new-lines with newlines
                  = no
```

We have to give a value to new-line template i.e. yes or no. If we give the value yes then a new line and tab spacing is outputted. Hence we get the output in pretty print format as follows:
<op><Rel>relation</Rel></op>

3.2 Testing ROC

Test Case -1

The test case contains elements that have already been wrapped in role tags. The test case contains rules to indicate whether a person is teacher or not. The first rule is a <Entails > statement and states if person owns a lab and person teaches subjects, then person is a teacher. The next rule is if the person is a student and person studies physics, then the person studies subject.

First ROC unwrapped the child nodes of the element <Assert> properly in the role tag <formula>. The test case also shows how the stylesheet allows elements to be present directly following the parent <Implies>. In the test case the last child is wrapped in <then> which means ROC will unwrap the second to last child in <if>. Inside the element <Implies>, the children of the element <And> are properly wrapped in <formula>. ROC also unwraps the children of <Atom> correctly by unwrapping <Rel> in <op> and <Var> in <arg> and transforms it in fully stripe -skipped form.

Input:

```
<?xml version="1.0" encoding="UTF-8"?>
<RuleML xmlns="http://ruleml.org/spec"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://ruleml.org/spec http://www.ruleml.org/1.0/xsd/datalog.xsd">
<!--  If X owns a Lab and X teaches subjects, then X is a Teacher -->
<act index="1">
   <Assert>
      <!-- A rule and two premise facts entail a conclusion fact-->
      <formula>
         <Entails>
            <if>
               <Rulebase>
                  <formula>
                     <Implies material="yes" direction="bidirectional" mapMaterial="yes"
mapDirection="bidirectional">
                        <if>
                           <And>
                              <formula>
                                 <Atom>
                                    <op>
                                       <Rel>owns</Rel>
                                    </op>
                                    <arg index="1">
                                       <Var>Person</Var>
                                    </arg>
                                    <arg index="2">
                                       <Var>Lab</Var>
                                    </arg>
                                 </Atom>
                              </formula>
                              <formula>
                                 <Atom>
                                    <op>
                                       <Rel>Physics</Rel>
                                    </op>
                                    <arg index="1">
                                       <Var>Lab</Var>
                                    </arg>
                                 </Atom>
```

```
                </formula>
              </And>
            </if>
            <then>
              <Atom>
                <op>
                  <Rel>teaches</Rel>
                </op>
                <arg index="1">
                  <Var>Person</Var>
                </arg>
                <arg index="2">
                  <Ind>subjects</Ind>
                </arg>
              </Atom>
            </then>
          </Implies>
        </formula>
        <formula>
          <Atom>
            <op>
              <Rel>owns</Rel>
            </op>
            <arg index="1">
              <Ind>X</Ind>
            </arg>
            <arg index="2">
              <Ind>Physics Lab</Ind>
            </arg>
          </Atom>
        </formula>
        <formula>
          <Atom>
            <op>
              <Rel>teaches</Rel>
            </op>
            <arg index="1">
              <Ind>physics</Ind>
            </arg>
          </Atom>
        </formula>
      </Rulebase>
    </if>
    <then>
      <Rulebase>
        <formula>
          <Atom>
            <op>
              <Rel>teaches</Rel>
            </op>
```

```
                <arg index="1">
                   <Ind>X</Ind>
                </arg>
                <arg index="2">
                   <Ind>students</Ind>
                </arg>
             </Atom>
          </formula>
       </Rulebase>
    </then>
 </Entails>
</formula>
<!--  The facts that Y is a Student and Y studies Physics entail that Y studies subjects -->
<formula>
   <Entails>
      <if>
         <Rulebase>
            <formula>
               <Atom>
                  <op>
                     <Rel>Student</Rel>
                  </op>
                  <arg index="1">
                     <Ind>Y</Ind>
                  </arg>
               </Atom>
            </formula>
            <formula>
               <Atom>
                  <op>
                     <Rel>Studies</Rel>
                  </op>
                  <arg index="1">
                     <Ind>Y</Ind>
                  </arg>
                  <arg index="2">
                     <Ind>Subjects</Ind>
                  </arg>
               </Atom>
            </formula>
         </Rulebase>
      </if>
      <then>
         <Rulebase>
            <formula>
               <Atom>
                  <op>
                     <Rel>buys</Rel>
                  </op>
                  <arg index="1">
```

```
                        <Ind>Jill</Ind>
                    </arg>
                    <arg index="2">
                        <Ind>produce</Ind>
                    </arg>
                </Atom>
            </formula>
        </Rulebase>
    </then>
</Entails>
</formula>
</Assert>
</act>
</RuleML>
```

Output:

```
<?xml version="1.0" encoding="UTF-8"?>
<RuleML xmlns="http://ruleml.org/spec"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
schemaLocation="http://ruleml.org/spec http://www.ruleml.org/1.0/xsd/datalog.xsd">
<!--  If X owns a Lab and X teaches subjects, then X is a Teacher -->
<Assert>
  <!-- A rule and two premise facts entail a conclusion fact-->
  <Entails>
    <Rulebase>
        <And>
          <Atom>
            <Rel>owns</Rel>
            <Var>Person</Var>
            <Var>Lab</Var>
          </Atom>
          <Atom>
            <Rel>Physics</Rel>
            <Var>Lab</Var>
          </Atom>
        </And>
        <Atom>
          <Rel>teaches</Rel>
          <Var>Person</Var>
          <Ind>subjects</Ind>
        </Atom>
      </Implies>
      <Atom>
        <Rel>owns</Rel>
        <Ind>X</Ind>
        <Ind>Physics Lab</Ind>
      </Atom>
      <Atom>
        <Rel>teaches</Rel>
```

```
            <Ind>physics</Ind>
          </Atom>
        </Rulebase>
        <Rulebase>
          <Atom>
            <Rel>teaches</Rel>
            <Ind>X</Ind>
            <Ind>students</Ind>
          </Atom>
        </Rulebase>
      </Entails>
      <!--  The facts that Y is a Student and Y studies Physics entail that Y studies subjects -->
      <Entails>
        <Rulebase>
          <Atom>
            <Rel>Student</Rel>
            <Ind>Y</Ind>
          </Atom>
          <Atom>
            <Rel>Studies</Rel>
            <Ind>Y</Ind>
            <Ind>Subjects</Ind>
          </Atom>
        </Rulebase>
        <Rulebase>
          <Atom>
            <Rel>buys</Rel>
            <Ind>Jill</Ind>
            <Ind>produce</Ind>
          </Atom>
        </Rulebase>
      </Entails>
    </Assert>
    </RuleML>
```

Observations from this test case: Following are four aspects extracted from the observations from the above test case which will give a general idea about the workings of ROC. The numbering of the steps mentioned below is in accordance with the phases of RON. Since we do not skip the default value attributes we do not mention step 3 below:

Step 1: The output is in canonical order. The order check is performed as first step in ROC.

Step 2: The role tags/edges in the input i.e. <formula>, <if>, <then>, <op> and <arg> has been skipped and output is in fully compact Stripe-skipped form.

Step 4: The pretty printing of the output has been done.

Test Case - 2

The test case contains elements that have already been wrapped in role tags. The test case contains rules to indicate whether a person is father or not. The first rule is a <Forall > statement and states If a male person has a child then he is a father. The next rule is a <Rulebase> statement that indicates that the dad relation and father relation are equivalent.

First, ROC transforms the test case, the children of Assert will be unwrapped in formula role tags. The children of Forall will be unwrapped by skipping the declare role tag. Inside Forall's subelement <Implies> the first <Atom> tag will be unwrapped by skipping the <if> and then>. Inside those <Atom> elements, <Rel> will be unwrapped by skipping <op> and <Var> will be unwrapped by skipping <arg> role tags that include an index attribute value. Here, phase 1 does not alter the <slot> elements. In the <Rulebase> statement, each of its subelements will be unwrapped by skipping the <formula> role tag. In the subelement <Equivalent> each of its children is unwrapped by skipping the <torso> role tag.

Input:

```
<?xml version="1.0" encoding="UTF-8"?>
<RuleML xmlns="http://ruleml.org/spec"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://ruleml.org/spec http://www.ruleml.org/1.0/xsd/datalog.xsd">
<act index="1">
   <Assert mapClosure="universal">
      <!-- If a male person has a child then he is a father -->
      <formula>
         <Forall>
            <declare>
               <Var type="Male">person</Var>
            </declare>
            <declare>
               <Var>child</Var>
            </declare>
            <formula>
               <Implies material="yes" direction="bidirectional" mapMaterial="yes"
mapDirection="bidirectional">
                  <if>
                     <Atom>
                        <op>
                           <Rel>haschild</Rel>
                        </op>
                        <arg index="1">
                           <Var>person</Var>
                        </arg>
                        <arg index="2">
                           <Var>child</Var>
                        </arg>
                     </Atom>
                  </if>
```

```
                <then>
                  <Atom>
                    <op>
                      <Rel>father</Rel>
                    </op>
                    <arg index="1">
                      <Var>person</Var>
                    </arg>
                  </Atom>
                </then>
              </Implies>
            </formula>
          </Forall>
        </formula>
        <!-- The relationship dad is equivalent to the relationship father -->
        <formula>
          <Rulebase>
            <formula>
              <Equivalent>
                <torso>
                  <Atom>
                    <op>
                      <Rel>dad</Rel>
                    </op>
                  </Atom>
                </torso>
                <torso>
                  <Atom>
                    <op>
                      <Rel>father</Rel>
                    </op>
                  </Atom>
                </torso>
              </Equivalent>
            </formula>
          </Rulebase>
        </formula>
      </Assert>
</act>
</RuleML>
```

Output:

```
<?xml version="1.0" encoding="UTF-8"?>
<RuleML xmlns="http://ruleml.org/spec"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
schemaLocation="http://ruleml.org/spec http://www.ruleml.org/1.0/xsd/datalog.xsd">
<Assert mapClosure="universal">
```

```
<!-- If a male person has a child then he is a father -->
<Forall>
   <Var type="Male">person</Var>
   <Var>child</Var>
      <Atom>
         <Rel>haschild</Rel>
         <Var>person</Var>
         <Var>child</Var>
      </Atom>
      <Atom>
         <Rel>father</Rel>
         <Var>person</Var>
      </Atom>
   </Implies>
</Forall>
<!-- The relationship dad is equivalent to the relationship father -->
<Rulebase>
   <Equivalent>
      <Atom>
         <Rel>dad</Rel>
      </Atom>
      <Atom>
         <Rel>father</Rel>
      </Atom>
   </Equivalent>
</Rulebase>
</Assert>
</RuleML>
```

Observations from this test case: Following are four aspects extracted from the observations from the above test case which will give a general idea about the workings of ROC. The numbering of the steps mentioned below is in accordance with the phases of RON. Since we do not skip the default value attributes we do not mention step 3 below:

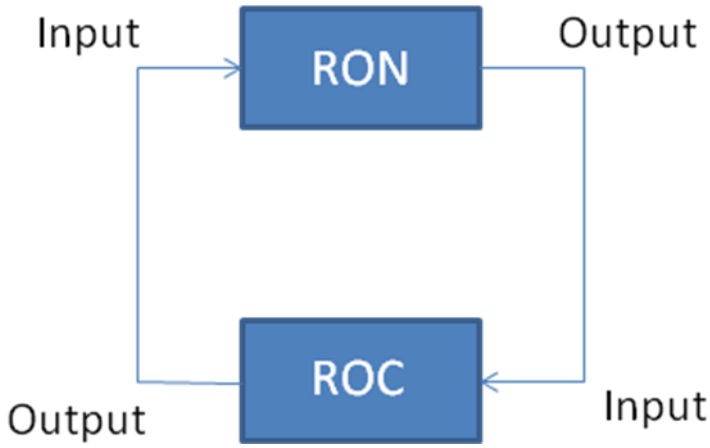
Step 1: The output is in canonical order. The order check is performed as first step in ROC.

Step 2: The role tags/edges in the input i.e. <formula>, <declare>, <if>, <then>, <op> ,<arg> and <torso> has been skipped and output is in fully compact Stripe-skipped form.

Step 4: The pretty printing of the output has been done.

4. Round Tripping

Round-tripping: As described in the diagram round tripping means we start from an input to RON and we get ouput for this input. Then the output obtained in the previous step is given as input to the ROC. The output obtained from the ROC should be same to the input with we started the trip. This complete cycle is known as round tripping.



Test Case - 1

The main function of round tripping is to complete the cycle as shown in the diagram above. We start round tripping from RON and end with ROC output.

ROC Input:

```
<?xml version="1.0" encoding="UTF-8"?>
<RuleML xmlns="http://ruleml.org/spec"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://ruleml.org/spec http://www.ruleml.org/1.0/xsd/datalog.xsd">
<act index="1">
   <Assert mapClosure="universal">
      <formula>
         <Implies material="yes" direction="bidirectional" mapMaterial="yes"
mapDirection="bidirectional">
            <if>
               <Atom>
                  <op>
                     <Rel>husband</Rel>
                  </op>
                  <arg index="1">
                     <Ind>Adam</Ind>
                  </arg>
                  <slot>
                     <Ind>wife</Ind>
```

```
                <Ind>Kathrina</Ind>
              </slot>
              <slot>
                <Data>seller</Data>
                <Var>vendor</Var>
              </slot>
            </Atom>
          </if>
          <then>
            <Atom>
              <op>
                <Rel>sells</Rel>
              </op>
              <arg index="1">
                <Var>vendor</Var>
              </arg>
              <arg index="2">
                <Var>food</Var>
              </arg>
            </Atom>
          </then>
        </Implies>
      </formula>
    </Assert>
</act>
</RuleML>
```

ROC Output/RON Input:

```
<?xml version="1.0" encoding="UTF-8"?>
<RuleML xmlns="http://ruleml.org/spec"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
schemaLocation="http://ruleml.org/spec http://www.ruleml.org/1.0/xsd/datalog.xsd">
<Assert mapClosure="universal">

      <Atom>
        <Rel>husband</Rel>
        <Ind>Adam</Ind>
        <slot>
            <Ind>wife</Ind>
             <Ind>Kathrina</Ind>
         </slot>
          <slot>
              <Data>seller</Data>
              <Var>vendor</Var>
          </slot>
      </Atom>
      <Atom>
```

```
        <Rel>sells</Rel>
        <Var>vendor</Var>
        <Var>food</Var>
      </Atom>
   </Implies>
</Assert>
</RuleML>
```

RON Output:

```
<?xml version="1.0" encoding="UTF-8"?>
<RuleML xmlns="http://ruleml.org/spec"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://ruleml.org/spec http://www.ruleml.org/1.0/xsd/datalog.xsd">
<act index="1">
   <Assert mapClosure="universal">
      <formula>
         <Implies material="yes" direction="bidirectional" mapMaterial="yes"
mapDirection="bidirectional">
            <if>
               <Atom>
                  <op>
                     <Rel>husband</Rel>
                  </op>
                  <arg index="1">
                     <Ind>Adam</Ind>
                  </arg>
                  <slot>
                     <Ind>wife</Ind>
                     <Ind>Kathrina</Ind>
                  </slot>
                  <slot>
                     <Data>seller</Data>
                     <Var>vendor</Var>
                  </slot>
               </Atom>
            </if>
            <then>
               <Atom>
                  <op>
                     <Rel>sells</Rel>
                  </op>
                  <arg index="1">
                     <Var>vendor</Var>
                  </arg>
                  <arg index="2">
                     <Var>food</Var>
                  </arg>
               </Atom>
            </then>
```

```
            </Implies>
          </formula>
        </Assert>
    </act>
    </RuleML>
```

Result: As we can see we get the exact same output as our first input. So, by performing round-tripping we are getting the expected results. It shows that for test case 1 both RON an ROC are working fine and are inverses of each other.

5. Conclusion and Future Work

The ROC and RON systems both are effective tools for XML serialization. RON helps us to get the fully normalized version of XML with all the proper role tags added in canonical order. ROC makes the XML document compact e.g. by skipping all the stripes which have been added by RON. The future work includes to embed this XSLT style sheet in a Java code or from any other scripting language. This can be achieved by using W3C online services by calling them from any java code or any other scripting language. Other improvement can be to make this code user-interactive by asking users to select the option whether they want to use ROC or RON according to their requirement.

6. Important Links:

RuleML Official Compactifier :
http://ruleml-roc.yolasite.com

ROC Fully Normalized Test case:
http://ruleml-roc.yolasite.com/resources/ROC_Fully%20Normalized_output.txt

Partial Normalized Test case:
http://ruleml-roc.yolasite.com/resources/ROC_Partial%20Normalized_output.txt

Relaxed Test case:
http://ruleml-roc.yolasite.com/resources/ROC_Relaxed_output.txt

7. References

1.RON
http://v37s3b4h7dn47s37hg1br4h7rs7n3du7s8nu.unbf.ca/~lbidlak1/

2.XSLT 2.0
http://www.xfront.com/rescuing-xslt.html
http://www.cafeconleche.org/books/bible3/chapters/ch15.html
http://www.w3.org/2005/08/online_xslt

3.RuleML
http://ruleml.org/

4.RuleML 1.0 Normalization
http://ruleml.org/1.0/xslt/normalizer/100_normalizer.xslt

5.Oxygen Editor
http://www.oxygenxml.com/