University of New Brunswick

CS6795 - Semantic Web Techniques
Project Report

Dec. 19th, 2011

Normalizers for RuleML 1.0 in XSLT 2.0

Instructor: Dr. Harold Boley
Advisor: Dr. Tara Athan

Team 4:
Nada Alsalmi
Leah Bidlake
Ao Cheng
Thea Gegenberg
Emily Wilson

Table of Contents

1. Introduction......1
1.1Background 2
2. Phased Approach 4
2.1 Upgrade to RuleML 1.0 4
2.2 Phase 1......4
2.2.1 Each Element's Role Tags 5
2.2.2 Implementation......7
2.2.3 Test Cases......7
2.3 Phase 2......9
2.3.1 Each Element's Canonical Order 9
2.3.2 Implementation......11
2.3.3 Test Cases......12
2.4 Phase 3......13
2.4.1 The Tasks of Phase 3......13
2.4.2 Implementation......15
2.5 Pretty Print......15
3. RuleML Official Normalizer (RON) 18
3.1 Overall Implementation......18
3.2 General Test Cases 18
3.2.1 Non-Normalized Test Cases......18
3.2.2 Partially Normalized Test Cases 19
3.2.3 Completely Normalized Test Cases 19
4. Conclusions and Future Work......20
5. References......21
Appendix A......22
Appendix B......23
Appendix C......27
Appendix D......28
Appendix E......29
Appendix F 30
Appendix G......32
Appendix H......34

1. Introduction

RuleML is a markup language for sharing rules in XML[2]. It is serialized as an XML tree whose elements alternate between representing classes or type tags (nodes), and representing methods 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 serialization is RuleML's main rule node <Implies> which can contain <if> and <then> edges. Because of XML's left-to-right ordering we can rely on the subelements' positions to tell us the implied roles of, say, the <if> and <then> edges; therefore we can remove them completely. Such a document with all edges removed is an (extreme) example of stripe-skipping. In the normalization of a RuleML document we want to transform 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.

The tool that we have developed is used to normalize RuleML documents. Ideally, we would like our tool to be able to normalize any RuleML 1.0 document, that is, we would like it to be able to make it applicable to documents that are not normalized at all (stripes missing and not in canonical order), partially normalized (stripes missing or not in canonical order but not both), and ones that are completely normalized (returning those unchanged).

We chose to implement our program in four phases. The first phase fills in the missing edge stripes wherever they are needed in the XML to achieve a fully striped document. The second phase assures that the subelements are in the correct canonical ordering. The third phase makes all attributes that have default values explicit. In the last phase we format our document using Pretty Print. The sequence of our phases is demonstrated in a workflow diagram in Figure 1.

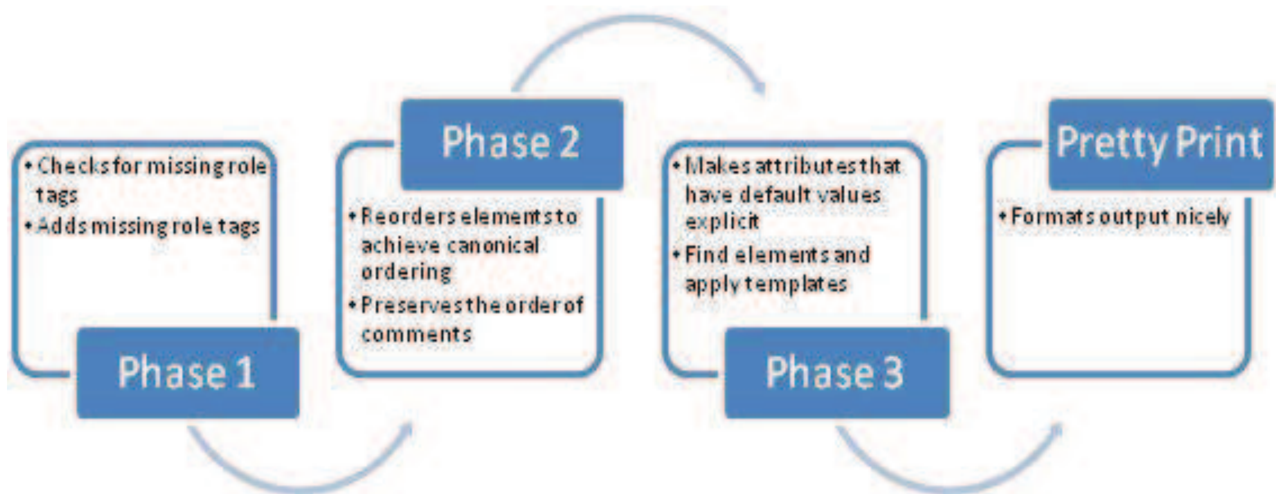

Figure 1: Workflow diagram

1.1 Background

A partially built normalizer for RuleML 0.91 queries has been developed by Dr. Tara Athan using an XSLT stylesheet. The current normalizer uses some features from XSLT 2.0 such as modes. Since our normalizer was divided into three phases we needed a method to be able to run a document through our stylesheet multiple times. This was done through the use of variables and modes. Each phase outputted to an appropriately named XSLT 2.0 variable and the subsequent phase would read from the variable for the previous phase and then apply templates with the mode being the given phase. The modes allowed us to have multiple templates with the same match attribute (i.e. that would be applied to the same piece of the input document) but doing different things since they were in different phases. So for phase 2 we would have

```
<xsl:variable name=”phase-2-output”>
        <xsl:apply-templates select=”$phase-1-output” mode=”phase-2“ />
</xsl:variable>
```

The other feature of XSLT that we used was parameter passing. This was used both in phase 1 and in the pretty-print phase. This allows templates to pass some value as a parameter to another template it calls. In phase 1 this was used for wrapping elements in specific tags and in pretty print was used to pass in the amount a line should be tabbed in.

There are several issues with the current normalizer that must be improved upon, as well as other tasks that the normalizer must be able to perform. Sometimes you want to include the same content from the source document in the output document multiple times. That is easy to do simply by applying templates multiple times, once in each place where you want the data to appear. However, suppose you want the data to be formatted differently in different locations. In this situation, the solution is to give each of the different rules a mode attribute. Then you can choose which template to apply by setting the mode attribute of the *xsl:apply-template* element. The format is shown as follows.

```
<xsl:apply-templates select=“XPath expression" mode="name">
 <!-- Content: (xsl:sort|xsl:with-param)* -->
</xsl:apply-templates>
```

An XSLT stylesheet has been developed by David Hirtle and Derek Smith for normalizing the syntax used in RuleML instances. David and Derek took a different approach in implementing their normalizer than we will take in ours. Instead of using multiple phases to implement each step of the normalization, David and Derek’s normalizer makes heavy use of wildcards for elements and explicit iteration. It is a catch-all template that runs for every tag. Each element in the RuleML document will run through a large choose statement and will be matched to the element’s name. To add missing role tags and achieve canonical ordering, the normalizer first checks to see if the missing role tags are already in place. If the role tags are already existent, then the elements are put into canonical ordering. However, if the role tags are missing then they are added to the output document and the elements are then put into canonical ordering. Therefore David and Derek’s normalizer performs the normalization in one pass or phase. Our normalizer will be implemented in multiple phases that will use the previous phase’s output as input for the current phase (e.g. phase 2 will use the output from phase 1). Therefore, the

normalization will be performed in several passes. Hence it is not surprising that the length of the code in our XSLT stylesheet is much longer than David and Derek's XSLT stylesheet. However, by making heavy use of wildcards for elements and explicit iteration David and Derek's normalizer uses XSLT in an untypical way and is not very readable an maintainable, therefore we have chosen to implement ours in several phases.

The tools that were used to develop the normalizer included Oxygen XML Editor 13.1 and the Online XSLT 2.0 Service [5] which is a service run by W3C Systems Team. The online validator service uses the XSLT stylesheet that is referenced to normalize the XML document that is also referenced. The result is a normalized XML document. This service is used to reference the links on the normalizer website in order to run the test cases and analyse the results.

The Oxygen XML Editor that was used to develop the normalizer has the capability of creating both XSLT stylesheets and XML documents. Oxygen allows for development in structured mark-up languages including XML and XSLT using Java technology [4]. Since both documents could be built within the editor and tested, this was much more efficient than using the online validator since the website did not have to be updated each time a change occurred that needed to be tested. However, the online validator is necessary as it allows the results of the normalizer to be replicated by third party testing.

2. Phased Approach

2.1 Upgrade to RuleML 1.0

In order to assist us in our development, Dr. Tara Athan developed a partially built normalizer for RuleML 0.91 using an XSLT stylesheet. This was the structure that the normalizer was built on. Initially, development started with a three-phase structure but a fourth phase was added in order to allow for the development of pretty print. Another change included upgrading the normalizer for RuleML 1.0. In order to do this, the namespace references had to be changed to reference RuleML 1.0 instead of 0.91. While revisions to the normalizer were being made, the namespace reference had to be changed again as there was another release of RuleML 1.0. Other changes that needed to be made in order to upgrade the stylesheet to RuleML 1.0 included changing the names of the element tags. The role tag names that had to be changed included: <head> changed to <then>; <body> changed to <if>, <lhs> changed to <left>, <rhs> changed to <right>. Also, the attribute in=”no|semi|yes” changed to per=”copy|open|value”, respectively [3].

2.2 Phase 1

In the first phase the syntax is checked for missing edge stripes using the XSLT stylesheet, and then the missing edge stripes are added. Edge stripes are also referred to as role tags or method tags. The elements are matched using the name of the parent tag. It then has to be determined if the children are either already wrapped with the appropriate role tag, or if they are ‘naked’ which means they are not wrapped. In the case where the children are already wrapped then the tags are copied unchanged. If the children are not wrapped then it is determined which tag is required and the children are wrapped in the appropriate tag. Test cases including not normlized, partially normalized and fully normalized elements are used to test the capability of the normalizer to add the missing role tags.

In phase 1 we initially had a different template for each of the tags we wanted to wrap something in, i.e. a template for each of op, arg, if, then, left, right, formula, etc. This quickly became clumsy and made the code long and harder to read through. So instead we created one template called wrap, which took a parameter called tag. Whenever one of our phase 1 templates wanted to wrap something in a specific tag we would call the wrap template with the desired tag. For example to wrap something in formula the code would be

```
<xsl:call-template name=”wrap”>
        <xsl:with-param name=”tag”>formula</xsl:with-param>
</xsl:call-template>
```

and the wrap template looks like

```
<xsl:template name=”wrap”>
        <xsl:param name=”tag” />
        <xsl:element name=”{$tag}”>
                <xsl:call-template name=”copy-1“ />
        </xsl:element>
</xsl:template>.
```

Where “copy-1” copies foreign elements unchanged.

The normalizer in its current state of development is unable to handle all cases of unexpected elements appearing as children within the elements Entails, Implies and Equal. Only the expected elements that are allowed to follow the parent are checked for. If an element follows a parent that is unexpected it will be wrapped regardless. The cases that the normalizer is able to transform correctly, as well as the cases the normalizer is unable to transform, are documented in the comments of the normalizer stylesheet for each parent. The reason that the normalizer does not handle all cases of Entails, Implies and Equal is because the stylesheet refers to the position of the children of these elements, so if the child is not in the second to last and last position then the foreign elements in those positions will be incorrectly wrapped in the role tag. Entails, Implies and Equal were revised during the development so they would correctly copy tags directly following the parent element such as <oid> which is allowed. The use of the last position and second to last position instead of the first and second position corrected the problem with <oid> being tagged with <if> or <then>. All cases that are handled correctly as well as cases that are not normalized correctly are documented directly in the stylesheet.

2.2.1 Each Element's Role Tags

The following elements are checked to ensure that their children are properly wrapped, or if they are not wrapped, then the correct role tags are added. For each element the name of the role tag it is to be wrapped in is given, and also the name of the role tag that its children are to be wrapped in.

<Retract> is checked for the <oid> tag which is allowed and if present, is copied unchanged. If the name tag of the child is <formula> then the role tag is copied unchanged. Otherwise, assuming there is no other role wrapper, the child is wrapped in the role tag <formula>.

<Query> is checked for the <oid> tag which is allowed and if present, is copied unchanged. If the name tag of the child is <formula> then the role tag is copied unchanged. Otherwise, assuming there is no other role wrapper, the child is wrapped in the role tag <formula>.

<Entails> copies foreign elements and <oid> unchanged as long as they are not in the second to last or last position. If neither child of <Entails> is wrapped, then the second to last child is wrapped in <if> and the last child is wrapped in <then>. If the second to last child is wrapped in <then>, the tag is copied unchanged, and the last child is wrapped in <if>. If the last child is wrapped in <if>, the tag is copied unchanged, and the second to last child is wrapped in <then>. In all other cases, the second to last child is wrapped in <if> and the last child is wrapped in <then>.

<Rulebase> is checked for the <oid> tag which is allowed and if present, is copied unchanged. If the name tag of the child is <formula> then the role tag is copied unchanged. Otherwise, assuming there is no other role wrapper, the child is wrapped in the role tag <formula>.

<Exists> is checked for the <oid> tag which is allowed and if present, is copied unchanged. If the children are already wrapped in <declare> or <formula> then they are copied unchanged. If

the child of <Exists> is <Var>, then <Var> is wrapped in <declare>. There may be more than one <Var> and all of them are wrapped. All other naked children are wrapped in <formula>.

<Forall> is checked for the <oid> tag which is allowed and if present, is copied unchanged. If the children are already wrapped in <declare> or <formula> then they are copied unchanged. If the child of <Forall> is <Var>, then <Var> is wrapped in <declare>. There may be more than one <Var> and all of them are wrapped. All other naked children are wrapped in <formula>.

<Implies> copies foreign elements and <oid> unchanged as long as they are not in the second to last or last position. If neither child of <Implies> is wrapped, then the second to last child is wrapped in <if> and the last child is wrapped in <then>. If the second to last child is wrapped in <then>, the tag is copied unchanged, and the last child is wrapped in <if>. If the last child is wrapped in <if>, the tag is copied unchanged, and the second to last child is wrapped in <then>. In all other cases, the second to last child is wrapped in <if> and the last child is wrapped in <then>.

<Equivalent> is checked for the <oid> tag which is allowed and if present, is copied unchanged. If the name tag of the children is not already <torso> then the role tag is copied unchanged. Otherwise, the children are wrapped in the role tag <torso>.

<And> is checked for the <oid> tag which is allowed and if present, is copied unchanged. If the name tag of the children is already <formula> then the role tag is copied unchanged. Otherwise, the children are wrapped in the role tag <formula>.

<Or> is checked for the <oid> tag which is allowed and if present, is copied unchanged. If the name tag of the children is already <formula> then the role tag is copied unchanged. Otherwise, the children are wrapped in the role tag <formula>.

<Atom> is checked for the tags <oid>, <op>, <arg>, <slot>, <degree>, <Reify>, and <Skolem>, which are allowed and if present, are copied unchanged. If the children are named <Ind> or <Var> they are wrapped in the <arg> tag and an index number is added, counting the children. If the child is named <Rel> then it is wrapped with the <op> tag.

<Assert> is checked for the <oid> tag which is allowed and if present, is copied unchanged. If the name tag of the children is already <formula> then the role tag is copied unchanged. Otherwise, the children are wrapped in the role tag <formula>.

<Equal> copies foreign elements and <oid> unchanged as long as they are not in the second to last or last position. If neither child of <Equal> is wrapped, then the second to last child is wrapped in <left> and the last child is wrapped in <right>. If the second to last child is wrapped in <right>, the tag is copied unchanged, and the last child is wrapped in <left>. If the last child is wrapped in <left>, the tag is copied unchanged, and the second to last child is wrapped in <right>. In all other cases, the second to last child is wrapped in <left> and the last child is wrapped in <right>.

<Neg> is checked for the <oid> tag which is allowed and if present, is copied unchanged. If the name tag of the children is already <strong> then the role tag is copied unchanged. Otherwise, the children are wrapped in the role tag <strong>.

<Naf> is checked for the <oid> tag which is allowed and if present, is copied unchanged. If the name tag of the children is already <weak> then the role tag is copied unchanged. Otherwise, the children are wrapped in the role tag <weak>.

<Expr> is checked for the tags <oid>, <op>, <arg>, <slot>, <degree>, <Reify>, and <Skolem>, which are allowed and if present, are copied unchanged. If the children are named <Ind> or <Var> they are wrapped in the <arg> tag and an index number is incremented, counting the children. If the children are named <Fun> then they are wrapped with the <op> tag.

<Plex> is checked for the tags <oid>, <op>, <arg>, <slot>, <degree>, <Reify>, and <Skolem>, which are allowed and if present, are copied unchanged. If the name tag of the children is already <arg> then the role tag is copied unchanged. Otherwise, the children are wrapped in the role tag <arg> and an index number is incremented, counting the children.

2.2.2 Phase 1 Implementation

In order to determine the children that need to be wrapped and what the role tag is they should be wrapped in is based on matching the parent name. Each element is matched with the appropriate template using the element name. The template then checks to verify if the node is wrapped. If the edge stripe is skipped, the template wraps the node in the appropriate role tag based either on the names of the child nodes, the position of the child nodes, or in some cases all child nodes of the element are wrapped in the same role tag. All foreign elements which are not matched are copied unchanged to the phase-1 output. Lastly, everything is copied to the phase-1 output, in which comments and foreign elements are preserved. Therefore, if more elements are added to RuleML then they are only being copied at this time, and will not be normalized.

2.2.3 Phase 1 Test Case

The test case that follows demonstrates the correct wrapping of the child nodes of the element <Assert> properly wrapped in the role tag <formula>. The test case also shows how the stylesheet allows elements to be present directly following the parent <Implies>. The element <oid> is not wrapped but instead is copied unchanged since the stylesheet refers to the last and second to last child as the nodes that are to be wrapped. In the test case the last child is wrapped in <then> which means the stylesheet will wrap the second to last child in <if>. Inside the element <Implies>, the element children of the element <And> are properly wrapped in <formula>. The stylesheet also wraps the children of <Atom> correctly by wrapping <Rel> in <op> and <Var> in <arg>, but the index attribute is not correct which is evident in the test cases. The results of the test case were found using the Online XSLT 2.0 Service.

The section of the stylesheet which is being used to wrap the child nodes of the <Implies> elements in the test case is shown below. This shows the use of position and its reference to the last and second to last node, which preserves the element <oid> unwrapped, but only because of its position.

```
<!-- Wraps the second to last RuleML child of Implies. -->
   <xsl:template match="r:Implies/*[namespace-uri(.)='http://ruleml.org/spec' and
position()=last()-1]" mode="phase-1">
      <xsl:comment>second to last</xsl:comment>
      <xsl:choose>
         <xsl:when test="local-name()='if' or local-name()='then'">
            <xsl:call-template name="copy-1" />
         </xsl:when>
         <xsl:when test="local-name(following-sibling::*[1])='if'">
            <xsl:call-template name="wrap">
               <xsl:with-param name="tag">then</xsl:with-param>
            </xsl:call-template>
         </xsl:when>
         <xsl:otherwise>
            <xsl:call-template name="wrap">
               <xsl:with-param name="tag">if</xsl:with-param>
            </xsl:call-template>
         </xsl:otherwise>
      </xsl:choose>
   </xsl:template>

   <!-- Wraps the last RuleML child of Implies. -->
   <xsl:template match="r:Implies/*[namespace-uri(.)='http://ruleml.org/spec' and
position()=last()]" mode="phase-1">
      <xsl:comment>last</xsl:comment>
      <xsl:choose>
         <xsl:when test="local-name()='if' or local-name()='then'">
            <xsl:call-template name="copy-1" />
         </xsl:when>
         <xsl:when test="local-name(preceding-sibling::*[1])='then'">
            <xsl:call-template name="wrap">
               <xsl:with-param name="tag">if</xsl:with-param>
            </xsl:call-template>
         </xsl:when>
         <xsl:otherwise>
            <xsl:call-template name="wrap">
               <xsl:with-param name="tag">then</xsl:with-param>
            </xsl:call-template>
         </xsl:otherwise>
      </xsl:choose>
   </xsl:template>
```

The test case can be found in Appendix A and at:
http://v37s3b4h7dn47s37hg1br4h7rs7n3du7s8nu.unbf.ca/~lbidlak1/test-caseEntails.xml

In this test case the first instance of the <Entails> element is partially normalized since the last child is wrapped in <then> but the second to last child is missing the role tag <if>. Inside the <Entails> element is an <Implies> element. The children of <Implies> are both missing their role tags.

The second instance of the <Entails> element nor of the children of any elements are wrapped in their appropriate role tag. Other elements that are missing their role tags include: <Assert>, <Rulebase>, <And>, and <Atom>.

The results of the test case can be found at in Appendix B and:
http://v37s3b4h7dn47s37hg1br4h7rs7n3du7s8nu.unbf.ca/~lbidlak1/resultsEntails.xml

In the results, the role tags for all elements that were missing are now present. The role tag for the first instance of <Entails> which was partially normalized was preserved and only the missing role tag was added according the tag that was already present.

2.3 Phase 2

The second phase of our transformation re-orders the subelements as necessary to achieve canonical ordering. Each element has certain rules pertaining to the order of their subelements in normal form. These order rules can be found in the "normal" schema (the schema that the output of the normalizer has to be validated against). This schema uses the Garden of Eden schema design pattern [1], that is, it declares all elements globally using named patterns. This pattern allows reuse of elements which makes the schema easier to maintain. This means that the schema was divided between multiple files. To find the canonical order of subelements for a particular element one must consult the main schema which includes a reference to another document containing the schema for that particular element [3].

2.3.1 Each Element's Canonical Order

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.

<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.

<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.

<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.

<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.

<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.

<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.

<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.

<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).

<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).

<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).

<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.

<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.

<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 <strong> role. Therefore the canonical ordering of the subelements of Neg is: oid, strong.

<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.

<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.

<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.

2.3.2 Implementation

To implement the canonical ordering for each element, we first have to copy the output from phase 1.

```
<xsl:template name="copy-1">
<xsl:copy><xsl:apply-templates select="node() | @*" mode="phase-1"/>
</xsl:copy>
</xsl:template>
```

Then we need to check the order of each element to determine if we need to put each element in the correct canonical order.

```
<xsl:variable name="phase-2-output">
<xsl:apply-templates select="$phase-1-output" mode="phase-2"/>
</xsl:variable>
```

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.

```
<xsl:template match="r:Implies" mode="phase-2">
   <xsl:copy>
         <xsl:apply-templates select="@*"/>
         <xsl:apply-templates select="r:oid" mode="phase-2"/>
           <xsl:apply-templates select="r:if" mode="phase-2"/>
          <xsl:apply-templates select="r:then" mode="phase-2"/>
<xsl:apply-templates select="*[namespace-uri(.)!='http://ruleml.org/spec' or (local-
name()!= 'oid' and local-name()!= 'if' and local-name()!= 'then')]" mode="phase-2"/>
     </xsl:copy>
</xsl:template>
```

Finally, we need to copy everything to the phase-2 output, in which comments are preserved without escaping. During this phase, both order and foreign elements need to be preserved because this is the most general of all templates.

2.3.3 Test Cases

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. The output schema used to infer canonical ordering content of Atom is shown below.

```
<xsl:template match="r:Atom" mode="phase-2">
   <xsl:copy>
      <xsl:apply-templates select="@*"/>
     <xsl:apply-templates select="r:op" mode="phase-2"/>
    <xsl:apply-templates select="r:arg" mode="phase-2">
    <xsl:sort select="@index"/>
    </xsl:apply-templates>
    <xsl:apply-templates select="r:repo" mode="phase-2"/>
   <xsl:apply-templates select="r:slot" mode="phase-2"/>
   <xsl:apply-templates select="r:resl" mode="phase-2"/>
   <xsl:apply-templates select="@*|*[namespace-uri(.)!='http://ruleml.org/spec' or (local-
name()!= 'oid' and local-name()!= 'op' and local-name()!= 'arg' and local-name()!='repo'
and local-name()!='slot' and local-name()!='resl')]" mode="phase-2"/>
   </xsl:copy>
</xsl:template>
```

The test case can be found in Appendix C and at:
http://v37s3b4h7dn47s37hg1br4h7rs7n3du7s8nu.unbf.ca/~lbidlak1/ImpliesSlots-testcasePhase2.xml

Before the normalization we can see that the <Atom> element contained in the <if> element contains first a relationship, followed by a slot, followed by an <Ind> element, followed by another slot. After phase 1 is complete, the <Ind> element will be wrapped in an indexed argument tag and the <Rel> element in an <op> tag. The canonical ordering for <Atom> elements is the oid, followed by all positional arguments, followed by all slotted arguments. Therefore the canonical ordering of this <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. The correct canonical ordering of elements is shown in the results which can be found in Appendix D and at:
http://v37s3b4h7dn47s37hg1br4h7rs7n3du7s8nu.unbf.ca/~lbidlak1/ImpliesSlots-resultsPhase2.xml

2.4 Phase 3

For phase 3 we will make all missing attributes explicit with their default values. Having to carefully map the hierarchy of an XML document in an XSLT style sheet may be inconvenient if the document does not follow a regular structure like the periodic table. Hence, we need general rules that can find an element and apply templates to it regardless of where it appears in the source document. To make this mapping more convenient, we make all attributes explicit with their default values in a (Deliberation, non-SWSL) RuleML instance, and copy their values into the output.

2.4.1 The Tasks of Phase 3

The attributes that are made explicit and their possible values, including their default values are outlined in this section.

The attribute @material is made explicit. This is an attribute indicating the kind of implication rule (<Implies>). The values that are allowed are “yes" and "no". The default value of @material is “yes”.

The attribute @mapMaterial is made explicit. This is an attribute indicating the kind of all implication rules (<Implies>) falling within its scope (i.e. child elements). The values that are allowed are "yes" and "no". The default value of @mapMaterial is “yes”.

The attribute @direction is made explicit. This is an attribute indicating the intended direction of an implication rule's (<Implies>) inferencing. It has a neutral default value of "bidirectional". The other allowed values are "forward" and "backward".

The attribute @mapDirection is made explicit. This is an attribute indicating the intended direction of implication rule (<Implies>) inferencing of elements falling within its scope (i.e. child elements). This attribute has a neutral default value of "bidirectional". The other allowed values are "forward" and "backward".

The attribute @oriented is made explicit. This is an attribute indicating whether an equation (<Equal>) is oriented (directed) or unoriented (symmetric). Allowed values are "no" (unoriented, the default) and "yes" (oriented).

For example, phase 3 transforms

```
<Equal>
   <left>
       <Expr>
              <Fun>father-of</Fun>
              <Ind>John</Ind>
              <Fun>fac</Fun>
       </Expr>
   </left>
   <right>
       <Ind>Mexico City</Ind>
   </right>
</Equal>
```

into:

```
<Equal oriented="no">
   <left>
      <Expr>
          <op>
              <Fun per="copy" val="0..">father-of</Fun>
          </op>
          <op>
              <Fun per="copy" val="0..">fac</Fun>
          </op>
              <arg index="1">
              <Ind>John</Ind>
              </arg>
       </Expr>
   </left>
   <right>
       <Ind>Mexico City</Ind>
   </right>
</Equal>
```

The value for @per is made explicit. This is an attribute used to indicate whether a function (<Fun>) or an expression (<Expr>) will be interpreted. In equality sublanguages, it has three values: "copy" (the default), "value" and "open", while in non-equality sublanguages it can only have the value "copy".

The value for @val is made explicit. This is an attribute used to indicate whether a function (<Fun>) is deterministic or non-deterministic. It has two values: "1" (deterministic: exactly one)

and "0.." (set-valued: zero or more, the default). For example, the function children(John, Mary) can be interpreted in a set-valued manner using a definition corresponding to children(John, Mary) = {Jory, Mahn}, so that the application children(John, Mary) returns {Jory, Mahn}:

For example phase 3 transforms <Expr><Fun per=”value”><Ind>John</Ind><Ind>Mary</Ind> </Expr> into:

```
<Expr>
 <Fun per="value" val="0..">children</Fun>
 <Ind>John</Ind>
 <Ind>Mary</Ind>
</Expr>
```

2.4.2 Implementation

To implement the treatment of attributes with default values, we firstly have to copy the output from phase 2.

```
<xsl:variable name="phase-3-output">
  <xsl:apply-templates select="$phase-2-output" mode="phase-3"/>
</xsl:variable>
```

Then we need to realize the default values for each designated attribute. For this part, the procedure is relatively easy. First, match the corresponding element and check the element to determine whether it includes the designated attributes or not. If the designated attributes are not included, then add the default values for the attributes in the element.

Finally, we need to copy everything to the phase-3 output, in which comments are preserved without escaping. During the phase, both order and foreign elements need to preserved because this is the most general of all templates. Any more specific template in phase 3 will over-ride this one. At the end of phase 3, the normalizer needs to copy everything to the transformation output.

2.5 Pretty Print

The pretty-print phase is a phase that was added later on when we noticed that the comments in the source document were not being outputted properly. In fact we would end up with no new line before or after the comments. If the comment was long enough then the Implies tag would run off the page and not be visible, as the following output shows:
<Assert><!-- Some implication rule --><Implies>.

The other 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 shown below made the outputted RuleML harder for a human to read.

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

Pretty print creates the standard formatting that is much easier to read, as shown:
<op><Rel>relation</Rel></op>

In order to do this we had to write the pretty-print mechanism entirely from scratch. This was done 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
```

The tabs parameter defines the amount of space a given element should be tabbed in and when this template is first called its value will be an empty string. Before each opening tag is outputted we first go to a new line and output the appropriate tab spacing. After outputting the node's attributes we then output the value of the node if it has no children or if the node does have children we apply the pretty print templates to them with the tabs parameter increased by the tab variable. 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>

The easy part for coding was then to have new lines before comments which was done by creating a template which would match any comment node and would output a new line (using the variable) and then output the comment.

The much trickier part was implementing the lack of spacing for certain tags. To do this, first the above template was named "new-line" indicating it was for when a new line was desired before the child. Two new templates were needed, one for deciding whether to call new-line or to call the other new template "no-new-line". The no-new-line template takes two parameters, tabs and

newlines. The newlines parameter is for determining if we want a new line before the tag or not. The pseudo-code for the two new 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
```

The no-new-lines template only outputs a new line and tab spacing if it is given the value yes for the newlines parameter. When it is first called by the main template it is passed yes but when it calls itself for each of the child nodes newlines is given the value no, so new line is outputted before the node and we get the desired output:
<op><Rel>relation</Rel></op>

3. RuleML Official Normalizer (RON)

3.1 Overall Implementation

The RuleML Official Normalizer (RON) works by running each of the four phases one after the other on a RuleML document. That is, the XSLT from phase 1 transforms the original document, then phase 2 runs on the output of phase 1, then phase 3 runs on the output of phase 2, and finally the output from phase 3 is formatted using pretty-print formatting. Therefore, the missing role tags are added first, subelements are put into the correct canonical order second, default values are made explicit third, and then the output is formatted. Although this is the ordering of phases we have chosen to implement in our normalizer an alternative ordering of phases also could have been implement. For phase 2 to successfully reorder subelements so that they are in the proper canonical order, the missing role tags must already be added since if they were added later they might not be added in the proper order, thus the subelements in the output would not be in proper canonical order. Similarly, for phase 3 to make default values explicit for the appropriate role tags, those role tags must have already been added to the output. Therefore we conclude that the first phase of our nomalization must always happen first. However, phase 2 and 3 are interchangeable. The attribute values used in phase 3 are not altered in phase 2, and the ordering of elements does not affect the attribute values. Therefore, after running phase 1 on the document, we could immediately run phase 3 followed by phase 2, and the output would remain the same as the output from our current normalizer.

3.2 General Test Cases

We have developed three test cases to demonstrate the capabilities of our RuleML normalizer.

3.2.1 Non-Normalized Test Case

The first test case contains no elements that have already been wrapped in role tags, and does not follow proper canonical order. Also, no default attribute values have been specified and thus must all be added. The test case can be found in Appendix E and at:
http://v37s3b4h7dn47s37hg1br4h7rs7n3du7s8nu.unbf.ca/~lbidlak1/RONTest1.xml
The test case contains rules to indicate whether a car is secure and safe. The first rule is a <Forall> statement and states if all car doors are locked, then the car is secure. The next rule is contained in an <Implies> statement and states if the trunk and car doors are closed, then car is safe. Next, there is a <Rulebase> statement that indicates that the safe relation and secure relation are equivalent.

Phase 1 transforms the test case first. In phase 1, the children of Assert will be wrapped in formula role tags. The children of Forall will be wrapped in declare role tags, except for the last one that is wrapped in a formula role tag. Inside Forall's subelement <Implies> the first <Atom> tag will be wrapped with <if> and the second <Atom> tag with <then>. Inside those <Atom> elements, <Rel> will be wrapped with <op> and <Var> will be wrapped with <arg> role tags that include an index attribute value. The children of the <Implies> element that describes the second rule will also be altered by phase 1. <Rel> will be wrapped in an <op> tag and <Var>

will be wrapped in <arg> role tags that include an index attribute. Here, phase 1 does not alter the <slot> elements. In the <Rulebase> statement, each of its subelements will be wrapped with a formula role tag. In the subelement <Equivalent> each of its children is wrapped in a <torso> role tag.

Once phase 1 is complete, phase 2 then performs the canonical ordering. In the output from phase 1 there are several places where subelements are not in the correct canonical order. In the <Implies> element that states that if trunk and car doors are closed, and the car alarm is on, then the car is safe, the <then> statement precedes the <if> statement, so to put these in canonical order we must rearrange them so that the <if> statement precedes the <then> statement. Also, in the first <Atom> element of the <if> we see that the slotted arguments come before the positional arguments, which is not the proper canonical ordering. Therefore in the output we want to output the <Var> elements which have been wrapped in <arg> role tags, before we output the slotted arguments.

Once phase 2 is complete, phase 3 then makes all default attribute values explicit. Since there are no attribute values already specified it will add default attribute values for <Assert>, <Implies>, and <Rulebase>. The results of this test case can be found in Appendix F and at: http://v37s3b4h7dn47s37hg1br4h7rs7n3du7s8nu.unbf.ca/~lbidlak1/out1.xml.
Each phase has successfully transformed the RuleML document.

3.2.2 Partially Normalized Test Case

The second test case is the same as the first one except that some normalization has occurred already. In the <Forall> statement <arg> role tags have been added to the <Atom>'s subelements with an index attribute. In the <Implies> element of the second rule, the <if> and <then> statements are now in canonical order. Also, in the <Atom> element the <Var> nodes have been wrapped with <arg> tags. In the <Rulebase> element <torso> tags have been added. Also an attribute for the <Assert> element has been added. This test case can be found in Appendix G and at: http://v37s3b4h7dn47s37hg1br4h7rs7n3du7s8nu.unbf.ca/~lbidlak1/RONTest2.xml
When we run the normalizer on this test case we get the same results as we got with the last test case, except this time the index attribute values of the <arg> role tags that wrap the <Var> tags in the <Atom> containing <slot>s are correct (since we have added them ourselves). The output can be found in Appendix F and at:
http://v37s3b4h7dn47s37hg1br4h7rs7n3du7s8nu.unbf.ca/~lbidlak1/out1.xml

3.2.3 Completely Normalized Test Case

Our last test case is the same as the other two test cases except that is a completely normalized version of those test cases. The test case can be found in Appendix H and at:
http://v37s3b4h7dn47s37hg1br4h7rs7n3du7s8nu.unbf.ca/~lbidlak1/RONTest3.xml
When this test case is normalized the result is the same as the original test case, as is expected.

4. Conclusions and Future Work

In phase one the children of the elements are wrapped in their appropriate role tags if they are not already wrapped. Foreign elements and comments are copied unchanged. The templates for <Implies>, <Entails> and <Equal> do not allow foreign elements to appear in the last or second to last position. Foreign elements are allowed and will be copied unchanged if they appear directly following the parent and not in the last or second to last position. If there are foreign elements added to RuleML that could appear as children of these elements in the last or second to last position but should not be wrapped, then the phase one stylesheet will need to be adjusted to check for these elements and either allow them to be copied unchanged, or to wrap them in the appropriate tags.

In phase two we reorder the subelements to achieve the canonical ordering outline in the "normal" schema (the schema that the output of our normalization is validated against). In the future, it may be beneficial to include more index attributes to be used for ordering or decide on some way to organize recurring elements (e.g. alphabetical). Many of the subelements are being outputted in the same order that they appear in the original document. For example, several formulas may be included in a <Query> element. Currently these formulas will appear in the output in the same order that they appear in the original document, but if there was some method to sort the formulas so that they would appear in a predictable order in every case, such a method may be beneficial.

In phase three the default values, for the attributes that have them, are made explicit. This functionality is best executed in phase three as all index attributes have already been assigned which will allow them to be correctly counted. However, there are still errors in the assignment of indexing values, especially when an <arg> tag is after <slot>. In the future work with the normalizer this issue will need to be addressed.

5. References:

[1] Ayub Khan and Marina Sum, Introducing Design Patterns in XML Schemas, http://developers.sun.com/jsenterprise/archive/nb_enterprise_pack/reference/techart/design_patterns.html, access date: Nov. 2011.

[2] Harold Boley et. al, The RuleML Markup Initiative, http://ruleml.org, access date: Nov. 2011

[3] Harold Boley et. al, Schema Specification of RuleML 1.0, http://ruleml.org/1.0/, access date: Dec. 2011

[4] IUWare, oXygen XML Editor 13.1, http://iuware.iu.edu/Windows/Package/1736, access date: Dec. 2011

[5] Online XSLT 2.0 Service http://www.w3.org/2005/08/online_xslt/ access date: Dec. 2011

Appendix A: Test Case for Entails

```
<?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 John owns a store and John sells products, then John is a Merchant -->
<Assert>
        <!-- A rule and two premise facts entail a conclusion fact-->
        <Entails>
                <Rulebase>
                        <Implies>
                                <And>
                                        <Atom>
                                                <Rel>owns</Rel>
                                                <Var>Person</Var>
                                                <Var>Property</Var>
                                        </Atom>
                                        <Atom>
                                                <Rel>grocery</Rel>
                                                <Var>Property</Var>
                                        </Atom>
                                </And>
                                <Atom>
                                        <Rel>sells</Rel>
                                        <Var>Person</Var>
                                        <Ind>produce</Ind>
                                </Atom>
                        </Implies>
                        <Atom>
                                <Rel>owns</Rel>
                                <Ind>john</Ind>
                                <Ind>goodyshop</Ind>
                        </Atom>
                        <Atom>
                                <Rel>grocery</Rel>
                                <Ind>goodyshop</Ind>
                        </Atom>
                </Rulebase>
                <then>
                        <Rulebase>
                                <Atom>
                                        <Rel>sells</Rel>
                                        <Ind>john</Ind>
                                        <Ind>produce</Ind>
```

```
					</Atom>
				</Rulebase>
			</then>
		</Entails>
		<!--  The facts that Jill is a consumer and Jill buys produce entail that Jill buys produce --
>
		<Entails>
			<Rulebase>
				<Atom>
					<Rel>consumer</Rel>
					<Ind>Jill</Ind>
				</Atom>
				<Atom>
					<Rel>buys</Rel>
					<Ind>Jill</Ind>
					<Ind>produce</Ind>
				</Atom>
			</Rulebase>
			<Rulebase>
				<Atom>
					<Rel>buys</Rel>
					<Ind>Jill</Ind>
					<Ind>produce</Ind>
				</Atom>
			</Rulebase>
		</Entails>

</Assert>
</RuleML>
```

Appendix B: Test Case Results for Entails

```
<?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 John owns a store and John sells products, then John is a Merchant -->
 <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>Property</Var></arg>
           </Atom>
          </formula>
          <formula>
           <Atom>
            <op><Rel>grocery</Rel></op>
            <arg index="1"><Var>Property</Var></arg>
           </Atom>
          </formula>
         </And>
        </if>
        <then>
         <Atom>
          <op><Rel>sells</Rel></op>
          <arg index="1"><Var>Person</Var></arg>
          <arg index="2"><Ind>produce</Ind></arg>
         </Atom>
        </then>
       </Implies>
      </formula>
      <formula>
       <Atom>
        <op><Rel>owns</Rel></op>
        <arg index="1"><Ind>john</Ind></arg>
        <arg index="2"><Ind>goodyshop</Ind></arg>
```

```
        </Atom>
      </formula>
      <formula>
        <Atom>
          <op><Rel>grocery</Rel></op>
          <arg index="1"><Ind>goodyshop</Ind></arg>
        </Atom>
      </formula>
    </Rulebase>
  </if>
  <then>
    <Rulebase>
      <formula>
        <Atom>
          <op><Rel>sells</Rel></op>
          <arg index="1"><Ind>john</Ind></arg>
          <arg index="2"><Ind>produce</Ind></arg>
        </Atom>
      </formula>
    </Rulebase>
  </then>
 </Entails>
</formula>
<!--  The facts that Jill is a consumer and Jill buys produce entail that Jill buys produce -->
<formula>
 <Entails>
  <if>
    <Rulebase>
      <formula>
        <Atom>
          <op><Rel>consumer</Rel></op>
          <arg index="1"><Ind>Jill</Ind></arg>
        </Atom>
      </formula>
      <formula>
        <Atom>
          <op><Rel>buys</Rel></op>
          <arg index="1"><Ind>Jill</Ind></arg>
          <arg index="2"><Ind>produce</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>
</RuleML>
```

Appendix C: Test Case for Implies Slots

```
<?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>buy</Rel>
          <Var>Cust</Var>
          <Var>book</Var>
        </Atom>
      </then>
      <if>
        <Atom>
          <Rel>father</Rel>
          <slot><Ind>daughter</Ind><Ind>Mary</Ind></slot>
          <Ind>John</Ind>
          <slot> <Data>buyer</Data> <Var>cust</Var> </slot>
        </Atom>
      </if>
    </Implies>
  </Assert>
</RuleML>
```

Appendix D: Test Case Results for Implies Slots

```
<?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">
    <formula>
      <Implies material="yes" direction="bidirectional" mapMaterial="yes"
mapDirection="bidirectional">
        <if>
          <Atom>
            <op><Rel>father</Rel></op>
            <arg index="1"><Ind>John</Ind></arg>
            <slot><Ind>daughter</Ind><Ind>Mary</Ind></slot>
            <slot><Data>buyer</Data><Var>cust</Var></slot>
          </Atom>
        </if>
        <then>
          <Atom>
            <op><Rel>buy</Rel></op>
            <arg index="1"><Var>Cust</Var></arg>
            <arg index="2"><Var>book</Var></arg>
          </Atom>
        </then>
      </Implies>
    </formula>
  </Assert>
</RuleML>
```

Appendix E: Non-Normalized Test Case


```xml
<?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">
    <formula>
      <Implies material="yes" direction="bidirectional" mapMaterial="yes"
mapDirection="bidirectional">
        <if>
          <Atom>
            <op><Rel>father</Rel></op>
            <arg index="1"><Ind>John</Ind></arg>
            <slot><Ind>daughter</Ind><Ind>Mary</Ind></slot>
            <slot><Data>buyer</Data><Var>cust</Var></slot>
          </Atom>
        </if>
        <then>
          <Atom>
            <op><Rel>buy</Rel></op>
            <arg index="1"><Var>Cust</Var></arg>
            <arg index="2"><Var>book</Var></arg>
          </Atom>
        </then>
      </Implies>
    </formula>
  </Assert>
</RuleML>
```

Appendix F: Non-Normalized Test Case Results


```xml
<?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 mapMaterial="yes" mapDirection="bidirectional">
    <!-- if all car doors are locked then the car is secure -->
    <formula>
      <Forall>
        <declare>
          <Var>door</Var>
        </declare>
        <formula>
          <Implies mapMaterial="yes" mapDirection="bidirectional" material="yes"
direction="bidirectional">
            <if>
              <Atom>
                <op><Rel>locked</Rel></op>
                <arg index="1"><Var>door</Var></arg>
              </Atom>
            </if>
            <then>
              <Atom>
                <op><Rel>secure</Rel></op>
                <arg index="1"><Var>car</Var></arg>
              </Atom>
            </then>
          </Implies>
        </formula>
      </Forall>
    </formula>
    <!--
            Alternatively, if trunk and doors are closed, and the car alarm is on, then car is safe
        -->
    <formula>
      <Implies mapMaterial="yes" mapDirection="bidirectional" material="yes"
direction="bidirectional">
        <if>
          <And>
            <formula>
              <Atom>
                <op><Rel>closed</Rel></op>
                <arg index="3"><Var>trunk</Var></arg>
                <arg index="4"><Var>doors</Var></arg>
                <slot><Ind>car</Ind><Ind>Nissan</Ind></slot>
                <slot><Ind>year</Ind><Ind>2007</Ind></slot>
```

```
          </Atom>
        </formula>
        <formula>
          <Atom>
            <op><Rel>on</Rel></op>
            <arg index="1"><Var>alarm</Var></arg>
          </Atom>
        </formula>
      </And>
    </if>
    <then>
      <Atom>
        <op><Rel>safe</Rel></op>
        <arg index="1"><Var>car</Var></arg>
      </Atom>
    </then>
  </Implies>
</formula>
<!--
        The relationship safe is equivalent to the relationship secure
    -->
<formula>
  <Rulebase mapMaterial="yes" mapDirection="bidirectional">
    <formula>
      <Equivalent>
        <torso>
          <Atom>
            <op><Rel>safe</Rel></op>
          </Atom>
        </torso>
        <torso>
          <Atom>
            <op><Rel>secure</Rel></op>
          </Atom>
        </torso>
      </Equivalent>
    </formula>
  </Rulebase>
</formula>
</Assert>
</RuleML>
```

Appendix G: Partially Normalized 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 mapMaterial="yes">

    <!--if all car doors are locked then the car is secure-->
    <Forall>
      <Var>door</Var>
      <Implies>
        <Atom>
          <Rel>locked</Rel>
          <Var>door</Var>
        </Atom>
        <Atom>
          <Rel>secure</Rel>
          <arg index="1">
            <Var>car</Var>
          </arg>
        </Atom>
      </Implies>
    </Forall>

    <!-- Alternatively, if trunk and doors are closed, and the car alarm is on, then car is safe -->
    <Implies>
      <if>
        <And>
          <Atom>
            <Rel>closed</Rel>
            <slot><Ind>car</Ind><Ind>Nissan</Ind></slot>
            <slot><Ind>year</Ind><Ind>2007</Ind></slot>
            <arg index="1">
              <Var>trunk</Var>
            </arg>
            <arg index="2">
              <Var>doors</Var>
            </arg>
          </Atom>
          <Atom>
            <Rel>on</Rel>
            <Var>alarm</Var>
          </Atom>
        </And>
```

```
        </if>
        <then>
            <Atom>
                <Rel>safe</Rel>
                <Var>car</Var>
            </Atom>
        </then>
    </Implies>

    <!--The relationship safe is equivalent to the relationship secure-->
    <Rulebase>
        <Equivalent>
            <torso>
                <Atom>
                    <Rel>safe</Rel>
                </Atom>
            </torso>
            <torso>
                <Atom>
                    <Rel>secure</Rel>
                </Atom>
            </torso>
        </Equivalent>
    </Rulebase>
  </Assert>
</RuleML>
```

Appendix H: Completely Normalized 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 mapMaterial="yes" mapDirection="bidirectional">
    <!--if all car doors are locked then the car is secure-->
    <formula>
      <Forall>
        <declare>
          <Var>door</Var>
        </declare>
        <formula>
          <Implies material="yes" direction="bidirectional">
            <if>
              <Atom>
                <op><Rel>locked</Rel></op>
                <arg index="1"><Var>door</Var></arg>
              </Atom>
            </if>
            <then>
              <Atom>
                <op><Rel>secure</Rel></op>
                <arg index="1"><Var>car</Var></arg>
              </Atom>
            </then>
          </Implies>
        </formula>
      </Forall>
    </formula>
    <!-- Alternatively, if trunk and doors are closed, and the car alarm is on, then car is safe -->
    <formula>
      <Implies material="yes" direction="bidirectional">
        <if>
          <And>
            <formula>
              <Atom>
                <op><Rel>closed</Rel></op>
                <arg index="1"><Var>trunk</Var></arg>
                <arg index="2"><Var>doors</Var></arg>
                <slot><Ind>car</Ind><Ind>Nissan</Ind></slot>
                <slot><Ind>year</Ind><Ind>2007</Ind></slot>
              </Atom>
            </formula>
```

```
                <formula>
                    <Atom>
                        <op><Rel>on</Rel></op>
                        <arg index="1"><Var>alarm</Var></arg>
                    </Atom>
                </formula>
            </And>
        </if>
        <then>
            <Atom>
                <op><Rel>safe</Rel></op>
                <arg index="1"><Var>car</Var></arg>
            </Atom>
        </then>
    </Implies>
</formula>
<!--The relationship safe is equivalent to the relationship secure-->
<formula>
    <Rulebase mapMaterial="yes" mapDirection="bidirectional">
        <formula>
            <Equivalent>
                <torso>
                    <Atom>
                        <op><Rel>safe</Rel></op>
                    </Atom>
                </torso>
                <torso>
                    <Atom>
                        <op><Rel>secure</Rel></op>
                    </Atom>
                </torso>
            </Equivalent>
        </formula>
    </Rulebase>
</formula>
</Assert>
</RuleML>
```