Syntax Reuse: XSLT as a Metalanguage for Knowledge Representation Languages

Tara Athan

Athan Services, Ukiah, CA, USA
taraathan AT gmail.com

Abstract. We present here MXSL, a subset of XSLT re-interpreted as a syntactic metalanguage for RuleML with operational semantics based on XSLT processing. This metalanguage increases the expressivity of RuleML knowledge bases and queries, with syntactic access to the complete XML tree through the XPath Data Model. The metalanguage is developed in an abstract manner, as a paradigm applicable to other KR languages, in XML or in other formats.

Keywords. Metalanguage, metamodel, RuleML, operational semantics, XSLT.

1 Introduction

One of the most important issues in the field of knowledge representation (KR) is the issue of semantic interoperability. This issue might be described as the compounding of information overload with representation-format overload. Representation formats abound because information collections (datasets, knowledge bases, metadata, ...) have a wide diversity of characteristics (numerical, textual, hierarchical, ...) and uses (quantitative processing, search, reasoning, ...), and diverse formats have been developed to meet these diverse needs.

One of the most widely-used formats is XML, due in part to its capability at representing both structured data and document markup, as well as the capability to specialize the syntax through schema languages, including XSD [1], Relax NG [2] and Schematron [3]. XML has been designated the foundational syntax of the semantic web [4], although other formats, such as JSON [5] may play a larger role in the future.

The need for compatibility between semantic and syntactic models of structured XML has been addressed from an RDF perspective in [6] and [7]. In the latter study, the syntactic XPath Data Model and the semantic RDFS model, are considered in parallel, as being alternate, and partially compatible, models of the same subject, an XML instance. This issue was considered again from the perspective of an XML-based Common Logic syntax in [8].

In this paper, we consider the case where the XML instance is itself a semantic representation, in particular instances of the RuleML [9] language, possibly containing embedded structured data. We investigate the use of a syntactic model of XML to describe correspondences between syntactic substructures of RuleML.

The goal of this investigation is to develop a syntactic metalanguage for RuleML in order to

1. enable reasoning over large, structured legacy datasets (e.g. in XML or flat files) without large-scale conversion of compact representations into the more verbose RuleML format;
2. allow user-specified deductive systems for reasoning over RuleML knowledge bases via axiom schemas;
3. enhance the RuleML query capability.

In addition, the principles of this metalanguage development are abstracted so that it may be viewed as a process that can be applied to the development of metalanguages for other (XML and non-XML) KR formats.

We define an operational semantics of the metalanguage as a "metainterpretation" transformation. This transformation maps the syntactic representations in the metalanguage into syntactic representations in the base language by substitution of (lexical) values for the metavariables, which we will call parameters to distinguish them from the logical variables.

Thus, a full interpretation of an instance of the metalanguage consists of two distinct phases;

1. the application of the metainterpretation to transform the metalanguage instance to another instance that is purely in the base language;
2. the application of the base language's model-theoretic interpretation to transform the derived instance to truth values.

The proof theory of the metalanguage is defined similarly. The deductive apparatus of the base language is extended to include deduction from metalanguage axioms to base language axioms obtained from metainterpretation with a suitable binding of the parameters.

The obvious model for a transformation language on XML-based metalanguage input is XSLT [10]. XSLT is a declarative language based on the same abstract data model underlying XPath and XQuery [11].

XSLT is a very rich language, which is in fact Turing complete [12]. We require only a small subset of the capabilities of XSLT for our purposes. In particular, our metalanguage should have the capability to

- add an element node to the output tree, given the local name and namespace of the element as parameters;
- add an attribute node to a particular element node, given the local name, namespace and value as parameters;
- add a text node as the child of an element;
- iteratively add, in a specified order, a number of nodes to the output tree, based on a set of parameter bindings represented in structured XML;
- express the hierarchical and sequential structure of the output tree isomorphically according to the corresponding structure of the metalanguage instance.

In addition, we have the following operational requirements: Instances in the meta-language, which we will call MXSL, will:

- validate against the XSLT schema,
- be self-contained, and
- be self-executing.

2 MXSL Syntax

The core MXSL syntax is a subset of XSLT 2.0 syntax which uses primarily the following elements:

- `<xsl:for-each>`, for iteration
- `<xsl:element>`, for construction of XML elements
- `<xsl:attribute>`, for addition of attributes to XML elements
- `<xsl:copy-of>` with `@select`, for generation of the values of attributes and contents of element

Additional elements which are allowed in MXSL in a restricted form (as indicated in parentheses) are:

- `<xsl:stylesheet>` (once as the root element)
- `<xsl:variable>` (once as a header to define the parameter bindings and also as the first children of `<xsl:for-each>`, once for each parameter in the iteration)
- `<xsl:template match="/">` (once for the MXSL body)
- `<xsl:processing-instruction>` (zero to many times as the first children of `<xsl:template>`)
- `<ruleml:RuleML>` (once as the last child of `<xsl:template>` to define the root for the output document)
- `<xsl:comment>` (arbitrarily within the `<ruleml:RuleML>` element)

A Relax NG schema that may be used to verify that an MXSL instance does not exceed these restrictions is given at `rnc/mxsl.rnc`.[1]

3 Example: Captured Strings

This example illustrates the definition of a function that takes as argument a character sequence, with some restrictions, and produces a name based on that string, equivalent to the captured string syntax of IKL [13].

[1] The path to the directory for all files referenced in this paper is http://athant.com/mxsl .

A sample RuleML axiom that will be used to generate the axiom schema is

```
<ruleml:Equal>
  <ruleml:Expr>
    <ruleml:Fun
        iri="http://athant.com/mxsl/vocab#constant"/>
    <ruleml:Data xsi:type="xs:string">stringParam</Data>
  </ruleml:Expr>
  <ruleml:Ind>stringParam</ruleml:Ind>
</ruleml:Equal>
```

RuleML syntax does not allow us to generalize the character sequence `stringParam` that appears as the simple content of both the `<ruleml:Data>` and `<ruleml:Ind>` elements. That is, we can replace the <ruleml:Data> data constant with a variable <Var> and the <ruleml:Ind> individual constant with a <Var>, but there is no means in RuleML to express the condition that the name of the individual constant should be the same character sequence as the lexical value of the data constant.

The first step in preparing the MXSL instance that expresses such a generalization is to apply the utility stylesheet `xsl/ruleml2mxsl.xsl`.

The `<ruleml:Ind>` element from the RuleML above is expressed in MXSL as

```
<xsl:element name="Ind"
     namespace="http://ruleml.org/spec">
  <xsl:copy-of select="'stringParam'"/>
</xsl:element>
```

An MXSL file is self-contained; that is, it is a stylesheet that takes itself as input. As written, the `<xsl:element>` declaration above will execute exactly once. In order to insert multiple `<ruleml:Ind>` elements into the output tree with different names, the content of the `<ruleml:Ind>` element may be generalized in MXSL as

```
<xsl:for-each select="$input/v:first/m:binding">
  <xsl:variable name="stringParam"
    select="v:stringParam" xsi:type="xs:string"/>
...
    <xsl:element name="Ind"
      namespace="http://ruleml.org/spec">
      <xsl:copy-of select="$stringParam/node()"/>
    </xsl:element>
...
```

An `<xsl:for-each>` contains an MXSL fragment that will be specialized multiple times, with different values substituted for the iteration parameter (`$stringParam` in the above case). The multiple specializations will be added to the output tree in the order in which the bindings appear in the input variable, as described in the next para-

graph. In the most general form of MXSL, the names and values of elements, attributes, processing instructions, and comments, may be generalized in this fashion.

Each `<xsl:for-each>` element should have a different designation, such as `v:first` above so that the variable bindings for the iteration may be uniquely identified. The XPath expression `$input/v:first/m:binding` is then used when "self-executing" an MXSL document to identify the parameter values to be substituted. A binding specification of the example above, with two different bindings for the parameter `stringParam`, takes the form

```
<variable name="input">
  <v:first>
    <m:binding>
      <v:stringParam xsi:type="xs:string"
        >a</v:stringParam>
    </m:binding>
    <m:binding>
      <v:stringParam xsi:type="xs:string"
        >b</v:stringParam>
    </m:binding>
  </v:first>
</variable>
```

Each MXSL file contains a self-referential xsl-processing instruction to initiate the XSLT transformation.

```
<?xml-stylesheet type="text/xsl" href="kb2.mxsl"?>
```

After XSLT processing of the MXSL example above, the output contains two formulas (with default RuleML namespace)

```
<Equal>
  <Expr>
    <Fun iri="http://athant.com/mxsl/vocab#constant"/>
    <Data xsi:type="xs:string">a</Data>
  </Expr>
  <Ind>a</Ind>
</Equal>
<Equal>
  <Expr>
    <Fun iri="http://athant.com/mxsl/vocab#constant"/>
    <Data xsi:type="xs:string">b</Data>
  </Expr>
  <Ind>b</Ind>
</Equal>
```

Notice that XSLT processing applied to the <xsl:for-each> element leads to multiple <Equal> elements occuring as siblings at the same depth in the tree rather than as a nesting to deeper levels of the tree. Recursive nesting, while possible in full XSLT through nesting of <xsl:call-template>, is not implemented in MXSL. The complete files for this example are available in the directory examples/kb2 .

4 Semantics for the Finitary Case

The semantics for the finitary case is based on the operational semantics of the XSLT processor, applied to the case when a non-empty but finite set of bindings is specified for all parameters. In this case, the output from XSLT processing of the MXSL file is guaranteed to produce well-formed XML, but not necessarily valid RuleML. Therefore, something more is required to fully define the semantics.

One approach for addressing this issue would be to require the user or generating program to pre-process the bindings to exclude those that lead to invalid RuleML. However, this would require complex conditionals to be generated for the input parameters, if it is even possible to specify such conditions. Instead, we define the metainterpretation transformation so that the invalid fragments are ignored (as described in the next paragraph), acting as an implicit syntactic-validation-based post-processor.

The implicit post-processing step will be described for the special case, called the "Rulebase-for-each" restriction, when every <xsl:for-each> element is the child of the MXSL equivalent of a <ruleml:Rulebase> element; that is

```
<xsl:element name="Rulebase" name-
space="http://ruleml.org/spec">
```

We consider each unit of XML produced by executing one iteration of XSLT processing on a single <xsl:for-each> element as a pre-formula. Any pre-formula that does not satisfy the (finitary) validity requirements for the syntax category of formulas is discarded, while all others are added to the XSLT output in the usual way (as siblings.)

It is an important consequence of the MXSL syntax that such invalid pre-formulas may be identified from the output of a standard XSLT processor applied to an MXSL instance. This is due to the removal of the XSLT feature (<xsl:text disable-output-escaping="yes">) that allows unescaped XML punctuation characters (<>'"&) to be inserted implicitly through parameters, rather than explictly with <xsl:element>. Because of the removal of this feature, every pre-formula is a well-formed XML child of a <ruleml:Rulebase> element. Therefore the post-processing step may be implemented on the output of a standard XLST processor by checking the validity of each child of every <ruleml:Rulebase> element against the RuleML schema (for the appropriate RuleML sublanguage.)[2]

[2] http://ruleml.org/spec

5 Semantics for the Infinitary Case

In the case when the input set of variable bindings for a particular <xsl:for-each> element is empty, we extend the operational semantics of the metalanguage to interpret the <xsl:for-each> element as an axiom schema. The theory of infinitary logic may be used to make this precise.

Again, we restrict our consideration to the "Rulebase-for-each"-restricted case. The default semantics of <ruleml:Rulebase> is an implicit conjunction, so that the theory of infinitary logics is applicable [14], whereby it is known that if the base (finitary) logic is first-order then the corresponding infinitary language with countable conjunctions and disjunctions but only finite sequences of quantifiers is complete with respect to the extended deductive apparatus that allows countably infinite sets of axioms and countably infinite proofs[3], but is not compact.

In more practical terms, the usual application is when a finitary knowledge base is extended with a finite number of axiom schemas. It is sufficient for reasoning purposes to consider only those instances of the axiom schemas that contain names from the vocabulary of the original finitary knowledge base. Consider, for example, the axioms schema for captured strings presented in Section 3. Forward reasoning of a finitary RuleML knowledge base yields at most a finite number of occurrences of names of the form

```
<ruleml:Ind>…</ruleml:Ind>
```

or expressions of the form

```
<ruleml:Expr>
  <ruleml:Fun
        iri="http://athant.com/mxsl/vocab#constant"/>
     …
</ruleml:Expr>
```

Therefore it is sufficient to generate a finite number of equations, corresponding to specializations of the axiom schema for the particular, finite, set of bindings relevant to the knowledge base. Because this axiom schema does not create any new names or expressions, there is no need to revisit the axiom schema during forward reasoning on the knowledge base extended by this finite set of equations. Therefore, this axiom schema does not undermine the finite-model property of the knowledge base to which it is added.

---

[3] According to Moore, G.H.: *The Emergence of First-Order Logic*. In P.K. William Aspray (ed.) *History and Philosophy of Modern Mathematics, Volume 11*. University of Minnesota Press, Minneapolis (1988), this result was first proved by Skolem in 1920 Skolem, T.: *Logisch-kombinatorische Untersuchungen über die Erfüllbarkeit oder Beweisbarkeit mathematischer Sätze nebst einem Theoreme über dichte Mengen*. *Videnskapsselskapet Skrifter, I. Matematisk-naturvidenskabelig Klasse 6*. p. 1–36 (1920).

In general, with proper design of the axiom schemas and naming conventions, i.e. avoiding an infinite cascade of new names or expressions, the set of relevant specializations of the axioms schemas will be finite, ensuring that reasoning on such extended knowledge bases will terminate provided the base logic is complete.

6 Extensions

The "Rulebase-for-each" restriction of MXSL may be safely relaxed to allow a `<ruleml:Assert> <xsl:for-each> <ruleml:Query>` tree. This is because such queries can be transformed to an equivalent "Rulebase-for-each" case by forming the negation of the existential closure of the formula contained in each query and inserting this into the rulebase which is being queried, where success of the query is equated with inconsistency of the resulting rulebase.

Further relaxation of the metalanguage incurs the risk of allowing the expression of infinite sequences of quantifiers. For example, suppose we allow a `<ruleml:And> <xsl:for-each>` tree. It is then possible to construct an instance such that every `<xsl:for-each>` iteration produces a new free variable, named according to value of the iteration parameter. For example,

```
<Exists closure="universal">
  <Var>F</Var>
  <And>
    <Atom>
      <Rel>Apply</Rel>
        <Var>F</Var>
        <Var>M1</Var>
        <Data>1</Data>
    </Atom>
  </And>
</Exists>
```

If the “And-for-each” construction is allowed, then we may create from the RuleML axiom above an MXSL axiom schema by generalizing the natural number 1 that appears both in the name of the variable `<Var>M1</Var>` and the content of the data element `<Data>1</Data>`. The resulting axiom schema can be applied to all natural numbers through the use of the datatype `xs:positiveInteger` for the schema parameter. See `examples/andForeach/andForeach/mxsl` for the complete axiom schema. The effect of the `closure` attribute is an infinite sequence of quantifiers, as this is necessary to implement closure on an infinite number of free variables. With this axiom schema we are well on our way to a characterization of the real numbers, and have thus left the realm of first-order logic and countable models, let alone finite models. Thus we cannot consider the “And-for-each” relaxation as a safe extension of MXSL.

Another possible extension of MXSL would be to allow recursive application of the metatransformation until a fixed point is reached, as was considered in [8]. Such recursion would implement a meta-circular evaluator [17]. In general it is an open

question as to how such extensions of the metalanguage would affect the logical properties of the resulting infinitary language.

7 Metalanguage Abstract Syntax

From the development and theory explored while developing MXSL, and described above, we have derived the following abstract syntax principles to form the foundation of a metalanguage development approach that is applicable to other KR formats, whether XML-based or not.

- The metalanguage uses a syntactic model of the base language, so that, for example, grouping symbols are entered as pairs through a single command.
- The hierarchical structure of a metalanguage instance mirrors the structure of the output document.
- An iteration command is available to generate a set of sibling items based on a pattern, instantiated with parameters.
- Text is always added through a metalanguage command, which implements output-escaping of punctuation characters appropriate to the base language.

All of the above can be implemented within XSLT for an arbitrary format. For example, a stylesheet with special purpose templates for an XSLT-based metalanguage for CLIF [18] has been implemented in `examples.cl1/cl-header.xsl`. This stylesheet may be imported into an MXSL-CLIF instance, allowing templates to be called, implementing, for example, the MXSL-CLIF extension syntax equivalent to `<xsl:element>`. That is, instead of XML tags, the expression is wrapped with parentheses and the name of the CLIF 'element' is placed into the first position of the list. MXSL-CLIF also has templates for the single quote and double quote containers for character sequences.

Of special importance is the handling of escaped punctuation. As was described in the previous section, proper handling of grouping and tokenizing symbols is a key feature for the output of the XSLT transformation to be parsable into pre-formulas, which will then be individually post-processed based on their syntactic validity in the base language. The imported MXSL-CLIF stylesheet uses the XSLT 2.0 character map and the XPath `translate` function to handling the escaping of the characters `' " ( ) \` that may appear in the value of iteration parameters.

An example illustrating the CLIF axiom schema for the captured string syntax is given in `examples/cl1/cl1.mxsl` . As an excerpt, the syntax for the MXSL command to wrap a character sequence in double quotes looks like

```
<xsl:call-template name="element-dquote">
  <xsl:with-param name="arg"
    select="$stringParam/node()"/>
</xsl:call-template>
```

Similar extensions to MXSL are possible for other formats, such as JSON. Although it appears awkward to have an XML metalanguage for rather different formats, such as CLIF or JSON, there are several advantages to this approach. For example, the common XML format of the metalanguages permits CLIF texts to be embedded into the RuleML XML syntax as data, while the semantic content is retained by defining axiom schemas relating parameterized CLIF sentences to the associated RuleML formulas.
Further, XML has an advantage that many of these other formats lack: prefixes and namespaces. The prefixes of the XML element names, and their associated namespaces, allow us to distinguish the components of the metalanguage from the components of the base language and avoid naming collisions.

8 Conclusions

We have shown that a (small) subset of the XSLT language, which we call MXSL can be used as a convenient, flexible and powerful metalanguage for XML-based KR languages. This metalanguage may be used to express axiom schemas, the semantics of structured data and generalized queries. The paradigm used to develop the metalanguage for the XML syntax can be applied to non-XML-based formats. Future investigations will explore the implementation of reasoning using the MXSL metalanguage.

References

1. *W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures (W3C Working Draft 3 December 2009)*, http://www.w3.org/TR/xmlschema11-1/#cTypeAlternative.
2. ISO/IEC: 19757-2 Document Schema Definition Language (DSDL) Part 2: Regular-grammar-based validation - RELAX NG. International Organization for Standardization, Geneva (2008)
3. ISO: 19757-3 Information technology -- Document Schema Definition Language (DSDL) -- Part 3: Rule-based validation -- Schematron. International Organization for Standardization, Geneva (2006)
4. *Semantic Web on XML: Architecture*, http://www.w3.org/2000/Talks/1206-xml2k-tbl/slide10-0.html.
5. *Introducing JSON*, http://json.org/.
6. *A Web Data Model Unifying XML and RDF*, http://www.dfki.uni-kl.de/~boley/xmlrdf.html.
7. Patel-Schneider, P., et al.: The Yin/Yang web: XML syntax and RDF semantics. Proceedings of the 11th international conference on World Wide Web. p. 443-453. ACM, Honolulu, Hawaii, USA (2002)
8. Athan, M.: XCLX: An XML-based Common Logic eXtension with Embedded Geography Markup Language. Geographic Information Systems. p. 77. University of Leeds, (2011)

9. Boley, H., A. Paschke, and O. Shafiq: *RuleML 1.0: The Overarching Specification of Web Rules*. In M. Dean, et al. (eds.) *Semantic Web Rules*. p. 162-178. Springer Berlin / Heidelberg (2010)
10. *XSL Transformations (XSLT) Version 1.0*, http://www.w3.org/TR/xslt.
11. *XQuery 1.0 and XPath 2.0 Data Model (XDM) (Second Edition)*, http://www.w3.org/TR/xpath-datamodel/.
12. *Universal Turing Machine in XSLT*, http://www.unidex.com/turing/utm.htm.
13. *IKL Guide*, http://www.ihmc.us/users/phayes/IKL/GUIDE/GUIDE.html.
14. *Infinitary Logic*, http://plato.stanford.edu/entries/logic-infinitary/.
15. Moore, G.H.: *The Emergence of First-Order Logic*. In P.K. William Aspray (ed.) *History and Philosophy of Modern Mathematics, Volume 11*. University of Minnesota Press, Minneapolis (1988)
16. Skolem, T.: Logisch-kombinatorische Untersuchungen über die Erfüllbarkeit oder Beweisbarkeit mathematischer Sätze nebst einem Theoreme über dichte Mengen. Videnskapsselskapet Skrifter, I. Matematisk-naturvidenskabelig Klasse 6. p. 1–36 (1920)
17. *The Metacircular Evaluator*, http://mitpress.mit.edu/sicp/full-text/book/book-Z-H-26.html#%_sec_4.1.
18. ISO/IEC: Information technology — Common Logic (CL): a framework for a family of logic based languages. International Organization for Standardization, Geneva (2007)