**Semantic Attributes and Metadata in NIEM 6**

Our goal is to support multiple serializations of NIEM data, and not just XML. For this to matter, each of those different serializations – NIEM JSON, NIEM RDF, etc. – must be accepted by the relevant developers. If our NIEM serializations are too ugly, unconventional, or difficult, then we might as well not bother.

These multiple serializations (and the lossless round-trip translations among them) are possible because the NIEM conceptual model is based on the RDF conceptual model, and because all NIEM data has an RDF interpretation. Two NIEM messages with different serializations are equivalent if they are each equivalent to the same RDF graph. For example, these two messages are equivalent:[[1]](#footnote-1)

|  |  |
| --- | --- |
| <nc:Email>  <nc:MessageID>M1</nc:MessageID>  <nc:MessageSentDate>  <nc:Date>2022-03-31</nc:Date>  </nc:MessageSentDate>  </nc:Email>  **NIEM XML** | {  "nc:Email": {  "nc:MessageID": "M1",  "nc:MessageSentDate": {  "nc:Date": "2022-03-31"  }  }  }  **NIEM JSON** |

Both messages resolve to the same NIEM RDF (Turtle syntax):

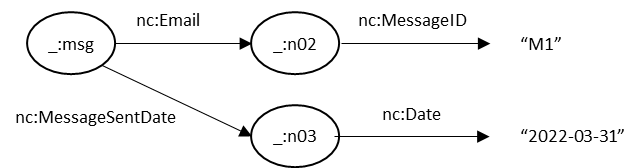
\_:msg nc:Email :\_n02 .

\_:n02 nc:MessageID "M1 .

\_:n02 nc:MessageSentDate \_:n03 .

\_:n03 nc:Date "2022-03-31" .

A diagram of that RDF graph looks like this:



The NIEM conceptual model has *objects*, *properties*, and *values*.[[2]](#footnote-2) In terms of the conceptual model, the above example contains:

* 3 objects: message object (\_:msg), email object (\_:n02), and sent-date object (\_:n03)
* 4 properties: nc:Email, nc:MessageID, nc:MessageSentDate, and nc:Date
* 2 literal values ("M1" and "2022-03-31")
* 2 object values (\_:n02 and \_:n03)

The objects, properties, and values in NIEM data are described by and conform to a *message model,* which is a NIEM data model. (NIEM core and domain models are also NIEM data models, but not message models.) A data model defines:

* The *object type* of each object, known as the object's *class*
* The *datatype* of each literal
* The *has-a* relationship denoted by properties; e.g. there is a *has-a-MessageID* relationship from the email object to the "M1" literal value.

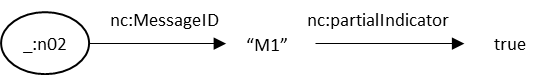
A property with an object value is an *object property.* A property with a literal value is a *datatype property.*

A NIEM data model specifies the optional and required properties in all objects of a class. It specifies the range of each object property (which is a class). It specifies the range of each datatype property (which is a datatype). At present, NIEM data models are expressed in XSD. In NIEM 6 they will also be expressed in the *Common Model Format (CMF).* The part of the message model defining the nc:MessageID property looks like this:

|  |  |
| --- | --- |
| <xs:element  name="MessageID"  type="niem-xs:string"/>  **XSD** | <Property>  <Name>MessageID</Name>  <Namespace s:uri="nc"/>  <Datatype s:uri="xs:string"/>  </Property>  **CMF** |

**NIEM without attributes is wonderful**

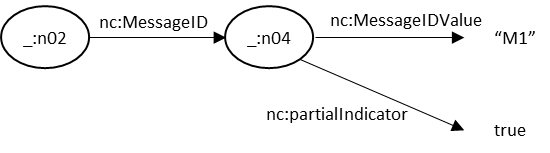
Let's begin by visiting the wonderful imaginary world without XML attributes, and see what happens when we need to add a property to a MessageID literal. Those strings can be very long,so we might want to truncate the string in our message, and put a "truncated=true" property on the shortened string value. We would like to create a message graph like this:



Bzzzt! We can't do that! Literals can't have a property. If you want to have a property like nc:partialIndicator, you must have an object to hang it from. What we must do instead is:

* Convert nc:MessageID from a datatype property to an object property
* Create an object class nc:MessageIDType for this new object property
* Create a new datatype property nc:MessageIDValue for the literal "M1"

The resulting graph looks like this:



The message data looks like this:

|  |  |
| --- | --- |
| <nc:MessageID>  <nc:MessageIDValue>M1  <nc:partialIndicator>true  </nc:MessageID>  **NIEM XML** | {  "nc:MessageID": {  "nc:MessageIDValue": "M1",  "nc:partialIndicator": true  }  }  **NIEM JSON** |

Both messages resolve to the same NIEM RDF (Turtle syntax):

\_:n02 nc:MessageID \_:n04 .

\_:n04 nc:MessageIDValue "M1" .

\_:n04 nc:partialIndicator "true" .

The message model looks like this in XSD:

<xs:element name="MessageID" type="nc:MessageIDType"/>

<xs:complexType name="MessageIDType>

<xs:complexContent>

<xs:extension base="s:ObjectType">

<xs:sequence>

<xs:element ref="nc:MessageIDValue"/>

<xs:element ref="nc:partialIndicator"/>

</xs:sequence>

</xs:extension>

</xs:complexContent>

</xs:complexType>

<xs:element name="MessageIDValue" type="niem-xs:string"/>

<xs:element name="partialIndicator" type="niem-xs:boolean" minOccurs="0"/>

In CMF the message model looks like this:

<Class>

<Name>MessageIDType</Name>

<HasProperty>

<Property s:uri="nc:MessageIDValue"/>

<MinOccursQuantity>1</MinOccursQuantity>

<MaxOccursQuantity>1</MaxOccursQuantity>

</HasProperty>

<HasProperty>

<Property s:uri="nc:partialIndicator"/>

<MinOccursQuantity>0</MinOccursQuantity>

<MaxOccursQuantity>1</MaxOccursQuantity>

</HasProperty>

<Class>

<Property>

<Name>MessageIDValue</Name>

<Datatype s:uri="xs:string"/>

</Property>

<Property>

<Name>partialIndicator</Name>

<Datatype s:uri="xs:boolean"/>

</Property>

Everybody is happy, because:

1. Programmers know exactly what they are going to get
2. All of the data components appear in the message model, in both XSD and CMF
3. They are the same components in all of the serializations; i.e. you find nc:MessageIDValue in the XML, JSON, and RDF data.

Now, the nc:partialIndicator property is optional. The message looks like this if we leave it out:

|  |  |
| --- | --- |
| <nc:MessageID>  <nc:MessageIDValue>M1  </nc:MessageID>  **NIEM XML** | {  "nc:MessageID": {  "nc:MessageIDValue": "M1"  }  }  **NIEM JSON** |

Both messages resolve to the same NIEM RDF (Turtle syntax):

\_:n02 nc:MessageID \_:n04 .

\_:n04 nc:MessageIDValue "M1" .

Everybody is still fairly happy, for the same three reasons. Some programmers will gripe about the "unnecessary" nesting . "Why must my message include both nc:MessageID and nc:MessageIDValue?" Well, that's part of the data model, which includes nc:partialIndicator because some people need it, even if you don't. (We might be able to satisfy this gripe with component aliases, later on.)

**NIEM with attributes means trouble**

Let us now return to the real world, which includes XML attributes. Attributes on complex content are nothing special. You can write

<nc:Email nc:someAttribute="foo">

<nc:MessageID>M1</nc:MessageID>

<nc:MessageSentDate>

<nc:Date>2022-03-31</nc:Date>

</nc:MessageSentDate>

</nc:Email>

and no one will bat an eye; the attribute is just one more property on the nc:Email object. However, attributes on simple content are trouble. XML is different from every other serialization we are likely to support, in that it is possible to put a property on a literal via an attribute. In the previous example we added a "truncated" property to nc:MessageID through an element. If you do it with an attribute instead, the XML message data looks like this:

<nc:MessageID nc:partialIndicator="true">M1</nc:MessageID>

In NIEM JSON the message looks like this:

{

"nc:MessageID": {

"rdf:value": "M1",

"nc:partialIndicator": "true"

}

}

Both messages resolve to this RDF:

\_:n02 nc:MessageID \_:n04 .

\_:n04 rdf:value "M1" .

\_:n04 nc:partialIndicator "true" .

The message model with attributes is different. In XSD, it looks like this:

<xs:element name="MessageID" type="nc:MessageIDType"/>

<xs:complexType name="MessageIDType>

<xs:simpleContent>

<xs:extension base="niem-xs:string">

<xs:attribute ref="nc:partialIndicator" use="optional"/>

</xs:extension>

</xs:complexContent>

</xs:complexType>

<xs:element name="partialIndicator" type="niem-xs:boolean" minOccurs="0"/>

At present the CMF representation looks like this:

<Class>

<Name>MessageIDType</Name>

<HasValue s:uri="xs:string"/>

<HasProperty>

<Property s:uri="nc:partialIndicator"/>

<MinOccursQuantity>0</MinOccursQuantity>

<MaxOccursQuantity>1</MaxOccursQuantity>

</HasProperty>

<Class>

<Property>

<Name>partialIndicator</Name>

<Datatype s:uri="xs:boolean"/>

</Property>

Finally, if the message omits the optional nc:partialIndicator attribute, the JSON serialization will look like this:

{

"nc:MessageID": "M1"

}

At this point the JSON developers are *not* happy – and I expect their emotions will be shared by developers using GPB, YAML, and most any other serialization we decide to support. They are unhappy because:

1. *Programmers know exactly what they are going to get.* Not any more, they don’t. The value of the nc:MessageID key can be either a literal or an object. JSON developers will have to write code to do one thing with a literal, another thing with an object, for every property that can have an attribute. They will hate that.
2. *All of the data components appear in the message model, in both XSD and CMF.* Not any more, they don't. The rdf:value component does not appear in the model at all. It will be completely alien to JSON developers. They will hate it. (Some RDF developers will recognize rdf:value as an old RDF idiom; I believe this was never widely adopted and is now out of favor, but I could be wrong.)
3. *The CMF model is slanted towards XML.* At present, CMF has this funky HasValue element to represent simple content with attributes. This is a model for an attribute; it only makes sense for XML. The technology-neutral way to represent a property on a literal value is to create a class – as shown in the CMF example on page 3.

**What to do about semantic attributes**

NIEM XML is packed full of attributes. Much as I would like to, we can't make them go away. However, there is some good news:

1. Relatively few are *semantic attributes* expressing some property of the message model*.*  Most are part of the plumbing, like s:id and s:ref. Plumbing attributes do not cause the problems described above.
2. Most elements in most messages do not have attributes, either because the element type does not define any attributes, or because the attributes defined were omitted from the message model subset.
3. The message model tells developers exactly where attributes can appear.

So if a change proposal for attributes causes any pain, it will only cause that pain in a few places.[[3]](#footnote-3)

My proposal is to change the RDF interpretation of simple content with semantic attributes. We can revise the NDR to say that if you have XML like

<nc:MessageID nc:partialIndicator="true">M1</nc:MessageID>

then the RDF interpretation is exactly what you would get if you had used an element instead, namely:

\_:n02 nc:MessageID \_:n04 .

\_:n04 nc:MessageIDLiteralValue "M1" .

\_:n04 nc:PartialIndicator "true" .

And since NIEM RDF and NIEM JSON are joined at the hip, this fixes the JSON as well:

{

"nc:MessageID": {

"nc:MessageIDLiteralValue": "M1",

"nc:PartialIndicator": true

}

}

We would also change the CMF representation of simple content with attributes, replacing HasValue with a datatype property, and indicating properties that should be XML attributes, like this:

<Class>

<Name>MessageIDType</Name>

<HasProperty>

<Property s:uri="nc:MessageIDLiteralValue"/>

<MinOccursQuantity>1</MinOccursQuantity>

<MaxOccursQuantity>1</MaxOccursQuantity>

</HasProperty>

<HasProperty>

<Property s:uri="nc:PartialIndicator"/>

<MinOccursQuantity>0</MinOccursQuantity>

<MaxOccursQuantity>1</MaxOccursQuantity>

<cmx:AttributeIndicator>true</cmx:AttributeIndicator>

</HasProperty>

<Class>

You may have noticed that nc:PartialIndicator uses lower camel case only in the XML. Everything else (CMF, JSON, RDF) uses upper camel case. The conversion would happen in CMFTool and NIEMTran. I think this approach is better for non-XML developers, but I'm not married to it.)

You may have noticed nc:MessageIDLiteralValue instead of nc:MessageIDValue. I don't like it, either -- but there are 18 places in the NIEM 5.0 model where there is a FooValue and also a FooType with simple content and attributes, and I don't think we are changing all of them.

And now, I think everyone is fairly happy again, because:

1. Programmers know exactly what they are going to get
2. All of the data components appear in the message model, in both XSD and CMF
3. They are the same components in all of the serializations; i.e. no ugly rdf:value for everything but XML.

**What to do about metadata on simple content**

There is no happy ending here. Adding a metadata attribute to simple content introduces all the problems described above. The problems are more severe, because a metadata attribute can appear *anywhere,* not on a relatively few elements that are all known in advance via the message model. I think we have three choices:

1. We can maintain the status quo, and hope that non-XML developers will come to accept rdf:value properties, and be willing to test every data value to see if it is an object or a literal.
2. We can apply the FooLiteralValue trick to every complex type with simple content (and not just those with attributes).
3. We can eliminate metadata on simple content.

I vote for Door #3. It is much like our solution for relationship metadata. There we said that if you want metadata on a relationship, then you must include a class for the relationship in your message model. Option #3 says that if you want metadata on a literal, then you must include a class with the literal as a property in your message model. Option #3, unlike the other two, has no impact on the vast majority of simple content elements.

If we combined our design pattern for relationship metadata with our idea of using elements for metadata, then metadata on simple content might look like this:

|  |  |
| --- | --- |
| <my:MessageIDRelationship>  <nc:MessageID>M1  <nc:Metadata>  <nc:EffectiveDate>2022-03-31  </nc:Metadata>  </my:MessageIDRelationship>  **NIEM XML** | {  "my:MessageIDRelationship": {  "nc:MessageID": "M1",  "nc:Metadata": {  "nc:EffectiveDate": "2022-03  }  }  }  **NIEM JSON** |

Both messages would resolve to this RDF:

\_:n01 nc:MessageIDRelationship \_:n02 .

\_:n02 nc:MessageID "M1" .

\_:n02 nc:Metadata \_:n03 .

\_:n03 nc:EffectiveDate "2022-03-31" .

The message model in XSD would contain the following in the "my" namespace:

<xs:complexType name="MessageIDRelationship>

<xs:complexContent>

<xs:extension base="s:ObjectType">

<xs:sequence>

<xs:element ref="nc:MessageID"/>

<xs:element ref="nc:Metadata" minOccurs="0"/>

</xs:sequence>

</xs:extension>

</xs:complexContent>

</xs:complexType>

Since we are creating a new class for the object that will hold the metadata property, we might as well skip the augmenation business. (Metadata for complex content would use augmentations in this approach.)

In CMF we would see:

<Class>

<Name>MessageIDRelationship</Name>

<Namespace s:uri="my"/>

<HasProperty>

<Property s:uri="nc:MessageID"/>

<MinOccursQuantity>1</MinOccursQuantity>

<MaxOccursQuantity>1</MaxOccursQuantity>

</HasProperty>

<HasProperty>

<Property s:uri="nc:Metadata"/>

<MinOccursQuantity>0</MinOccursQuantity>

<MaxOccursQuantity>1</MaxOccursQuantity>

</HasProperty>

<Class>

It's not elegant, but if you really need metadata on simple content, it would work. We need more study on exactly what such metadata would mean, and when it would be needed.

**What to do about the plumbing attributes (s:id, s:ref, s:uri)**

These attributes are even worse than semantic or metadata attributes. All of the old problems apply. The new problem is that there is no RDF equivalent of these attributes. Literals do not have identifiers in RDF. If you need an identifier, you must have an object. The RDF equivalent for XML like this:

<nc:MessageID s:id="foo">ABCD</nc:messageID>

<nc:MessageID s:ref="foo"/>

would be something like this:

\_:n1 nc:MessageID ABCD .

\_:n2 nc:MessageID *the exact same literal in that tuple over there*

There's no way to say that in RDF. Either we get rid of plumbing attributes on simple content, or we give up on complete convertibility among NIEM serializations. (By the way, writing software to process NIEM XML is *much easier* if you do not have to worry about s:ref pointers on simple content elements. I mean, a lot. I doubt if one developer in ten is actually writing code that meets our specification.)

**What to do about s:sequenceID**

I want to get rid of this in NIEM 6 and replace it with an "order is significant" property on object classes in CMF, and equivalent appinfo in XSD.

**What if we do nothing?**

I believe the status quo is somewhat hostile to JSON developers. I think the rdf:value key, the ubiquitous object-or-literal test, and the XML-centric CMF will all be an obstacle to adoption. I also believe the long-term future for machine-to-machine data exchange is JSON and YAML, not XML. So I think we should find some alternatives to the status quo – those proposed above, or something even better.

1. Examples in this paper may omit closing tags, shorten component names, etc. for clarity. [↑](#footnote-ref-1)
2. These correspond to *subjects, predicates,* and *objects* in the RDF conceptual model. [↑](#footnote-ref-2)
3. Festus: *Doc, Doc, my arm hurts in three places!*

   Doc: *Well, then stay outta those three places!*

   You're welcome 😊. [↑](#footnote-ref-3)