**NIEM Message Specifications**

This is a proposal for message specifications in NIEM 6.0. It’s for NTAC discussion.

One goal is to accommodate any combination of these three things:

* A specification for a single kind of message (a *message class*), or more than one
* A single serialization (XML, JSON) for a message class, or more than one
* The use of canonical properties (nc:PersonSurName) or simple properties (lname)

Thus a message specification can be simple: one message class, one serialization, only canonical properties. A message specification can be complex: several message classes, with multiple serializations, and either canonical or simple properties (or both).

Another goal is to promote convention over configuration. A simple specification should require only a few configuration details. I’ve tried to distinguish between three parts of a message specification:

* Information that must be provided by the message designer in a configuration
* Information that may be inferred from convention or provided by explicit configuration
* Information that can be generated by tools and build automation

Finally, I’d like to support some form of build automation for message specification, so that message designers make their changes to the manifest and model artifacts, and then the other artifacts (validation schemas, translation scripts, etc.) are generated from those.

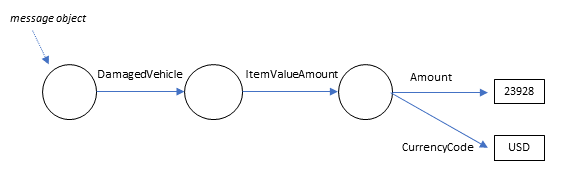
**Definitions**

I am going to define and describe the following:

* message
* message specification
* message class
* message format
* model constraint file
* property mapping file

*Message***:** A message is an information exchange package: a structured collection of facts that may be shared among applications or systems. Conceptually, a message is a directed graph, in which the nodes are objects or literal values, and the edges are properties. There is a distinguished node, the *message object*, from which all other nodes are reachable.

Physically, a message is a serialization of that conceptual graph in some data syntax; for example, XML or JSON. Two messages are equivalent if they serialize the same conceptual graph. There is always an equivalent message in every supported data syntax. For example, the following XML and JSON messages are serializations of the graph above, and are equivalent to each other.



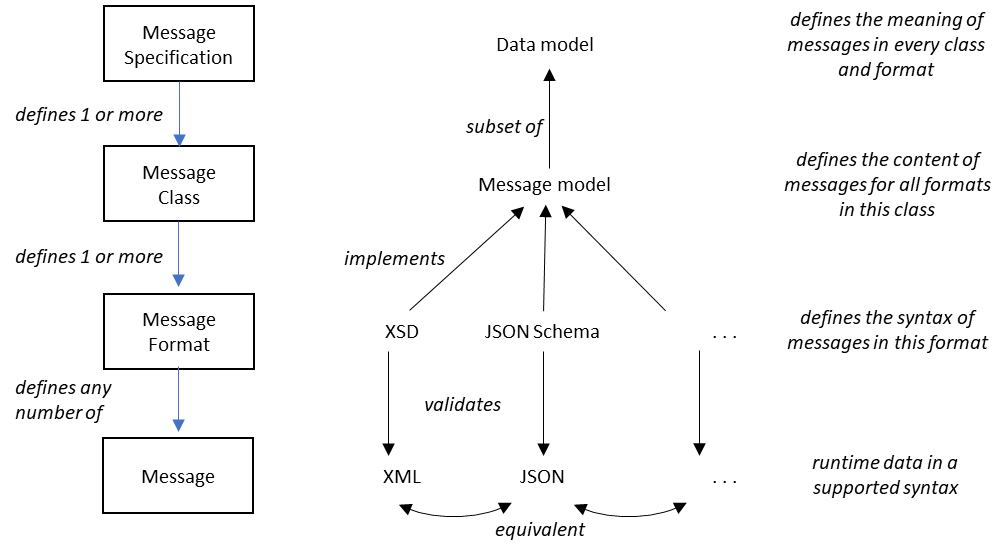
<c:Claim> | {  
 <c:DamagedVehicle> | "c:DamagedVehicle": {  
 <nc:ItemValue> | "nc:ItemValue": {  
 <nc:ItemValueAmount> | "nc:ItemValueAmount": {  
 <nc:Amount>23928</Amount> | "nc:Amount": 23928,  
 <nc:CurrencyCode>USD | "nc:CurrencyCode": "USD"  
 </nc:ItemValueAmount> | }  
 </nc:ItemValue> | }  
 </c:DamagedVehicle> | }  
</c:Claim> | }

*Message specification:* A message specification defines the content and meaning of messages that are instances of one or more related *message classes*. The meaning is described by the specification’s *data model*, which is expressed in CMF or XSD.

*Message class:* At the conceptual level, every message object is an instance of a *message class*. A message class defines the mandatory and optional content of its instance messages. It has a *message model,* which is a subset of the data model for the message specification. It specifies a *message property,* and through that a message type. In the above example, the message property is Claim, and the message type is ClaimType.

*Message format:* At the physical level, every message is a instance of a *message format*. A message format defines the data syntax for the information content in a message class; for example XML or JSON. It usually provides a schema for assessing the validity of a message: XSD for XML messages, JSON Schema for JSON messages.

The following figure shows the relationship between message specification, message class, message format, and messages.

****

*Model constraint file:* Specifies a subset of a CMF data model, which is generated from a model file plus the model constraint file. Everything valid in the subset is also valid in the original model. Definitions do not change. A constraint can change the cardinality of a property in an object type, or in a particular object property. A constraint can change a data type or particular data properties by adding or changing facets, or by removing enumerations. Every message class has an optional model constraint file to specify how its message model is derived from the message specification’s data model.

*Property mapping file:* Every property has a canonical form, such as nc:PersonSurName. Message designers sometimes wish to use equivalent simple names for some or all of the properties in their messages; for example, lname instead of nc:PersonSurName. A property mapping file specifies these equivalences. It is used to generate a @context for JSON formats, and to generate translation XSLT between simple and canonical XML formats.

**Example message specifications**

These examples run from very simple to fairly complex. I’m using them to describe three things:

* Required manifest content. For instance, there has to be some way to specify the message property of each message class. That either goes into some sort of manifest, or into the model file (as a property in CMF, as appinfo in XSD). I think the manifest is a better choice.
* Conventional directory structure and file names for simple and complex message specifications. Anything that follows the convention does not have to be configured in the manifest.
* Manifest content needed for a NIEM-bespoke build automation tool. I’d like to provide one. I don’t think it will be all that hard.

**The simplest specification: XSD model, one class, one canonical XML format**

Let’s look at the layout of the specification for the insurance claim examples above. This is the simplest message specification, having one class, one format, and canonical properties. The finished specification would look like this:

|  |  |  |
| --- | --- | --- |
| **Directory / File name** | **Description** | **G** |
| Claim.mspec/ | message specification directory |  |
| manifest.{xml,json,yaml} | it’s a NIEM message; can be any supported syntax |  |
| model.xsd/ | data model in a NIEM-conforming XML schema |  |

This is the simplest possible message specification. The schema in *model.xsd* is constructed by loading every XML catalog file, followed by every XSD file, in the directory. That schema is also used for message validation. There is nothing for build automation to do, so the only information required in the manifest is the URI of the message element.

messageProperty: http://example.org/claim/1.0/#Claim *required*

**Almost as simple: CMF model, one class, one canonical XML format**

When the data model is in CMF, we have to generate an XML schema to validate XML messages. Build automation should do that for us, generating the constraint schema for validation whenever the model is changed.

|  |  |  |
| --- | --- | --- |
| **Directory / File name** | **Description** | **G** |
| Claim.mspec/ | message specification directory |  |
| manifest.{xml,json,yaml} | it’s a NIEM message; can be any supported syntax |  |
| model.cmf | data model in CMF |  |
| validate.xsd/ | constraint schema for message validation | X |

A constraint schema does not have to capture the entire data model and does not have to follow the NDR, and so it can be simplified by replacing subsitution groups with xs:choice, and by reducing long type derivation chains. The manifest now contains:

messageProperty: http://example.org/claim/1.0/#Claim *required*

syntax: xml *for build auto*

**Simple simple: CMF model, one class, one simple XML format**

If we decide to support the Simple XML format, then we would generate a constraint schema having a single namespace, single schema document, all element declarations local except the message element, and canonical properties replaced with programmer-friendly names. We would also generate translation scripts between the simple XML format and the canonical XML. We would get a message specification like this:

|  |  |  |
| --- | --- | --- |
| **Directory / File name** | **Description** | **G** |
| Claim.mspec/ | message specification directory |  |
| manifest.{xml,json,yaml} | it’s a NIEM message; can be any supported syntax |  |
| model.cmf | data model in CMF |  |
| propertyMap.\* | optional; a NIEM message, or maybe just CSV |  |
| trans2canon.xslt | conversion script from simple to canonical | X |
| trans2simple.xslt | conversion script from canonical to simple | X |
| validate.xsd | constraint schema, now a single file | X |

The manifest now contains:

messageProperty: http://example.org/claim/1.0/#Claim *required*

uriSuffix: simple.xml *convention?*

syntax: simple.xml *for build auto*

**Still very simple: CMF model, one class, JSON format**

This time we are generating JSON Schema and a JSON-LD context from the CMF model.

|  |  |  |
| --- | --- | --- |
| **Directory / File name** | **Description** | **G** |
| Claim.mspec/ | message specification directory |  |
| context.json | JSON-LD context for this format | X |
| manifest.{xml,json,yaml} | it’s a NIEM message; can be any supported syntax |  |
| model.cmf | data model in CMF |  |
| propertyMap.\* | optional; a NIEM message, or maybe just CSV |  |
| validate.json | JSON Schema generated from CMF | X |

We generate JSON Schema for message validation, and a JSON-LD context to supply URIs for all of the model components. Property mappings, if supplied, become part of the context. JSON messages [don’t have a message property](#properties); however, the data model must have one anyway for the URI of the message class. The manifest now contains:

messageProperty: http://example.org/claim/1.0/#Claim *required*

syntax: json *for build auto*

**Complexity! CMF model, one class, three message formats**

Things get a bit complicated when the message designer wants artifiacts for more than one format. The convention is a separate directory for each format.

.

|  |  |  |
| --- | --- | --- |
| **Directory / File name** | **Description** | **G** |
| Claim.mspec/ | message specification directory |  |
| manifest.{xml,json,yaml} | it’s a NIEM message; can be any supported syntax |  |
| model.cmf | data model in CMF |  |
| propertyMap.\* | optional; applies to JSON and simple XML format |  |
| json/ |  |  |
| context.json | JSON-LD context for this format | X |
| propertyMap.\* | optional; different/additional mappings for JSON |  |
| validate.json | JSON Schema generated from CMF | X |
| simple.xml/ |  |  |
| propertyMap.\* | optional; different/additional mappings for simple XML |  |
| trans2canon.xslt | conversion script from simple to canonical | X |
| trans2simple.xslt | conversion script from canonical to simple | X |
| validate.xsd | constraint schema, now a single file | X |
| xml/ |  |  |
| validate.xsd/ | constraint schema for message validation | X |

The manifest now contains:

messageProperty: http://example.org/claim/1.0/#Claim *required*

formats:

- directory: json *convention?*

syntax: json *for build auto*

uriSuffix: json *convention?*

- directory: simple.xml *convention?*

syntax: simple.xml*for build auto*

uriSuffix:simple.xml *convention?*

- directory: xml *convention?*

syntax: xml *for build auto*

**More complexity! CMF model, two classes, one format apiece**

This example specifies two message classes: a claim, and a claim response. We’ll use JSON syntax for the message formats.

|  |  |  |
| --- | --- | --- |
| **Directory / File name** | **Description** | **G** |
| Claim.mspec/ | message specification directory |  |
| manifest.{xml,json,yaml} | it’s a NIEM message; can be any supported syntax |  |
| model.cmf | data model in CMF |  |
| class.Claim/ |  |  |
| constraint.cmf | optional model constraint file for this class |  |
| context.json | JSON-LD context for this format | X |
| propertyMap.\* | optional; a NIEM message, or maybe just CSV |  |
| validate.json | JSON Schema generated from CMF | X |
| class.ClaimResponse/ |  |  |
| constraint.cmf | optional model constraint file for this class |  |
| context.json | JSON-LD context for this format | X |
| propertyMap.\* | optional; a NIEM message, or maybe just CSV |  |
| validate.json | JSON Schema generated from CMF | X |

The manifest now contains:

classes: *required*

- directory: class.Claim *convention?*

messageProperty: http://example.org/claim/1.0/#Claim *required*

syntax: json *for build auto*

- directory: class.ClaimResponse *convention?*

messageProperty: http://example.com/Claim/1.0/#ClaimResponse *required*

syntax: json *for build auto*

**All the complexity! CMF model, two classes, two formats apiece**

This example specifies two message classes: a claim, and a claim response. We’ll use JSON and simple XML syntax for the message formats.

|  |  |  |
| --- | --- | --- |
| **Directory / File name** | **Description** | **G** |
| Claim.mspec/ | message specification directory |  |
| manifest.{xml,json,yaml} | it’s a NIEM message; can be any supported syntax |  |
| model.cmf | data model in CMF |  |
| class.Claim/ |  |  |
| constraint.cmf | optional model constraint file for this class |  |
| propertyMap.\* | optional; a NIEM message, or maybe just CSV |  |
| json/ |  |  |
| context.json | optional model constraint file for this class | X |
| validate.json | JSON-LD context for this format | X |
| simple.xml/ |  |  |
| propertyMap.\* | optional; different/additional mappings for simple XML |  |
| trans2canon.xslt | conversion script from simple to canonical | X |
| trans2simple.xslt | conversion script from canonical to simple | X |
| validate.xsd | constraint schema, now a single file | X |
| class.ClaimResponse/ |  |  |
| constraint.cmf | optional model constraint file for this class |  |
| propertyMap.\* | optional; a NIEM message, or maybe just CSV |  |
| json/ |  |  |
| context.json | optional model constraint file for this class | X |
| validate.json | JSON-LD context for this format | X |
| simple.xml/ |  |  |
| propertyMap.\* | optional; different/additional mappings for simple XML |  |
| trans2canon.xslt | conversion script from simple to canonical | X |
| trans2simple.xslt | conversion script from canonical to simple | X |
| validate.xsd | constraint schema, now a single file | X |

The manifest now contains:

classes: *required*

- directory: class.Claim *convention?*

messageProperty: http://example.org/claim/1.0/#Claim *required*

formats: *required*

- directory: json *convention?*

syntax: json *for build auto*

uriSuffix: json *convention?*

- directory: simple.xml *convention?*

syntax: simple.xml *for build auto*

uriSuffix: simple.xml *convention?*

- directory: class.ClaimResponse *convention?*

messageProperty: http://example.com/Claim/1.0/#ClaimResponse *required*

formats: *required*

- directory: json *convention?*

syntax: json *for build auto*

uriSuffix: json *convention?*

- directory: simple.xml *convention?*

syntax: simple.xml *for build auto*

uriSuffix: simple.xml *convention?*

**Properties, classes, and types, oh my!**

I want a rule that says each message class must have its own message type; that is, no two message classes in a specfication can have the same type. Otherwise we have to require a usually-useless message property in JSON messages. The story goes like this…

|  |  |
| --- | --- |
| The message property in an XML message is usually just a wrapper around the real contents. | <Claim>  <DamagedVehicle>  <VehicleIdentification>  <IdentificationID>JYAVN01E7CA042113  </VehicleIdentification>  </DamagedVehicle>  <VehicleDamageAmount>  <Amount>1432.28</nc:Amount>  <CurrencyCode>USD</nc:CurrencyCode>  </VehicleDamageAmount> </Claim> |
| In JSON syntax, the message property is useless. It’s just an extra object that nobody needs. | {  “@context”: “http://example.com/claim/”,  “Claim”: {  “DamagedVehicle”: {  “VehicleIdentification”: {  “IdentificationID”: “JYAVN01…  }  },  “VehicleDamageAmount”: {  “Amount”: 1432.28,  “CurrencyCode”: “USD”  }  }  } |
| And in RDF, it produces a blank node with an unknown type. | \_:n0 Claim \_:n1 .  \_:n1 a ClaimType .  \_:n1 “DamagedVehicle” \_:n2 .  \_:n1 “VehicleDamageAmount” \_:n3 . |
| So the sensible thing to do is omit the message property from JSON, RDF, and probably every other syntax | {  “@context”: “http://example.com/claim/”,  “DamagedVehicle”: {  “VehicleIdentification”: {  “IdentificationID”: “JYAVN01…  }  “VehicleDamageAmount”: {  “Amount”: 1432.28,  “CurrencyCode”: “USD”  }  } |
| Which works great until someone writes a message specification with two message classes that share the message type, like this | <xs:element name=”Claim” type=”MsgType”/>  <xs:element name=”Resp” type=”MsgType”/> |
| Presto! The message property is now meaningful. So do we have to stick it into all our JSON and RDF, even though it is almost always useless there?.  *G$@!FFD!!!* | |
| Instead, let’s have a rule that no two message properties can have the same type. You have to do this, instead: | <xs:element name=”Claim” type=”ClaimType”/>  <xs:element name=”Resp” type=”RespType”/> |
| You can then tell the message types apart in JSON, because they will have different @context values | {  “@context”: “http://example.com/claim/”,  }  *vs*  {  “@context”: “http://example.com/resp/”,  } |
| And in RDF, the message blank nodes will have different types | \_:n0 a ClaimType .  *vs*  \_:n0 a RespType . |

**Self-describing data**

NIEM XML is about as self-describing as data can be. Every element and attribute has a URI that can be formed from the namespace declaration plus its local name. The URI identifies a documentation resource in XSD or CMF. That resource can’t always be retrived by HTTP GET, but the URI does give a good idea of where to start looking. Given the URI, a NIEM registry should be able to supply at least a POC for the message specification, if registered.

I want to retain this self-describing property in NIEM JSON and other serializations, by means of the following rules and conventions:

1. *Every message specification has a URI.* The URI of the namespace that declares most or all of the message properties is a good choice. In the insurance claim example, it’s http://example.org/claim/1.0/
2. *Every message class has a URI.* It’s the URI of the message property. In the insurance claim example above, it’s http://example.org/claim/1.0/#Claim
3. *Every message format has a URI.* If there is only one format for a particular class, then that format URI is the same as the class URI. When a class has more than one format, then the format URI is created by appending the uriSuffix property from the manifest to the class URI – except that canonical XML formats do not need or have a uriSuffix. In the three-format example above, the format URIs are:
   * Canonical XML: http://example.org/claim/1.0/#Claim
   * Simple XML: http://example.org/claim/1.0/#Claim/simple.xml
   * JSON: http://example.org/claim/1.0/#Claim/json
4. *Every message contains the URI of its message format*.
   * In canonical XML, this is the URI of the message element, constructed from the namespace declaration and local name:

<c:Claim

xmlns:c=”http://example.com/claim/1.0/”>

* + In simple XML, this is the URI of the default namespace:

<claim

xmlns=”http://example.com/claim/1.0/#Claim/simple.xml”>

* + In JSON, this is the value of the @context key:

{

“@context”: “http://example.com/claim/1.0/#Claim/json”,

...

}

**Tool support**

These are the operations I think we need. I think I know how to implement them.

* Transforming CMF to CMF
  + UNION: combine several model files into one model file  
    For example, to separate the model into *extension.cmf* and *niem-subset.cmf*
  + SELECT: extract compnents from a model file, based on a wantlist
  + CONSTRAIN: apply a model constraint file to a model file
* Transforming CMF to XSD
  + To a model representation in NIEM-conforming XSD
  + To a constraint schema document pile that validates a message format
  + To a single constraint schema document that validates a simple XML format,   
    plus transformation XSLT scripts to/from the canonical XML
* Transforming CMF to JSON
  + To a JSON Schema that validates a message format
  + To a JSON-LD context for a message format  
    (using either canonical or simple properties)