Proposal: Functional Node Definitions (Non-Graph Implementations)

This proposal extends the former proposal to support `functiona nodedefs` to include non-graph implementations.

Motivation

The main motivation is still the same to have a single Element contain bot the interface and implementation declaration of a node definition. This makes it easier to manage and distribute node definitions as atomic units.

Atomic Elements would also fit well with any definition templating used to handle polymorphic node definitions.

Generalization

Currently `NodeGraphs` are derived from `Implementation` Elements. The natural extension is to allow any Element which is either a `Implementation` or derived from it.

Non-nodegraph implementations generally have a `target` specifier to indicate that the implementation is specific to a target deployment or language.
While it is not required to have full encapsulation a way to include all "targets" implementations within a single `functiona nodedef` is desirable.

Implementation Variants

- `nodedef` : Both nodegraph and non-graph implementations can have a back-reference to their `nodedef` (1)
- `target` : Implementations are per-target specific.
- `file` : Implementations may reference external files (2)

```
<implementation name="IM_rotate2d_vector2_genglsl"
nodedef="ND_rotate2d_vector2" file="mx_rotate_vector2.glsl"
function="mx_rotate_vector2" target="genglsl" />
```

- `sourcecode` : Implementations may inline source code. Example:

```
<implementation name="IM_floor_float_genglsl"
nodedef="ND_floor_float" target="genglsl"
sourcecode="floor({{in}})" />
```

- if there is not source reference then these implementations fall into the category of have an externally defined implementation -- possible as
part of a code generator.

Notes:

- 1. This would no longer be required as the implementation is contained within the `nodedef`.
- 2. Handling external file references is not part of this proposal. There have been separate discussions if if and how this should be handled, including:
  - Encoding the file content into something that is storable as XML uft-8 text which would be basically inlining the file content.

Examples:

Below is an example of the definition for float `image` which has per target non-graph implementations (`ND_image_float`):

Current separate definition + implementation:

```
<nodedef name="ND_image_float" node="image" nodegroup="texture2d">
  <input name="file" type="filename" value="" uiname="Filename"
uniform="true">
  <input name="layer" type="string" value="" uiname="Layer"
uniform="true">
  <input name="default" type="float" value="0.0" uiname="Default Color">
  <input name="texcoord" type="vector2" defaultgeomprop="UV0"
uiname="Texture Coordinates">
  <input name="uaddressmode" type="string" value="periodic"
enum="constant,clamp,periodic,mirror" uiname="Address Mode U"
uniform="true">
  <input name="vaddressmode" type="string" value="periodic"
enum="constant,clamp,periodic,mirror" uiname="Address Mode V"
uniform="true">
  <input name="filtertype" type="string" value="linear"
enum="closest,linear,cubic" uiname="Filter Type" uniform="true">
  <input name="framerange" type="string" value="" uiname="Frame Range"
uniform="true">
  <input name="frameoffset" type="integer" value="0" uiname="Frame
Offset" uniform="true">
  <input name="frameendaction" type="string" value="constant"
enum="constant,clamp,periodic,mirror" uiname="Frame End Action"
uniform="true">
  <output name="out" type="float" default="0.0">
</nodedef>
```

```
<implementation name="IM_image_float_genglsl" nodedef="ND_image_float"
file="mx_image_float.glsl" function="mx_image_float" target="genglsl">
<input name="default" type="float" implname="default_value" />
</implementation>
```

Combined functional definition with the default standard library targets added:

```
<nodedef name="ND_image_float" node="image" nodegroup="texture2d">
  <input name="file" type="filename" value="" uiname="Filename"
uniform="true">
  <input name="layer" type="string" value="" uiname="Layer"
uniform="true">
  <input name="default" type="float" value="0.0" uiname="Default Color">
  <input name="texcoord" type="vector2" defaultgeomprop="UV0"
uiname="Texture Coordinates">
  <input name="uaddressmode" type="string" value="periodic"
enum="constant,clamp,periodic,mirror" uiname="Address Mode U"
uniform="true">
  <input name="vaddressmode" type="string" value="periodic"
enum="constant,clamp,periodic,mirror" uiname="Address Mode V"
uniform="true">
  <input name="filtertype" type="string" value="linear"
enum="closest,linear,cubic" uiname="Filter Type" uniform="true">
  <input name="framerange" type="string" value="" uiname="Frame Range"
uniform="true">
  <input name="frameoffset" type="integer" value="0" uiname="Frame
Offset" uniform="true">
  <input name="frameendaction" type="string" value="constant"
enum="constant,clamp,periodic,mirror" uiname="Frame End Action"
uniform="true">
  <output name="out" type="float" default="0.0">

  <implementation name="IM_image_float_genglsl"
file="mx_image_float.glsl" function="mx_image_float" target="genglsl">
    <input name="default" type="float" implname="default_value">
  </implementation>
  <implementation name="IM_image_float_genmdl"
sourcecode="materialx::stdlib_{{MDL_VERSION_SUFFIX}}::mx_image_float({{f
ile}}, {{layer}}, {{default}}, {{texcoord}}, {{uaddressmode}},
{{vaddressmode}}, {{filtertype}}, {{framerange}}, {{frameoffset}},
{{frameendaction}}, mxp_flip_v:{{flip_v}})" target="genmdl">
  </implementation>
    <input name="default" type="float" implname="default_value">
  <implementation name="IM_image_float_genmsl"
file="../genglsl/mx_image_float.glsl" function="mx_image_float"
target="genmsl">
    <input name="default" type="float" implname="default_value">
  </implementation>
  <implementation name="IM_image_float_genosl" file="mx_image_float.osl"
function="mx_image_float" target="genosl">
    <input name="default" type="float" implname="default_value">
  </implementation>
  <implementation name="IM_image_float_genslang"
file="../genglsl/mx_image_float.glsl" function="mx_image_float"
target="genslang">
    <input name="default" type="float" implname="default_value">
```

```
  </implementation>
</nodedef>
```

Second example for `ND_normal_vector3` with a mix of source code and external implementationss

```
<nodedef name="ND_position_vector3" node="position"
nodegroup="geometric">
  <input name="space" type="string" value="object"
enum="model,object,world" uniform="true">
  <output name="out" type="vector3" default="0.0, 0.0, 0.0">
  <implementation name="IM_position_vector3_genglsl" target="genglsl" />
  <implementation name="IM_position_vector3_genmdl"
sourcecode="materialx::stdlib_{{MDL_VERSION_SUFFIX}}::mx_position_vector
3(mxp_space:{{space}})" target="genmdl" />
  <implementation name="IM_position_vector3_genmsl" target="genmsl" />
  <implementation name="IM_position_vector3_genosl" target="genosl"
sourcecode="transform({{space}}, P)" />
  <implementation name="IM_position_vector3_genslang" target="genslang"
/>
</nodedef>
```

Mixed Graph and Non-Graph Implementations

It is possible to have both graph and non-graph implementations within a single functional nodedef. In this case all the implementations would
be added as children of the `nodedef`.

Precedence

- As this is an "additive" change existing functional `nodedef`s with separated implementations can still be used.
- Un-encapsulated implementations have higher precedence over encapsulated ones. (Same as with functional `nodedef`s with graph implementations).

Modifications Required

- Thd Document definition / implementation association cache would need to consider non-graph implementations. The final cached content would
  still look the same and all existing API logic and entry point interfaces would not change.
- Any utilities to create functional node definitions would need to be updated to handle non-graph implementations. As there are none currently these
  would be new interfaces.

Further Consolidation

It is possible to have further consolidation by allow for multiple targets to be specified within a single `implementation` Element. In the example
above the `genglsl`, `genmsl`, and `genslang` implementations could be combined, however if any of them implementations diverge in the future it would require explicitly splitting.

Appendix: Test Code

Below is sample logic on how to append to a node definition using any implementation variant. It was
used to create the previous examples.

```
def make_functional_definition(test_name, target=''):
    test_def = stdlib.getNodeDef(test_name)
    if test_def:
        impl = test_def.getImplementation(target)
        if impl:
            new_impl = None
            ngname = impl.getNodeDefString()
            qualname = impl.getQualifiedName(test_def.getName())
            if ngname == qualname:
                new_impl_name = impl.getName()
                new_impl =
test_def.addChildOfCategory(impl.getCategory(), new_impl_name)
                if not new_impl:
                    print("Failed to create new functional def child:",
new_impl_name)
                    return None
                else:
                    # Create new functional implementation
                    new_impl.copyContentFrom(impl)

new_impl.removeAttribute(mx.InterfaceElement.NODE_DEF_ATTRIBUTE)

                    # Clear out back-reference to nodedef on original
impl

impl.removeAttribute(mx.InterfaceElement.NODE_DEF_ATTRIBUTE)

    return test_def
```