Technical Report NEESgrid-2003-15

www.neesgrid.org

(Draft Whitepaper Version: 0.5

Last modified: August 5, 2003)

NEESML Reference User Guide

Joe Futrelle[1]

[1] National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign

Feedback on this document should be directed to futrelle@ncsa.uiuc.edu

Acknowledgment: This work was supported primarily by the George E. Brown, Jr. Network for Earthquake Engineering Simulation (NEES) Program of the National Science Foundation under Award Number CMS-0117853.

1 Summary

The NEESML Reference User Guide provides a comprehensive reference manual for the NEESML language, which is used to populate a NEESgrid metadata repository with objects and definitions of object types. The User Guide specifies the syntax for NEESML documents, describes the meaning of every NEESML construct, and explains how NEESML may be extended. A glossary of NEESML-specific terms is included.

2 Introduction

NEESML is a means for populating a NEESgrid metadata repository with objects and definitions of object types. It provides a syntax for defining object types and specifying objects, the values of their properties, and the relationships between objects. The NEESgrid metadata ingestion tool accepts NEESML files and uses them to populate a repository with objects and type definitions, communicating with the repository with the NEESgrid metadata service (NMDS) interface.

NEESML can be used to share type definitions and objects between repositories, and as an interface between other applications and repositories, because applications can either be written to read and write NEESML files, or can be augmented with translation utilities that translate the data formats they can read or write to and/or from NEESML. Finally, NEESML can be used to archive metadata to files.

2.1 About this document

This document is intended as a comprehensive reference manual for the NEESML language. It does not explain how to use NEESML to represent real-world objects, but rather completely specifies the syntax and meaning of every NEESML construct. Using this document, you will be able to write NEESML documents that conform to the proper syntax, and will be able to understand syntax errors and other problems with non-conforming NEESML documents or document fragments.

For a more general introduction to NEESML, please refer to (Futrelle, 2003).

Some NEESML-specific terminology is used in this document. The first time every such term is used, it is *italicized*. Definitions of all special terms are given in the glossary in section 8.

NEESML is a general-purpose metadata description language. Throughout this document, examples are used that do not directly pertain to earthquake engineering. This is done merely in the interest of simplicity. All the constructs used in the examples can be applied to metadata describing earthquake engineering experiments and simulations.

2.2 Some XML "gotchas"

NEESML documents are XML documents. There are some aspects of XML that constrain the syntax of NEESML documents in important ways:

- XML *element* and *attribute* names are case-sensitive. "`ID`" is not equivalent to "`id`", and "`MyTypeName`" is not equivalent to "`mytypename`".
- *Namespaces*, if they are used as relation ID's or the ID's of types for which objects are created, must be declared in an enclosing element.
- Some characters are not allowed in element names. It is common practice to only use alphabetic characters and dashes. Colons and other punctuation are not allowed.

For details on XML syntax, consult the XML specification (Bray, Paoli, Sperberg-McQueen & Maler, 2000) or a good XML reference or tutorial.

3 NEESML document syntax

A NEESML document consists of a top-level "`neesml`" element containing any number of *type* definitions and *object* specifications. The simplest NEESML document, which defines no types and contains no objects, is:

```
<neesml>
</neesml>
```

The examples in the following sections are assumed to be contained within the top-level "`neesml`" element of a NEESML document.

4 Defining types

Types are defined in NEESML using the `type` *directive*. Every type must have a unique *ID*, which is defined with the `id` directive. You may use any ID that is not already in use in the repository, except for the ID "`type`", because that is the name of the `type` directive.

A type may also have a title, defined with the `title` directive. The title need not be unique, and simply serves as a convenient and easily readable name for the type. For example:

```
<type id="lvdt" title="Linear variable displacement transducer"/>
```

Type definitions are essentially collections of *relation* definitions.

4.1 Defining relations

Relations are defined using sub-elements of the `type` directive. Each sub-element that defines a relation is named after the relation's ID. For instance, if you want to define a type with ID "`person`"with a relation with ID "`name`", use the following construct:

```
<type id="person">
  <name/>
</type>
```

Any number of relations may be defined for a type, but no more than one relation with a given ID may be defined. Like types, relations can have titles, specified with the `title` directive.

A relation definition can specify *relation constraints*, including *relation type* and *cardinality*.

4.1.1 Specifying relation type

Relation types constrain the kinds of values a relation may hold for each object of a given type. Relation types can be either *primitive* types or *reference* types. Primitive types are described in Table 1. Reference types constrain the types of objects a relation can contain a reference to.

Table 1: Primitive types in NEESML

Name	Description	Examples
`string`	Text	`"Hello, world."` `"BN# 493-2584x"`

`int`	Integer	`3`
		`-2`
		`2147483647`

`long`	Long integer. Can exceed the size of an integer.	`-5782347562427` `9223372036854775807`
`double`	Double precision floating point number.	`523425.4568574636` `-0.0000000435234`
`date`[1]	A moment in time, represented as a date and time stamp in UTC with 1ms resolution.	`2002-10-27 15:40:32.048` `1969-01-12 00:03:48.774`

4.1.1.1 Specifying primitive relation types

Primitive relation types are specified with the (relation) `type` directive. For instance, if you want to define a type with ID "`engine`" with a relation with ID "`numberOfCylinders`" of type integer, use the following construct:

```
<type id="engine">
  <numberOfCylinders type="int"/>
</type>
```

If no primitive type for a relation is defined, it is assumed to be of the default relation type, `string`.

4.1.1.2 Specifying reference relation types

Reference relation types are specified with the `allow` directive. The `allow` directive specifies one or more object types that are allowable as values of the relation. For instance, if you want to define a type with ID "`person`" with a relation with ID "`dwelling`" whose value may be a reference to either an object of type "`apartment`" or an object of type "`house`", use the following construct:

```
<type id="person">
  <dwelling>
    <allow type="apartment"/>
    <allow type="house"/>
  </dwelling>
</type>
```

NEESML provides a shorter form of the `allow` directive when only one reference type is allowed. For instance:

[1] NEESML does not currently support dates.

```
<type id="apartmentDweller">
  <dwelling allow="apartment"/>
</type>
```

4.1.2 Specifying relation cardinality

Relation cardinality constrains the number of values a relation may hold for each object of a given type. Relation cardinality is specified with the `min` and `max` directives. The `min` directive specifies the minimum number of values allowed; it may be either 0 or 1. The `max` directive specifies the maximum number of values allowed; it may be any number greater than or equal to the minimum number of values allowed, or `unbounded`, indicating that there is no limit on the number of values allowed.

For example, if you want to define a type with ID "`boltedConnection`" with a relation with ID "`bolts`" that allows one or more references to objects of type "`bolt`", use the following construct:

```
<type id="boltedConnection">
  <bolts allow="bolt" min="1" max="unbounded"/>
</type>
```

The default values for `min` and `max` are 0 and 1. This means zero or one values are allowed, meaning that the relation is an *optional* relation, i.e., it is not required to have a value for each object. To make a relation *required*, set `min` to 1:

```
<type id="person">
  <name min="1"/>
</type>
```

`min` and `max` are ***not*** used to set the minimum and maximum values allowed for a relation of a numeric type such as `int`, `long`, or `double`. NEESML does not currently provide this capability.

4.1.3 Nested type definitions

NEESML allows types to be defined inside relation definitions, as a convenience. For instance, suppose you want to define two types, one to represent a person and one to represent a person's phone number(s), and then define a relation between those two types. You can do that without nested type definitions, like this:

```
<type id="phoneNumber">
  <countryCode type="int"/>
  <areaCode type="int"/>
  <exchange type="int" min="1"/>
  <number type="int" min="1"/>
</type>

<type id="person">
  <phoneNumber allow="phoneNumber" max="unbounded"/>
</type>
```

With nested type definitions, you can define the same set of types and the relation like this:

```
<type id="person">
  <phoneNumber>
    <type id="phoneNumber" max="unbounded">
      <countryCode type="int"/>
      <areaCode type="int"/>
      <exchange type="int" min="1"/>
      <number type="int" min="1"/>
    </type>
  </phoneNumber>
</type>
```

Relations with nested type definitions cannot include the `allow` directive, because the presence of a nested type definition implies that only objects of the type defined in the nested type definition are allowed as values of the relation whose definition contains the nested type definition.

4.2 Inheritance

A type can *inherit* relation definitions from another type by *extending* another type, which is then called its *parent* type. The effect of extending a type is that the parent type's relation definitions are implicitly duplicated in ("inherited by") the *child* type, which can then add other relation definitions to the set of inherited relation definitions. This can be used to specialize types. For instance, if you define a type representing sensors:

```
<type id="sensor">
  <manufacturer/>
  <model/>
  <serialNumber/>
</type>
```

You can specialize it by extending it to represent particular kinds of sensors:

```
<type id="lvdt" extends="sensor">
  <range type="double"/>
</type>
```

In this case, the "`lvdt`" type has four relations defined for it: the three defined by "`sensor`" and the additional "`range`" relation it defines.

A type can extend only one other type. A parent type ***must be defined before every type that extends it***. A type may extend any type that exists in the repository, not just the ones defined in the same NEESML file that it is defined in. A type may be extended by any number of other types. Types that extend other types may themselves be extended. The effect is to create a hierarchy of types.

If a type (call it "A") extends a type ("B"), all objects of type A are also considered to be of type B. If an `allow` directive allows objects of type B, it also implicitly allows objects of type A, and any descendants of A in the type hierarchy.

5 Creating objects

To create an *object* of a given type, use an element with the same name as the ID of the type. To set relation values for an object, use sub-elements with the same name as relations defined on that type. For instance, to create an object of type "`lvdt`", defined in the example in section 4.2, use a construct like this:

```
<lvdt>
  <manufacturer string="Ltd. Mfg. Co"/>
  <serialNumber string="X49-AAB-083"/>
  <range double="34.2"/>
</lvdt>
```

Sub-elements representing relations specify values using directives named after primitive types (`string`, `int`, `long`, `double`, and `date`), or using reference value constructs described later in this section.

Like types and relations, objects can have titles, specified with the `title` directive.

5.1 Setting values for primitive relation types

Use sub-elements to set values for relations with primitive types, as in the example above. To set more than one value for a relation, use sub-elements named after primitive types, like this:

```
<person>
  <name>
    <string>Mark Twain</string>
    <string>Samuel Clemens</string>
  </name>
</person>
```

The values of a relation with multiple values are unordered; it does not matter what order you specify them in.

5.2 Setting values for reference relation types

Setting values for reference relation types requires 1) identifying objects, and 2) referring to them.

5.2.1 Identifying objects

Objects are identified in two very different ways, by *ID* and by *alias*. ID's and aliases are unique strings. ID's must be unique in the repository; aliases need only be unique within a single NEESML document. Object ID's are specified using the `id` directive:

```
<lvdt id="lvdt1">
  <serialNumber string="42-ABX"/>
</lvdt>
```

Aliases are specified using the alias directive:

```
<lvdt alias="lvdt1">
  <serialNumber string="42-ABX"/>
</lvdt>
```

The purpose of ID's and aliases is to allow objects to refer to each other. If an object is created with no ID or alias, then no other objects can refer to it in their relation values. There are several situations where this may be OK, including nested objects and objects added to a container, both of which are described in later sections.

5.2.2 Referring to objects

For an object to refer to another object, it must have defined a relation that is of a reference type. For instance you can define two types to represent cameras and lenses:

```
<type id="camera">
  <lens>
    <type id="lens">
      <fStop type="double"/>
      <focalLength type="int"/>
    </type>
  </lens>
</type>
```

To define a lens, and link it to a camera, you can give the lens an object ID and use the `id` directive on the camera's "`lens`" relation to make the link:

```
<lens id="myWideAngle">
  <fStop double="1.4"/>
  <focalLength int="24"/>
</lens>

<camera>
  <lens id="myWideAngle"/>
</camera>
```

You can also use an alias:

```
<lens alias="myWideAngle">
  <fStop double="1.4"/>
  <focalLength int="24"/>
</lens>

<camera>
  <lens alias="myWideAngle"/>
</camera>
```

The difference is that if you use an alias, the two objects must be contained in the same NEESML file. If you use an ID, the referring object can be in any file, provided that the referred-to object exists in the repository before it is referred to. In this way, ID's can be used to build libraries of objects that can be reused in many NEESML files.

NEESML provides a shorthand construct for inter-object reference. Instead of creating objects separately, an object can be created inside another object's specification, like this:

```
<camera>
  <lens>
    <lens>
      <fStop double="1.4"/>
      <focalLength int="24"/>
    </lens>
  </lens>
</camera>[2]
```

This is accomplished by creating an implicit alias for the inner object. If you want to use the enclosed object elsewhere, you can give it an explicit alias:

```
<camera>
  <lens>
    <lens alias="myWideAngle">
      <fStop double="1.4"/>
      <focalLength int="24"/>
    </lens>
  </lens>
</camera>

<type id="lensCollection">
  <lenses allow="lens" max="unbounded"/>
</type>

<lensCollection>
  <lenses>
    <lens alias="myWideAngle"/>
    <lens alias="myZoom"/>
  </lenses>
</lensCollection>
```

You can also use an ID instead of an alias in this kind of construct.

This construct can be applied for relations with any number of values:

```
<lensCollection>
  <lenses>
    <lens alias="myWideAngle">
      <fStop double="1.4"/>
      <focalLength int="24"/>
    </lens>
    <lens alias="mySlowTele">
      <fStop double="4.1"/>
      <focalLength double="400"/>
    </lens>
  </lenses>
</lensCollection>
```

---

[2] The reason "`lens`" is repeated at two levels in this example is because camera defines a relation with ID "`lens`", whose value must be of the type whose ID is also "`lens`". The outer "`lens`" element identifies the relation, and the inner "`lens`" element creates an object of type "`lens`".

It can also be applied recursively:

```
<type id="person">
  <name/>
  <siblings allow="person" max="unbounded"/>
  <pet allow="animal"/>
</type>

<type id="dog" extends="animal">
  <answersTo max="unbounded"/>
</type>

<person>
  <name string="Joe"/>
  <siblings>
    <person>
      <name string="Veeves"/>
      <pet>
        <dog>
          <answersTo string="Samson"/>
        </dog>
      </pet>
    </person>
  </siblings>
</person>
```

5.3 Updating objects

If an object with the same ID as an object specified in a NEESML file already exists in the repository, the NEESML ingestion tool will create a new version of the existing object that matches the object specification in the NEESML file. This can be useful when you want to modify a set of objects defined in a NEESML file and re-ingest the file.

To make this possible, the objects you want to update must have an ID defined with the `id` directive. If instead they have no id, or have an alias but no ID, then a new object is created and the original object is not updated.

If existing objects had reference relations that linked to the original version of the object, you must also update those objects so that they link to the new version; otherwise, they will continue to link to the old version.

5.4 Adding objects to containers

Containers are special metadata objects that can contain other objects. They can be used to organize objects into groups. Containers can contain other containers, forming a hierarchy. An object can be in any number of containers. The container type is a special, pre-defined type in the repository, with the ID "`md:container`". If you want to define you own types of containers, you can extend this type.

An object can be added to a container with the "`container`" directive. The directive specifies the ID of the container to which you want to add the object. The NEESML ingestion tool will add the object to the latest version of the container. If an object with the same ID as the object being added is already in the container, the object in the container will be updated rather than re-added.

For instance, if there was a container with the ID "`uiuc:mostData`", you could add an object to it like this:

```
<io:file title="MOST data file" container="uiuc:mostData">
  <io:pathname string="/usr/local/data/mostData.txt"/>
</io:file>
```

6 Namespaces

Any ID (for a type, relation, or object) can consist of a *namespace part* followed by a *local part*, separated by a colon. The namespace part serves to distinguish ID's with identical local parts from one another. For instance, the ID's "`uiuc:actuator`" and "`uc:actuator`" have the same local part, but different namespace parts.

Namespaces that are used in type and relation ID's must be declared in an enclosing XML element. It is probably simplest to define them all on the top-level "`neesml`" element. NEESML ignores the URL's used to identify NEESML namespaces, so it does not matter what they are. A proposed convention is to use a URL consisting of `http://www.nees.org/md/ns/` followed by the namespace part. For example:

```
<neesml xmlns:io="http://www.nees.org/md/ns/io">

  <type id="io:file">
    <io:pathname/>
  </type>

  <io:file title="My data file">
    <io:pathname string="/tmp/aFile.txt"/>
  </io:file>

</neesml>
```

7 NEESML Extensions

NEESML can be extended by developing new constructs along with rules for transforming them into standard NEESML. The most natural technology for implementing such rules is XSLT (Clark, 2000). Several example extensions are provided with the ingestion tool, and are supported by the CHEF-based NEESgrid repository browser.

7.1 Unit extensions

In scientific domains, numerical values are often associated with units of measure. It is convenient to be able to convert values between different units of measure. To do this, it is necessary to represent the units of measure in terms of base units. Because this would be inconvenient to do for every value, it is convenient to maintain a library of commonly used units that describes them in terms of algebraic combinations of base units.

The NEESML unit extensions provide a set of types representing units of measure and their components, as well as a simplified syntax for defining new units of measure and associating a value with a unit.

7.1.1 Defining units

Each unit definition consists of a set of *unit terms*. Each unit term relates a base unit with a factor, exponent, and offset. Units are defined with the `defUnit` directive, and terms are defined with the `term` directive. Allowable base units are `m` (meter), `s` (second), `kg` (kilogram), `A` (ampere), `K` (Kelvin), `mol` (mole), and `cd` (candela). For definitions of these base units see (Taylor, 2001). Base units, factors, exponents, and offsets are specified with `term`'s `baseUnit`, `factor`, `exponent`, and `offset` directives. For example, meters per second can be defined as follows:

```
<defUnit id="mps" title="meters per second">
  <term baseUnit="m"/>
  <term baseUnit="s" exponent="-1"/>
</defUnit>
```

And inches can be defined like this:

```
<defUnit id="ft" title="feet">
  <term baseUnit="m" factor="0.0254"/>
</defUnit>
```

The `id` directive specifies the local part of the ID of the object that is created to represent the unit. All units are defined in the `unit` namespace; you cannot include a namespace part in `term`'s `id` directive. You also cannot use slashes or carets (^) in the ID. The title directive can be used to give a descriptive title.

Offsets can be used for units that are offset by a constant from the origin of their base unit. For instance Fahrenheit can be defined like this:

```
<defUnit id="degF" title="degrees Fahrenheit">
  <term baseUnit="K" factor="1.8" offset="-459.67"/>
</defUnit>
```

7.1.2 Combining units

It's convenient to define units in terms of other derived units, rather than base units. This can be done with the `unit` directive of `term`. For instance, suppose we define G (gravitational acceleration) like this:

```
<defUnit id="G">
  <term baseUnit="m" factor="9.80665"/>
  <term baseUnit="s" exponent="-2"/>
</defUnit>
```

We can now define pounds (of force) like this:

```
<defUnit id="lb">
  <term baseUnit="kg" factor="0.43559237"/>
  <term unit="G"/>
</defUnit>
```

And kips like this:

```
<defUnit id="kips">
  <term unit="lb" factor="1000"/>
</defUnit>
```

7.1.3 Using units

To enable units to be used, NEESML unit extensions define a special relation type called "`quant`". For example, suppose you want to represent the maximum amount of force that can be applied by an actuator:

```
<type id="actuator">
  <maximumForce type="quant"/>
</type>
```

In object specifications, NEESML unit extensions provide `value` and `unit` directives for relation values. For example:

```
<actuator title="My actuator">
  <maximumForce value="324.1" unit="kips"/>
</actuator>
```

In order to use a unit in an object specification, a definition for it must be present in the repository. The NEESML ingestion tool provides an example "unit dictionary" file containing a set of unit definitions.

7.2 Geometric extensions

The representation of geometry is a very complex problem. NEESML geometry extensions in their current form provide just a tiny example of how NEESML could begin to be used to address this problem. They are by no means a comprehensive treatment of the issue.

NEESML geometry extensions provide two types, `geom:point` and `geom:orientation`, along with simplified syntax to create these types.

`geom:point` represents a point in 3d Cartesian space in which the components are related to a unit of measure. NEESML geometry extensions allow points to be represented using the `point` directive.

For instance suppose you want to represent the location of a sensor. You can define the sensor type like this:

```
<type id="sensor">
  <manufacturer/>
  <serialNumber/>
  <location allow="geom:point"/>
</type>
```

And you can specify the location of a particular sensor like this:

```
<sensor title="My sensor #24">
  <manufacturer string="ltd. mfg. co."/>
  <serialNumber string="43-XJDH-HDB"/>
  <location>
    <point x="43.2" y="12.3" z="0" unit="in"/>
  </location>
</sensor>
```

Again, the unit you use must have been previously defined. No attempt is made to describe coordinate systems or transformations. This will obviously be necessary for any comprehensive treatment of geometry.

Orientation is represented very similarly, except that instead of *x*, *y*, and *z* components, orientation consists of *rx*, *ry*, and *rz* components, measured in radians. Because the components are measured in radians, there is no unit of measure associated with them. For example:

```
<type id="beam">
  <material/>
  <beamOrientation allow="geom:orientation"/>
</type>

<beam title="supporting beam">
  <material string="grade 50 steel"/>
  <beamOrientation>
    <orientation rx="0" ry="0" rz="3.1415"/>
  </beamOrientation>
</beam>
```

8 Glossary

Alias – a unique string identifying an *object* within a NEESML document.

Attribute – an XML attribute (see XML specification)

Cardinality (relation) – the range of number of values allowed for a *relation*, specified using the *min* and *max* directives.

Constraints (relation) – limits on the *type* and *cardinality* of a *relation*, specified in an *object type* definition.

Construct – a specific combination of syntactic elements. NEESML allows certain constructs and disallows others.

Container – a special kind of metadata object that can contain other objects. All containers are of type "`md:container`".

Directive – an *element* (e.g., `type`, `allow`) or *attribute* with special meaning to NEESML.

Element – an XML element (see XML specification)

ID – a unique string identifying an *object*, *type*, or *relation*.

Inheritance – the mechanism by which a *type* can extend another *type*, adding additional *relations* or modifying *relation constraints*.

Namespace – a prefix on an *ID*, followed by a colon. The prefix is called the namespace part and the rest of the *ID* is called the local part. Namespaces allow *ID*'s with the same local part to be distinguished from one another.

Optional relation – a *relation* with a minimum *cardinality* of 0.

Primitive relation type – one of a small set of special *relation types* used to represent attributes or properties of *objects*. They are `string` (text), `int` (integer), `long` (large integer), `double` (double precision floating point number), `date` (moment in time), and `reference` (link to another *object* – see *reference relation type*).

Reference relation type – a *relation type* representing a link from an *object* to another *object*. The *types* of other *objects* that may be linked to can be constrained using the `allow` *directive* in the *type* definition.

Relation – an attribute or property of an *object*, identified by a *relation ID* and having a value. The relation's *type* and *cardinality* is constrained by *constraints* specified in the *object*'s *type* definition.

Required relation – a *relation* with a minimum *cardinality* of 1.

Type (object) – a kind of metadata *object*, defined using the `type` *directive*, which specifies the *type*'s allowable *relations* and their *constraints*

Type (relation) – the kind of value allowed for a relation, which is either a *primitive relation type* or a set of *object types*, defined by the `allow` *directive*.

Value (relation) – the value, either of a *primitive* or *reference* type, of a *relation* for a given *object*.

9 References

Bray, T., Paoli, J., Sperberg-McQueen, C. M., & Maler, E. (Ed.). (2000). Extensible markup language (xml) 1.0 (second edition) [On-line]. Available at: http://www.w3.org/TR/REC-xml.

Clark, J. (Ed.). (2000). Xsl transformations (xslt) version 1.0 [On-line]. Available at: http://www.w3.org/TR/xslt.

Futrelle, J. (Ed.). (2003). Overview of neesml whitepaper v1.0 [On-line]. Available at: http://www.neesgrid.org/documents/NEESgrid_TR_2003-12.pdf.

Taylor, B. N. (Ed.). (2001). Guide for the use of the international system of units (si) [On-line]. Available at: http://physics.nist.gov/Pubs/SP811/contents.html.