Building the National Virtual Collaboratory for Earthquake Engineering Research

NEESgrid

Technical Report NEESgrid-2003-12

www.neesgrid.org

Draft Whitepaper Version: 1.0

Last modified: June 17, 2003

Overview of NEESML

Joe Futrelle[1]

[1] National Center for Supercomputing Applications, Urbana-Champaign, IL 61820

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.

Summary

NEESML is an XML format for defining NEES metadata schemas and uploading metadata objects into the NEES metadata repository. NEESML provides a syntax for defining object types, object attributes, and relationships between object types, including inheritance. It also provides a means for creating instances of object types, including the ability to link objects together in a variety of ways. The purpose of the type definition and instantiation mechanisms is to allow users to describe real-world objects of interest to them in order to enable browsing and retrieval of data and information within and across sites. This paper describes NEESML and gives examples of its use, but does not include a comprehensive reference for the format.

1 Overview

NEESML is an XML format for defining NEES metadata schemas and uploading metadata objects into the NEES metadata repository. NEESML provides a syntax for defining object types, object attributes, and relationships between object types, including inheritance. It also provides a means for creating instances of object types, including the ability to link objects together in a variety of ways. The purpose of the type definition and instantiation mechanisms is to allow users to describe real-world objects of interest to them and link them together in a meaningful structure. These representations can then be uploaded into a repository where they can be linked to other objects, browsed, and used for a variety of applications. The metadata stored in the repository can thus be used not only to describe scientific work for posterity, but also can serve as "glue" to link scientific datasets with each other and with detailed descriptions, to enable browsing and retrieval of data and information within and across sites.

This paper describes NEESML and gives examples of its use, but does not include a comprehensive reference for the format.

2 Object model

The NEES metadata repository uses an object-oriented metadata model. Objects are defined as instances of object types, and relations are used to link objects to values or to other objects. These concepts are elaborated further in the sections below. The object model was chosen as a simple, generic model that can accommodate both relational and hierarchical modes of description. A later section in the document gives examples of how to migrate relational and XML schemas into NEESML.

Each NEESML feature is illustrated using examples. The examples serve only to explain the syntax and use of NEESML, not to prescribe which metadata types to use for NEES applications.

There is no correct or incorrect way to represent information with metadata. It is up to communities of use to develop metadata structures that enable the applications and services that are most useful for them. These can range from generalized services such as information retrieval to more specialized applications such as analysis—in short, any task that requires structured, descriptive information.

3 Defining types

A fundamental aspect of designing metadata for the NEES repository is defining *types*. Types can be used to represent kinds of real-world objects (e.g., sensors, specimens, materials), as well as abstract entities (e.g., material properties, geometry, policies). Which kinds of objects are represented is up to users; the repository provides a generalized representational mechanism.

3.1 Types and relations

Each object type is defined as a set of *relations*. A relation is a way of associating an object instance with a value or with another object. There are several types of values:

strings, integers, long integers, double precision floating-point numbers, dates[1], and references. We can call these types "primitive types" to distinguish them from user-defined types. The following table describes each primitive type:

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`	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`

`reference`	A reference to another instance.	*[described later in this document]*

By defining sets of relations of these types, we can represent a variety of kinds of information. For example, suppose we want to represent units of measure. We can represent a unit of measure as a type with two relations: the base unit and a factor. Here's how that would look in NEESML:

```
<type id="unitOfMeasure">
 <baseUnit type="string"/>
 <factor type="double"/>
</type>
```

Each type has an ID, in this case "`unitOfMeasure`". Each relation for each type has an ID and a primitive type. In this case there are two relations, one with an ID of "`baseUnit`" and a primitive type of "`string`", and one with an ID of "`factor`" and a primtive type of "`double`".

To represent the unit of measure "foot", we could create an *instance* of the "`unitOfMeasure`" object type, like this:

```
<unitOfMeasure id="ft">
 <baseUnit string="m"/>
 <factor double="0.3048"/>
</unitOfMeasure>
```

Note that there is nothing in our type definition that specifies what values are allowed for each of these relations[2]. To use this object[3], I need to know what is meant when a base

[1] NEESML does not currently support dates.

unit of “m” is used, and what it means to associate a base unit with a factor. Let us assume for the purpose of this example that we may only use standard international units as base units: m, s, kg, K, A, mol, and cd.

3.2 Complex types

Now, suppose we want to represent a unit of measure such as miles per hour. Our current definition for `unitOfMeasure` is insufficient for this purpose because it only allows us to describe a unit in terms of a single base unit. So we can use it to represent miles:

```
<unitOfMeasure id=”mile”>
 <baseUnit string=”m”/>
 <factor double=”1609.344”/>
</unitOfMeasure>
```

and hours:

```
<unitOfMeasure id=”hr”>
 <baseUnit string=”s”/>
 <factor double=”3600”/>
</unitOfMeasure>
```

but we cannot use it to describe the relationship between them. In order to accomplish this, we can conceptualize units of measure as consisting of one or more unit terms, each of which specifies a base unit, a scaling factor, and an exponent. We can define unit terms like this:

```
<type id=”unitTerm”>
 <baseUnit type=”string”/>
 <factor type=”double”/>
 <exponent type=”int[4]”/>
</type>
```

Now, we can define a unit of measure as a collection of one or more unit terms:

```
<type id=”unitOfMeasure”>
 <terms type=”reference” min=”1” max=”unbounded”>
  <allow type=”unitTerm”/>
 </terms>
</type>
```

[2] A future document will describe how to use XML Schema to restrict values to ranges or sets of legal values.

[3] The terms “object” and “instance” can be used interchangeably.

[4] “`int`” is short for integer.

The “`min`” and “`max`” directives restrict the number of allowable values for each relation. In this case, we are specifying that for every instance of `unitOfMeasure`, the relation “`terms`” must have one or more values. The “`allow`” directive specifies that the values must be references to objects of type `unitTerm`. For convenience, NEESML provides a shorthand for defining types that are used as the values of the relations of other types. In that shorthand, we can define both types like this:

```
<type id=”unitOfMeasure”>
 <terms min=”1” max=”unbounded”>
  <type id=”unitTerm”>
   <baseUnit type=”string”/>
   <factor type=”double”/>
   <exponent type=”int”/>
  </type>
 </terms>
</type>
```

Here’s how we can use this schema to represent miles per hour:

```
<unitOfMeasure id=”mph”>
 <terms>
  <unitTerm>
   <baseUnit string=”m”>
   <factor double=”1609.344”/>
   <exponent int=”1”/>
  </unitTerm>
  <unitTerm>
   <baseUnit string=”s”/>
   <factor double=”3600”/>
   <exponent int=”-1”/>
  </unitTerm>
 </terms>
</unitOfMeasure>
```

This NEESML code creates three objects: 1) a `unitOfMeasure` representing miles per hour, 2) a `unitTerm` representing miles, and 3) a `unitTerm` representing hours. It also creates the appropriate links between those objects, so that an application can discover the `unitTerms` representing miles and hours given the `unitOfMeasure` representing miles per hour. The code above is a shorthand alternative for NEESML’s more general non-hierarchical syntax for linking objects together, in which each object is separately defined. In that syntax, the same set of objects and relationships can be represented like this:

```
<unitTerm alias=”miles”>
 <baseUnit string=”m”/>
 <factor double=”1609.344”/>
 <exponent int=”1”/>
</unitTerm>

<unitTerm alias=”per hour”>
 <baseUnit string=”s”/>
 <factor double=”3600”/>
 <exponent int=”-1”/>
</unitTerm>

<unitOfMeasure id=”mph”>
 <terms>
  <unitTerm alias=”miles”/>
  <unitTerm alias=”per hour”/>
 </terms>
</unitOfMeasure>
```

The “`alias`” directive provides a way to link objects together. Another similar directive called “`id`” provides a simple way for an object in one NEESML file to be reused in other NEESML files. Aliases and ID’s are described further later in this paper.

3.3 Inheritance

Types can be defined as extensions of more general types using *inheritance*. When an extended type is defined, it inherits all the relations from the type it extends, which is also referred to as its *parent* type. For instance, suppose we define several types to represent beams and columns:

```
<type id=”beam”>
 <length type=”double”/>
 <material allow=”material”/>
</type>

<type id=”column”>
 <length type=”double”/>
 <material allow=”material”/>
 <baseFixation allow=”boundaryCondition”/>
</type>
```

(For this example, let’s assume that we’ve already created types called “`material`” and “`boundaryCondition`” elsewhere.) It is clear that `beam` and `column` have several relations in common: `length` and `material`. Instead of defining each one of them twice, we can define a parent type with those relations and then extend it, like this:

```
<type id=”structuralElement”>
 <length type=”double”/>
 <material allow=”material”/>
</type>

<type id=”beam” extends=”structuralElement”/>
```

```
<type id=”column” extends=”structuralElement”>
 <baseFixation allow=”boundaryCondition”/>
</type>
```

This is not only efficient, but it also allows us to define other extensions of `structuralElement`, should we require them. Extended types may themselves be extended (i.e., we could extend `beam` and `column`), and there is no limit to the depth of the resulting type hierarchy. Every instance of an extended type is also implicitly defined as an instance of the extended type’s parent type (i.e., all instances of `column` are also considered instances of `structuralElement`), and also its parent type if any, and so on. There is a single, “root” type, of which all other types are descendants. If you do not include an “`extends`” directive in a type definition, you are implicitly extending the root type.

4 ID’s, namespaces, and aliases

4.1 ID’s

Each type, instance, and relation is uniquely identified with an *identifier*, or ID. The ID is used by the repository system to locate types, objects, and relations. NEESML makes it possible to control the ID used for each type or instance. For types, the ID must be specified. This is done with the “`id`” directive, like this:

```
<type id=”lens”>
 <length type=”int”/>
 <fStop type=”double”/>
</type>
```

For instances, the `id` directive is optional. If an ID is not specified, the repository will assign a unique ID automatically. Relation ID’s need only be unique for the type for which they are defined.

In NEESML, ID’s are required to be valid XML names, i.e., strings that can be used as the names of XML elements.

ID’s can be used as a means by which objects can refer to each other. For instance, suppose two houses are on the same street:

```
<street id=”southRace”>
 <direction string=”n/s”/>
 <surface string=”brick”/>
</street>

<house id=”myHouse”>
 <number int=”1205”/>
 <street id=”southRace”/>
 <color string=”green”/>
</house>
```

```
<house id=”yourHouse”>
 <number int=”1207”/>
 <street id=”southRace”/>
 <color string=”red”/>
</house>
```

In this example, the value of the “`street`” relation for each house is the instance with the ID “`southRace`”.

4.2 Namespaces

Maintaining the uniqueness of ID’s across several different NEESML files developed independently requires an impractical amount of coordination. For this reason, NEESML and the repository support *namespaces*. Namespaces are ID prefixes. An ID consists of a *namspace part* (the prefix), a separator (a colon), and a *local part*. Namespaces can be conceptualized as spaces that contain ID’s, which are only required to be unique within each namespace. Here is an example name:

```
structures:column
```

In this case, “`structures`” is the namespace part and “`column`” is the local part. This allows us to distinguish the ID from, for instance:

```
newspaper:column
```

which has an identical local part but a different namespace part.

In NEESML, any namespaces used must be declared, as in all XML documents. XML namespace declarations are described in the XML specification and in any comprehensive documentation on XML.

4.3 Aliases

When an object is created, it can be given an *alias*. Like ID’s, aliases can be used to link objects together. Unlike ID’s, aliases are only valid within a single NEESML document. Also unlike ID’s, aliases can contain spaces and other characters. Here’s an earlier example rewritten to use aliases:

```
<street alias=”S. Race St.”>
 <direction string=”n/s”/>
 <surface string=”brick”/>
</street>

<house id=”myHouse”>
 <number int=”1205”/>
 <street alias=”S. Race St.”/>
 <color string=”green”/>
</house>
```

```
<house id=”yourHouse”>
 <number int=”1207”/>
 <street alias=”S. Race St.”/>
 <color string=”red”/>
</house>
```

Used properly, aliases can make a NEESML file easier to read by giving descriptive names to objects that are referred to several places in a file.

5 Other features

NEESML supports several other features of the repository. One is that every object can have a title. Unlike an ID, a title need not be unique, nor need it be a legal XML name. Therefore it can include spaces and other characters. For instance:

```
<facility title=”Wave Tank Lab #4”>
 … etc …
</facility>
```

Another is that a relation can have more than one value. We saw this in the `unitOfMeasure` example above. This can be done when the values are references to other objects:

```
<bridge>
 <columns>
  <column alias=”east column”/>
  <column alias=”central column”/>
  <column alias=”west column”/>
 </column>
</bridge>
```

or simple values:

```
<racer>
 <times>
  <double>133.4</double>
  <double>132.9</double>
  <double>129.8</double>
 </times>
</racer>
```

In this format, the name of the primitive type (i.e., `string`, `int`, `long`, `double`, and `date`) must be used as the name of an enclosing element for each value.

6 Shortcuts

NEESML provides a number of shortcuts to make the syntax simpler and more compact. This section describes several of the shortcuts. One shortcut is that in instance descriptions, for relations of type `string`, the value can be given in an element body rather than in a “`string`” directive. For instance, you can use this form:

```
<person>
 <name>Joe Futrelle</name>
</person>
```

instead of this form:

```
<person>
 <name string=”Joe Futrelle”/>
</person>
```

This only applies to relations of type `string`. For relations of other types, like `int` or `double`, NEESML requires the latter format. There is also a shortcut for defining relations of type `string`. Instead of this syntax:

```
<type id=”person”>
 <name type=”string”/>
</type>
```

a shorter form without the “type” directive can be used instead:

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

The longer syntax is still required for relations of type `int`, `long`, `double`, and `date`.

Another set of shortcuts concerns the “`allow`” directive. Normally it is included as a subelement of a relation definition:

```
<type id=”car”>
 <engine type=”reference”>
  <allow type=”engine”/>
 </engine>
</type>
```

There are two shortcuts that can be used here. First, since all relations for which `allow` applies are of type `reference`, the relation type can be left off, like this:

```
<type id=”car”>
 <engine>
  <allow type=”engine”/>
 </engine>
</type>
```

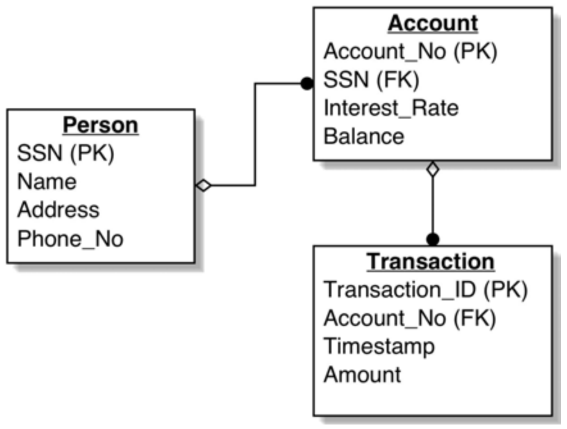
Second, if only one type is allowed for a relation, the allow directive can be rewritten more simply, like this:

```
<type id=”car”>
 <engine allow=”engine”/>
</type>
```

7 Migrating from other models

7.1 Relational models

The relational model is a familiar and mature model for designing data schemas. NEESML is similar enough to a relational model that many aspects of relational models translate fairly easily into NEESML. For instance, suppose we have the following relational model:



In this model, there are three tables allowing us to represent information about people, bank accounts, and transactions. Foreign key relationships enable us to link transaction information to account information and account information to personal information.

In NEESML, we can accomplish the same thing by defining three object types, each of which has a set of relations. We need not use keys, since all objects have unique ID's as an inherent feature of the NEES repository. Instead, we can link objects together using relations of type `reference`. Here is a possible way to represent the relational schema shown above in NEESML:

```
<type id=”person”>
 <ssn type=”int”/>
 <name type=”string”/>
 <address type=”string”/>
 <phoneNo type=”string”/>
</type>

<type id=”account”>
 <accountNo type=”int”/>
 <owner allow=”person”/>
 <interestRate type=”double”/>
 <balance type=”double”/>
</type>
```

```
<type id=”transaction”>
 <account allow=”account”/>
 <timestamp type=”date”/>
 <amount type=”double”/>
</type>
```

In this example, the relation “`owner`” in the “`account`” type takes the place of the `SSN` foreign key in the `Account` table, and the relation “`account`” in the “`transaction`” type takes the place of the `Account_No` foreign key in the `transaction` table. Here are some example NEESML objects using these types:

```
<person alias=”John”>
 <ssn int=”342514723”/>
 <name>John Q. Public</name>
 <address>77 Sunset Strip</address>
 <phoneNo>867-5309</phoneNo>
</person>

<account alias=”John’s savings”>
 <accountNo int=”239472”/>
 <owner alias=”John”/>
 <interestRate double=”0.02”/>
 <balance double=”3241.32”/>
</account>

<transaction>
 <account alias=”John’s savings”/>
 <timestamp date=”2003-05-12 14:53:02”/>
 <amount double=”-45.20”/>
</transaction>
```

7.2 XML models

XML formats are a convenient way of exchanging information. Although advanced features for type definition and non-hierarchical organization are available in XML technologies such as XML schema, typical XML models are strictly hierarchical and have relatively few constraints on the format of non-markup data. These two characteristics map into NEESML somewhat differently. The rigidly hierarchical nature of most XML formats is relatively easy to represent in NEESML. The lack of type constraints requires more attention, since the values of NEESML relations must conform to the set of primitive types it allows. There is always a simple solution, however, which is to declare relations as type `string`.

Suppose we have an XML format for which this is an example document:

```
<server>
 <processors number=”4”>
  <manufacturer>AMD</manufacturer>
  <model>Athlon</model>
  <clockSpeed>1.4 Ghz</clockSpeed>
 </processors>
 <memory totalSize=”1GB”/>
 <disks>
  <disk interface=”SCSI”>
   <manufacturer>IBM</manufacturer>
   <rpm>7200</rpm>
   <capacity>256GB</capacity>
  </disk>
 </disks>
</server>
```

There are many ways we could represent this information in NEESML. Here is an example type definition:

```
<type id=”server”>
 <processors>
  <type id=”processors”>
   <number type=”int”/>
   <manufacturer type=”string”/>
   <model type=”string”/>
   <clockSpeedGhz type=”double”/>
  </type>
 </processors>
 <memoryTotalSizeGB type=”double”/>
 <disks min=”1” max=”unbounded”>
  <type id=”disk”>
   <interface type=”string”/>
   <manufacturer type=”string”/>
   <rpm type=”int”/>
   <capacityGB type=”int”/>
  </type>
 </disk>
</type>
```

And here is how we could represent the first example in NEESML, given the type definition above:

```
<server>
 <processors>
  <processors>
   <number int=”4”/>
   <manufacturer>AMD</manufacturer>
   <model>Athlon</model>
   <clockSpeedGhz double=”1.4”/>
  </processors>
 </processors>
 <memoryTotalSizeGB int=”1”/>
 <disks>
  <disk>
   <interface>SCSI</interface>
   <manufacturer>IBM</manufacturer>
   <rpm int=”7200”/>
   <capacityGB int=”256”/>
  </disk>
 </disks>
</server>
```

Although the representation is slightly more complex than the original XML format (for instance, multiple tags called “`processors`” are required in order to accomplish the reference between the “`server`” object and the “`processors`” object), the hierarchical structure of the original document is largely intact in the NEESML version. Also, the numerical types implicit in the original format have been made explicit, allowing the format to be more easily validated.