nXML Mode

This manual documents nxml-mode, an Emacs major mode for editing XML with RELAX NG support. This manual is not yet complete.

1 Completion

Apart from real-time validation, the most important feature that nxml-mode provides for assisting in document creation is "completion". Completion assists the user in inserting characters at point, based on knowledge of the schema and on the contents of the buffer before point.

The traditional GNU Emacs key combination for completion in a buffer is *M-*⟨TAB⟩. However, many window systems and window managers use this key combination themselves (typically for switching between windows) and do not pass it to applications. It's hard to find key combinations in GNU Emacs that are both easy to type and not taken by something else. *C-*⟨RET⟩ (i.e. pressing the Enter or Return key, while the Ctrl key is held down) is available. It won't be available on a traditional terminal (because it is indistinguishable from Return), but it will work with a window system. Therefore we adopt the following solution by default: use *C-*⟨RET⟩ when there's a window system and *M-*⟨TAB⟩ when there's not. In the following, I will assume that a window system is being used and will therefore refer to *C-*⟨RET⟩.

Completion works by examining the symbol preceding point. This is the symbol to be completed. The symbol to be completed may be the empty. Completion considers what symbols starting with the symbol to be completed would be valid replacements for the symbol to be completed, given the schema and the contents of the buffer before point. These symbols are the possible completions. An example may make this clearer. Suppose the buffer looks like this (where ⋆ indicates point):

```
<html xmlns="http://www.w3.org/1999/xhtml">
<h⋆
```

and the schema is XHTML. In this context, the symbol to be completed is '`h`'. The possible completions consist of just '`head`'. Another example, is

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<⋆
```

In this case, the symbol to be completed is empty, and the possible completions are '`base`', '`isindex`', '`link`', '`meta`', '`script`', '`style`', '`title`'. Another example is:

```
<html xmlns="⋆
```

In this case, the symbol to be completed is empty, and the possible completions are just '`http://www.w3.org/1999/xhtml`'.

When you type *C-*⟨RET⟩, what happens depends on what the set of possible completions are.

- If the set of completions is empty, nothing happens.
- If there is one possible completion, then that completion is inserted, together with any following characters that are required. For example, in this case:

  ```
  <html xmlns="http://www.w3.org/1999/xhtml">
  <⋆
  ```

  C-⟨RET⟩ will yield

  ```
  <html xmlns="http://www.w3.org/1999/xhtml">
  <head⋆
  ```

- If there is more than one possible completion, but all possible completions share a common non-empty prefix, then that prefix is inserted. For example, suppose the buffer is:

  ```
  <html x⋆
  ```

  The symbol to be completed is '`x`'. The possible completions are '`xmlns`' and '`xml:lang`'. These share a common prefix of '`xml`'. Thus, *C-*⟨RET⟩ will yield:

  ```
  <html xml⋆
  ```

  Typically, you would do *C-*⟨RET⟩ again, which would have the result described in the next item.

- If there is more than one possible completion, but the possible completions do not share a non-empty prefix, then Emacs will prompt you to input the symbol in the minibuffer, initializing the minibuffer with the symbol to be completed, and popping up a buffer showing the possible completions. You can now input the symbol to be inserted. The symbol you input will be inserted in the buffer instead of the symbol to be completed. Emacs will then insert any required characters after the symbol. For example, if it contains:

  ```
  <html xml⋆
  ```

  Emacs will prompt you in the minibuffer with

  ```
  Attribute: xml⋆
  ```

  and the buffer showing possible completions will contain

  ```
  Possible completions are:
  xml:lang    xmlns
  ```

  If you input *xmlns*, the result will be:

  ```
  <html xmlns="⋆
  ```

  (If you do *C-*⟨RET⟩ again, the namespace URI will be inserted. Should that happen automatically?)

2 Inserting end-tags

The main redundancy in XML syntax is end-tags. nxml-mode provides several ways to make it easier to enter end-tags. You can use all of these without a schema.

You can use *C-*⟨RET⟩ after '`</`' to complete the rest of the end-tag.

C-c C-f inserts an end-tag for the element containing point. This command is useful when you want to input the start-tag, then input the content and finally input the end-tag. The '`f`' is mnemonic for finish.

If you want to keep tags balanced and input the end-tag at the same time as the start-tag, before inputting the content, then you can use *C-c C-i*. This inserts a '`>`', then inserts the end-tag and leaves point before the end-tag. *C-c C-b* is similar but more convenient for block-level elements: it puts the start-tag, point and the end-tag on successive lines, appropriately indented. The '`i`' is mnemonic for inline and the '`b`' is mnemonic for block.

Finally, you can customize nxml-mode so that `/` automatically inserts the rest of the end-tag when it occurs after '`<`', by doing

M-x customize-variable ⟨RET⟩ *nxml-slash-auto-complete-flag* ⟨RET⟩

and then following the instructions in the displayed buffer.

3 Paragraphs

Emacs has several commands that operate on paragraphs, most notably `M-q`. nXML mode redefines these to work in a way that is useful for XML. The exact rules that are used to find the beginning and end of a paragraph are complicated; they are designed mainly to ensure that `M-q` does the right thing.

A paragraph consists of one or more complete, consecutive lines. A group of lines is not considered a paragraph unless it contains some non-whitespace characters between tags or inside comments. A blank line separates paragraphs. A single tag on a line by itself also separates paragraphs. More precisely, if one tag together with any leading and trailing whitespace completely occupy one or more lines, then those lines will not be included in any paragraph.

A start-tag at the beginning of the line (possibly indented) may be treated as starting a paragraph. Similarly, an end-tag at the end of the line may be treated as ending a paragraph. The following rules are used to determine whether such a tag is in fact treated as a paragraph boundary:

- If the schema does not allow text at that point, then it is a paragraph boundary.
- If the end-tag corresponding to the start-tag is not at the end of its line, or the start-tag corresponding to the end-tag is not at the beginning of its line, then it is not a paragraph boundary. For example, in

  ```
  <p>This is a paragraph with an
  <emph>emphasized</emph> phrase.
  ```

  the '`<emph>`' start-tag would not be considered as starting a paragraph, because its corresponding end-tag is not at the end of the line.
- If there is text that is a sibling in element tree, then it is not a paragraph boundary. For example, in

  ```
  <p>This is a paragraph with an
  <emph>emphasized phrase that takes one source line</emph>
  ```

  the '`<emph>`' start-tag would not be considered as starting a paragraph, even though its end-tag is at the end of its line, because there the text '`This is a paragraph with an`' is a sibling of the '`emph`' element.
- Otherwise, it is a paragraph boundary.

4 Outlining

nXML mode allows you to display all or part of a buffer as an outline, in a similar way to Emacs' outline mode. An outline in nXML mode is based on recognizing two kinds of element: sections and headings. There is one heading for every section and one section for every heading. A section contains its heading as or within its first child element. A section also contains its subordinate sections (its subsections). The text content of a section consists of anything in a section that is neither a subsection nor a heading.

Note that this is a different model from that used by XHTML. nXML mode's outline support will not be useful for XHTML unless you adopt a convention of adding a `div` to enclose each section, rather than having sections implicitly delimited by different `hn` elements. This limitation may be removed in a future version.

The variable `nxml-section-element-name-regexp` gives a regexp for the local names (i.e. the part of the name following any prefix) of section elements. The variable `nxml-heading-element-name-regexp` gives a regexp for the local names of heading elements. For an element to be recognized as a section

- its start-tag must occur at the beginning of a line (possibly indented);
- its local name must match `nxml-section-element-name-regexp`;
- either its first child element or a descendant of that first child element must have a local name that matches `nxml-heading-element-name-regexp`; the first such element is treated as the section's heading.

You can customize these variables using *M-x customize-variable*.

There are three possible outline states for a section:

- normal, showing everything, including its heading, text content and subsections; each subsection is displayed according to the state of that subsection;
- showing just its heading, with both its text content and its subsections hidden; all subsections are hidden regardless of their state;
- showing its heading and its subsections, with its text content hidden; each subsection is displayed according to the state of that subsection.

In the last two states, where the text content is hidden, the heading is displayed specially, in an abbreviated form. An element like this:

```
<section>
<title>Food</title>
<para>There are many kinds of food.</para>
</section>
```

would be displayed on a single line like this:

```
<-section>Food...</>
```

If there are hidden subsections, then a `+` will be used instead of a `-` like this:

```
<+section>Food...</>
```

If there are non-hidden subsections, then the section will instead be displayed like this:

```
<-section>Food...
  <-section>Delicious Food...</>
```

```
  <-section>Distasteful Food...</>
</-section>
```

The heading is always displayed with an indent that corresponds to its depth in the outline, even it is not actually indented in the buffer. The variable `nxml-outline-child-indent` controls how much a subheading is indented with respect to its parent heading when the heading is being displayed specially.

Commands to change the outline state of sections are bound to key sequences that start with *C-c C-o* (*o* is mnemonic for outline). The third and final key has been chosen to be consistent with outline mode. In the following descriptions current section means the section containing point, or, more precisely, the innermost section containing the character immediately following point.

- *C-c C-o C-a* shows all sections in the buffer normally.
- *C-c C-o C-t* hides the text content of all sections in the buffer.
- *C-c C-o C-c* hides the text content of the current section.
- *C-c C-o C-e* shows the text content of the current section.
- *C-c C-o C-d* hides the text content and subsections of the current section.
- *C-c C-o C-s* shows the current section and all its direct and indirect subsections normally.
- *C-c C-o C-k* shows the headings of the direct and indirect subsections of the current section.
- *C-c C-o C-l* hides the text content of the current section and of its direct and indirect subsections.
- *C-c C-o C-i* shows the headings of the direct subsections of the current section.
- *C-c C-o C-o* hides as much as possible without hiding the current section's text content; the headings of ancestor sections of the current section and their child section sections will not be hidden.

When a heading is displayed specially, you can use ⟨RET⟩ in that heading to show the text content of the section in the same way as *C-c C-o C-e*.

You can also use the mouse to change the outline state: *S-mouse-2* hides the text content of a section in the same way as*C-c C-o C-c*; *mouse-2* on a specially displayed heading shows the text content of the section in the same way as *C-c C-o C-e*; *mouse-1* on a specially displayed start-tag toggles the display of subheadings on and off.

The outline state for each section is stored with the first character of the section (as a text property). Every command that changes the outline state of any section updates the display of the buffer so that each section is displayed correctly according to its outline state. If the section structure is subsequently changed, then it is possible for the display to no longer correctly reflect the stored outline state. *C-c C-o C-r* can be used to refresh the display so it is correct again.

5 Locating a schema

nXML mode has a configurable set of rules to locate a schema for the file being edited. The rules are contained in one or more schema locating files, which are XML documents.

The variable '`rng-schema-locating-files`' specifies the list of the file-names of schema locating files that nXML mode should use. The order of the list is significant: when file x occurs in the list before file y then rules from file x have precedence over rules from file y. A filename specified in '`rng-schema-locating-files`' may be relative. If so, it will be resolved relative to the document for which a schema is being located. It is not an error if relative file-names in '`rng-schema-locating-files`' do not not exist. You can use *M-x customize-variable* ⟨RET⟩ *rng-schema-locating-files* ⟨RET⟩ to customize the list of schema locating files.

By default, '`rng-schema-locating-files`' list has two members: '`schemas.xml`', and '*dist-dir*`/schema/schemas.xml`' where '*dist-dir*' is the directory containing the nXML distribution. The first member will cause nXML mode to use a file '`schemas.xml`' in the same directory as the document being edited if such a file exist. The second member contains rules for the schemas that are included with the nXML distribution.

5.1 Commands for locating a schema

The command *C-c C-s C-w* will tell you what schema is currently being used.

The rules for locating a schema are applied automatically when you visit a file in nXML mode. However, if you have just created a new file and the schema cannot be inferred from the file-name, then this will not locate the right schema. In this case, you should insert the start-tag of the root element and then use the command *C-c C-a*, which reapplies the rules based on the current content of the document. It is usually not necessary to insert the complete start-tag; often just '*<name*' is enough.

If you want to use a schema that has not yet been added to the schema locating files, you can use the command *C-c C-s C-f* to manually select the file contaiing the schema for the document in current buffer. Emacs will read the file-name of the schema from the minibuffer. After reading the file-name, Emacs will ask whether you wish to add a rule to a schema locating file that persistently associates the document with the selected schema. The rule will be added to the first file in the list specified '`rng-schema-locating-files`'; it will create the file if necessary, but will not create a directory. If the variable '`rng-schema-locating-files`' has not been customized, this means that the rule will be added to the file '`schemas.xml`' in the same directory as the document being edited.

The command *C-c C-s C-t* allows you to select a schema by specifying an identifier for the type of the document. The schema locating files determine the available type identifiers and what schema is used for each type identifier. This is useful when it is impossible to infer the right schema from either the file-name or the content of the document, even though the schema is already in the schema locating file. A situation in which this can occur is when there are multiple variants of a schema where all valid documents have the same document element. For example, XHTML has Strict and Transitional variants. In a situation like this, a schema locating file can define a type identifier for each variant. As with *C-c C-s C-f*, Emacs will ask whether you wish to add a rule to a schema locating file that persistently associates the document with the specified type identifier.

The command *C-c C-s C-l* adds a rule to a schema locating file that persistently associates the document with the schema that is currently being used.

5.2 Schema locating files

Each schema locating file specifies a list of rules. The rules from each file are appended in order. To locate a schema each rule is applied in turn until a rule matches. The first matching rule is then used to determine the schema.

Schema locating files are designed to be useful for other applications that need to locate a schema for a document. In fact, there is nothing specific to locating schemas in the design; it could equally well be used for locating a stylesheet.

5.2.1 Schema locating file syntax basics

There is a schema for schema locating files in the file '`locate.rnc`' in the schema directory. Schema locating files must be valid with respect to this schema.

The document element of a schema locating file must be '`locatingRules`' and the namespace URI must be '`http://thaiopensource.com/ns/locating-rules/1.0`'. The children of the document element specify rules. The order of the children is the same as the order of the rules. Here's a complete example of a schema locating file:

```
<?xml version="1.0"?>
<locatingRules xmlns="http://thaiopensource.com/ns/locating-rules/1.0">
  <namespace ns="http://www.w3.org/1999/xhtml" uri="xhtml.rnc"/>
  <documentElement localName="book" uri="docbook.rnc"/>
</locatingRules>
```

This says to use the schema '`xhtml.rnc`' for a document with namespace '`http://www.w3.org/1999/xhtml`', and to use the schema '`docbook.rnc`' for a document whose local name is '`book`'. If the document element had both a namespace URI of '`http://www.w3.org/1999/xhtml`' and a local name of '`book`', then the matching rule that comes first will be used and so the schema '`xhtml.rnc`' would be used. There is no precedence between different types of rule; the first matching rule of any type is used.

As usual with XML-related technologies, resources are identified by URIs. The '`uri`' attribute identifies the schema by specifying the URI. The URI may be relative. If so, it is resolved relative to the URI of the schema locating file that contains attribute. This means that if the value of '`uri`' attribute does not contain a '`/`', then it will refer to a filename in the same directory as the schema locating file.

5.2.2 Using the document's URI to locate a schema

A '`uri`' rule locates a schema based on the URI of the document. The '`uri`' attribute specifies the URI of the schema. The '`resource`' attribute can be used to specify the schema for a particular document. For example,

```
<uri resource="spec.xml" uri="docbook.rnc"/>
```

specifies that that the schema for '`spec.xml`' is '`docbook.rnc`'.

The '`pattern`' attribute can be used instead of the '`resource`' attribute to specify the schema for any document whose URI matches a pattern. The pattern has the same syntax as an absolute or relative URI except that the path component of the URI can use a '`*`'

character to stand for zero or more characters within a path segment (i.e. any character other '/'). Typically, the URI pattern looks like a relative URI, but, whereas a relative URI in the 'resource' attribute is resolved into a particular absolute URI using the base URI of the schema locating file, a relative URI pattern matches if it matches some number of complete path segments of the document's URI ending with the last path segment of the document's URI. For example,

```
<uri pattern="*.xsl" uri="xslt.rnc"/>
```

specifies that the schema for documents with a URI whose path ends with '.xsl' is 'xslt.rnc'.

A 'transformURI' rule locates a schema by transforming the URI of the document. The 'fromPattern' attribute specifies a URI pattern with the same meaning as the 'pattern' attribute of the 'uri' element. The 'toPattern' attribute is a URI pattern that is used to generate the URI of the schema. Each '*' in the 'toPattern' is replaced by the string that matched the corresponding '*' in the 'fromPattern'. The resulting string is appended to the initial part of the document's URI that was not explicitly matched by the 'fromPattern'. The rule matches only if the transformed URI identifies an existing resource. For example, the rule

```
<transformURI fromPattern="*.xml" toPattern="*.rnc"/>
```

would transform the URI 'file:///home/jjc/docs/spec.xml' into the URI 'file:///home/jjc/docs/spec.rnc'. Thus, this rule specifies that to locate a schema for a document '*foo*.xml', Emacs should test whether a file '*foo*.rnc' exists in the same directory as '*foo*.xml', and, if so, should use it as the schema.

5.2.3 Using the document element to locate a schema

A 'documentElement' rule locates a schema based on the local name and prefix of the document element. For example, a rule

```
<documentElement prefix="xsl" localName="stylesheet" uri="xslt.rnc"/>
```

specifies that when the name of the document element is 'xsl:stylesheet', then 'xslt.rnc' should be used as the schema. Either the 'prefix' or 'localName' attribute may be omitted to allow any prefix or local name.

A 'namespace' rule locates a schema based on the namespace URI of the document element. For example, a rule

```
<namespace ns="http://www.w3.org/1999/XSL/Transform" uri="xslt.rnc"/>
```

specifies that when the namespace URI of the document is 'http://www.w3.org/1999/XSL/Transform', then 'xslt.rnc' should be used as the schema.

5.2.4 Using type identifiers in schema locating files

Type identifiers allow a level of indirection in locating the schema for a document. Instead of associating the document directly with a schema URI, the document is associated with a type identifier, which is in turn associated with a schema URI. nXML mode does not constrain the format of type identifiers. They can be simply strings without any formal structure or they can be public identifiers or URIs. Note that these type identifiers have nothing to do with the DOCTYPE declaration. When comparing type identifiers, whitespace is normalized in the same way as with the 'xsd:token' datatype: leading and

trailing whitespace is stripped; other sequences of whitespace are normalized to a single space character.

Each of the rules described in previous sections that uses a '`uri`' attribute to specify a schema, can instead use a '`typeId`' attribute to specify a type identifier. The type identifier can be associated with a URI using a '`typeId`' element. For example,

```
<locatingRules xmlns="http://thaiopensource.com/ns/locating-rules/1.0">
  <namespace ns="http://www.w3.org/1999/xhtml" typeId="XHTML"/>
  <typeId id="XHTML" typeId="XHTML Strict"/>
  <typeId id="XHTML Strict" uri="xhtml-strict.rnc"/>
  <typeId id="XHTML Transitional" uri="xhtml-transitional.rnc"/>
</locatingRules>
```

declares three type identifiers '`XHTML`' (representing the default variant of XHTML to be used), '`XHTML Strict`' and '`XHTML Transitional`'. Such a schema locating file would use '`xhtml-strict.rnc`' for a document whose namespace is '`http://www.w3.org/1999/xhtml`'. But it is considerably more flexible than a schema locating file that simply specified

```
<namespace ns="http://www.w3.org/1999/xhtml" uri="xhtml-strict.rnc"/>
```

A user can easily use *C-c C-s C-t* to select between XHTML Strict and XHTML Transitional. Also, a user can easily add a catalog

```
<locatingRules xmlns="http://thaiopensource.com/ns/locating-rules/1.0">
  <typeId id="XHTML" typeId="XHTML Transitional"/>
</locatingRules>
```

that makes the default variant of XHTML be XHTML Transitional.

5.2.5 Using multiple schema locating files

The '`include`' element includes rules from another schema locating file. The behavior is exactly as if the rules from that file were included in place of the '`include`' element. Relative URIs are resolved into absolute URIs before the inclusion is performed. For example,

```
<include rules="../rules.xml"/>
```

includes the rules from '`rules.xml`'.

The process of locating a schema takes as input a list of schema locating files. The rules in all these files and in the files they include are resolved into a single list of rules, which are applied strictly in order. Sometimes this order is not what is needed. For example, suppose you have two schema locating files, a private file

```
<locatingRules xmlns="http://thaiopensource.com/ns/locating-rules/1.0">
  <namespace ns="http://www.w3.org/1999/xhtml" uri="xhtml.rnc"/>
</locatingRules>
```

followed by a public file

```
<locatingRules xmlns="http://thaiopensource.com/ns/locating-rules/1.0">
  <transformURI pathSuffix=".xml" replacePathSuffix=".rnc"/>
  <namespace ns="http://www.w3.org/1999/XSL/Transform" typeId="XSLT"/>
</locatingRules>
```

The effect of these two files is that the XHTML '`namespace`' rule takes precedence over the '`transformURI`' rule, which is almost certainly not what is needed. This can be solved by adding an '`applyFollowingRules`' to the private file.

```
<locatingRules xmlns="http://thaiopensource.com/ns/locating-rules/1.0">
  <applyFollowingRules ruleType="transformURI"/>
  <namespace ns="http://www.w3.org/1999/xhtml" uri="xhtml.rnc"/>
</locatingRules>
```

6 DTDs

nxml-mode is designed to support the creation of standalone XML documents that do not depend on a DTD. Although it is common practice to insert a DOCTYPE declaration referencing an external DTD, this has undesirable side-effects. It means that the document is no longer self-contained. It also means that different XML parsers may interpret the document in different ways, since the XML Recommendation does not require XML parsers to read the DTD. With DTDs, it was impractical to get validation without using an external DTD or reference to an parameter entity. With RELAX NG and other schema languages, you can simulataneously get the benefits of validation and standalone XML documents. Therefore, I recommend that you do not reference an external DOCTYPE in your XML documents.

One problem is entities for characters. Typically, as well as providing validation, DTDs also provide a set of character entities for documents to use. Schemas cannot provide this functionality, because schema validation happens after XML parsing. The recommended solution is to either use the Unicode characters directly, or, if this is impractical, use character references. nXML mode supports this by providing commands for entering characters and character references using the Unicode names, and can display the glyph corresponding to a character reference.

7 Limitations

nXML mode has some limitations:

- DTD support is limited. Internal parsed general entities declared in the internal subset are supported provided they do not contain elements. Other usage of DTDs is ignored.
- The restrictions on RELAX NG schemas in section 7 of the RELAX NG specification are not enforced.
- Unicode support has problems. This stems mostly from the fact that the XML (and RELAX NG) character model is based squarely on Unicode, whereas the Emacs character model is not. Emacs 22 is slated to have full Unicode support, which should improve the situation here.