Review Q: Coding practices

Table of contents

1 Coding practices for DITA grammar files

This section contains information about creating modular DTD- or RELAX NG-based grammar files. It explains how document-type shells, specialization modules, and element-configuration modules (constraint and expansion) are organized.

1.1 DTD coding requirements

This section explains how to implement DTD-based document-type shells, specializations, and element-configuration modules (constraint and expansion).

1.1.1 DTD: Use of entities

DITA-based DTDs use entities to implement specialization and element configuration. Therefore, an understanding of entities is critical when working with DTD-based document-type shells, vocabulary modules, or element-configuration modules (constraint and expansion).

Entities can be defined multiple times within a single document type, but only the first definition is effective. How entities work shapes DTD coding practices. The following list describes a few of the more important entities that are used in DITA DTDs:

Elements defined as entities

Every element in a DITA DTD is defined as an entity. When elements are added to a content model, they are added using the entity. This enables extension with domain specializations.

For example, the entity `%ph;` usually just means the `<ph>` element, but it can be defined in a document-type shell to mean "`<ph>` plus the elements from the highlighting domain". Because the document-type shell places that entity definition before the usual definition, every element that includes `%ph;` in its content model now includes `<ph>` plus every element in the highlighting domain that is specialized from `<ph>`.

Content models defined as entities

Every element in a DITA DTD defines its content model using an entity. This enables element configuration.

For example, the content model for the `<p>` element is set to `%ph.content;`, and the `%ph.content;` entity defines the actual content model. A constraint module then can redefine the `%ph.content;` entity to remove selected elements from the content model, or an expansion module can redefine the `%ph.content;` entity to add elements to the content model.

Attribute sets defined as entities

Every element name in a DITA DTD defines its attributes using a parameter entity. This enables element configuration.

For example, the attribute list for the `<ph>` element is set to `%ph.attributes;`, and the `%ph.attributes;` entity defines the actual attribute list. A constraint module then can redefine the entity to remove attributes from the attribute list, or an expansion module can redefine the entity to add attributes to the attribute list.

Note When constructing an element-configuration module or document-type shell, new entities are usually viewed as "redefinitions" because they redefine entities that already exist. However, these new definitions only work because they are added to a document-type shell before the existing definitions. Most topics about DITA DTDs, including others in this specification, describe these overrides as redefinitions to ease understanding.

1.1.2 DTD: Coding requirements for document-type shells

A DTD-based document-type shell is organized into sections. Each section contains entity declarations that follow specific coding rules.

The DTD-based approach to configuration, specialization, and element configuration (constraint and expansion) relies heavily upon parameter entities. Several of the parameter entities that are declared in document-type shells contain references to other parameter entities. Because parameter entities must be declared before they are used, the order of the sections in a DTD-based document-type shell is significant.

A DTD-based document-type shell contains the following sections:

1. Topic [or map] entity declarations (4)
2. Domain constraint integration (4)
3. Domain entity declarations (5)
4. Domain attributes declarations (5)
5. Domain extensions (5)
6. Domain attribute extensions (6)
7. Topic nesting override (6)
8. Specializations attribute override (6)
9. Element-type configuration integration (7)
10. Topic [or map] element integration (7)
11. Domain element integration (7)

Each of the sections in a DTD-based document-type shell follows a pattern. These patterns help ensure that the shell follows XML parsing rules for DTDs. They also establish a modular design that simplifies creation of new document-type shells.

Topic [or map] entity declarations

This section declares and references an external parameter entity for each of the following items:

- The entity declaration module (`.ent` file) for the top-level topic or map type that the document-type shell configures
- The entity declaration modules for any additional structural modules that are used by the document-type shell

The parameter entity is named *typeName*`-dec`.

In the following example, the entity declaration module for the `<concept>` specialization is integrated into a document-type shell:

```
<!-- ============================================================ -->
<!--              TOPIC ENTITY DECLARATIONS                       -->
<!-- ============================================================ -->

<!ENTITY % concept-dec
  PUBLIC "-//OASIS//ENTITIES DITA 2.0 Concept//EN"
         "concept.ent"
>%concept-dec;
```

Domain constraint integration

This section declares and references an external parameter entity for each domain-constraint module that is integrated into the document-type shell.

The parameter entity is named *descriptorDomainName*`-c-dec`.

In the following example, the entity file for a constraint module that reduces the highlighting domain to a subset is integrated in a document-type shell:

```
<!-- ============================================================= -->
<!--                    DOMAIN CONSTRAINT INTEGRATION              -->
<!-- ============================================================= -->

<!ENTITY % HighlightingDomain-c-dec
  PUBLIC "-//ACME//ENTITIES DITA Highlighting Domain Constraint//EN"
  "acme-HighlightingDomainConstraint.mod"
>%basic-HighlightingDomain-c-dec;
```

Domain entity declarations

This section declares and references an external parameter entity for each element-domain module that is integrated into the document-type shell.

The parameter entity is named *shortDomainName*-dec.

In the following example, the entity file for the highlighting domain is included in a document-type shell:

```
<!-- ============================================================= -->
<!--             DOMAIN ENTITY DECLARATIONS                        -->
<!-- ============================================================= -->

<!ENTITY % hi-d-dec PUBLIC
    "-//OASIS//ENTITIES DITA 2.0 Highlight Domain//EN"
    "highlightDomain.ent"
>%hi-d-dec;
```

Domain attributes declarations

This section declares and references an external parameter entity for each attribute domain that is integrated into the document-type shell.

The parameter entity is named *domainName*Att-dec.

In the following example, the entity file for the @deliveryTarget attribute domain is included in a document-type shell:

```
<!-- ============================================================= -->
<!--             DOMAIN ATTRIBUTES DECLARATIONS                    -->
<!-- ============================================================= -->

<!ENTITY % deliveryTargetAtt-d-dec
  PUBLIC "-//OASIS//ENTITIES DITA 2.0 Delivery Target Attribute Domain//EN"
         "deliveryTargetAttDomain.ent"
>%deliveryTargetAtt-d-dec;
```

Domain extensions

This section declares and references a parameter entity for each element that is extended by one or more domain modules. These entities are used by the element-domain modules that are declared later in the document-type shell to redefine the content models. Redefining the content models adds domain specializations wherever the base element is allowed.

In the following example, the entity for the <pre> element is redefined to add specializations from the programming, software, and user interface domains:

```
<!-- ============================================================= -->
<!--                    DOMAIN EXTENSIONS                          -->
<!-- ============================================================= -->

<!ENTITY % pre
    "pre        |
     %pr-d-pre; |
```

```
      %sw-d-pre; |
      %ui-d-pre;">
```

Domain attribute extensions

This section redefines the parameter entity for each attribute domain that is integrated globally into the document-type shell. The redefinition adds an extension to the parameter entity for the relevant attribute.

In the following example, the parameter entities for the `@base` and `@props` attributes are redefined to include the `@newfrombase`, `@othernewfrombase`, `@new`, and `@othernew` attributes:

```
<!-- ============================================================= -->
<!--                    DOMAIN ATTRIBUTE EXTENSIONS                -->
<!-- ============================================================= -->

<!ENTITY % base-attribute-extensions
        "%newfrombaseAtt-d-attribute;
         %othernewfrombaseAtt-d-attribute;">

<!ENTITY % props-attribute-extensions
        "%newAtt-d-attribute;
         %othernewAtt-d-attribute;">
```

Topic nesting override

This section redefines the entity that controls topic nesting for each topic type that is integrated into the document-type shell.

The parameter entity is named *topictype*`-info-types`.

The definition usually is an "OR" list of the topic types that can be nested in the parent topic type. Use the literal root-element name, not the corresponding parameter entity. Topic nesting can be disallowed completely by specifying the `<no-topic-nesting>` element.

In the following example, the parameter entity specifies that `<concept>` can nest any number of `<concept>` or `<myTopicType>` topics, in any order:

```
<!-- ============================================================= -->
<!--                     TOPIC NESTING OVERRIDE                    -->
<!-- ============================================================= -->

<!ENTITY % concept-info-types "concept | myTopicType">
```

Specializations attribute override

This section redefines the `included-domains` entity to include the text entity for each attribute domain that is included in the document-type shell. The redefinition sets the effective value of the `@specializations` attribute for the top-level document type that is configured by the document-type shell.

In the following example, parameter entities are included for the DITA conditional-processing attributes:

```
<!-- ============================================================= -->
<!--                 SPECIALIZATIONS ATTRIBUTE OVERRIDE            -->
<!-- ============================================================= -->

<!ENTITY included-domains
                          "&audienceAtt-d-att;
                           &deliveryTargetAtt-d-att;
                           &otherpropsAtt-d-att;
                           &platformAtt-d-att;
                           &productAtt-d-att;"
>
```

Element-type configuration integration

This section declares and references the parameter entity for each element-configuration module (constraint and expansion) that is integrated into the document-type shell.

The parameter entity is named *descriptionElement*-c-def.

In the following example, the module that constrains the content model for the <taskbody> element is integrated into the document-type shell for strict task:

```
<!ENTITY % strictTaskbody-c-def
  PUBLIC "-//OASIS//ELEMENTS DITA 2.0 Strict Taskbody Constraint//EN"
  "strictTaskbodyConstraint.mod"
>%strictTaskbody-c-def;
```

Topic [or map] element integration

This section declares and references an external parameter entity for the element declaration module (.mod file) for each structural module that is integrated into the document-type shell.

The parameter entity is named *structuralType*-type.

The structural modules are included in ancestry order, so that the parameter entities that are used in an ancestor module are available for use in specializations. When a structural module depends on elements from a vocabulary module that is not part of its ancestry, the module upon which the structural module has a dependency (and any ancestor modules not already included) need to be included before the module with a dependency.

In the following example, the structural modules that are required by the troubleshooting topic are integrated into the document-type shell:

```
<!-- ============================================================= -->
<!--                    TOPIC ELEMENT INTEGRATION                  -->
<!-- ============================================================= -->

<!ENTITY % topic-type
  PUBLIC "-//OASIS//ELEMENTS DITA 2.0 Topic//EN"
         "../../base/dtd/topic.mod"
>%topic-type;

<!ENTITY % task-type
  PUBLIC "-//OASIS//ELEMENTS DITA 2.0 Task//EN"
         "task.mod"
>%task-type;

<!ENTITY % troubleshooting-type
  PUBLIC "-//OASIS//ELEMENTS DITA 2.0 Troubleshooting//EN"
         "troubleshooting.mod"
>%troubleshooting-type;
```

Domain element integration

This section declares and references an external parameter entity for each element domain that is integrated into the document-type shell.

The parameter entity is named *domainName*-def.

In the following example, the element-definition file for the highlighting domain is integrated into the document-type shell:

```
<!-- ============================================================= -->
<!--                    DOMAIN ELEMENT INTEGRATION                 -->
<!-- ============================================================= -->

<!ENTITY % hi-d-def PUBLIC
    "-//OASIS//ELEMENTS DITA 2.0 Highlight Domain//EN"
```

```
    "highlightDomain.mod"
>%hi-d-def;
```

If a structural module depends on a domain, the domain module needs to be included before the structural module. This erases the boundary between the final two sections of the DTD-based document-type shell, but it is necessary to ensure that modules are embedded after their dependencies. Technically, the only solid requirement is that both domain and structural modules be declared after all other modules that they specialize from or depend on.

1.1.3 DTD: Coding requirements for structural and element-domain modules

This topic covers general coding requirements for defining element types in both structural and element-domain vocabulary modules.

Module files

A vocabulary module that defines a structural or element-domain specialization is composed of two files:

Definition module file

This (`.mod`) file declares the element names, content models, and attribute lists for the element types that are defined in the vocabulary module.

Entity declaration file

This (`.ent`) file declares the general and parameter entities that are used to integrate the vocabulary module into a document-type shell.

Element definitions

A structural or element-domain vocabulary module contains a declaration for each element type that is named by the module. While the XML standard allows content models to refer to undeclared element types, the DITA standard does not permit this. All element types or attribute lists that are named within a vocabulary module are declared in one of the following objects:

- The vocabulary module
- A base module of which the vocabulary module is a direct or indirect specialization
- (For structural modules) A required domain module

The following components make up a single element definition in a DITA DTD-based vocabulary module.

Element name entities

For each element type, there is a parameter entity with a name that matches the element-type name. The value is the element-type name.

For example:

```
<!ENTITY % topichead "topichead">
```

The parameter entity provides a layer of abstraction when setting up content models. It can be redefined in a document-type shell in order to create domain extensions or implement element configuration (constraint and expansion).

Element name entities for a vocabulary module typically are grouped together at the top of the vocabulary module. They can occur in any order.

Content-model parameter entity

For each element type, there is a parameter entity that defines the content model. The name of the parameter entity is *tagname*.content, and the value is the content model definition.

For example:

```
<!ENTITY % topichead.content
  "((%topicmeta;)?,
    (%data.elements.incl; |
     %navref; |
     %topicref;)*)
">
```

Attribute-list parameter entity

For each element type, there is a parameter entity that declares the attributes that are available on the element. The name of the parameter entity is *tagname*.attributes, and the value is a list of the attributes that are used by the element type (except for @class).

For example:

```
<!ENTITY % topichead.attributes
 "keys CDATA #IMPLIED
  %topicref-atts;
  %univ-atts;"
>
```

Consistent use and naming of the *tagname*.content parameter entity enables the use of element-configuration modules (constraint and expansion) to redefine the content model.

Element declaration

For each element type, there is an element declaration that consists of a reference to the content-model parameter entity.

For example:

```
<!ELEMENT topichead    %topichead.content;>
```

Attribute list declaration

For each element type, there is an attribute-list declaration that consists of a reference to the attribute-list parameter entity.

For example:

```
<!ATTLIST topichead    %topichead.attributes;>
```

Specialization attribute declarations

A vocabulary module defines a @class attribute for every element that is declared in the module. The value of the attribute is constructed according to the rules in The class attribute rules and syntax.

For example, the ATTLIST definition for the <topichead> element (a specialization of the <topicref> element in the base map type) includes the definition of the @class attribute, as follows:

```
<!ATTLIST topichead  class CDATA "+ map/topicref  mapgroup-d/topichead ">
```

Definition of the <topichead> element

The following code sample shows how the `<topichead>` element is defined in `mapGroup.mod`. Ellipses indicate where the code sample has been snipped for brevity.

```
<!-- ============================================================= -->
<!--                    ELEMENT NAME ENTITIES                      -->
<!-- ============================================================= -->

<!ENTITY % topichead      "topichead"                                    >

...

<!-- ============================================================= -->
<!--                     ELEMENT DECLARATIONS                      -->
<!-- ============================================================= -->

<!--                     LONG NAME: Topichead                      -->
<!ENTITY % topichead.content
                        "((%topicmeta;)?,
                          (%data.elements.incl; |
                           %navref; |
                           %topicref;)*)"
>
<!ENTITY % topichead.attributes
               "keys
                           CDATA
                                      #IMPLIED
                %topicref-atts;
                %univ-atts;"
>
<!ELEMENT  topichead %topichead.content;>
<!ATTLIST  topichead %topichead.attributes;>

...

<!-- ============================================================= -->
<!--             SPECIALIZATION ATTRIBUTE DECLARATIONS             -->
<!-- ============================================================= -->

...

<!ATTLIST  topichead      class CDATA "+ map/topicref mapgroup-d/topichead ">

<!-- ================== End of DITA Map Group Domain ==================== -->
```

1.1.4 DTD: Coding requirements for structural modules

This topic covers general coding requirements for DTD-based structural modules.

Required topic and map element attributes

The topic or map element type sets the `@ditaarch:DITAArchVersion` attribute to the version of DITA in use, typically by referencing the `arch-atts` parameter entity. It also sets the `@specializations` attribute to the `included-domains` entity.

The `@DITAArchVersion` and `@specializations` attributes give processors a reliable way to check the architecture version and look up the list of attribute domains that are available in the document type.

The following example shows how the `@DITAArchVersion` and `@specializations` attributes are defined for the `<concept>` element in DITA 2.0. Ellipses indicate where the code is snipped for brevity:

```
<!-- ============================================================= -->
<!--                     ELEMENT DECLARATIONS                      -->
<!-- ============================================================= -->

...
```

```
<!--                    LONG NAME: Concept                         -->

...

<!ATTLIST concept
  %concept.attributes;
  %arch-atts;
  specializations  CDATA  "&included-domains;">
```

Controlling nesting in topic types

A structural module that defines a new topic type typically uses a parameter entity to define a default for what topic types are permitted to nest. When this is done consistently, a shell that includes multiple structural modules can set common nesting rules for all topic types by setting `%info-types;` entity.

The following rules apply when using parameter entities to control nesting.

Parameter entity name

The name of the parameter entity is the topic element name plus the `-info-types` suffix.

For example, the name of the parameter entity for the concept topic is `concept-info-types`.

Parameter entity value

To set up default topic nesting rules, set the entity to the desired topic elements. The default topic nesting is used when a document-type shell does not set up different rules.

For example, the following parameter entity sets up default nesting so that `<concept>` will nest only other `<concept>` topics:

```
<!-- ============================================================ -->
<!--                    ELEMENT DECLARATIONS                      -->
<!-- ============================================================ -->

<!ENTITY % concept-info-types
             "%info-types;"
>
```

As an additional example, the following parameter entity sets up a default that will not allow any nesting:

```
<!ENTITY % glossentry-info-types "no-topic-nesting">
```

Content model of the root element

The last position in the content model defined for the root element of a topic type should be the *topictype*`-info-types` parameter entity.

A document-type shell then can control how topics are allowed to nest for this specific topic type by redefining the *topictype*`-info-types` entity for each topic type.

For example, with the following content model defined for `<concept>`, a document-type shell that uses the concept specialization can control which topics are nested in `<concept>` by redefining the `concept-info-types` parameter entity:

```
<!ENTITY % concept.content
  "((%title;),
    (%abstract; | %shortdesc;)?,
    (%prolog;)?,
    (%conbody;)?,
    (%related-links;)?,
    (%concept-info-types;)*)"
>
```

In certain cases, you do not need to use an `info-types` parameter entity to control topic nesting:

- If you want a specialized topic type to disallow nested topics, regardless of context, it can be defined without any entity or any nested topics.
- If you want a specialized topic type to only allow specific nesting patterns, such as allowing only other topic types that are defined in the same module, it can nest those topics directly in the same way that other nested elements are defined.

1.1.5 DTD: Coding requirements for element-domain modules

The vocabulary modules that define element domains have an additional coding requirement. The entity declaration file must include a parameter entity for each element that the domain extends.

Parameter entity name

The name of the parameter entity is the abbreviation for the domain, followed by a hyphen ("-") and the name of the element that is extended.

For example, the name of the parameter entity for the highlighting domain that extends the `<ph>` element is `hi-d-ph`.

Parameter entity value

The value of the parameter entity is a list of the specialized elements that can occur in the same locations as the extended element. Each element is separated by the vertical line (|) symbol.

For example, the value of the `%hi-d-ph;` parameter entity is `"b | u | i | line-through | overline | tt | sup | sub"`.

Example

The following code sample shows the parameter entity for the highlight domain, as declared in `highlightDomain.ent`:

```
<!-- ============================================================= -->
<!--                    ELEMENT EXTENSION ENTITY DECLARATIONS      -->
<!-- ============================================================= -->

<!ENTITY % hi-d-ph "b | i | line-through | overline | sup | sub | tt | u">

<!-- ================ End DITA Highlight Domain ================== -->
```

1.1.6 DTD: Coding requirements for attribute-domain modules

The vocabulary modules that define attribute domains have additional coding requirements. The module must include a parameter entity for the new attribute, which can be referenced in document-type shells, as well as a general entity that specifies the contribution to the `@specializations` attribute for the attribute domain.

The name of an attribute domain is the name of the attribute plus "Att". For example, for the attribute named `@deliveryTarget`, the attribute-domain name is "deliveryTargetAtt". The attribute-domain name is used to construct entity names for the domain.

Parameter entity name and value

The name of the parameter entity is the attribute-domain name, followed by the literal `-d-attribute`. The value of the parameter entity is a DTD declaration for the attribute.

General entity name and value

The general entity name is the attribute-domain name, followed by the literal `-d-Att`. The value of the text entity is the `@specializations` attribute contribution for the module. See The specializations attribute rules and syntax for details on how to construct this value.

Example

The `@deliveryTarget` attribute can be defined in a vocabulary module using the following two entities.

```
<!ENTITY % deliveryTargetAtt-d-attribute
  "deliveryTarget  CDATA  #IMPLIED"
>

<!ENTITY deliveryTargetAtt-d-att "@props/deliveryTarget" >
```

1.1.7 DTD: Coding requirements for element-configuration modules

Element-configuration modules (constraint and expansion) have specific coding requirements.

The *tagname*.attributes parameter entity

When the attribute list for an element is constrained or expanded, there is a declaration of the *tagname*`.attributes` parameter entity that defines the modified attributes.

The following list provides examples for both constraint and expansion modules:

Constraint module

The following parameter entity defines a constrained attributes list for the `<note>` element that removes most of the values defined for `@type`. It also removes `@othertype`:

```
<!ENTITY % note.attributes
       "type  (attention | caution | note ) #IMPLIED
        %univ-atts;">
```

The following parameter entity restricts the highlighting domain to `<b>` and `<i>`:

```
<!ENTITY % HighlightingDomain-c-ph     "b | i"  >
```

Expansion module

The following parameter entity defines a new attribute intended for use with various table elements:

```
<!ENTITY % cellPurposeAtt-d-attribute-expansion
  "cell-purpose  (sale | out-of-stock | new | last-chance | inherit | none)  #IMPLIED"
>
```

For expansion modules, note the following considerations. The *tagname*`.attributes` parameter entity can be defined in an attribute-specialization module, or it can be defined directly in the expansion module.

The *tagname*.content parameter entity

When the content model for an element is constrained or expanded, there is a declaration of the *tagname*`.content` parameter entity that defines the modified content model.

The following list provides examples for both constraint and expansion modules:

Constraint module

The following parameter entity defines a more restricted content model for `<topic>`, in which the `<shortdesc>` element is required.

```
<!ENTITY % topic.content
  "((%title;),
    (%shortdesc;),
    (%prolog;)?,
    (%body;)?,
```

```
    (%topic-info-types;)*)"
>
```

Note that replacing a base element with domain extensions is a form of constraint that can be accomplished directly in the document-type shell. No constraint module is required.

In the following example, the <pre> base type is removed from the entity declaration, effectively allowing specializations of <pre> but not <pre> itself.

```
<!ENTITY % pre
    "%pr-d-pre; |
     %sw-d-pre; |
     %ui-d-pre;">
```

Expansion module

The redefinition of the content model references the parameter entity that was defined in the element-specialization module.

The following code sample shows the entity declaration file for an element-specialization module that defines a <section-shortdesc> element, which is intended to be added to the content model of <section>:

```
<!ENTITY % section-shortdesc "section-shortdesc">
```

When the content model for <section> is redefined in the expansion module, it references the parameter entity defined in the entities file for the element specialization:

```
<!ENTITY % section.content
              "(#PCDATA |
               %dl; |
               %div; |
               %fig; |
               %image; |
               %note; |
               %ol; |
               %p; |
               %simpletable; |
               %ul; |
               %title; |
               %draft-comment;|
               %sectionShortdesc;)*"
>
```

Note that this expansion module also constrains the content model of <section> to only include certain block elements.

Related concepts

Examples: Constraints implemented using DTDs

Examples: Expansion implemented using DTDs

1.2 RELAX NG coding requirements

This section explains how to implement RNG-based document-type shells, specializations, and element-configuration modules (constraints and expansions).

If you plan to generate DTD- or XSD-based modules from RELAX NG modules, avoid RELAX NG features that cannot be translated into DTD or XSD constructs. Such features include lexical patterns for attributes and elements, interleave patterns, and context-specific patterns for content models or attribute lists.

When RELAX NG is used directly for DITA document validation, the document-type shells for those documents can integrate constraint modules that use the full power of RELAX NG to enforce constraints that cannot be enforced by DTDs or XSDs.

1.2.1 RELAX NG: Overview of coding requirements

This topic contains general information about the self-integrating aspect of domain specialization modules, RELAX NG grammar files, and the two RNG syntaxes

Self-integration of RELAX NG domain modules

Domain modules coded in RELAX NG are self-integrating; they automatically add to the content models and attribute lists that they extend. This aspect of RELAX NG results in the following coding practices:

- Each domain module consists of a single file, unlike the two required for DTDs.
- The domain modules directly extend elements, unlike DTDs, which rely on an extra file and extensions within the document-type shell.
- Element-configuration modules (constraint and expansion) directly include the modules that they extend, which means that just by referencing an element-configuration module, the document-type shell gets everything it needs to redefine a vocabulary module.

General RELAX NG information

RELAX NG grammars for DITA document-type shells, vocabulary modules, and element-configuration modules (constraint and expansion) can do the following:

- Use the `<a:documentation>` element anywhere that foreign elements are allowed by RELAX NG. The `<a:documentation>` element refers to the `<documentation>` element type from the `http://relaxng.org/ns/compatibility/annotations/1.0` as defined by the DTD compatibility specification. The prefix "a" is used by convention.
- Use `<div>` to group pattern declarations.
- Include embedded Schematron rules or any other foreign vocabulary. Processors can ignore any foreign vocabularies within DITA grammars that are not in the `http://relaxng.org/ns/compatibility/annotations/1.0` or `http://dita.oasis-open.org/architecture/2005/` namespaces.

Syntaxes for RELAX NG grammars

The RELAX NG specification defines two syntaxes for RELAX NG grammars: the XML syntax and the compact syntax. The two syntaxes are functionally equivalent, and either syntax can be reliably converted into the other by using, for example, the open-source Trang tool.

The DITA coding requirements are defined for the RELAX NG XML syntax. Document-type shells, vocabulary modules, and element-configuration modules (constraints and expansion) that use the RELAX NG compact syntax can use the same organizational structures as those defined for the RELAX NG XML syntax.

DITA practitioners can author DITA modules using one RELAX NG syntax, and then use tools to generate modules in the other syntax. The resulting RELAX NG modules are equivalent if there is a one-to-one file correspondence.

1.2.2 RELAX NG: Coding requirements for document-type shells

A RNG-based document-type shell is organized into sections; each section follows a pattern. These patterns help ensure that the shell follows XML parsing rules for RELAX NG; they also establish a modular design that simplifies creation of new document-type shells.

An RNG-based document-type shell contains the following sections:

1. Root element declaration (16)
2. specializations attribute (16)
3. Element-type configuration integration (16)
4. Module inclusions (17)
5. ID-defining element overrides (17)

Root element declaration

Document-type shells use the RELAX NG start declaration to specify the root element of the document type. The `<start>` element defines the root element, using a reference to a *tagname*`.element` pattern.

For example:

```
<div>
  <a:documentation>ROOT ELEMENT DECLARATION</a:documentation>
  <start combine="choice">
    <ref name="topic.element"/>
  </start>
</div>
```

@specializations attribute

This section lists the tokens that attribute-domain and element-configuration modules contribute to the `@specializations` attribute.

For example:

```
<div>
  <a:documentation>SPECIALIZATIONS ATTRIBUTE</a:documentation>
  <define name="specializations-att">
    <optional>
      <attribute name="specializations"
                 a:defaultValue="@props/audience
                                 @props/deliveryTarget
                                 @props/otherprops
                                 @props/platform
                                 @props/product"
      />
    </optional>
  </define>
</div>
```

Element-type configuration integration

This section of the document-type shell contains includes for element-type configuration modules (constraint and expansion). Because an element-configuration module imports the module that it overrides, any module that is configured in this section (including the base topic or map modules) is left out of the following "Module inclusion" section.

The following code sample shows the section of an RNG-based document-type shell that redefines the `<taskbody>` element to create the strict task topic.

```
<div>
<a:documentation>ELEMENT-TYPE CONFIGURATION INTEGRATION</a:documentation>
```

```
  <include href="strictTaskbodyConstraintMod.rng"/>
</div>
```

Module inclusions

This section of the RNG-based document-type shell includes all unconstrained domain or structural modules.

For example:

```
<div>
  <a:documentation>MODULE INCLUSIONS</a:documentation>
  <include href="topicMod.rng">
    <define name="topic-info-types">
      <ref name="topic.element"/>
    </define>
  </include>
  <include href="audienceAttDomain.rng" dita:since="2.0"/>
  <include href="deliveryTargetAttDomain.rng"/>
  <include href="otherpropsAttDomain.rng" dita:since="2.0"/>
  <include href="platformAttDomain.rng" dita:since="2.0"/>
  <include href="productAttDomain.rng" dita:since="2.0"/>
  <include href="alternativeTitlesDomain.rng" dita:since="2.0"/>
  <include href="emphasisDomain.rng" dita:since="2.0"/>
  <include href="hazardstatementDomain.rng"/>
  <include href="highlightDomain.rng"/>
  <include href="utilitiesDomain.rng"/>
</div>
```

ID-defining element overrides

This section declares any element in the document type that uses an `@id` attribute with an XML data type of "ID". This declaration is required in order to prevent RELAX NG parsers from issuing errors.

If the document-type shell includes domains for foreign vocabularies such as SVG or MathML, this section also includes exclusions for the namespaces used by those domains.

For example, the following code sample is from an RNG-based document-type shell for a task topic. It declares that both the `<topic>` and `<task>` elements have an `@id` attribute with an XML data type of ID. These elements and any elements in the SVG or MathML namespaces are excluded from the "any" pattern by being placed within the `<except>` element:

```
<div>
    <a:documentation> ID-DEFINING-ELEMENT OVERRIDES </a:documentation>
    <define name="any">
       <zeroOrMore>
          <choice>
             <ref name="idElements"/>
             <element>
                <anyName>
                   <except>
                      <name>topic</name>
                      <name>task</name>
                      <nsName ns="http://www.w3.org/2000/svg"/>
                      <nsName ns="http://www.w3.org/1998/Math/MathML"/>
                   </except>
                </anyName>
                <zeroOrMore>
                   <attribute>
                      <anyName/>
                   </attribute>
                </zeroOrMore>
                <ref name="any"/>
             </element>
             <text/>
          </choice>
       </zeroOrMore>
```

```
      </define>
  </div>
```

1.2.3 RELAX NG: Coding requirements for structural and element-domain modules

This topic covers general coding requirements for defining element types in both structural and element-domain vocabulary modules.

Module files

Each RELAX NG vocabulary module consists of a single module file.

Element definitions

A structural or element-domain vocabulary module contains a declaration for each element type that is named in the module. While the XML standard allows content models to refer to undeclared element types, the DITA standard does not permit it. All element types or attribute lists that are named in a vocabulary module are declared in one of the following objects:

- The vocabulary module
- A base module of which the vocabulary module is a direct or indirect specialization
- (If the vocabulary module is a structural module) A required domain or structural module

The element type patterns are organized into the following sections:

Element type name patterns

For each element type that is declared in the vocabulary module, there is a pattern whose name is the element type name and whose content is a reference to the *tagname*`.element` pattern for the element type.

The following example shows the pattern for the `<b>` element:

```
<div>
  <a:documentation>ELEMENT TYPE NAME PATTERNS</a:documentation>
    <!-- ... -->
    <define name="b">
      <ref name="b.element"/>
    </define>
    <!-- ... -->
</div>
```

The element-type name pattern provides a layer of abstraction that facilitates redefinition. The element-type name patterns are referenced from content model and domain extension patterns. Specialization modules can re-declare the patterns to include specializations of the type, allowing the specialized types in all contexts where the base type is allowed.

The declarations can occur in any order.

Common content-model patterns

Structural and element-domain modules can include a section that defines the patterns that contribute to the content models of the element types that are defined in the module.

Common attribute sets

Structural and element-domain modules can include a section that defines patterns for attribute sets that are common to one or more of the element types that are defined in the module.

Element type declarations

For each element type that is declared in the vocabulary module, the following set of patterns are used to define the content model and attributes for the element type. Each set of patterns typically is grouped within a `<div>` element.

***tagname*.content**
Defines the complete content model for the element *tagname*. The content model pattern can be overridden in element-configuration modules (constraint and expansion).

***tagname*.attributes**
Defines the complete attribute list for the element *tagname*, except for `@class`. The attribute list declaration can be overridden in element-configuration modules (constraint and expansion).

***tagname*.element**
Provides the actual element-type definition. It contains an `<element>` element whose `@name` value is the element type name and whose content is a reference to the *`tagname`*`.content` and *`tagname`*`.attlist` patterns.

***tagname*.attlist**
Contains an additional attribute-list pattern with a `@combine` attribute set to the value "interleave". This pattern contains only a reference to the *`tagname`*`.attributes` pattern. This pattern enables the integration of attribute specializations.

The following example shows the declaration for the `<topichead>` element, including the definition for each pattern described above.

```
<div>
  <a:documentation>Topic Head</a:documentation>
  <define name="topichead.content">
    <optional>
      <ref name="topicmeta"/>
    </optional>
    <zeroOrMore>
      <choice>
        <ref name="data.elements.incl"/>
        <ref name="navref"/>
        <ref name="topicref"/>
      </choice>
    </zeroOrMore>
  </define>
  <define name="topichead.attributes">
    <optional>
      <attribute name="keys"/>
    </optional>
    <ref name="topicref-atts"/>
    <ref name="univ-atts"/>
  </define>
  <define name="topichead.element">
    <element name="topichead">
      <a:documentation/>
      <ref name="topichead.attlist"/>
      <ref name="topichead.content"/>
    </element>
  </define>
  <define name="topichead.attlist" combine="interleave">
    <ref name="topichead.attributes"/>
  </define>

</div>
```

idElements pattern contribution

Element types that declare the `@id` attribute as type "ID", including all topic and map element types, provide a declaration for the `idElements` pattern. This is required to correctly configure the "any"

pattern override in document-type shells and avoid errors from RELAX NG parsers. The declaration is specified with a `@combine` attribute set to the value "choice".

For example:

```
<div>
  <a:documentation>LONG NAME: Map</a:documentation>
  <!-- ... -->
  <define name="idElements" combine="choice">
    <ref name="map.element"/>
  </define>
</div>
```

Specialization attribute declarations

A vocabulary module must define a `@class` attribute for every specialized element. This is done in a section at the end of each module that includes a *tagname*`.attlist` pattern for each element type that is defined in the module. The declarations can occur in any order.

The *tagname*`.attlist` pattern for each element defines the value for the `@class` attribute for the element. `@class` is declared as an optional attribute. The default value is declared using the `@a:defaultValue` attribute, and the value of the attribute is constructed according to the rules in The class attribute rules and syntax.

For example:

```
<define name="topichead.attlist" combine="interleave">
  <optional>
    <attribute name="class"
        a:defaultValue="+ map/topicref mapgroup-d/topichead "
    />
  </optional>
</define>
```

1.2.4 RELAX NG: Coding requirements for structural modules

A structural vocabulary module defines a new topic or map type as a specialization of a topic or map type.

Required topic and map element attributes

The topic or map element type references the `arch-atts` pattern, which defines the `@DITAArchVersion` attribute in the DITA architecture namespace and sets the attribute to the version of DITA. In addition, the topic or map element type references the `specializations-att` pattern, which pulls in a definition for the `@specializations` attribute.

For example, the following definition references the `arch-atts` and `specializations-att` patterns as part of the definition for the `<concept>` element.

```
<div>
  <a:documentation> LONG NAME: Concept </a:documentation>
  <!-- ... -->
  <define name="concept.attlist" combine="interleave">
    <ref name="concept.attributes"/>
    <ref name="arch-atts"/>
    <ref name="specializations-att"/>
  </define>
  <!-- ... -->
</div>
```

The `@DITAArchVersion` and `@specializations` attributes give processors a reliable way to check the DITA version and the attribute domains that are used.

Controlling nesting in topic types

A structural module that defines a new topic type typically defines an `info-types` style pattern to specify a default for what topic types are permitted to nest. Document-type shells then can control how topics are allowed to nest by redefining the pattern.

The following rules apply when using a pattern to control topic nesting.

Pattern name

The pattern name is the topic element name plus the suffix `-info-types`.

For example, the info-types pattern for the concept topic type is `concept-info-types`.

Pattern value

To set up default topic-nesting rules, set the pattern to the desired topic elements. The default topic nesting is used when a document-type shell does not set up different rules.

For example:

```
<div>
  <a:documentation>INFO TYPES PATTERNS</a:documentation>
  <define name="mytopic-info-types">
    <ref name="subtopictype-01.element"/>
    <ref name="subtopictype-02.element"/>
  </define>
  <!-- ... -->
</div>
```

To disable topic nesting, specify the `<empty>` element.

For example:

```
<define name="learningAssessment-info-types">
  <empty/>
</define>
```

The `info-types` pattern also can be used to refer to common nesting rules across the document-type shell.

For example:

```
<div>
  <a:documentation>INFO TYPES PATTERNS</a:documentation>
  <define name="mytopic-info-types">
    <ref name="info-types"/>
  </define>
  <!-- ... -->
</div>
```

Content model of the root element

In the declaration of the root element of a topic type, the last position in the content model is the *`topictype`*`-info-types` pattern.

For example, the `<concept>` element places the pattern after `<related-links>`:

```
<div>
  <a:documentation>LONG NAME: Concept</a:documentation>
  <define name="concept.content">
    <!-- ... -->
    <optional>
      <ref name="related-links"/>
    </optional>
```

```
    <zeroOrMore>
      <ref name="concept-info-types"/>
    </zeroOrMore>
  </define>
</div>
```

In certain cases, you do not need to use the `info-types` pattern to control topic nesting:

- If a topic type will never permit topic nesting, regardless of context, it can be defined without any pattern or any nested topics.
- If a topic type will allow only specific nesting patterns, such as allowing only other topic types that are defined in the same module, it can nest those topics directly in the same way that other nested elements are defined.

1.2.5 RELAX NG: Coding requirements for element-domain modules

Element-domain modules declare an extension pattern for each element that is extended by the domain. These patterns are used when including the domain module in document-type shells.

Pattern name

The name of the pattern is the abbreviation for the domain, followed by a hyphen ("-"), and the name of the element that is extended.

For example, the name of the pattern for the highlighting domain that extends the `<ph>` element is `hi-d-ph`.

Pattern definition

The pattern consists of a choice group that contains references to element-type name patterns. Each extension of the base element type is referenced.

The following code sample shows the pattern for the elements defined in the highlighting domain:

```
<a:documentation>DOMAIN EXTENSION PATTERNS</a:documentation>

<define name="hi-d-ph">
  <choice>
    <ref name="b.element"/>
    <ref name="i.element"/>
    <ref name="line-through.element"/>
    <ref name="overline.element"/>
    <ref name="sup.element"/>
    <ref name="sub.element"/>
    <ref name="tt.element"/>
    <ref name="u.element"/>
  </choice>
</define>
```

Extension pattern

For each element type that is extended by the element-domain module, the module extends the element-type pattern with a `@combine` value of "choice" that contains a reference to the domain pattern.

For example, the following pattern adds the highlight domain specializations of the `<ph>` element to the content model of the `<ph>` element:

```
<define name="ph" combine="choice">
  <ref name="hi-d-ph"/>
</define>
```

Because the pattern uses a `@combine` value of "choice", the effect is that the domain-provided elements automatically are added to the effective content model of the extended element in any grammar that includes the domain module.

1.2.6 RELAX NG: Coding requirements for attribute-domain modules

An attribute-domain vocabulary module declares a new attribute specialized from either the `@props` or `@base` attribute.

The name of an attribute domain is the name of the attribute plus "Att". For example, for the attribute named `@deliveryTarget`, the attribute-domain name is "deliveryTargetAtt". The attribute-domain name is used to construct pattern names for the domain.

An attribute-domain module consists of a single file, which has three sections:

Specializations attribute contribution

The contribution to the `@specializations` attribute is documented in the module. The value is constructed according to the rules found in The specializations attribute rules and syntax.

The OASIS grammar files use a `<domainsContribution>` element to document the contribution; this element is used to help enable generation of DTD and XSD grammar files. An XML comment or `<a:documentation>` element also can be used.

Attribute declaration pattern

The specialized attribute is declared in a pattern named *domainName*`-d-attribute`. The attribute is defined as optional.

For example, the following code samples shows the the `@audience` specialization of `@props`:

```
<define name="audienceAtt-d-attribute">
  <optional>
    <attribute name="audience" dita:since="2.0">
      <a:documentation>Specifies the audience to which an element applies.</
a:documentation>
    </attribute>
  </optional>
</define>
```

Attribute extension pattern

The attribute extension pattern extends either the `@props` or `@base` attribute-list pattern to include the attribute specialization.

Specializations of @props

The pattern is named `props-attribute-extensions`. The pattern specifies a `@combine` value of "interleave", and the content of the pattern is a reference to the specialized-attribute declaration pattern.

For example:

```
<define name="props-attribute-extensions" combine="interleave">
  <ref name="audienceAtt-d-attribute"/>
</define>
```

Specializations of @base

The pattern is named `base-attribute-extensions`. The pattern specifies a `@combine` value of "interleave", and the content of the pattern is a reference to the specialized-attribute declaration pattern.

For example:

```
<define name="base-attribute-extensions" combine="interleave">
    <ref name="myBaseSpecializationAtt-d-attribute"/>
</define>
```

1.2.7 RELAX NG: Coding requirements for element-configuration modules

An element-configuration module (constraint and expansion) redefines the content model or attribute list for one or more elements.

Implementation of element-configuration modules

Element-configuration modules are implemented by importing the element-configuration modules into a document-type shell in place of the vocabulary module that is redefined. The element-configuration module itself imports the base vocabulary module; within the import, the module redefines the patterns as needed to implement the constraint, expansion, or both.

Constraint modules

For example, a constraint module that modifies the `<topic>` element imports the base module `topicMod.rng`. Within that import, it constrains the `topic.content` pattern:

```
<div>
  <a:documentation>ATTRIBUTES AND CONTENT MODEL OVERRIDES</a:documentation>
  <include href="urn:pubid:oasis:names:tc:dita:rng:topicMod.rng:2.0">
    <define name="topic.content">
      <ref name="title"/>
      <ref name="shortdesc"/>
      <optional>
        <ref name="prolog"/>
      </optional>
      <optional>
        <ref name="body"/>
      </optional>
    </define>
  </include>
</div>
```

Expansion modules

For example, an expansion module that modifies the content model of `<section>` imports the base module `topicMod.rng`. Within that import, it expands the `section.content` pattern:

```
  <a:documentation>CONTENT MODEL AND ATTRIBUTE LIST OVERRIDES</a:documentation>
  <include href="urn:pubid:oasis:names:tc:dita:rng:topicMod.rng:2.0">
    <define name="section.content">
      <optional>
        <ref name="title"/>
      </optional>
      <optional>
        <ref name="sectionDesc"/>
      </optional>
      <zeroOrMore>
        <ref name="section.cnt"/>
      </zeroOrMore>
    </define>
  </include>
</div>
```

Note that the specialized element `<sectionDesc>` must be declared in an element-domain module that also is integrated into the document-type shell.

Combining multiple element-configuration modules

Because the element-configuration module imports the module that it modifies, only one element-configuration module can be used per vocabulary module; otherwise the vocabulary module would be imported multiple times. If multiple element configurations are combined for a single vocabulary module, they need to be implemented in one of the following ways:

Combined into a single element-configuration module

The element configurations can be combined into a single module.

For example, when combining separate constraints for `<section>` and `<shortdesc>`, a single module can be defined as follows:

```
<include href="topicMod.rng">
  <define name="section.content">
    <!-- Constrained model for section -->
  </define>
  <define name="shortdesc.content">
    <!-- Constrained model for shortdesc -->
  </define>
</include>
```

Chaining element-configuration modules

Element-configuration modules can be chained so that each element-configuration module imports another, until the final element-configuration module imports the base vocabulary module.

For example, when combining separate constraints for `<section>`, `<shortdesc>`, and `<li>` from the base vocabulary, the `<section>` constraint can import the `<shortdesc>` constraint, which in turn imports the `<li>` constraint, which finally imports `topicMod.rng`.

Related concepts

Examples: Constraints implemented using RNG

Examples: Expansion implemented using RNG

A Aggregated RFC-2119 statements

This appendix contains all the normative statements from the DITA 2.0 specification. They are aggregated here for convenience in this non-normative appendix.

Index