Stage 3: #15 Loosen specialization rules

Loosen specialization rules to make it possible to target specializations to specific element types.

Champions

- Kristen James Eberlein, Eberlein Consulting LLC
- Chris Nitchie, Individual member

Tracking information: Stage 2

Event	Date	Links
Initial suggestion	30 March 2017	E-mail
Stage 1 proposal accepted	02 May 2017	Minutes, 02 May 2017 GitHub issue
Stage 2 proposal submitted to TC for early feedback	Not applicable	
Stage 2 proposal submitted to reviewers	Not applicable	
Stage 2 proposal submitted to TC	08 June 2017	E-mail
Stage 2 proposal discussed by TC	11 July 2017	Minutes
Stage 2 proposal approved by TC	18 July 2017	Minutes

Tracking information: Stage three

Event	Date	Links
Discussed by TC for early feedback	26 May 2020	E-mail Minutes, 26 May 2020
Stage 3 proposal submitted to reviewers	O5 July 2020	Robert Anderson, Oracle Kristen James Eberlein, Eberlein Consulting LLC Eliot Kimber, Individual member
Discussed by TC for early feedback	03 November 2020 30 March 2021 20 April 2021	E-mail, 01 November 2020 Minutes, 03 November 2020 E-mail, 30 March 2021 Minutes, 30 March 2021 E-mail, 20 April 2021 Minutes, 20 April 2021
Stage 3 proposal submitted to reviewers	14 May 2021	Robert Anderson, Oracle Eliot Kimber, Individual member Chris Nitchie, Individual member
Stage 3 proposal submitted to TC	07 August 2022	E-mail, 07 August 2022

Event	Date	Links
Stage 3 proposal discussed		
Stage 3 proposal approved		

Approved technical requirements

This proposal enables the following:

- Specializations of `@base` or `@props` can be added to the attribute lists of specified elements, instead of being made globally available. Such specializations contribute a token to the `@specializations` attribute.
- Specialized elements can be added to the content models of specified elements, instead of being available in all places where the specialization base is allowed. Such specializations do not contribute a token to the `@specializations` attribute.

Dependencies or interrelated proposals

None.

Modified grammar files

We need to change the "header" information in the document-type shells:

DTD-based document-type shells

Old

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

New

```
<!-- ============================================================= -->
<!--           ELEMENT-TYPE CONFIGURATION INTEGRATION              -->
<!-- ============================================================= -->
```

RNG-based document-type shells

Old

```
<div>
  <a:documentation>CONTENT CONSTRAINT INTEGRATION</a:documentation>
</div>
```

New

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

Modified terminology

The specification should add the following terminology:

- Document-type configuration (This replaces what we previously just called "configuration".
- Element-type configuration

- Element-type configuration module
- Expansion module

Modified specification documentation

This proposal requires extensive changes to the following chapters of the DITA specification:

- "Configuration, specialization, generalization, and constraints"
- "Coding practices for DITA grammar files"

Using a table to compare the DITA 1.3 and proposed DITA 2.0 text would be onerous. Instead, see the accompanying PDF which has revisions flagged.

New specification documentation

This proposal requires the addition of the following terms:

document-type configuration

The process of configuring a document-type shell so that it includes the desired specializations, domains, and element-configuration modules (constraint and expansion)

Note This term will replace "configuration," as it was termed in the DITA 1.3 specification.

element-type configuration
The process of configuring an element type so that it has the desired content model and attribute list

element-type configuration module
A vocabulary module that performs element-type configuration (constraint and expansion)

expansion module
A vocabulary module that expands the content module or attribute list for a specified element or sets of elements

In addition, the specification will have the following topics added:

"Configuration, specialization, generalization, and constraints"

1. Expansion modules
    a. Overview of expansion modules
    b. Expansion module rules
    c. Examples: Expansion implemented using DTD
        1. Example: Adding an element to the `<section>` element using DTD
        2. Example: Adding an attribute to certain table elements using DTDs
        3. Example: Adding an attribute domain to certain elements using DTDs
        4. Example: Aggregating constraint and expansion modules using DTDs
    d. Examples: Expansion implemented using RNG
        1. Example: Adding an element to the `<section>` element using RNG
        2. Example: Adding an attribute to certain table elements using RNG
        3. Example: Adding an attribute domain to certain elements using RNG
        4. Example: Aggregating constraint and expansion modules using RNG

"Coding practices for DITA grammar files"

1. DTD: Coding requirements for expansion modules
2. RNG: Coding requirements for expansion modules

Migration plans for backwards incompatibilities

Not applicable.

8 Configuration, specialization, generalization, constraints, and expansion

The extension facilities of DITA allow document-type shells, vocabulary modules, and element-configuration modules (constraint and expansion) to be combined to create specific DITA document types.

8.1 Overview of DITA extension facilities

DITA provides three extension facilities: Document-type configuration, specialization, and element-type configuration.

Document-type configuration

Document-type configuration enables the definition of DITA document types that include only the vocabulary modules that are required for a given set of documents. There is no need to modify the vocabulary modules. Document-type configurations are implemented as document-type shells.

Specialization

Specialization enables the creation of new element types in a way that preserves the ability to interchange those new element types with conforming DITA applications. Specializations are implemented as vocabulary modules, which are integrated into document-type shells.

Specializations declare the elements and entities that are unique to a specialization. The separation of the vocabulary and its declarations into modules makes it easy to extend existing modules, because new modules can be added without affecting existing document types. It also makes it easy to assemble elements from different sources into a single document-type shell and to reuse specific parts of the specialization hierarchy in more than one document-type shell.

DITA content that uses specializations can be treated as or converted to unspecialized markup through the process of generalization. The information about the original specialized form can be retained.

Element-type configuration

Element-type configuration enables DITA architects to modify the content models and attribute lists for individual elements, without modifying the vocabulary modules in which the elements are defined.

There are two types of element configuration: Constraint and expansion. Both constraint and expansion are implemented as modules that are integrated into document-type shells:

Constraint

Constraint modules enable the restriction of content models and attribute lists for individual elements.

Expansion

Expansion modules enable the expansion of content models and attribute lists for individual elements.

8.2 Document-type configuration

Document-type configuration enables the definition of DITA document types that include only the vocabulary modules that are required for a given set of documents. There is no need to modify the vocabulary modules. Document-type configurations are implemented using document-type shells.

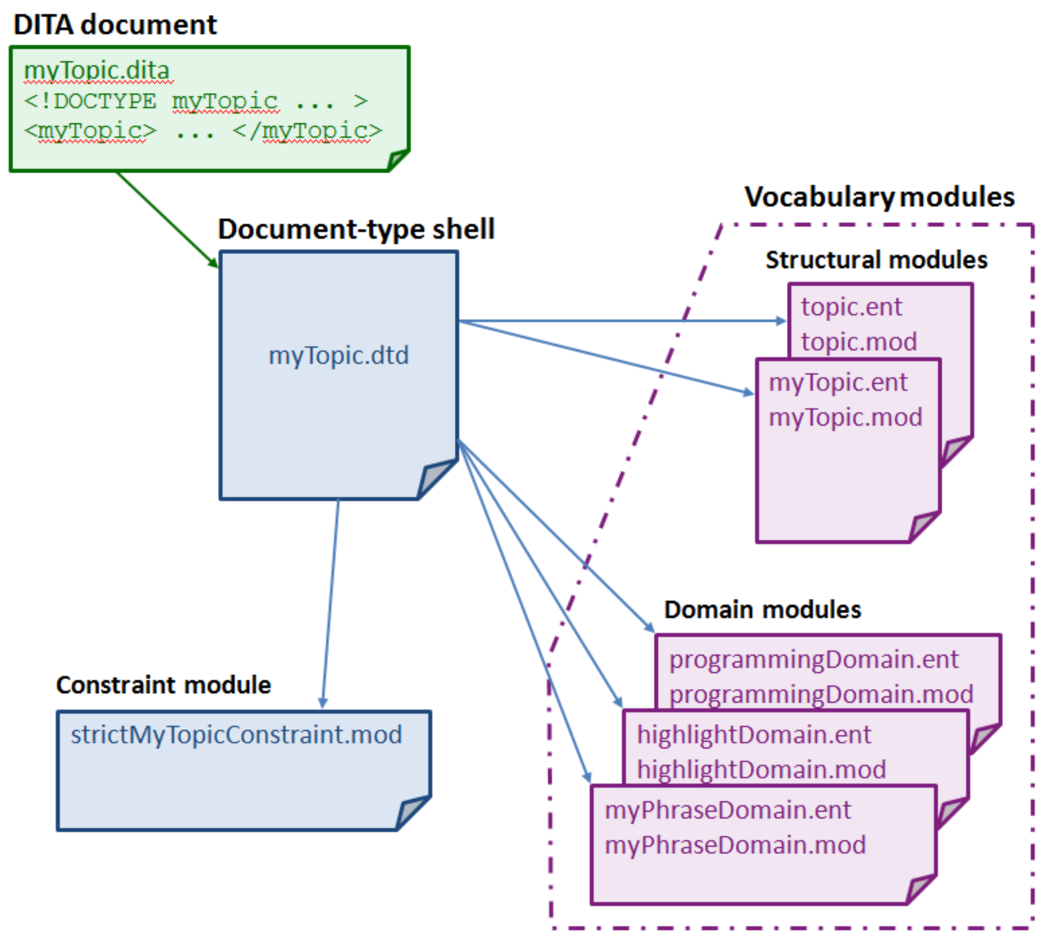
8.2.1 Overview of document-type shells

A document-type shell is an XML grammar file that specifies the elements and attributes that are allowed in a DITA document. The document-type shell integrates structural modules, domain modules, and element-configuration modules. In addition, a document-type shell specifies whether and how topics can nest.

A DITA document either must have an associated document-type definition or all required attributes must be made explicit in the document instances. Most DITA documents have an associated document-type shell. DITA documents that reference a document-type shell can be validated using most standard XML processors. Such validation enables processors to read the XML grammar files and determine default values for the `@specializations` and `@class` attributes.

The following figure illustrates the relationship between a DTD-based DITA document, its document-type shell, the vocabulary modules that it uses, and the element-configuration modules (constraint and expansion) that it integrates. Similar structure applies to DITA documents that use other XML grammars.

Figure 81: Document type shell



Comment by Kristen J Eberlein on 28 March 2021

The illustration needs to be updated to include expansion modules.

The DITA specification contains a starter set of document-type shells. These document-type shells are commented and can be used as templates for creating custom document-type shells.

While the OASIS-provided document-type shells can be used without any modification, creating custom document-type shells is a best practice. If the document-type shells need to be modified in the future, for example, to include a specialization or integrate an element-configuration module (constraint or expansion), the existing DITA documents will not need to be modified to reference a new document-type shell.

8.2.2 Rules for document-type shells

This topic collects the rules that concern DITA document-type shells.

XML grammars

053 (410)

While the DITA specification only defines coding requirements for DTD and RELAX NG, conforming DITA documents **MAY** use other document-type constraint languages, such as Schematron.

Defining element or attribute types

054 (410)

With two exceptions, a document-type shell **MUST NOT** directly define element or attribute types; it only includes vocabulary and element-configuration modules (constraint and expansion). The exceptions to this rule are the following:

- The ditabase document-type shell directly defines the `<dita>` element.
- RNG-based document-type shells directly specify values for the `@specializations` attribute. These values reflect the details of the attribute domains that are integrated by the document-type shell.

Document-type shells not provided by OASIS

055 (410)

Document-type shells that are not provided by OASIS **MUST** have a unique public identifier, if public identifiers are used.

056 (410)

Document-type shells that are not provided by OASIS **MUST NOT** indicate OASIS as the owner. The public identifier or URN for such document-type shells **SHOULD** reflect the owner or creator of the document-type shell.

For example, if `example.com` creates a copy of the document-type shell for topic, an appropriate public identifier would be "-//EXAMPLE//DTD DITA Topic//EN", where "EXAMPLE" is the component of the public identifier that identifies the owner. An appropriate URN would be "urn:example.com:names:dita:rng:topic.rng".

8.2.3 Equivalence of document-type shells

Two distinct DITA document types that are taken from different tools or environments might be functionally equivalent.

A DITA document type is defined by the following:

- The set of vocabulary and element-configuration modules (constraint and expansion) that are integrated by the document type shell
- The values of the `@class` attributes of all the elements in the document
- Rules for topic nesting

Two document-type shells define the same DITA document type if they integrate identical vocabulary modules, element-configuration modules (constraint and expansion), and rules for topic nesting. For example, a document type shell that is an unmodified copy of the OASIS-provided document-type shell for topic defines the same DITA document type as the original document-type shell. However, the new document-type shell has the following differences:

- It is a distinct file that is stored in a different location.
- It has a distinct system identifier.
- If it has a public identifier, the public identifier is unique.

Note The public or system identifier that is associated with a given document-type shell is not necessarily distinguishing. Two different people or groups might use the same modules and constraints to assemble equivalent document type shells, while giving them different names or public identifiers.

8.2.4 Conformance of document-type shells

DITA documents typically are governed by a conforming DITA document-type shell. However, the conformance of a DITA document is a function of the document instance, not its governing grammar. Conforming DITA documents are not required to use a conforming document-type shell.

Conforming DITA documents are not required to have any governing document type declaration or schema. There might be compelling or practical reasons to use non-conforming document-type shells. For example, a document might use a document-type shell that does not conform to the DITA requirements for shells in order to meet the needs of a specific application or tool. Such a non-conforming document-type shell still might enable the creation of conforming DITA content.

8.3 Specialization

The specialization feature of DITA allows for the creation of new element types and attributes that are explicitly and formally derived from existing types. This facilitates interchange of conforming DITA content and ensures a minimum level of common processing for all DITA content. It also allows specialization-aware processors to add specialization-specific processing to existing base processing.

8.3.1 Overview of specialization

Specialization allows information architects to define new kinds of information (new structural types or new domains of information), while reusing as much of existing design and code as possible, and minimizing or eliminating the costs of interchange, migration, and maintenance.

Specialization modules enable information architects to create new element types and attributes. These new element types and attributes are derived from existing element types and attributes.

In traditional XML applications, all semantics for a given element instance are bound to the element type, such as `<para>` for a paragraph or `<title>` for a title. The XML specification provides no built-in mechanism for relating two element types to say "element type B is a subtype of element type A".

In contrast, the DITA specialization mechanism provides a standard mechanism for declaring that an element type or attribute is derived from an ancestor type. This means that a specialized type inherits the semantics and default processing behavior from its ancestor type. Additional processing behavior optionally can be associated with the specialized descendant type.

For example, the `<section>` element type is part of the DITA base. It represents an organizational division in a topic. Within the task information type (itself a specialization of `<topic>`), the `<section>` element type is further specialized to other element types (such as `<prereq>` and `<context>`) that provide more precise semantics about the type of organizational division that they represent. The specialized element types inherit both semantic meaning and default processing from the ancestor elements.

There are two types of DITA specializations:

Structural specialization

Structural specializations are developed from either topic or map types. Structural specializations enable information architect to add new document types to DITA. The structures defined in the new document types either directly use or inherit from elements found in other document types. For example; concept, task, and reference are specialized from topic, whereas bookmap is specialized from map.

Domain specialization

Domain specializations are developed from elements defined with topic or map, or from the `@props` or `@base` attributes. They define markup for a specific information domain or subject area. Domain specializations can be added to document-type shells.

Each type of specialization module represents an “is a” hierarchy, in object-oriented terms, with each structural type or domain being a subclass of its parent. For example, a specialization of task is still a task, and a specialization of the user interface domain is still part of the user interface domain. A given domain can be used with any map or topic type. In addition, specific structural types might require the use of specific domains.

Use specialization when you need a new structural type or domain. Specialization is appropriate in the following circumstances:

- You need to create markup to represent new semantics (meaningful categories of information). This might enable you to have increased consistency or descriptiveness in your content model.
- You have specific needs for output processing and formatting that cannot be addressed using the current content model.

Do not use specialization to simply eliminate element types from specific content models. Use constraint modules to restrict content models and attribute lists without changing semantics.

8.3.2 Modularization

Modularization is at the core of DITA design and implementation. It enables reuse and extension of the DITA specialization hierarchy.

The DITA XML grammar files are a set of module files that declare the markup and entities that are required for each specialization. The document-type shell then integrates the modules that are needed for a particular authoring and publishing context.

Because all the pieces are modular, the task of developing a new information type or domain is easy. An information architect can start with existing base types (topic or map)—or with an existing specialization if it comes close to matching their business requirements—and only develop an extension that adds the extra semantics or functionality that is required. A specialization reuses elements from ancestor modules, but it only needs to declare the elements and attributes that are unique to the specialization. This saves considerable time and effort; it also reduces error, enforces consistency, and makes interoperability possible.

Because all the pieces are modular, it is easy to reuse different modules in different contexts.

For example, a company that produces machines can use the hazard statement domain, while a company that produces software can use the software, user interface, and programming domains. A company that produces health information for consumers can avoid using the standard domains; instead, it develops a new domain that contains the elements necessary for capturing and tracking the comments made by medical professionals who review information for accuracy and completeness.

Because all the pieces are modular, new modules can be created and put into use without affecting existing document-type shells.

For example, a marketing division of a company can develop a new specialization for message campaigns and have their content authors begin using that specialization, without affecting any of the other information types that they have in place.

8.3.3 Vocabulary modules

A DITA element type or attribute is declared in exactly one vocabulary module.

The following terminology is used to refer to DITA vocabulary modules:

structural module
: A vocabulary module that defines a top-level map or topic type.

element domain module
: A vocabulary module that defines one or more specialized element types that can be integrated into maps or topics.

attribute domain module
: A vocabulary module that defines exactly one specialization of either the `@base` or `@props` attribute.

For structural types, the module name is typically the same as the root element. For example, "task" is the name of the structural vocabulary module whose root element is `<task>`.

For element domain modules, the module name is typically a name that reflects the subject domain to which the domain applies, such as "highlight" or "software". Domain modules often have an associated short name, such as "hi-d" for the highlighting domain or "sw-d" for the software domain.

The name (or short name) of an element domain module is used to identify the module in `@class` attribute values. While module names need not be globally unique, module names must be unique within the scope of a given specialization hierarchy. The short name must be a valid XML name token.

057 (410) Structural modules based on topic **MAY** define additional topic types that are then allowed to occur as subordinate topics within the top-level topic. However, such subordinate topic types **MAY NOT** be used as the root elements of conforming DITA documents.

For example, a top-level topic type might require the use of subordinate topic types that would only ever be meaningful in the context of their containing type and thus would never be candidates for standalone authoring or aggregation using maps. In that case, the subordinate topic type can be declared in the module for the top-level topic type that uses it. However, in most cases, potential subordinate topics are best defined in their own vocabulary modules.

058 (410) Domain elements intended for use in topics **MUST** ultimately be specialized from elements that are defined in the topic module. Domain elements intended for use in maps **MUST** ultimately be specialized from elements defined by or used in the map module. Maps share some element types with topics but no map-specific elements can be used within topics.

Structural modules also can define specializations of, or reuse elements from, domain or other structural modules. When this happens, the structural module becomes dependent.

8.3.4 Specialization rules for element types

There are certain rules that apply to element type specializations.

A specialized element type has the following characteristics:

Comment by Kristen J Eberlein on 07 August 2022

Robert Anderson and I both think that the 2nd bullet point needs expansion, to cover the effects of expansion modules.

- A properly-formed `@class` attribute that specifies the specialization hierarchy of the element
- A content model that is the same or less inclusive than that of the element from which it was specialized
- A set of attributes that are the same or a subset of those of the element from which it was specialized, except for specializations of `@base` or `@props`
- Values or value ranges of attributes that are the same or a subset of those of the element from which it was specialized

DITA elements are never in a namespace. Only the `@DITAArchVersion` attribute is in a DITA-defined namespace. All other attributes, except for those defined by the XML standard, are in no namespace.

This limitation is imposed by the details of the `@class` attribute syntax, which makes it impractical to have namespace-qualified names for either vocabulary modules or individual element types or attributes. Elements included as descendants of the DITA `<foreign>` element type can be in any namespace.

Note Domain modules that are intended for wide use should define element type names that are unlikely to conflict with names used in other domains, for example, by using a domain-specific prefix on all names.

8.3.5 Specialization rules for attributes

There are certain rules that apply to attribute specializations.

A specialized attribute has the following characteristics:

- It is specialized from `@props` or `@base`.
- It can be integrated into a document-type shell either globally, which makes it available on all elements, or it can be assigned to specific elements by using an expansion module.
- It does not have values or value ranges that are more extensive than those of the attribute from which it was specialized.
- Its values must be alphanumeric space-delimited values. In generalized form, the values must conform to the rules for attribute generalization.

8.3.6 @class attribute rules and syntax

The specialization hierarchy of each DITA element is declared as the value of the `@class` attribute. The `@class` attribute provides a mapping from the current name of the element to its more general equivalents, but it also can provide a mapping from the current name to more specialized equivalents. All specialization-aware processing can be defined in terms of `@class` attribute values.

The `@class` attribute tells a processor what general classes of elements the current element belongs to. DITA scopes elements by module type (for example topic type, domain type, or map type) instead of document type, which lets document type developers combine multiple module types in a single document without complicating transformation logic.

The sequence of values in the `@class` attribute is important because it tells processors which value is the most general and which is most specific. This sequence is what enables both specialization aware processing and generalization.

Syntax

Values for the `@class` attribute have the following syntax requirements:

- An initial "-" or "+" character followed by one or more spaces. Use "-" for element types that are defined in structural vocabulary modules, and use "+" for element types that are defined in domain modules.
- A sequence of one or more tokens of the form `"`*`modulename/typename`*`",` with each token separated by one or more spaces, where *modulename* is the short name of the vocabulary module and *typename* is the element type name. Tokens are ordered left to right from most general to most specialized.

  These tokens provide a mapping for every structural type or domain in the ancestry of the specialized element. The specialization hierarchy for a given element type must reflect any intermediate modules between the base type and the specialization type, even those in which no element renaming occurs.
- At least one trailing space character (" "). The trailing space ensures that string matches on the tokens can always include a leading and trailing space in order to reliably match full tokens.

Rules

059 (410)	Every DITA element (except the `<dita>` element that is used as the root of a ditabase document) **MUST** declare a `@class` attribute.
060 (410)	When the `@class` attribute is declared in an XML grammar, it **MUST** be declared with a default value. In order to support generalization round-tripping (generalizing specialized content into a generic form and then returning it to the specialized form) the default value **MUST NOT** be fixed. This allows a generalization process to overwrite the default values that are defined by a general document type with specialized values taken from the document being generalized.
061 (410)	A vocabulary module **MUST NOT** change the `@class` attribute for elements that it does not specialize, but simply reuses by reference from more generic levels.
062 (410)	Authors **SHOULD NOT** modify the `@class` attribute.

Example: DTD declaration for @class attribute for the <step> element

The following code sample lists the DTD declaration for the `@class` attribute for the `<step>` element:

```
<!ATTLIST step          class  CDATA "- topic/li task/step ">
```

This indicates that the `<step>` element is specialized from the `<li>` element in a generic topic. It also indicates explicitly that the `<step>` element is available in a task topic; this enables round-trip migration between upper level and lower level types without the loss of information.

Example: Element with @class attribute made explicit

The following code sample shows the value of the `@class` attribute for the `<wintitle>` element:

```
<wintitle class="+ topic/keyword ui-d/wintitle ">A specialized keyword</wintitle>
```

The `@class` attribute and its value is generally not surfaced in authored DITA topics, although it might be made explicit as part of a processing operation.

Example: @class attribute with intermediate value

The following code sample shows the value of a `@class` attribute for an element in the guiTask module, which is specialized from `<task>`. The element is specialized from `<keyword>` in the base topic vocabulary, rather than from an element in the task module:

```
<windowName class="- topic/keyword task/keyword guiTask/windowname ">...</windowName>
```

The intermediate values are necessary so that generalizing and specializing transformations can map the values simply and accurately. For example, if `task/keyword` was missing as a value, and a user decided to generalize this guiTask up to a task topic, then the transformation would have to guess whether to map to keyword (appropriate if task is more general than guiTask, which it is) or leave it as windowName (appropriate if task were more specialized, which it isn't). By always providing mappings for more general values, processors can then apply the simple rule that missing mappings must by default be to more specialized values than the one we are generalizing to, which means the last value in the list is appropriate. For example, when generalizing `<guitask>` to `<task>`, if a `<p>` element has no target value for `<task>`, we can safely assume that `<p>` does not specialize from `<task>` and does not need to be generalized.

8.3.7 @specializations attribute rules and syntax

The `@specializations` attribute enables processors to determine what attribute specializations are available in a document. The attribute is declared on the root element for each topic or map type. Each attribute domain defines a token to declare the extension; the effective value of the `@specializations` attribute is composed of these tokens.

Syntax and rules

The `@props` and `@base` attributes are the only two core attributes available for specialization.

063 (410) Each specialization of the `@props` and `@base` attributes **MUST** provide a token for use by the `@specializations` attribute.

The `@specializations` token for an attribute specialization begins with either `@props` or `@base` followed by a slash, followed by the name of the new attribute:

```
'@', props-or-base, ('/', attname)+
```

For example:

- If `@props` is specialized to create `@myNewProp`, this results in the following token: `@props/myNewProp`
- If `@base` is specialized to create `@myFirstBase`, this results in the following token: `@base/myFirstBase`
- If that specialized attribute `@myFirstBase` is further specialized to create `@mySecondBase`, this results in the following token: `@base/myFirstBase/mySecondBase`

Example: @specializations attribute for a task with multiple domains

In this example, a document-type shell integrates the task structural module and the following domain modules:

Domain	Domain short name
User interface	ui-d
Software	sw-d
`@deliveryTarget` attribute	deliveryTarget
`@platform` attribute	platform
`@product` attribute	product

The value of the `@specializations` attribute includes one value from each attribute module; the effective value is the following:

```
specializations="@props/deliveryTarget @props/platform @props/product"
```

If the document-type shell also used a specialization of the `@platform` attribute that describes the hardware platform, the new `@hardwarePlatform` attribute domain would add an additional value to the `@specializations` attribute:

```
specializations="@props/deliveryTarget @props/platform @props/platform/hardwarePlatform
@props/product"
```

Note that the value for the `@specializations` attribute is not authored. Instead, the value is defaulted based on the modules that are included in the document type shell.

8.3.8 Specializing to include non-DITA content

You can extend DITA to incorporate standard vocabularies for non-textual content, such as MathML and SVG, as markup within DITA documents. This is done by specializing the `<foreign>` or `<unknown>` elements.

There are three methods of incorporating foreign content into DITA.

- A domain specialization of the `<foreign>` or `<unknown>` element. This is the usual implementation.
- A structural specialization using the `<foreign>` or `<unknown>` element. This affords more control over the content.
- Directly embedding the non-DITA content within `<foreign>` or `<unknown>` elements. If the non-DITA content has interoperability or vocabulary naming issues such as those that are addressed by specialization in DITA, they must be addressed by means that are appropriate to the non-DITA content.

Do not use `<foreign>` or `<unknown>` elements to include textual content or metadata in DITA documents, except where such content acts as an example or display, rather than as the primary content of a topic.

Example: Creating an element domain specialization for SVG

The following code sample, which is from the `svgDomain.ent` file, shows the domain declaration for the SVG domain.

```
<!-- ============================================================= -->
<!--                    SVG DOMAIN ENTITIES                         -->
<!-- ============================================================= -->

<!-- SVG elements must be prefixed, otherwise they conflict with
     existing DITA elements (e.g., <desc> and <title>.
  -->
<!ENTITY % NS.prefixed "INCLUDE" >
<!ENTITY % SVG.prefix "svg" >

<!ENTITY % svg-d-foreign
   "svg-container
   "
>
```

Note that the SVG-specific `%SVG.prefix;` parameter entity is declared. This establishes the default namespace prefix to be used for the SVG content embedded with this domain. The namespace can be overridden in a document-type shell by declaring the parameter entity before the reference to the `svgDomain.ent` file. Other foreign domains might need similar entities when required by the new vocabulary.

For more information, see the `svgDomain.mod` file that is shipped with the DITA Technical Content edition. For an example of including the SVG domain in a document-type shell, see `task.dtd`.

8.3.9 Sharing elements across specializations

Specialization enables easy reuse of elements from ancestor specializations. However, it is also possible to reuse elements from non-ancestor specializations, as long as the dependency is properly declared in order to prevent invalid generalization or conref processing.

A structural specialization can incorporate elements from unrelated domains or other structural specializations by referencing them in the content model of a specialized element. The elements included in this manner must be specialized from ancestor content that is valid in the new context. If the reusing and reused specializations share common ancestry, the reused elements must be valid in the reusing context at every level they share in common.

Although a well-designed structural specialization hierarchy with controlled use of domains is still the primary means of sharing and reusing elements in DITA, the ability to also share elements declared elsewhere in the hierarchy allows for situations where relevant markup comes from multiple sources and would otherwise be developed redundantly.

Example: A specialization of <concept> reuses an element from the task module

A specialized concept topic could declare a specialized `<process>` section that contains the `<steps>` element that is defined in the task module. This is possible because of the following factors:

- The `<steps>` element is specialized from `<ol>`.
- The `<process>` element is specialized from `<section>`, and the content model of `<section>` includes `<ol>`.

The `<steps>` element in `<process>` always can be generalized back to `<ol>` in `<section>`.

Example: A specialization of <reference> reuses an element from the programming domain

A specialized reference topic could declare a specialized list (`<apilist>`) in which each `<apilistitem>` contains an `<apiname>` element that is borrowed from the programming domain.

8.4 Generalization

Generalization is the process of reversing a specialization. It converts specialized elements or attributes into the original types from which they were derived.

8.4.1 Overview of generalization

Specialized content can be generalized to any ancestor type. The generalization process can preserve information about the former level of specialization to allow round-tripping between specialized and unspecialized forms of the same content.

All DITA documents contain a mix of markup from at least one structural type and zero or more domains. When generalizing the document, any individual structural type or domain can be left as-is, or it can be generalized to any of its ancestors. If the document will be edited or processed in generalized form, it might be necessary to have a document-type shell that includes all non-generalized modules from the original document-type shell.

Generalization serves several purposes:

- It can be used to migrate content. For example, if a specialization is unsuccessful or is no longer needed, the content can be generalized back to a less specialized form.
- It can be used for temporary round-tripping. For example, if content is shared with a process that is not specialization aware, it can be temporarily generalized for that process and then returned to specialized form.
- It can allow reuse of specialized content in an environment that does not support the specialization. Similar to round-tripping, content can be generalized for sharing, without the need to re-specialize.

When generalizing for migration, the `@class` attribute and `@specializations` attribute need to be absent from the generalized instance document, so that the default values in the document-type shell are used.

064 (410) When generalizing for round-tripping, the `@class` attribute and `@specializations` attribute **SHOULD** retain the original specialized values in the generalized instance document.

Note that when using constraints, a document instance can always be converted from a constrained document type to an unconstrained document type merely by switching the binding of the document instance to the less restricted document type shell. No renaming of elements is needed to remove constraints.

However, a document whose document-type shell uses expansion modules might not be interchangeable without first generalizing the element and attribute types that were introduced by the expansion modules.

8.4.2 Element generalization

Elements are generalized by examining the `@class` attribute. When a generalization process detects that an element belongs to one of the modules that is being generalized, the element is renamed to a more general form.

For example, the `<step>` element has a `@class` attribute value of `"- topic/li task/step "`. If the task module is generalized, the `<step>` element is renamed to its more general form from the topic module: `<li>`.

For specific concerns when generalizing structural types with dependencies on non-ancestor modules, see 8.4.5 Generalization with cross-specialization dependencies (190).

While the tag name of a given element is normally the same as the type name of the last token in the `@class` value, this is not required. For example, if a generalization process has already run on the element, the `@class` attribute could contain tokens from two or more modules based on the original specialization. In that case, the element name could already match the first token or an intermediate token in the `@class` attribute. A second generalization process could end up renaming the element again or could leave it alone, depending on the target module or document type.

8.4.3 Processor expectations when generalizing elements

Generalization processors convert elements from one or more modules into their less specialized form. The list of modules can be supplied to a generalization processor, or it can be inferred based on knowledge of a target document-type shell.

The person or application initiating a generalization process can supply the source and target modules for each generalization, for example, "generalize from reference to topic". Multiple target modules can be specified, for example, "generalize from reference to topic and from user-interface domain to topic". When the source and target modules are not supplied, the generalization process is assumed to be from all structural types to the base (topic or map), and no generalization is performed for domains.

The person or application initiating a generalization process also can supply the target document-type shell. When the target document-type shell is not supplied, the generalized document will not contain a reference to a document-type shell.

065 (410)

A generalization processor **SHOULD** be able to handle cases where it is given:

- Only source modules for generalization (in which case the designated source types are generalized to topic or map)
- Only target modules for generalization (in which case all descendants of each target are generalized to that target)
- Both (in which case only the specified descendants of each target are generalized to that target)

For each structural type instance, the generalization processor checks whether the structural type instance is a candidate for generalization, or whether it has domains that are candidates for generalization. It is important to be selective about which structural type instances to process; if the process simply generalizes every element based on its `@class` attribute values, an instruction to generalize "reference" to "topic" could leave a specialization of reference with an invalid content model, since any elements it reuses from "reference" would have been renamed to topic-level equivalents.

The `@class` attribute for the root element of the structural type is checked before generalizing structural types:

	Source module unspecified	**Source module specified**
Target module unspecified	Generalize this structural type to its base ancestor	Check whether the root element of the topic type matches a specified source module; generalize to its base ancestor if it does, otherwise ignore the structural type instance unless it has domains to generalize.
Target module specified	Check whether the `@class` attribute contains the target module. If it does contain the target, rename the element to the value associated with the target module. Otherwise, ignore the element.	It is an error if the root element matches a specified source but its `@class` attribute does not contain the target. If the root element matches a specified source module and its `@class` attribute does contain the target module, generalize to the target module. Otherwise, ignore the structural type instance unless it has domains to generalize.

For each element in a candidate structural type instance:

	Source module unspecified	**Source module specified**
Target module unspecified	If the `@class` attribute starts with "-" (part of a structural type), rename the element to its base ancestor equivalent. Otherwise ignore it.	Check whether the last value of the `@class` attribute matches a specified source; generalize to its base ancestor if it does, otherwise ignore the element.
Target module specified	Check whether the `@class` attribute contains the target module; rename the element to the value associated with the target module if it does contain the target, otherwise ignore the element.	It is an error if the last value in the `@class` attribute matches a specified source but the previous values do not include the target. If the last value in the `@class` attribute matches a specified source module and the previous values do include the target module, rename the element to the value associated with the target module. Otherwise, ignore the element.

066 (411) When renaming elements during round-trip generalization, the generalization processor **SHOULD** preserve the values of all attributes. When renaming elements during one-way or migration generalization, the process **SHOULD** preserve the values of all attributes except the `@class` attribute, which is supplied by the target document type.

8.4.4 Attribute generalization

DITA provides a syntax to generalize attributes that have been specialized from the `@props` or `@base` attribute.

067 (411) Specialization-aware processors **MUST** process both the specialized and generalized forms of an attribute as equivalent in their values.

When a specialized attribute is generalized to an ancestor attribute, the value of the ancestor attribute consists of the name of the specialized attribute followed by its specialized value in parentheses.

For example, if `@jobrole` is an attribute specialized from `@person`, which in turn is specialized from `@props`:

- `jobrole="programmer"` can be generalized to `person="jobrole(programmer)"` or to `props="jobrole(programmer)"`
- `props="jobrole(programmer)"` can be respecialized to `person="jobrole(programmer)"` or to `jobrole="programmer"`

In this example, processors performing generalization and respecialization can use the `@specializations` attribute to determine the ancestry of the specialized `@jobrole` attribute, and therefore the validity of the specialized `@person` attribute as an intermediate target for generalization.

If more than one attribute is generalized, the value of each is separately represented in this way in the value of the ancestor attribute.

Generalized attributes are typically not expected to be authored or edited directly. They are used by processors to preserve the values of the specialized attributes during the time or in the circumstances in which the document is in a generalized form.

068 (411) A single element **MUST NOT** contain both generalized and specialized values for the same attribute.

For example, the following `<p>` element provides two values for the `@jobrole` attribute, one in a generalized syntax and the other in a specialized syntax:

```
<p person="jobrole(programmer)" jobrole="admin">
    <!-- ... -->
</p>
```

This is an error condition, since it means the document has been only partially generalized, or that the document has been generalized and then edited using a specialized document type.

8.4.5 Generalization with cross-specialization dependencies

Dependencies across specializations limit generalization targets to those that either preserve the dependency or eliminate them. Some generalization targets will not be valid and need to be detected before generalization occurs.

When a structural specialization has a dependency on a domain specialization, then the domain cannot be generalized without also generalizing the reusing structural specialization.

For example, a structural specialization `<codeConcept>` might incorporate and require the `<codeblock>` element from the programming domain. A generalization process that turns programming domain elements back into topic elements would convert `<codeblock>` to `<pre>`, making a document that uses `<codeConcept>` invalid. However, codeConcept<> could be generalized to concept or topic, without generalizing programming domain elements, as long as the target document type includes the programming domain.

When a structural specialization has a dependency on another structural specialization, then both must be generalized together to a common ancestor.

For example, if the task elements in checklist were generalized without also generalizing checklist elements, then the checklist content models that referenced task elements would be broken. And if the checklist elements were generalized to topic without also generalizing the task elements, then the task elements would be out of place, since they cannot be validly present in topic. However, checklist and task can be generalized together to any ancestor they have in common: in this case topic.

069 (411) When possible, generalizing processes **SHOULD** detect invalid generalization target combinations and report them as errors.

8.5 Constraints

Constraint modules define additional constraints for vocabulary modules in order to restrict content models or attribute lists for specific element types, remove certain extension elements from an integrated domain module, or replace base element types with domain-provided, extension element types.

8.5.1 Overview of constraints

Constraint modules enable information architects to restrict the content models or attributes of DITA elements. A constraint is a simplification of an XML grammar such that any instance that conforms to the constrained grammar also will conform to the original grammar.

A constraint module can perform the following functions:

Restrict the content model for an element

Constraint modules can modify content models by removing optional elements, making optional elements required, or requiring unordered elements to occur in a specific sequence. Constraint modules cannot make required elements optional or change the order of element occurrence for ordered elements.

For example, a constraint for `<topic>` can require `<shortdesc>`, can remove `<abstract>`, and can require that the first child of `<body>` be `<p>`. A constraint cannot allow `<shortdesc>` to follow `<prolog>`, because the content model for `<topic>` requires that `<shortdesc>` precedes `<prolog>`.

Restrict the attributes that are available on an element

Constraint modules can restrict the attributes that are available on an element. They also can limit the set of permissible values for an attribute.

For example, a constraint for `<note>` can limit the set of allowed values for the `@type` attribute to "note" and "tip". It also can omit the `@othertype` attribute, since it is needed only when the value of the `@type` attribute is "other".

Restrict the elements that are available in a domain

Constraint modules can restrict the set of extension elements that are provided in a domain. They also can restrict the content models for the extension elements.

For example, a constraint on the programming domain can reduce the list of included extension elements to `<codeph>` and `<codeblock>`.

Note For DITA implementations that use RNG-based grammar file, restricting the set of extension elements that are provided in a domain can be handled simply by document-type configuration.

Replace base elements with domain extensions

Constraint modules can replace base element types with the domain-provided extension elements.

For example, a constraint module can replace the `<ph>` element with the domain-provided elements, making `<ph>` unavailable.

8.5.2 Constraint rules

There are certain rules that apply to the design and implementation of constraints.

Content model

The content model for a constrained element must be at least as restrictive as the unconstrained content model for the element.

Domain constraints

When a domain module is integrated into a document-type shell, the base domain element can be omitted from the domain extension group or parameter entity. In such a case, there is no separate constraint declaration, because the content model is configured directly in the document-type shell.

A domain module can be constrained by only one constraint module. This means that all restrictions for the extension elements that are defined in the domain must be contained within that one constraint module.

Structural constraints

Each constraint module can constrain elements from only one vocabulary module. For example, a single constraint module that constrains `<refsyn>` from `reference.mod` and constrains `<context>` from `task.mod` is not allowed. This rule maintains granularity of reuse at the module level.

Constraint modules that restrict different elements from within the same vocabulary module can be combined with one another. Such combinations of constraints on a single vocabulary module have no meaningful order or precedence.

Aggregation of constraint modules

The content model of an element can be modified by either of the following element-configuration modules:

- Constraint module
- Expansion module

The content model of an element only can be modified by a single element-type configuration module. If multiple constraints or extensions need to be applied to a single element, the element configurations must be combined into a single module that reflects all the constraints and expansions that were defined in the original separate modules.

8.5.3 Constraints, processing, and interoperability

Because constraints can make optional elements required, documents that use the same vocabulary modules might have incompatible constraints. Thus the use of constraints can affect the ability for content from one topic or map to be used in another topic or map.

A constraint does not change basic or inherited element semantics. The constrained instances remain valid instances of the unconstrained element type, and the element type retains the same semantics and `@class` attribute declaration. Thus, a constraint never creates a new case to which content processing might need to react.

For example, a document type constrained to require the `<shortdesc>` element allows a subset of the possible instances of the unconstrained document type with an optional `<shortdesc>` element. Thus, the content processing for topic still works when `<topic>` is constrained to require a short description.

For example, an unconstrained task is compatible with an unconstrained topic, because the `<task>` element can be generalized to `<topic>`. However, if the topic is constrained to require the `<shortdesc>` element, a document type with an unconstrained task is not compatible with the constrained document type, because some instances of the task might not have a `<shortdesc>` element. However, if the task document type also has been constrained to require the `<shortdesc>` element, it is compatible with the constrained topic document type.

8.5.4 Examples: Constraints implemented using DTDs

This section of the specification contains examples of constraints implemented using DTD.

8.5.4.1 Example: Redefine the content model for the <topic> element using DTD

In this scenario, the DITA architect for Acme, Incorporated wants to redefine the content model for the topic document type. They want to omit the `<abstract>` element and make the `<shortdesc>` element required; they also want to omit the `<related-links>` element and disallow topic nesting.

1. The DITA architect creates a constraint module: `acme-TopicConstraint.mod`.
2. They add the following content to `acme-TopicConstraint.mod`:

```
<!-- ============================================================= -->
<!--                    CONSTRAINED TOPIC ENTITIES                 -->
<!-- ============================================================= -->

<!-- Declares the entities referenced in the constrained content  -->
<!-- model.                                                       -->

<!ENTITY % title            "title">
<!ENTITY % shortdesc        "shortdesc">
<!ENTITY % prolog           "prolog">
<!ENTITY % body             "body">

<!-- Defines the constrained content model for <topic>.            -->

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

3. They add the constraint module to the `catalog.xml` file.
4. They then integrate the constraint module into the document-type shell for topic by adding the following section above the "TOPIC ELEMENT INTEGRATION" comment:

```
<!-- ============================================================= -->
<!--             ELEMENT-TYPE CONFIGURATION INTEGRATION            -->
<!-- ============================================================= -->

<!ENTITY % topic-constraints-c-def
  PUBLIC "-//ACME//ELEMENTS DITA Topic Constraint//EN"
  "acme-TopicConstraint.mod">
%topic-constraints-c-def;
```

5. After checking the test topic to ensure that the content model is modified as expected, the work is done.

8.5.4.2 Example: Constrain attributes for the <section> element using DTD

In this scenario, a DITA architect wants to redefine the attributes for the `<section>` element. They want to make the `@id` attribute required.

1. The DITA architect creates a constraint module: `idRequiredSectionContraint.mod`.
2. They add the following content to `idRequiredSectionContraint.mod`:

```
<!-- Declares the entities referenced in the constrained content  -->
<!-- model.                                                       -->

<!ENTITY % localization-atts
              "translate
                          (no |
                           yes |
                           -dita-use-conref-target)
```

```
                                  #IMPLIED
               xml:lang
                          CDATA
                                  #IMPLIED
               dir
                          (lro |
                           ltr |
                           rlo |
                           rtl |
                           -dita-use-conref-target)
                                  #IMPLIED"
>
<!ENTITY % filter-atts
              "props
                          CDATA
                                  #IMPLIED
               %props-attribute-extensions;"
>
<!ENTITY % select-atts
              "%filter-atts;
               base
                          CDATA
                                  #IMPLIED
               %base-attribute-extensions;
               importance
                          (default |
                           deprecated |
                           high |
                           low |
                           normal |
                           obsolete |
                           optional |
                           recommended |
                           required |
                           urgent |
                           -dita-use-conref-target)
                                  #IMPLIED
               rev
                          CDATA
                                  #IMPLIED
               status
                          (changed |
                           deleted |
                           new |
                           unchanged |
                           -dita-use-conref-target)
                                  #IMPLIED"
>
<!ENTITY % conref-atts
              "conref
                          CDATA
                                  #IMPLIED
               conrefend
                          CDATA
                                  #IMPLIED
               conaction
                          (mark |
                           pushafter |
                           pushbefore |
                           pushreplace |
                           -dita-use-conref-target)
                                  #IMPLIED
               conkeyref
                          CDATA
                                  #IMPLIED"
>
<!-- Redefines the attributes available on section  -->

<!ENTITY % section.attributes
              "id
                          ID
                                  #REQUIRED
               %conref-atts;
               %select-atts;
```

```
                %localization-atts;
                outputclass
                           CDATA
                                     #IMPLIED"
>
```

Note The DITA architect had to declare all the parameter entities that are referenced in the redefined attributes for `<section>`. If they did not do so, none of the attributes that are declared in the parameter entities would be available on the `<section>` element. Furthermore, since the `%select-atts;` parameter entity references the `%filter-atts;` parameter entity, the `%filter-atts;` must be declared and it must precede the declaration for the `%select-atts;` parameter entity. The `%props-attribute-extensions;` and `%base-attribute-extensions;` parameter entities do not need to be declared in the constraint module, because they are declared in the document-type shells before the inclusion of the constraint module.

3. They add the constraint module to the `catalog.xml` file.
4. They then integrate the constraint module into the applicable document-type shells.
5. After checking the test topics to ensure that the content model is modified as expected, the work is done.

8.5.4.3 Example: Constrain a domain module using DTD

In this scenario, a DITA architect wants to use only a subset of the elements defined in the highlighting domain. They want to use `<b>` and `<i,>` but not `<line-through>`, `<overline>`, `<sup>`, `<sup>`, `<tt>`, or `<u>`. They want to integrate this constraint into the document-type shell for task.

1. TheDITA architect creates a constraint module: `reducedHighlightingDomainConstraint.mod`.
2. They add the following content to `reducedHighlightingDomainConstraint.mod`:

```
<!-- ============================================================= -->
<!--      CONSTRAINED HIGHLIGHT DOMAIN ENTITIES                     -->
<!-- ============================================================= -->

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

3. They add the constraint module to the `catalog.xml` file.
4. They then integrate the constraint module into the company-specific, document-type shell for the task topic by adding the following section directly before the "DOMAIN ENTITY DECLARATIONS" comment:

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

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

5. In the "DOMAIN EXTENSIONS" section, they replace the parameter entity for the highlighting domain with the parameter entity for the constrained highlighting domain:

```
<!ENTITY % ph             "ph |
                           %HighlightDomain-c-ph; |
                           %sw-d-ph; |
                           %ui-d-ph;
                          ">
```

6. After checking the test topic to ensure that the content model is modified as expected, the work is done.

8.5.4.4 Example: Replace a base element with the domain extensions using DTD

In this scenario, a DITA architect wants to remove the `<ph>` element but allow the extensions of `<ph>` that exist in the highlighting, programming, software, and user interface domains.

1. In the "DOMAIN EXTENSIONS" section, the DITA architect removes the reference to the `<ph>` element:

   ```
   <!-- Removed "ph | " so as to make <ph> not available, only the domain extensions. -->
   <!ENTITY % ph            "%pr-d-ph; |
                             %sw-d-ph; |
                             %ui-d-ph;
                            ">
   ```

Note Because no other entities are modified or declared outside of the usual "DOMAIN EXTENSIONS" section, this completes the architect's task. Because no new grammar file or entity is created that would highlight this change, adding a comment to highlight the constraint becomes particularly important (as shown in the example above).

8.5.4.5 Example: Apply multiple constraints to a single document-type shell using DTD

You can apply multiple constraints to a single document-type shell. However, there can be only one constraint for a given element or domain.

Here is a list of constraint modules and what they do:

File name	What it constrains	Details
`example-TopicConstraint.mod`	`<topic>`	• Removes `<abstract>` • Makes `<shortdesc>` required • Removes `<related-links>` • Disallows topic nesting
`example-SectionConstraint.mod`	`<section>`	Makes `@id` required
`example-HighlightingDomainConstraint.mod`	Highlighting domain	Reduces the highlighting domain elements to `<b>` and `<i>`
N/A	`<ph>`	Remove the `<ph>` element, allowing only domain extensions (does not require a `.mod` file)

All of these constraints can be integrated into a single document-type shell for `<topic>`, since they constrain distinct element types and domains. The constraint for the highlighting domain must be integrated before the "DOMAIN ENTITIES" section, but the order in which the other constraints are listed does not matter.

8.5.5 Examples: Constraints implemented using RNG

This section of the specification contains examples of constraints implemented using RNG

8.5.5.1 Example: Redefine the content model for the <topic> element using RNG

In this scenario, a DITA architect for Acme, Incorporated wants to redefine the content model for the topic document type. They want to omit the `<abstract>` element and make the `<shortdesc>` element required; they also want to omit the `<related-links>` element and disallow topic nesting.

1. The DITA architect creates a constraint module: `acme-TopicConstraintMod.rng`.
2. They update the `catalog.xml` file to include the new constraint module.
3. They add the following content to `acme-TopicConstraint.mod`:

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

Comment by Kristen J Eberlein on 21 April 2021

I know that the override won't happen without `combine="interleave"`, but I don't know if we cover that in the coding requirements topic. If people start with copying-and-pasting from the module that they are overriding, they won't have that and will get errors.

4. They then integrate the constraint module into the document-type shell for topic by adding an `<include>` element in the "ELEMENT-TYPE CONFIGURATION INTEGRATION" section:

```
<div>
  <a:documentation>ELEMENT-TYPE CONFIGURATION INTEGRATION</a:documentation>
  <include href="acme-TopicConstraintMod.rng"/>
</div>
```

5. They then remove the `<include>` statement that references `topicMod.rng` from the "MODULE INCLUSIONS" section:

```
<div>
  <a:documentation>MODULE INCLUSIONS </a:documentation>
    ~~<include href="urn:oasis:names:tc:dita:rng:topicMod.rng:2.0"/>~~
    <include href="urn:oasis:names:tc:dita:rng:audienceAttDomain.rng:2.0"/>
    <include href="urn:oasis:names:tc:dita:rng:deliveryTargetAttDomain.rng:2.0"/>
    <include href="urn:oasis:names:tc:dita:rng:platformAttDomain.rng:2.0"/>
    <include href="urn:oasis:names:tc:dita:rng:productAttDomain.rng:2.0"/>
    <include href="urn:oasis:names:tc:dita:rng:otherpropsAttDomain.rng:2.0"/>
    <include href="urn:oasis:names:tc:dita:rng:highlightDomain.rng:2.0"/>
  </div>
```

6. After checking the test topic to ensure that the content model is modified as expected, the work is done.

8.5.5.2 Example: Constrain attributes for the <section> element using RNG

In this scenario, a DITA architect wants to redefine the attributes for the `<section>` element. They want to make the `@id` attribute required.

1. The DITA architect creates a constraint module: `id-requiredSectionContraintMod.rng`.
2. They update the `catalog.xml` file to include the new constraint module.
3. They add the following content to the constraint module:

```
<?xml version="1.0" encoding="UTF-8"?>
<grammar
  xmlns="http://relaxng.org/ns/structure/1.0"
  xmlns:dita="http://dita.oasis-open.org/architecture/2005/"
  xmlns:a="http://relaxng.org/ns/compatibility/annotations/1.0"
  datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">

  <div>
    <a:documentation>ATTRIBUTE LIST OVERRIDES</a:documentation>
    <include href="urn:oasis:names:tc:dita:rng:topicMod.rng:2.0">
      <define name="section.attributes">
          <attribute name="id">
            <data type="NMTOKEN"/>
          </attribute>
        <ref name="conref-atts"/>
        <ref name="select-atts"/>
        <ref name="localization-atts"/>
        <optional>
          <attribute name="outputclass"/>
        </optional>
      </define>
    </include>
  </div>

</grammar>
```

   Note that unlike a constraint module that is implemented using DTD, this constraint module did not need to re-declare the patterns that are referenced in the redefinition of the content model for `<section>`

4. They then integrate the constraint module into the document-type shell for topic by adding an `<include>` element in the "CONTENT CONSTRAINT INTEGRATION" section:

```
<div>
  <a:documentation>CONTENT CONSTRAINT INTEGRATION</a:documentation>
  <include href="id-requiredSectionConstraintMod.rng"/>
</div>
```

5. They then remove the `<include>` statement that references `topicMod.rng` from the "MODULE INCLUSIONS" section:

```
<div>
  <a:documentation>MODULE INCLUSIONS </a:documentation>
    ~~<include href="urn:oasis:names:tc:dita:rng:topicMod.rng:2.0"/>~~
    <include href="urn:oasis:names:tc:dita:rng:audienceAttDomain.rng:2.0"/>
    <include href="urn:oasis:names:tc:dita:rng:deliveryTargetAttDomain.rng:2.0"/>
    <include href="urn:oasis:names:tc:dita:rng:platformAttDomain.rng:2.0"/>
    <include href="urn:oasis:names:tc:dita:rng:productAttDomain.rng:2.0"/>
    <include href="urn:oasis:names:tc:dita:rng:otherpropsAttDomain.rng:2.0"/>
    <include href="urn:oasis:names:tc:dita:rng:highlightDomain.rng:2.0"/>
  </div>
```

6. After checking the test topic to ensure that the attribute list is modified as expected, the work is done.

8.5.5.3 Example: Constrain a domain module using RNG

In this scenario, a DITA architect wants to use only a subset of the elements defined in the highlighting domain. They want to use `<b>` and `<i>` but not `<line-through>`, `<overline>`, `<sup>`, `<sup>`, `<tt>`, or `<u>`. Their implementation uses RNG for its grammar files.

When using RNG, domains can be constrained directly in the document-type shells.

1. They open the document-type shell for topic in an XML editor, and then they modify the "MODULE INCLUSIONS" division to exclude the elements that they do not want the implementation to use:

```
<div>
  <a:documentation>MODULE INCLUSIONS</a:documentation>
  ...
  <include href="highlightDomain.rng">
    <define name="line-through.element">
      <notAllowed/>
    </define>
    <define name="overline.element">
      <notAllowed/>
    </define>
    <define name="sub.element">
      <notAllowed/>
    </define>
    <define name="sup.element">
      <notAllowed/>
    </define>
    <define name="tt.element">
      <notAllowed/>
    </define>
    <define name="u.element">
      <notAllowed/>
    </define>
  </include>
  ..
</div>
```

   Note The DITA architect made a choice as to where in the document-type shell they would implement the constraint. It can be placed either in the "Element-type configuration integration" or the "Module inclusions" section.

2. They make similar changes to all the other document-type shells in which they want to constrain the highlighting domain.

8.5.5.4 Example: Replace a base element with the domain extensions using RNG

In this scenario, the DITA architect wants to remove the `<ph>` element but allow the extensions of `<ph>` that exist in the highlight, programming, software, and user interface domains.

1. They open the document-type shell for topic in an XML editor, and then they modify the "MODULE INCLUSIONS" division to exclude `<ph>`:

```
<div>
  <a:documentation>MODULE INCLUSIONS</a:documentation>
  <include href="urn:oasis:names:tc:dita:rng:topicMod.rng:2.0">
    <define name="ph.element">
      <notAllowed/>
    </define>
  </include>
  ...
</div>
```

2. They make similar changes to all the other document-type shells in which they want `<ph>` to not be available

8.5.5.5 Example: Apply multiple constraints to a single document-type shell using RNG

In this scenario, the DITA architect wants to apply multiple constraints to a document-type shell.

Here is a list of the constraint modules and what they do:

File name	What it constrains	Details
`example-TopicConstraint.mod`	`<topic>`	• Removes `<abstract>` • Makes `<shortdesc>` required • Removes `<related-links>` • Disallows topic nesting
`example-SectionConstraint.mod`	`<section>`	Makes `@id` required
Not applicable	Highlighting domain	Reduces the highlighting domain elements to `<b>` and `<i>`
Not applicable	`<ph>`	Remove the `<ph>` element, allowing only domain extensions

The constraint modules that target the `<topic>` and `<section >` elements must be combined, since both elements are defined in `topicMod.rng`. The other constraints can be implemented directly in the document-type shell.

1. The DITA architect creates a constraint module that combines the constraints from `example-TopicConstraint.mod` and `example-SectionConstraint.mod`:

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-model href="urn:oasis:names:tc:dita:rng:vocabularyModuleDesc.rng"
                         schematypens="http://relaxng.org/ns/structure/1.0"?>
<grammar xmlns="http://relaxng.org/ns/structure/1.0"
  xmlns:dita="http://dita.oasis-open.org/architecture/2005/"
  xmlns:a="http://relaxng.org/ns/compatibility/annotations/1.0"
  datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">
  <div>
    <a:documentation>CONTENT MODEL AND ATTRIBUTE LIST OVERRIDES</a:documentation>
    <include href="topicMod.rng">
      <define name="section.attributes">
          <attribute name="id">
            <data type="NMTOKEN"/>
          </attribute>
        <ref name="conref-atts"/>
        <ref name="select-atts"/>
        <ref name="localization-atts"/>
        <optional>
          <attribute name="outputclass"/>
        </optional>
      </define>
      <define name="topic.content">
        <ref name="title"/>
        <ref name="shortdesc"/>
        <optional>
          <ref name="prolog"/>
        </optional>
        <optional>
          <ref name="body"/>
        </optional>
      </define>
    </include>
  </div>
</grammar>
```

2. In the document-type shell, they integrate the constraint module (and remove the inclusion statement for `topicMod.rng`):

```
<div>
  <a:documentation>ELEMENT-TYPE CONFIGURATION INTEGRATION</a:documentation>
  <include href="acme-SectionTopicContraintMod.rng"/>
</div>
```

3. To constrain the highlight domain, they modify the include statement for the domain module:

```
<div>
  <a:documentation>MODULE INCLUSIONS</a:documentation>
  ...
  <include href="highlightDomain.rng">
    <define name="line-through.element">
      <notAllowed/>
    </define>
    <define name="overline.element">
      <notAllowed/>
    </define>
    <define name="sub.element">
      <notAllowed/>
    </define>
    <define name="sup.element">
      <notAllowed/>
    </define>
    <define name="tt.element">
      <notAllowed/>
    </define>
    <define name="u.element">
      <notAllowed/>
    </define>
  </include>
  ..
</div>
```

4. Finally, to disallow `<ph>`, they add the following statement to the constraint module:

```
      <define name="ph.element">
        <notAllowed/>
      </define>
```

8.6 Expansion modules

Expansion modules enable the extension of content models and attribute lists for individual elements. Expansion modules are the opposite of constraints. They add elements and attributes to specific content models and attribute lists, rather than removing them.

8.6.1 Overview of expansion modules

Expansion modules enable information architects to include specialized attributes or elements in specific element types, without making them globally available.

An expansion module can perform the following functions:

Expand content models

Expansion modules extend the content models of specific elements, without making the specialized elements available wherever the specialization base is permitted.

For example, an expansion for `<section>` can make a new element (`<sectionDesc>`) available as an optional, child element. The `<sectionDesc>` element is specialized from `<p>`, but it is available only within `<section>`.

The elements are defined in a separate element domain that declares the content models and attribute lists for the new elements.

Expand attribute lists

Expansion modules extend the attribute lists of specific elements by adding attributes specialized from either `@base` or `@props`.

For example, an expansion for `<entry>`, `<row>`, and `<colspec>` can make `@cell-purpose` available only on those elements. The `@cell-purpose` attribute is specialized from `@base`.

The additional attribute can be either defined directly within the expansion module, or it can be defined in a separate attribute-specialization module. In either case, the token used as value for the `@specializations` attribute must be defined.

8.6.2 Expansion module rules

There are certain rules that apply to the design and implementation of expansion modules. These rules all stem from the requirement that the content model of a specialized element must be consistent with the content model of the specialization base. After generalization, the content model of an element affected by an expansion module must match the original content model for that element.

Specialization base of expanded elements

Elements that are added to content models by expansion models must be specializations of existing elements that are permitted in the original content model.

Content model of expanded elements

Elements that are added to content models by expansion models must be allowed only where their specialization base is allowed.

For example, when creating an expansion model that adds a specialization of `<data>` to `<ol>`, the specialization of `<data>` must only be allowed before any `<li>` elements, as that is the only place that the `<data>` element is allowed in the content model for an ordered list.

Ordinality of expanded elements

Elements that are added to content models by expansion modules must not violate the ordinality of the original content model. If the original content model requires a child element to occur at least once, then the expanded content model cannot break this requirement. If the original content model only permits a child element to occur once, then the expanded content model cannot break this requirement.

For example, in the expansion module that adds a specialization of `<data>` to `<ol>`, the redefined content model for `<ol>` cannot make the `<li>` element optional.

However, a expansion module that adds a specialization of `<li>` (`<listIntro>`) to `<ol>` can redefine the content model of `<ol>` in the following ways:

- Make `<listIntro>` the first child element and be required
- Make `<li>` the second child element and optional

When a DITA topic affected by this expansion module is generalized, the resulting markup would be valid; the content model of `<ol>` would be respected.

Aggregation of expansion modules

The content model of an element can be modified by either of the following element-configuration modules:

- Constraint module
- Expansion module

The content model of an element can be modified only by a single element-type configuration module. If multiple constraints or extensions need to be applied to a single element, the element configurations must be combined into a single module that reflects all the constraints and expansions that were defined in the original separate modules.

8.6.3 Examples: Expansion implemented using DTDs

This section of the specification contains examples of extension modules that are implemented using DTDs.

8.6.3.1 Example: Adding an element to the <section> element using DTDs

In this scenario, a DITA architect wants to modify the content model for the `<section>` element. The DITA architect wants to add an optional `<sectionDesc>` element that is specialized from `<p>`.

To accomplish this, the DITA architect needs to create the following modules and integrate them into the document-type shell:

- An element specialization module that defines the `<sectionDesc>` element
- An expansion module that adds the `<sectionDesc>` element to the content model for `<section>`

1. First, the DITA architect creates the element specialization module: `sectionDescDomain.mod`. This single `.mod` file defines the parameter entity, content model, attributes, and value for the `@class` attribute for `<sectionDesc>`.

   ```
   <?xml version="1.0" encoding="UTF-8"?>

   <!ENTITY % sectionDesc "sectionDesc">

   <!ENTITY % sectionDesc.content "(%para.cnt;)*">
   <!ENTITY % sectionDesc.attributes "%univ-atts;">

   <!ELEMENT sectionDesc %sectionDesc.content;>
   <!ATTLIST sectionDesc %sectionDesc.attributes;>

   <!ATTLIST sectionDesc    class CDATA "+ topic/p sectionShortdesc-d/sectionDesc ">
   ```
2. The DITA architect adds the element specialization module to the `catalog.xml` file.
3. Next, the DITA architect modifies the applicable document-type shell to integrate the applicable element specialization module:

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

   <!-- ... other domains ... -->

   <!ENTITY % sectionDesc-d-def
     PUBLIC "-//ACME//ELEMENTS DITA 2.0 Section Description Domain//EN"
            "sectionDescDomain.mod"
   >%sectionDesc-d-def;
   ```

   At this point, the new domain is integrated into the topic document-type shell. However, the new element is not added to the content model for `<section>`.
4. Next, the DITA architect creates an expansion module: `acme-SectionExpansion.mod`. This module adds the `<sectionDesc>` element to the content model of `<section>`.

   ```
   <?xml version="1.0" encoding="UTF-8"?>

   <!-- Declares the entities referenced in the modified content  -->
   ```

```
<!-- model.                                                    -->

<!ENTITY % dl "dl">
<!ENTITY % div "div">
<!ENTITY % fig "fig">
<!ENTITY % image "image">
<!ENTITY % lines "lines">
<!ENTITY % lq "lq">
<!ENTITY % note "note">
<!ENTITY % object "object">
<!ENTITY % ol "ol">
<!ENTITY % p "p">
<!ENTITY % pre "pre">
<!ENTITY % simpletable "simpletable">
<!ENTITY % sl "sl">
<!ENTITY % table "table">
<!ENTITY % ul "ul">
<!ENTITY % cite "cite">
<!ENTITY % include "include">
<!ENTITY % keyword "keyword">
<!ENTITY % ph "ph">
<!ENTITY % q "q">
<!ENTITY % term "term">
<!ENTITY % text "text">
<!ENTITY % tm "tm">
<!ENTITY % xref "xref">
<!ENTITY % state "state">
<!ENTITY % data "data">
<!ENTITY % foreign "foreign">
<!ENTITY % unknown "unknown">
<!ENTITY % title "title">
<!ENTITY % draft-comment "draft-comment">
<!ENTITY % fn "fn">
<!ENTITY % indexterm "indexterm">
<!ENTITY % required-cleanup "required-cleanup">
<!ENTITY % sectionDesc "sectionDesc">

<!-- Defines the modified content model for <section>.       -->

<!ENTITY % section.content
              "(#PCDATA |
               %dl; |
               %div; |
               %fig; |
               %image; |
               %lines; |
               %lq; |
               %note; |
               %object; |
               %ol; |
               %p; |
               %pre; |
               %simpletable; |
               %sl; |
               %table; |
               %ul; |
               %cite; |
               %include; |
               %keyword; |
               %ph; |
               %q; |
               %term; |
               %text; |
               %tm; |
               %xref; |
               %state; |
               %data; |
               %foreign; |
               %unknown; |
               %title; |
               %draft-comment; |
               %fn; |
               %indexterm; |
               %required-cleanup; |
               %sectionDesc;)*"
>
```

Note that the DITA architect needed to explicitly declare all the elements, rather than using the `%section.cnt;` parameter entity that is used in the definition of `<section>`. Because the element-configuration modules are integrated into the document-type shell before the base grammar modules, none of the parameter entities that are used in the base DITA vocabulary modules are available.

5. Finally, the DITA architect integrates the expansion module into the document-type shell:

```
<!-- ============================================================ -->
<!--           ELEMENT-TYPE CONFIGURATION INTEGRATION              -->
<!-- ============================================================ -->

<!-- Other constraint and expansion modules -->

<!ENTITY % acmeSection-def
  PUBLIC "-//ACME//ELEMENTS DITA 2.0 Section Expansion//EN"
         "acme-SectionExpansion.mod"
>%acmeSection-def;
```

6. After updating the `catalog.xml` file to include the expansion module and testing, the work is done.

8.6.3.2 Example: Adding an attribute to certain table elements using DTDs

In this scenario, a company makes extensive use of complex tables to present product listings. They occasionally highlight individual cells, rows, or columns for various purposes. The DITA architect wants to implement a semantically meaningful way to identify the purpose of various table elements.

The DITA architect decides to create a new attribute (`@cell-purpose`) and add it to the attribute lists of the following elements:

- `<colspec>`
- `<entry>`
- `<row>`
- `<stentry>`
- `<strow>`

The new attribute will be specialized from `@base`, and it will take a small set of tokens as values.

The DITA architect decides to integrate the attribute declaration and its assignment to elements into a single expansion module. An alternate approach would be to put each `<!ATTLIST` declaration in its own separate expansion module, thus allowing DITA architects who construct document-type shells to decide the elements to which to apply the attribute.

1. First, the DITA architect creates the expansion module for the `@cell-purpose` attribute: `acme-cellPurposeAttExpansion.ent`.

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

<!-- Declare the entity to be used in the @specializations attribute -->
<!ENTITY cellPurposeAtt-d-att "@base/cell-purpose" >

<!-- Add the attribute to the elements. -->
<!ATTLIST entry %cellPurposeAtt-d-attribute-expansion;>
<!ATTLIST row %cellPurposeAtt-d-attribute-expansion;>
<!ATTLIST colspec %cellPurposeAtt-d-attribute-expansion;>
<!ATTLIST strow %cellPurposeAtt-d-attribute-expansion;>
<!ATTLIST stentry %cellPurposeAtt-d-attribute-expansion;>
```

Note The attribute definition entity is optional. It is used here to enable the DITA architect to add the same attribute with the same tokens to several elements.

2. They then update the `catalog.xml` file to include the expansion module.
3. They integrate this module into the applicable document-type shell.

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

<!-- ... other domains ... -->

<!ENTITY % cellPurposeAttExpansion-d-dec
  PUBLIC "-//ACME//ENTITIES DITA Cell Purpose Attribute Expansion//EN"
         "cellPurposeAttExpansion.ent"
>%cellPurposeAttExpansion-d-dec;
```

4. They add the entity for the contribution to the `@specializations` attribute.

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

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

5. After checking the test topic to ensure that the attribute lists are modified as expected, the work is done.

8.6.3.3 Example: Adding an existing domain attribute to certain elements using DTDs

In this scenario, a company wants to use the `@otherprops` attribute specialization. However, they want to make the attribute available **only** on certain elements: `<p>`, `<div>`, and `<section>`.

The DITA architect will need to create an extension module and integrate it into the appropriate document-type shells.

1. The DITA architect creates an expansion module that adds the `@otherprops` attribute to the selected elements: `acme-otherpropsAttExpansion.mod`. The expansion module contains the following code:

```
<?xml version="1.0" encoding="UTF-8"?>

<!-- Add the otherprops attribute to certain elements -->
<!ATTLIST p %otherpropsAtt-d-attribute;>
<!ATTLIST div %otherpropsAtt-d-attribute;>
<!ATTLIST section %otherpropsAtt-d-attribute;>
```

   Note that the `%otherpropsAtt-d-attribute;` is defined in the separate attribute-specialization module that defines the `@otherprops` attribute.

2. They then update the `catalog.xml` file to include the expansion module.
3. They integrate the extension module into the applicable document-type shell, **after** the declaration for the `@otherprops` attribute-specialization module:

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

```
...

<!ENTITY % otherpropsAtt-d-dec
  PUBLIC "-//OASIS//ENTITIES DITA 2.0 Otherprops Attribute Domain//EN"
         "otherpropsAttDomain.ent"
>%otherpropsAtt-d-dec;

<!ENTITY % otherprops-expansion-e-def
  PUBLIC "-//ACME//DITA 2.0 Otherprops Expansion//EN"
         "acme-otherpropsAttExpansion.mod"
  >%otherprops-expansion-e-def;

...
```

4. They remove the reference to the `@otherprops` attribute from the `props-attribute-extension` entity:

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

<!ENTITY % base-attribute-extensions
  ""
>

<!ENTITY % props-attribute-extensions
  "%audienceAtt-d-attribute;
   %deliveryTargetAtt-d-attribute;
   ~~%otherpropsAtt-d-attribute;~~
   %platformAtt-d-attribute;
   %productAtt-d-attribute;"
>
```

5. They ensure that the `included-domains` entity contains the `@otherprops` contribution to the `@specializations` attribute:

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

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

6. After checking the test topic to ensure that the attribute lists are modified as expected, the work is done.

8.6.3.4 Example: Aggregating constraint and expansion modules using DTDs

The DITA architect wants to add some extension modules to the document-type shell for topic. The document-type shell already integrates a number of constraint modules.

The following table lists the constraints that are currently integrated into the document-type shell:

File name	What it constrains	Details
`example-TopicConstraint.mod`	`<topic>`	• Removes `<abstract>` • Makes `<shortdesc>` required • Removes `<related-links>` • Disallows topic nesting

File name	What it constrains	Details
example-SectionConstraint.mod	<section>	• Makes <title> required • Reduces the content model of <section> to a smaller subset
example-HighlightingDomainConstraint.mod	Highlighting domain	Reduces the highlighting domain elements to <b> and <i>

The following table lists the expansion modules that the DITA architect wants to add to the document-type shell:

File name	What it modifies	Details
example-sectionSectionShortdescExpansion.mod	<section>	Adds an optional <sectionDesc> element to <section>. The <sectionDesc> element can only appear directly after <title>.
example-dlentryModeAttExpansion.ent	<dlentry>	Adds @dlentryMode to the attributes of <dlentry>.

The constraint and expansion modules that target the <section> element must be combined into a single element-configuration module. An element can only be targeted by a single element-configuration module.

8.6.4 Examples: Expansion implemented using RNG

This section of the specification contains examples of extension modules implemented using RNG.

8.6.4.1 Example: Adding an element to the <section> element using RNG

In this scenario, a DITA architect wants to modify the content model for the <section> element. He wants to add an optional <sectionDesc> element that is specialized from <p>; the <sectionDesc> can occur once and must be directly after the section title.

To accomplish this, the DITA architect needs to create the following modules and integrate them into the document-type shells:

- An element domain module that defines the <sectionDesc> element
- An expansion module that adds the <sectionDesc> element to the content model for <section>

1. First, the DITA architect creates the element domain module: sectionDescDomain.rng. It contains the following code:

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-model href="urn:oasis:names:tc:dita:rng:vocabularyModuleDesc.rng"
                         schematypens="http://relaxng.org/ns/structure/1.0"?>
<grammar xmlns:a="http://relaxng.org/ns/compatibility/annotations/1.0"
  xmlns:dita="http://dita.oasis-open.org/architecture/2005/"
  xmlns="http://relaxng.org/ns/structure/1.0">
  <div>
    <a:documentation>DOMAIN EXTENSION PATTERNS</a:documentation>
  </div>
  <div>
    <a:documentation>ELEMENT TYPE NAME PATTERNS</a:documentation>
    <define name="sectionDesc">
      <ref name="sectionDesc.element"/>
    </define>
  </div>
  <div>
    <a:documentation>ELEMENT TYPE DECLARATIONS</a:documentation>
    <div>
      <a:documentation>LONG NAME: Section Description</a:documentation>
      <define name="sectionDesc.content">
        <zeroOrMore>
            <ref name="para.cnt"/>
          </zeroOrMore>
      </define>
      <define name="sectionDesc.attributes">
        <ref name="univ-atts"/>
      </define>
      <define name="sectionDesc.element">
        <element name="sectionDesc" dita:longName="Section Description">
          <a:documentation/>
          <ref name="sectionDesc.attlist"/>
          <ref name="sectionDesc.content"/>
        </element>
      </define>
      <define name="sectionDesc.attlist" combine="interleave">
        <ref name="sectionDesc.attributes"/>
      </define>
    </div>
  </div>
  <div>
    <a:documentation>SPECIALIZATION ATTRIBUTE DECLARATIONS</a:documentation>
    <define name="sectionDesc.attlist" combine="interleave">
      <optional>
        <attribute name="class" a:defaultValue="+ topic/p sectionDesc-d-p/sectionDesc
"/>
      </optional>
    </define>
  </div>
</grammar>
```

2. The DITA architect adds the element domain module to the `catalog.xml` file.
3. Next, the DITA architect modifies the document-type shell (in this example, the one for topic) to integrate the element domain module:

```
    <div>
      <a:documentation>MODULE INCLUSIONS</a:documentation>
      ...
      <include href="urn:example:names:tc:dita:rng:sectionDescDomain.rng:2.0"/>
    </div>
```

   At this point, the new domain is integrated into the document-type shell. However, the new element is not added to the content model for `<section>`.
4. Next, the DITA architect created an expansion module (`sectionExpansionMod.rng`) that adds the `<sectionDesc>` element to the content model of `<section>`:

```
  <?xml version="1.0" encoding="UTF-8"?>
  <?xml-model href="urn:oasis:names:tc:dita:rng:vocabularyModuleDesc.rng"
                           schematypens="http://relaxng.org/ns/structure/1.0"?>
  <grammar xmlns="http://relaxng.org/ns/structure/1.0"
    xmlns:dita="http://dita.oasis-open.org/architecture/2005/"
    xmlns:a="http://relaxng.org/ns/compatibility/annotations/1.0"
    datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">
    <div>
      <a:documentation>CONTENT MODEL AND ATTRIBUTE LIST OVERRIDES</a:documentation>
      <include href="urn:oasis:names:tc:dita:rng:topicMod.rng:2.0">
        <define name="topic-info-types">
          <ref name="topic.element"/>
        </define>
        <define name="section.content" combine="interleave">
          <optional>
            <ref name="title"/>
          </optional>
          <optional>
            <ref name="sectionDesc"/>
          </optional>
          <zeroOrMore>
            <ref name="section.cnt"/>
          </zeroOrMore>
        </define>
      </include>
    </div>
  </grammar>
```

Note that the expansion module directly integrates `topicMod.rng`.

5. Finally, the DITA architect integrates the expansion module into the document-type shell and removes the inclusion statement for `topicMod.rng`:

```
  <div>
    <a:documentation>ELEMENT-TYPE CONFIGURATION INTEGRATION</a:documentation>
    <include href="sectionExpansionMod.rng"/>
  </div>
  <div>
    <a:documentation>MODULE INCLUSIONS</a:documentation>
    ~~<include href="urn:oasis:names:tc:dita:rng:topicMod.rng:2.0">~~
~~      <define name="topic-info-types">~~
~~        <ref name="topic.element"/>~~
~~      </define>~~
~~    </include>~~
    ...
    <include href="urn:example:names:tc:dita:rng:sectionDescDomain.rng:2.0"/>
  </div>
```

6. After updating the `catalog.xml` file to include the expansion module and testing, the work is done.

8.6.4.2 Example: Adding an attribute to certain table elements using RNG

In this scenario, a company makes extensive use of complex tables to present product listings. They occasionally highlight individual cells, rows, or columns for various purposes. The DITA architect wants to implement a semantically meaningful way to identify the purpose of various table elements.

The DITA architect decides to create a new attribute (`@cell-purpose`) and add it to the content model of the following elements:

- `<entry>`
- `<row>`
- `<colspec>`
- `<stentry>`
- `<strow>`

The new attribute will be specialized from `@base`, and it will take a small set of tokens as values.

The DITA architect decides to integrate the attribute declaration and its assignment to elements into a single expansion module. An alternate approach would be to put each `<!ATTLIST` declaration in its own separate expansion module, thus allowing DITA architects who construct document-type shells to decide the elements to which to apply the attribute.

1. The DITA architect creates an expansion module: `cellPurposeAtt.rng`. It contains the following code:

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-model href="urn:oasis:names:tc:dita:rng:vocabularyModuleDesc.rng"
                         schematypens="http://relaxng.org/ns/structure/1.0"?>
<grammar
  xmlns="http://relaxng.org/ns/structure/1.0"
  xmlns:a="http://relaxng.org/ns/compatibility/annotations/1.0"
  xmlns:dita="http://dita.oasis-open.org/architecture/2005/"
  datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">

  <!-- DEFINE THE ATTRIBUTE SPECIALIZATION -->
  <define name="cellPurposeAtt">
    <optional>
      <attribute name="cellPurpose">
        <a:documentation>Specifies the purpose of the table cell. This is a specialized
          attribute for Acme Corporation.
        </a:documentation>
        <choice>
          <value>sale</value>
          <value>out-of-stock</value>
          <value>new</value>
          <value>last-chance</value>
          <value>inherit</value>
          <value>none</value>
        </choice>
      </attribute>
    </optional>
  </define>

  <!-- ASSIGN THE ATTRIBUTE TO CERTAIN ELEMENTS -->
  <define name="entry.attributes" combine="interleave">
    <ref name="cellPurposeAtt"/>
  </define>
  <define name="stentry.attributes" combine="interleave">
    <ref name="cellPurposeAtt"/>
  </define>
  <define name="row.attributes" combine="interleave">
    <ref name="cellPurposeAtt"/>
  </define>
  <define name="strow.attributes" combine="interleave">
    <ref name="cellPurposeAtt"/>
  </define>
  <define name="colspec.attributes" combine="interleave">
    <ref name="cellPurposeAtt"/>
  </define>
</grammar>
```

2. They then update the `catalog.xml` file to include the expansion module.
3. They integrate the expansion module into the document-type shell:

```
<div>
    <a:documentation>MODULE INCLUSIONS</a:documentation>
    ...
    <include href="urn:example:names:tc:dita:rng:cellPurposeAtt.rng:2.0"/>
  </div>
```

4. They specify the value that the `@cellPurpose` attribute contributes to the `@specializations` attribute:

```
  <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
                         @base/cellPurpose"/>
      </optional>
    </define>
  </div>
```

5. After checking the test file to ensure that the attribute lists are modified as expected, the work is done.

8.6.4.3 Example: Adding an existing domain attribute to certain elements using RNG

In this scenario, a company wants to use the `@otherprops` attribute specialization. However, they want to make the attribute available **only** on certain elements: `<p>`, `<div>`, and `<section>`.

The DITA architect will need to create an extension module and integrate it into the appropriate document-type shells.

1. The DITA architect creates an expansion module that adds the `@otherprops` attribute to the selected elements: `acme-otherpropsAttExpansion.mod`. The expansion module contains the following content:

   ```
   <?xml version="1.0" encoding="UTF-8"?>
   <?xml-model href="urn:oasis:names:tc:dita:rng:vocabularyModuleDesc.rng"
                            schematypens="http://relaxng.org/ns/structure/1.0"?>
   <grammar xmlns="http://relaxng.org/ns/structure/1.0"
     xmlns:a="http://relaxng.org/ns/compatibility/annotations/1.0"
     xmlns:dita="http://dita.oasis-open.org/architecture/2005/"
     datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">
     <div>
       <a:documentation>CONTENT MODEL AND ATTRIBUTE LIST OVERRIDES</a:documentation>
       <include href="urn:oasis:names:tc:dita:rng:topicMod.rng:2.0">
         <define name="topic-info-types">
           <ref name="topic.element"/>
         </define>
         <define name="p.attributes" combine="interleave">
           <optional>
             <attribute name="otherprops"/>
           </optional>
         </define>
         <define name="div.attributes" combine="interleave">
           <optional>
             <attribute name="otherprops"/>
           </optional>
         </define>
         <define name="section.attributes" combine="interleave">
           <optional>
             <attribute name="otherprops"/>
           </optional>
         </define>
       </include>
     </div>
   </grammar>
   ```

2. They then update the `catalog.xml` file to include the expansion module.
3. They integrate the extension module into the applicable document-type shell, and remove the `<include>` element for `topicMod.rng`:

   ```
     <div>
       <a:documentation>ELEMENT-TYPE CONFIGURATION INTEGRATION</a:documentation>
         <include href="acme-otherpropsAttExpansion.rng"/>
     </div>
     <div>
       <a:documentation>MODULE INCLUSIONS</a:documentation>
       ~~<include href="urn:oasis:names:tc:dita:rng:topicMod.rng:2.x"/>~~
       ...
       <include href="urn:oasis:names:tc:dita:rng:otherpropsAttDomain.rng:2.0">
       </include>
     </div>
   ```

4. They remove the reference to the `@otherprops` attribute from the `props-attribute-extension` pattern:

   ```
     <div>
       <a:documentation>MODULE INCLUSIONS</a:documentation>
         ...
   ```

```
     <include href="urn:oasis:names:tc:dita:rng:otherpropsAttDomain.rng:2.0">
       <define name="props-attribute-extensions" combine="interleave">
       <empty/>
       </define>
     </include>
```

5. They ensure that the `included-domains` entity contains the `@otherprops` contribution to the `@specializations` attribute:

```
   <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>
```

6. After checking the test topic to ensure that the attribute lists are modified as expected, the work is done.

8.6.4.4 Example: Aggregating constraint and expansion modules using RNG

The DITA architect wants to add some extension modules to the document-type shell for topic. The document-type shell already integrates a constraint module.

The following table lists the constraint module and the extension modules that the DITA architect wants to integrate into the document-type shell for topic.

Type of element configuration	File name	What it does
Constraint	`topicSectionConstraint.rng`	Constrains `<topic>`: • Removes `<abstract>` • Makes `<shortdesc>` required • Removes `<related-links>` • Disallows topic nesting Constrains `<section>`: • Makes `@id` required
Expansion	`sectionExpansionMod.rng`	Adds `<sectionDesc>` to the content model of `<section>`
Expansion	`tableCellAttExpansion.rng`	Adds `@cellPurpose` to the attribute lists for certain table elements

Because all of these element-configuration modules target elements declared in `topicMod.rng`, the DITA architect needs to combine them into a single element-configuration module like the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-model href="urn:oasis:names:tc:dita:rng:vocabularyModuleDesc.rng"
                         schematypens="http://relaxng.org/ns/structure/1.0"?>
<grammar xmlns="http://relaxng.org/ns/structure/1.0"
  xmlns:dita="http://dita.oasis-open.org/architecture/2005/"
  xmlns:a="http://relaxng.org/ns/compatibility/annotations/1.0"
  datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">
  <div>
```

```
    <a:documentation>CONTENT MODEL AND ATTRIBUTE LIST OVERRIDES</a:documentation>
    <include href="urn:oasis:names:tc:dita:rng:topicMod.rng:2.0">
      <!-- Redefines attribute list for section: Makes @id required -->
      <define name="section.attributes">
        <attribute name="id">
          <data type="ID"/>
        </attribute>
        <ref name="conref-atts"/>
        <ref name="select-atts"/>
        <ref name="localization-atts"/>
        <optional>
          <attribute name="outputclass"/>
        </optional>
      </define>
      <!-- Adds sectionDesc to the content model of section -->
      <define name="section.content" combine="interleave">
        <optional>
          <ref name="title"/>
        </optional>
        <optional>
          <ref name="sectionDesc"/>
        </optional>
        <zeroOrMore>
          <ref name="section.cnt"/>
        </zeroOrMore>
      </define>
      <!-- Adds @cellPurpose to certain table and simple table elements -->
      <define name="colspec.attributes" combine="interleave">
        <optional>
          <attribute name="cellPurpose"/>
        </optional>
      </define>
      <define name="entry.attributes" combine="interleave">
        <optional>
          <attribute name="cellPurpose"/>
        </optional>
      </define>
      <define name="row.attributes" combine="interleave">
        <optional>
          <attribute name="cellPurpose"/>
        </optional>
      </define>
      <define name="stentry.attributes" combine="interleave">
        <optional>
          <attribute name="cellPurpose"/>
        </optional>
      </define>
      <define name="strow.attributes" combine="interleave">
        <optional>
          <attribute name="cellPurpose"/>
        </optional>
      </define>
      <!-- Redefines topic: removes abstract and related-links; makes shortdesc -->
      <!--                 required; disallows topic nesting                   -->
      <define name="topic.content">
        <ref name="title"/>
        <ref name="shortdesc"/>
        <optional>
          <ref name="prolog"/>
        </optional>
        <optional>
          <ref name="body"/>
        </optional>
      </define>
    </include>
  </div>
</grammar>
```

When the DITA architect edits the document-type shell to integrate the element configuration module, they also need to do the following:

- Remove the include statement for `topicMod.rng`

- Add `<section>` to the "ID-DEFINING ELEMENT OVERRIDES" division

9 Coding practices for DITA grammar files

This section collects all of the rules for creating modular DTD or RELAX NG-based grammar files that represent DITA document types, specializations, constraints, and expansion modules.

9.1 Recognized XML-document grammar mechanisms

Conforming DITA document types and vocabulary modules can be constructed using several XML document-grammar mechanisms. The DITA specification provides coding requirements for DTDs and RNG; it also includes grammar files constructed using those mechanisms.

Of these document-grammar mechanisms, RELAX NG grammars offer the easiest-to-use syntax and the most precise constraints. For this reason, the RELAX NG definitions of the standard DITA vocabularies are the normative versions.

Related information

Tools for generating DTD or XSD from RELAX NG

9.2 Normative versions of DITA grammar files

The OASIS DITA Technical Committee uses the RELAX NG XML syntax for the normative versions of the XML grammar files that comprise the DITA release.

The DITA Technical Committee chose the RELAX NG XML syntax for the following reasons:

Easy use of foreign markup

The DITA grammar files maintained by OASIS depend on this feature of RELAX NG in order to capture metadata about document-type shells and modules.

The foreign vocabulary feature also can be used to include Schematron rules directly in RELAX NG grammars. Schematron rules can check for patterns that either are not expressible with RELAX NG directly or that would be difficult to express.

RELAX NG <div> element

This general grouping element allows for arbitrary organization and grouping of patterns within grammar documents. Such grouping tends to make the grammar documents easier to work with, especially in XML-aware editors. The use or non-use of the RELAX NG `<div>` element does not affect the meaning of the patterns that are defined in a RELAX NG schema.

Capability of expressing precise restrictions

RELAX NG is capable of expressing constraints that are more precise than is possible with either DTDs or XSDs. For example, RELAX NG patterns can be context specific such that the same element type can allow different content or attributes in different contexts.

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. 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. The grammar files provided by the OASIS DITA Technical Committee do not use any features of RELAX NG that cannot be translated into equivalent DTD or XSD constructs.

Comment by Kristen J Eberlein on 31 March 2021

Regarding the above paragraph, do we want to remove the mentions of XSD? The advice is still sound, although I think we want to encourage people who must use XSDs to generate a single file for validation from either DTD or RNG.

The DITA use of RELAX NG depends on the *RELAX NG DTD Compatibility* specification, which provides a mechanism for defining default attribute values and embedded documentation. Processors that use RELAX NG for DITA documents in which required attributes (for example, `@class` attribute) are not explicitly present must implement the DTD compatibility specification in order to get default attribute values.

Related information

Tools for generating DTD or XSD from RELAX NG

9.3 DTD coding requirements

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

9.3.1 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, rather than directly setting what is allowed in `<ph>`, that element sets its content model to `%ph.content;`; that entity defines the actual content model. A constraint module can redefine the `%ph.content;` model to remove selected elements, or an expansion module can redefine the `%ph.content;` module to add elements.

Attribute sets defined as entities

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

For example, rather than directly defining attributes for `<ph>`, that element sets its attributes using the `%ph.attributes;` entity; that entity defines the actual attributes. A constraint module can remove attributes from the attribute list, or an expansion module can 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.

9.3.2 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 (218)
2. Domain constraint integration (218)
3. Domain entity declarations (219)
4. Domain attributes declarations (219)
5. Domain extensions (219)
6. Domain attribute extensions (220)
7. Topic nesting override (220)
8. Specializations attribute override (220)
9. Element-type configuration integration (221)
10. Topic [or map] element integration (221)
11. Domain element integration (221)

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 top-level topic or map type that the document-type shell configures
- Any additional structural modules that are used by the document type shell

The parameter entity is named `*type-name*-dec`.

In the following example, 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-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 later modules to define content models; redefining the entity 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 a parameter entity 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 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 a parameter entity for each element domain that is integrated tinto 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.

9.3.3 DTD: Coding requirements for element-type declarations

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:

- A definition module (`.mod`) file, which declares the element names, content models, and attribute lists for the element types that are defined in the vocabulary module
- An entity declaration (`.ent`) file, which declares the text 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
- (If the vocabulary module is a structural module) 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 default 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 single 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 8.3.6 class attribute rules and syntax (182).

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 ==================== -->
```

9.3.4 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 `@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 modules that defines a new topic type typically uses a parameter entity to define what topic types are permitted to nest. While there are known exceptions described below, 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">
```

Default topic nesting in a structural module often is set up to use the `%info-types;` parameter entity rather than using a specific element. 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 example shows a structural module that uses `%info-types;` for default topic nesting:

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

Content model of the root element

070 (411) 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. If default nesting rules reference the `info-types` parameter entity, a shell can efficiently create common nesting rules by redefining the `info-types` entity.

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;),
    (%titlealts;)?,
    (%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 never allow any 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.

Comment by Kristen J Eberlein on 07 April 2021

Examples? I think the wording needs to be crisped up.

9.3.5 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 highlight 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 ================== -->
```

9.3.6 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 text 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.

Text entity name and value

The text 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 8.3.7 specializations attribute rules and syntax (184) 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" >
```

9.3.7 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 set 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 set of attributes 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;),
    (%titlealts;)?,
    (%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 only 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 sectionShortdesc-d-p-expansion "section-shortdesc">

<!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-d-p-expansion;)*"
>
```

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

9.4 RELAX NG coding requirements

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

9.4.1 RELAX NG: Overview of coding requirements

RELAX NG modules are self-integrating; they automatically add to the content models and attribute lists that they extend. This means that information architects do not have much work to do when integrating vocabulary modules and element-configuration modules (constraints and expansion) into document-type shells.

Self-integration of RELAX NG

In addition to simplifying document-type shells, the self-integrating aspect of RELAX NG results in the following coding practices:

- Each specialization module consists of a single file, unlike the two required for DTDs.
- 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.

071 (411)

RELAX NG grammars for DITA document-type shells, vocabulary modules, and element-configuration modules (constraint and expansion) **MAY** 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 **MAY** 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.

Syntax 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.

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 conforming if there is a one-to-one file correspondence.

072 (411)

Conforming RELAX NG-based DITA modules **MAY** omit the annotations and foreign elements that are used in the OASIS grammar files to enable generation of other XML grammars, such as DTDs and XML Schema. When such annotations are used, conversion from one RELAX NG syntax to the other might lose the information, as processors are not required to process the annotations and information from foreign vocabularies.

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 organization requirements as those defined for the RELAX NG XML syntax.

9.4.2 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.

Because RELAX NG modules are self-integrating, RNG-based document-type shells need only to include vocabulary modules and element-configuration modules (constraint and expansion).

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

1. Root element declaration (230)
2. specializations attribute (230)
3. Element-type configuration integration (231)
4. Module inclusions (231)
5. ID-defining element overrides (231)

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 the element-configuration module imports the module that it override, any module that is configured in this section (including the base topic or map modules) is left out of the following "Module inclusion" section.

For example, 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">
    <define name="task-info-types">
      <ref name="task.element"/>
    </define>
  </include>
</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 a 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>
```

9.4.3 RELAX NG: Coding requirements for element-type declarations

This topic covers general coding requirements for element types in 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 element-type `*tagname*.element` pattern.

For example:

```
<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**
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.

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>The &lt;topichead> element provides a title-only entry in a
navigation map, as an alternative to the fully-linked title provided by the &lt;topicref>
element. Category:
            Mapgroup elements</a:documentation>
          <ref name="topichead.attlist"/>
          <ref name="topichead.content"/>
        </element>
      </define>
      <define name="topichead.attlist" combine="interleave">
        <ref name="topichead.attributes"/>
      </define>

    </div>
```

Comment by robander
Reminder to update this example with a 2.0 version of the declaration.

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 that element's `@class` attribute. `@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 8.3.6 class attribute rules and syntax (182).

For example:

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

9.4.4 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 default topic nesting. Document-type shells then can control how topics are allowed to nest by redefining the pattern. While there are known exceptions described below, 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.

Comment by Kristen J Eberlein on 07 April 2021

What does the above sentence mean?

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.

Comment by Kristen J Eberlein on 07 April 2021

Examples? I think the wording needs to be crisped up.

9.4.5 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 highlight 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.

For example:

```
<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.

Example

The following code sample shows the extension pattern for the highlight domain, as declared in `highlightDomain.rng`:

```
<div>
  <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>

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

9.4.6 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 8.3.7 specializations attribute rules and syntax (184).

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 set 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>
```

9.4.7 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.

Comment by Kristen J Eberlein on 14 July 2022

The above contradicts what we show in the "Example: Adding an attribute to certain table elements using RNG," where the exansion modules does not import the base vocabulary module." The current wording is true for constraints, inaccurate for expansion.

The same issue crops up several times in this topic ...

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:oasis:names:tc:dita:rng:topicMod.rng:2.0">
    <define name="topic.content" combine="interleave">
      <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:oasis:names:tc:dita:rng:topicMod.rng:2.0">
    <define name="section.content" combine="interleave">
      <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`.