Review P: Configuration

Table of contents

1 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.

1.1 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.

Specialization modules 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.

1.2 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.

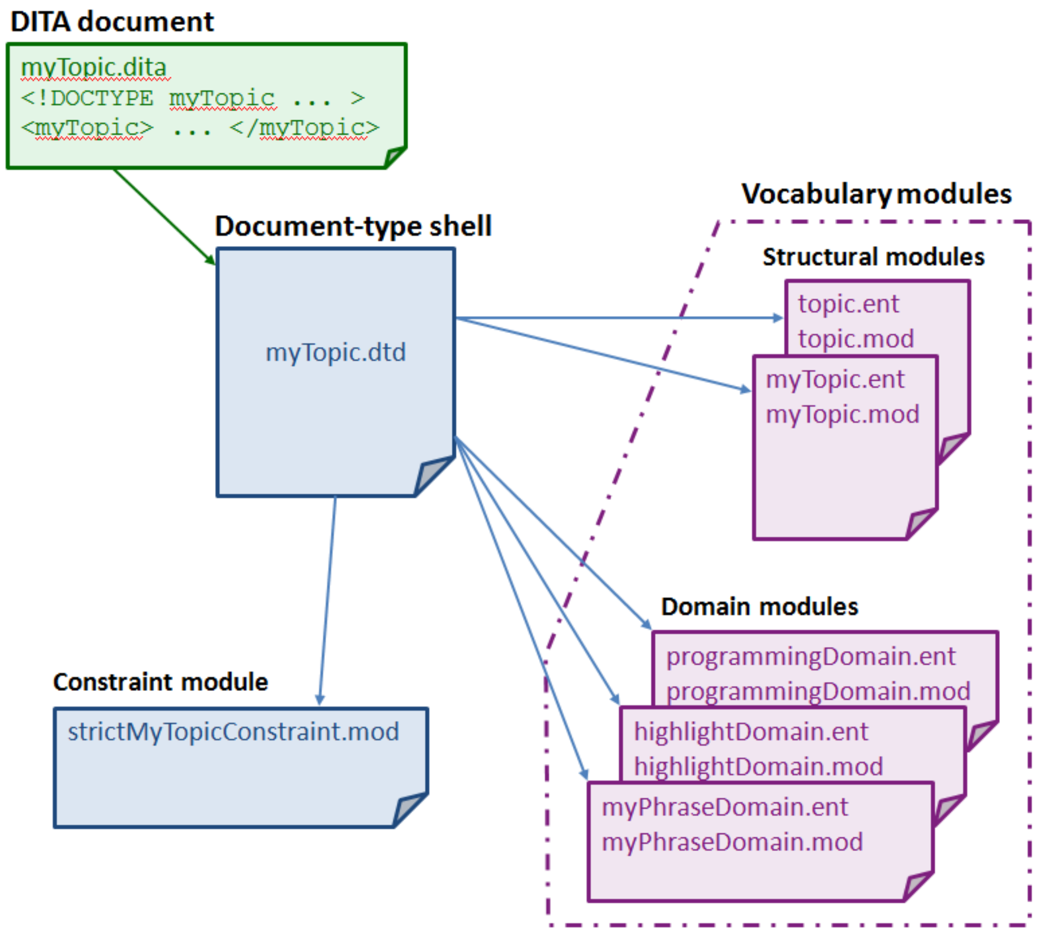
1.2.1 Overview of document-type shells

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 1: 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.

1.2.2 Rules for document-type shells

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

XML grammars

001 (39) 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 XSD or Schematron.

Defining element or attribute types

002 (39) 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

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

004 (39) 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".

1.2.3 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.

1.2.4 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.

1.3 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.

1.3.1 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 architects 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, and bookmap is specialized from map.

Domain specialization

Domain specializations are developed from elements defined within 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.

1.3.2 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 simplified. 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, reduces error, enforces consistency, and makes interoperability possible.

Because all the pieces are modular, it is simpler 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.

1.3.3 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.

005 (39) 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.

006 (39) 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.

1.3.4 Specialization rules for element types

There are certain rules that apply to element type specializations.

Characteristics

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

Namespaces

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.

Comment by Kristen J Eberlein on 31 August 2022

We should consider moving the above note elsewhere.

1.3.5 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.

1.3.6 The @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. The `@class` attribute 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 instead of document type. Examples of module types are topic type, domain type, or map type. This enables document-type developers to 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

007 (39)	Every DITA element (except the `<dita>` element that is used as the root of a ditabase document) **MUST** declare a `@class` attribute.
008 (39)	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.
009 (39)	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.
010 (39)	Authors **SHOULD NOT** modify the `@class` attribute. 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: 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 the topic module. It also indicates explicitly that the `<step>` element is available in a task topic. This declaration 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>
```

Example: @class attribute with intermediate value

Comment by Kristen J Eberlein on 01 September 2022

This example needs to be reviewed as part of the generalization content.

In addition, it would benefit by an edit that started by asking "What exactly are we trying to convey here? What are the critical points?

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.

1.3.7 The @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.

011 (39) 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`

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.

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"
```

1.3.8 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 model.

- 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.

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`.

1.3.9 Sharing elements across specializations

Specialization enables reuse of elements from ancestor specializations. However, it is also possible to reuse elements from non-ancestor specializations.

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.

1.4 Constraints

Constraint modules 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.

1.4.1 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 files, 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.

1.4.2 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

For any document-type shell, the content model of an element can only 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.

1.4.3 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. Similarly, an unconstrained task is compatible with an unconstrained topic, because the `<task>` element can be generalized to `<topic>`.

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

1.4.4 Examples: Constraints implemented using DTDs

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

1.4.4.1 Example: Restrict 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 certain elements, make the `<shortdesc>` element required, and disallow topic nesting.

Specifically, the DITA architect wants to redefine the content model in the following ways:

- Remove `<abstract>`
- Require `<shortdesc>`
- Remove `<related-links>`
- Remove the `%task-info-types;` entity in order to 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 content to the "Element-Type Configuration Integration section:

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

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

5. They check their test topic to ensure that the content model is modified as expected.

1.4.4.2 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 that 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 by adding the following code:

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

<!ENTITY % section-constraints-c-def
  PUBLIC "-//ACME//ELEMENTS DITA 2.0 Section Constraint//EN"
  "idRequiredSectionContraint.mod">
%section-constraints-c-def;
```

5. They check their test topic to ensure that the attribute list is modified as expected.

1.4.4.3 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 any other of the elements in the domain. They want to integrate this constraint into the document-type shell for task.

Specifically, the DITA architect wants to redefine the content model in the following ways:

- Use `<b>` and `<i>`
- Remove `<line-through>`, `<overline>`, `<sup>`, `<sup>`, `<tt>`, and `<u>`

1. The DITA 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 content in the "DOMAIN CONSTRAINT INTEGRATION" section:

```
<!-- ============================================================= -->
<!--                   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. They check their test topic to ensure that the content model is modified as expected.

1.4.4.4 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.

1.4.4.5 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

File name	What it constrains	Details
`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 typically is located in the "DOMAIN CONSTRAINT INTEGRATION" section, and it must be integrated before the "DOMAIN ENTITIES" section. The other constraints typically are located in the "ELEMENT-TYPE CONFIGURATION INTEGRATION" section, and the order in which they are listed does not matter.

1.4.5 Examples: Constraints implemented using RNG

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

1.4.5.1 Example: Restrict the content model for the <topic> element using RNG

In this scenario, the DITA architect for Acme Incorporated wants to redefine the content model for the topic document type. They want to omit certain elements, make the `<shortdesc>` element required, and disallow topic nesting.

Specifically, the DITA architect wants to redefine the content model in the following ways:

- Remove `<abstract>`
- Require `<shortdesc>`
- Remove `<related-links>`
- Remove the `task-info-types`pattern in order to 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. They check their test topic to ensure that the content model is modified as expected.

1.4.5.2 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. They check their test topic to ensure that the content model is modified as expected.

1.4.5.3 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 any other of the elements in the domain. They want to integrate this constraint into the document-type shell for task.

Specifically, the DITA architect wants to redefine the content model in the following ways:

- Use `<b>` and `<i>`
- Remove `<line-through>`, `<overline>`, `<sup>`, `<sup>`, `<tt>`, and `<u>`

Note that 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.

1.4.5.4 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

1.4.5.5 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>
```

1.5 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.

1.5.1 Overview of expansion modules

Expansion modules enable information architects to include specialized attributes or elements in specific element types, without making the specialized attributes or elements 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 a value for the `@specializations` attribute must be defined.

1.5.2 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, an 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.

1.5.3 Examples: Expansion implemented using DTDs

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

1.5.3.1 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-domain 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 sectionDesc-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 it, the work is done.

1.5.3.2 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 attribute-list pattern 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.

1.5.3.3 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.

1.5.3.4 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
`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
`acme-SectionExpansion.mod`	`<section>`	Adds an optional `<sectionDesc>` element to `<section>`.
`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.

1.5.4 Examples: Expansion implemented using RNG

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

1.5.4.1 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">
        <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.

1.5.4.2 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.

1.5.4.3 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.

1.5.4.4 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">
        <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

A Aggregated RFC-2119 statements

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

Item	Conformance statement
001 (5)	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 XSD or Schematron.
002 (5)	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.
003 (5)	Document-type shells that are not provided by OASIS **MUST** have a unique public identifier, if public identifiers are used.
004 (5)	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.
005 (8)	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.
006 (8)	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.
007 (10)	Every DITA element (except the `<dita>` element that is used as the root of a ditabase document) **MUST** declare a `@class` attribute.
008 (10)	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.
009 (10)	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.
010 (10)	Authors **SHOULD NOT** modify the `@class` attribute. 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.
011 (11)	Each specialization of the `@props` and `@base` attributes **MUST** provide a token for use by the `@specializations` attribute.

Index