Extreme Markup Languages 2005®

Montréal, Québec
August 1-5, 2005

A unified type hierarchy for discourse
A potential direction for DITA 2

Erik Hennum
IBM

Abstract

A type hierarchy enables extension of general types by specialized types for more precise semantics, thus allowing for flexibility of design and processing. A formal type hierarchy for discourse can realize benefits similar to those already seen for type hierarchies in OO design and ontologies. Because schema languages declare elements and content models, discourse types may be defined by specifying relationships between more specific (often pragmatic) models and more general models. Because refinement of discourse models may entail changes in containment structure, the type relationships must declare a mapping between containment structures. A Design Annotation Specialization Language can be used to declare type relationships and can be embedded as annotation elements within XML Schema or RelaxNG files or maintained externally as an annotation file for DTDs. Using the declared type hierarchy, an instance of a discourse model can be processed against base types to inherit formatting or behaviors and can be generalized to base types for interoperability. The author is proposing these strategies to the OASIS DITA Technical Committee as a possible direction for a future release of DITA.

A Conference of IDEAlliance
Extreme Markup 2005 Languages®

A unified type hierarchy for discourse

A potential direction for DITA 2

Table of Contents

A Conference of IDEAlliance
Extreme Markup Languages® 2005

A unified type hierarchy for discourse

A potential direction for DITA 2

Erik Hennum

§ Benefits of a type hierarchy

Type hierarchies have had great success in the Object-Oriented Design, ontology, and other areas. For instance, an API for UI programming often defines a general UI component that serves as the base type for specialized UI components such as a button, list, or text box. Some benefits of defining a type hierarchy:

Shared design	In the UI example, the size dimensions and position coordinates can be defined for the base component and inherited by the specialized button, list, and text box.
Shared processing	The basic layout functions can be developed for the base component and applied to a specialized button, list, or text box.
Easier understanding	The base component expresses the commonality of the button, list, and text box.
Interoperability	A component developer can define a new type of specialized component such as a UI thermometer, and an application developer can use the unknown component in existing UI form logic because the unknown component can be treated as a base component.
Composability	An instance of a specialized type anywhere that allows an instance of the general type. This approach, called polymorphism in object-oriented systems, allows for design and processing flexibility without compromising the validity of designs.

A formal type hierarchy can realize similar benefits for discourse. By *discourse*, this paper means a discussion with sequential flow whose content is structured and classified by constructs such as paragraphs, tips, phrases, product names, and so on. These structuring and classifying constructs can be considered *types* defined in a markup languages. Thus, a formal type hierarchy establishes relationships between more general and more specialized constructs. For instance, a product name might be considered a special kind of phrase. Overall, a set of information conforming to a formal type hierarchy would be composed of discourse objects (each of which retains its type integrity) as opposed to an indefinite mix of namespaced content.

This paper proposes a strategy for building on the existing DITA type mechanisms to provide a more flexible type hierarchy. Thus, a review of the existing mechanism provides the background for the proposal.

§ Current DITA and substitution of specialized types in discourse

The DITA standard has two fundamental aspects:

- An architecture [DITA Architecture 2005] for strongly typed content objects with inheritance relationships between types.
- A set of general types [DITA Language 2005] and some initial specialized types that populate the architecture to support technical content (in particular, User Assistance) as well as other information deliverables that lend themselves to aggregation.

In the DITA architecture, a specialized type distinguishes a subset of the instances of a more general type. For example, we might notice that the instances of a general ordered list include chronological, structural, and task lists:

Table 1

Chronological list	Structural list	Task list
The lifecycle: - Four legs in the morning - Two legs in the afternoon - Three legs in the evening 	The management chain: - Sancho Panza - Don Quixote - Cervantes 	Edit the file: - Execute emacs. Emacs displays. - Open the file.

A DITA *specialization* recognizes a subset of instances with a common semantic by substituting a new element for the existing element when marking up those instances. For example, we might substitute a new `steps` element for the existing `ol` element for marking up the subset of ordered lists that have a task semantic:

Table 2

General type	Specialized type
  - Execute emacs. Emacs displays. - Open the file. 	<steps> <step><cmd>Execute emacs.</cmd> <stepresult>Emacs displays.</stepresult> </step> <step><cmd>Open the file.</cmd></step> </steps>

Besides declaring a more precise semantic, the substituted element can also add constraints to the general content model. In the example, the content model for the new `step` element can require a specialized `cmd` inline phrase to delimit the imperative content of the step.

By refining the markup semantics and restricting the content model, DITA specialization realizes the benefits of type hierarchies discussed earlier. The particular benefits for discourse include:

Better authoring
: The specialized markup language reinforces the semantics of the content instead of merely declaring the structure of the content. The constrained content models minimize semantic errors such as providing a step without instruction, reducing the need for low-level editorial review. The user experience for the author can take on some qualities of a wizard where a validating editor prompts the author for the next piece of content.

Easier processing
: The developer for the processing can work with well-defined input instead of having to handle cases that are valid in the markup but don't match the intent. In addition, because the markup declares a more precise semantics, the rules that match the markup declare a more precise semantic as well, making the purpose of the processing easier to understand. Effectively, the specialized markup becomes a better contract between the author and the developer. Finally, because the special instances remain valid general instances, the general processing still applies. The developer only needs to override the general processing where desirable.

Interoperability on a general base
: Because a specialized type restricts a general type to recognize a subset of its instances, every instance of the specialized type is guaranteed to be valid for the general type. As a result, the general element can always be substituted for the specialized element, a process known as *generalization*.

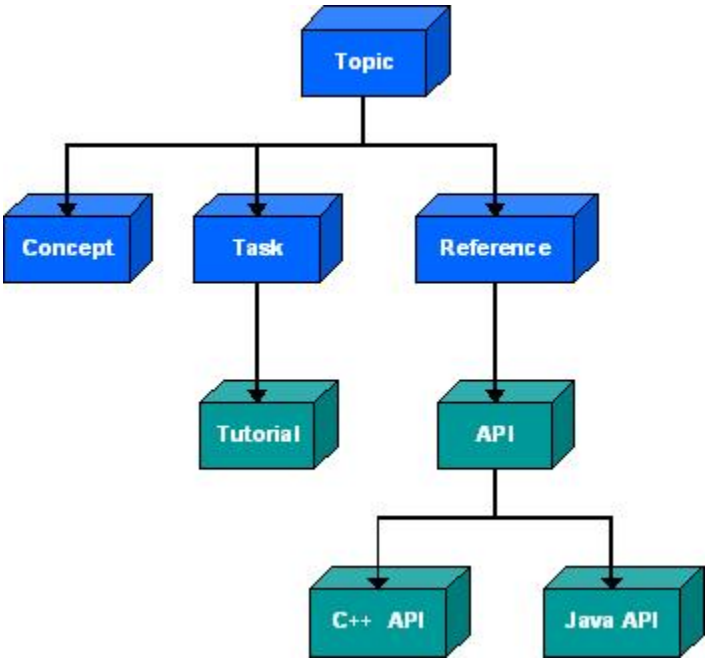
  Generalization of specializations has benefits over transforms between arbitrary document types because it provides a reliable, standard operation. By contrast, a transform between document types is an empirical exercise in finding an acceptable equivalent for every possible instance. Such transforms typically have special cases where the equivalence between the source and target element is strained. In addition, a new document type entails new transforms to every other known document type.

  The practical consequence of generalization is that content can be shared across communities with the semantics accepted generally by that community. The visual equivalent would be a set of Russian nesting dolls where the innermost doll has the narrowest community and the most precise

semantic and the outermost doll has the broadest community and the most general semantic. For instance, a telecommunications company could specialize to represent the unique knowledge culture of its company but still generalize to accepted telecommunications semantics when sharing content with other telecommunications companies and or generalize to generic technical semantics when sharing content outside the telecommunications industry.

The following illustrations shows the inheritance relationships for the core DITA topic types and some specialized topic types:

Figure 1: The type hierarchy for some topic type specializations



In XML Schema [XML Schema 2004], the DITA definition for type specialization can be implemented with a combination of restriction and substitution groups. The architecture also provides design patterns for implementing the DITA architecture using entities in DTDs.

The DITA architecture identifies types in XML using schema-agnostic architectural attributes. In particular, the schema defines a class attribute by which each element identifies its type ancestry. For instance, when the class attribute is normalized in the instance, the steps element resembles the following example:

```
<steps class="- topic/ol task/steps ">
```

In the value of the class attribute, the initial hyphen distinguishes the type of specialization (topic as opposed to domain). The value can contain any number of ancestry tokens, which are value pairs that consist of the identifier of the module supplying the type, a separating solidus, and the name of the supplied type. Thus, in the example above, the class attribute declares that the current element:

1. Is provided through topic specialization.
2. Has a base type of ol from the topic module.
3. Has a type of the steps from the task module.

For XSLT processing, DITA uses a standard idiom to match elements based on a type listed in the class attribute rather than on the element name. As a result, processing rules written against base types match

specialized types as well by default. For instance, the processing rule for ordered lists also processes task steps because the class attribute contains the "topic/ol" string:

```
<xsl:template match="*[contains(@class, ' topic/ol ')]">
```

A specialized type can override the base rule, however, in an XSLT module that imports the base processing:

```
<xsl:template match="*[contains(@class, ' task/steps ')]">
```

In the same way that classes can be assembled for an application, the type and processing modules for a specialization are pluggable. That is, the specialization modules can be assembled to create application-specific document types. For instance, topic type specializations that define content structure can be combined with domain specializations that define vocabularies for specific subject areas.

This pragmatic approach has worked well. DITA content has been authored and deployed by both large and small organizations for web sites, help systems, books, papers, and so on. The base types have been extended for communities, problem domains, and organizational cultures including special vocabularies for the telecommunications industry, reference types for API libraries, and so on.

§ Design motivations for more flexible specialization

As designers have gone deeper into the DITA architecture, the benefit of additional capabilities has become clear. Designers want the flexibility to meet goals such as the following through their specializations:

Element aliasing	Conforming to community culture by using a different label for an existing element. For instance, different communities may prefer the "p," "para," or "paragraph" label (or a localized label) but the paragraph element should be processed in the same way.
Content model restriction	Simplifying the discourse instances by restricting the model of existing elements. For instance, a community might prefer to restrict the content models of block elements to prevent nesting of blocks (and thus paragraphs can't nest lists).
Element addition	Adding new elements to content models that introduce a new kind of content without a semantic ancestor. For instance, a code example might contain a structure that provides parameters for checking out the source file and extracting the code fragment.
Attribute addition	Adding new properties to existing elements without losing interoperability with others. Many problem domains have special metadata that must be represented in the document instance. For instance, warning notes for hardware might have an attribute that identifies the regulation that motivates the warning.
Attribute specialization	Refining the attribute semantic by renaming and restricting the value. For instance, an organization might want to specialize the platform attribute to provide an enumeration of operating systems or to distinguish separate machine, programming language, and operating system attributes.
Token specialization	Adding tokens to an enumeration to indicate a semantic subset of an existing token. For instance, the audience element has a type attribute with an administrator token, which might legitimately have system administrator and system operator specializations.
Contextual specialization	Substituting a specialized element within a specific, existing content model rather than globally. For instance, a community might need to introduce legal inline phrases that should only appear in the content models of admonition notes.

These design requirements strain the capabilities of the existing approach to specialized typing in the DITA architecture and suggest a reexamination of discourse typing. The challenge for this reexamination is to meet as many of these requirements as possible without losing the integrity of the type hierarchy or the pragmatic benefits of the existing approach.

§ Perspectives on discourse typing

A number of initiatives and investigations are relevant to discourse typing:

- Ontological models, which have a standard representation in the Web Ontology Language [OWL], define a set of classes that constitute a type hierarchy. The literal content in the instances of these classes, however, are data values rather than discourse text. That is, because the Web Ontology Language is designed for a machine-processable web, it doesn't provide a useful representation for discourse.
- Architectural Forms [HyTime] provides for mappings between a client document type and a meta document type, but this relationship constitutes a subset relationship rather than a type hierarchy. In particular, a single client document type can have any number of meta document types.
- The DocBook and TEI communities have defined a way to substitute appropriate elements in a context using RelaxNG [Rahtz et al 2004]. As the cited paper notes, however, this solution doesn't address the larger question of semantic and structural validity of those substitutions
- The Datatype Library Language (DTLL) initiative championed by Jeni Tennison [DTLL] defines a type hierarchy mechanism for RelaxNG, but for atomic values rather than discourse semantics.
- The Markup Semantics initiative of the BECHAMEL project recognizes the value of a type hierarchy for markup [Renear et al. 2002] as part of a set of limitations on the semantic transparency of markup but solves these limitations through more sophisticated strategies of interpretation [Dubin and Birnbaum 2004]. This paper suggests that a subset of pragmatic benefits can be realized at the markup level through additional annotation.
- The Secondary Structuring initiative [Sasaki 2004], like the approach suggest in this paper, seeks to identify semantic equivalence between markups but at arbitrary points of equivalence rather than through the design discipline of a type hierarchy.
- The Schematron notion of abstract patterns [Ogbuji 2004] provides for the concept of a general type but provides for only a single generation of derivation and requires that the general type be abstract.
- Object Oriented Design [OOP] has a well-defined strategy for type specialization. Because this strategy has been so successful, is important to consider carefully the extent to which it applies or doesn't apply for any system of type specialization including discourse typing.

§ The Object Oriented perspective and addition of properties to discourse

In the Object Oriented approach, a specialized class inherits all of the properties of the base class. These properties are often accessed through extensible behaviors, but that nuance doesn't alter the basic principle. The specialized class introduces variation by adding new properties (in XML Schema parlance, through extension by addition).

The following example shows the specialization of a class for generic structural nodes to define a class for tree nodes.

Table 3

General class	Specialized class

`Node`
`  data: Object`
`  next: Node`
	`TreeNode`
	`  data:  Object`
	`  next:  Node`
	`  parent: TreeNode`

A program can treat objects of the specialized type as objects of the general type through a casting operation that hides the added properties. For instance, the parent property of the TreeNode class isn't visible when a program is treating a TreeNode object as a Node object. Such casting makes it easy for a program to process objects in shared or distinct ways as appropriate.

Adding properties to a discourse object can be important for metadata processing and for hybrid documents that include record data as well as discourse text. For instance, a lab report type might need metadata about the institution that produced the report or record data expressing the raw data analyzed in the report. If added content is restricted to properties outside the main flow of discourse, the standard object-oriented strategy of hiding the additions can maintain the validity of the discourse when generalizing to a type that doesn't have the added properties. That is, after the added properties are hidden, the remaining discourse remains a valid instance of the general type.

One strategy is to put the addition inside a processing instruction that occupies the position of the hidden content during generalization. It should even be possible to add properties to a specialization of an empty element because, when generalized, the empty element should be able to contain the processing instruction for the hidden addition.

Table 4

Special type	General type after hiding the addition

`<fig>`
`  <title>Quantum`
`  engines</title>`
`  <labloc>B52-FA-RA13</lablo`
`  <image href="qengines.jpg"`
`</fig>`
	`<fig>`
	`  <title>Lab report</title>`
	`  <?HIDDEN-ELEMENT <labloc>B52-FA-RA13</labloc> ?>`
	`  <image href="qengines.jpg"/>`
	`</fig>`

Thus, addition complements substitution by supporting extensible properties about discourse.[1]

§ Semantic precision through subdivision of discourse

The discourse text itself, however, doesn't lend itself to specialized typing by addition. Specialized typing of discourse adds more precise semantic markup without adding to the discourse text.

For instance, compare a fragment of discourse with progressively more specialized markup:

Table 5

Discourse text	Discourse structure	Discourse semantic structure

`<content>`
`Execute emacs.`
`Emacs displays.`
`Open the file.`
`</content>`
	`<ol>`
	`<li>Execute emacs.`
	`  <p>Emacs`
	`  displays.</p>`
	`</li>`
	`<li>Open the file.</li>`
	`</ol>`
		`<steps>`
		`  <step>`
		`  <cmd>Execute`
		`  <cmdname>emacs</cmdname>.`
		`  </cmd>`
		`  <stepresult>`
		`  <wintitle>Emacs</wintitle>`
		`  displays.`
		`  </stepresult>`
		`  </step>`
		`  <step>`
		`  <cmd>Open the file.</cmd>`
		`  </step>`
		`</steps>`

Increasing the semantic precision requires additional levels of containment and more fine-grained pieces of text. That is, the semantic precision results from subdivision of the existing value rather than addition of new values as in the Object Oriented approach.

The gap between the object-oriented perspective and the discourse perspective becomes evident when examining the DOM (Document Object Model). The markup above would result in the following DOM trees:

Table 6

Text DOM	Structure DOM	Semantic structure DOM

`<content>`
`  "Execute emacs.`
`  Emacs displays`
`  Open the file.`
	`<ol>`
	`  <li>`
	`  "Execute emacs."`
	`  <p>`
	`  "Emacs displays."`
	`  <li>`
	`  "Open the file."`
		`<steps>`
		`  <step>`
		`  <cmd>`
		`  "Execute "`
		`  <cmdname>`
		`  "emacs"`
		`  "."`
		`  <stepresult>`
		`  <wintitle>`
		`  "Emacs"`
		`  " displays."`
		`  <step>`
		`  <cmd>`
		`  "Open the file."`

If the object-oriented strategy of extension by addition explained semantic refinement, we would be able to revert to a more general semantic by pruning the branches added to the DOM tree. Applied to the third case above, pruning the `cmd` and `wintitle` branches would prune most of the text as well, producing the following result:

Table 7

Generalizing from 3rd to 2nd DOM by pruning added branches and renaming elements	Generalization from 3rd to 2nd case through DOM-based pruning
      " displays." 	 " displays."

As with added properties, a subdivision container element must be hidden in the general form. What's different is that the contents of the subdivision container must not be hidden. In the following example,

Table 8

Special type with subdivision	General type after hiding the subdivision
<productFeatures> <productModule function="editor"> <productFeature>XML support</productFeature> <productFeature>Styled</productFeature> </productModule> <productModule function="composer"> <productFeature>Dynamic </productFeature> <productFeature>Conditional </productFeature> </productModule> </productFeatures>	 <?HIDDEN-CONTAINER-START <productModule function="editor"> ?> - XML support - Styled <?HIDDEN-CONTAINER-END </productModule> ?> <?HIDDEN-CONTAINER-START <productModule function="composer"> ?> - Dynamic - Conditional <?HIDDEN-CONTAINER-END </productModule> ?>

Elements introduced by subdivision can remove the need for textual delimiters. For instance, dates are commonly expressed with solidus or hyphen separators between the fields of the date, which become unnecessary if those fields are declared by the markup:

Table 9

General date with textual delimiters	Specialized date with subdivision elements
<date>2005-06-24</date>	<date> <year>2005</year> <month>06</month> <day>24</day> </date>

An address provides another example because the linebreaks, spacing, and punctuation that provide a parseable format for a single-value address are no longer part of the subdivided fields of the address.[2] These textual delimiters pose an additional requirement for generalization from a specialized subdivision. Textual delimiters may need to be inserted during generalization and deleted during respecialization.

Because semantic precision is obtained through subdivision of the text, discourse typing differs from object-oriented typing. A complete strategy for specializing discourse types should include not only substitution of specialized types and addition of properties but also subdivision of content models by new elements. In other words, specialization of discourse must support changes in containment within the specialized XML representation.

§ Variation in XML expression

In practice, an actual document type often makes accommodations for authoring or processing considerations that are unrelated to the fundamental semantics or structure of the discourse model. Such pragmatic accommodations can include the following:

- Specifying a sequence for properties so authors have a more limited set of options in a validating XML editor even if the properties have no actual sequential relationship.
- Adding grouping for the convenience of processing or so authors can consider a smaller set of options.

- Removing semantic containment so authors see more text than markup because, from an author's perspective, too much markup looks like noise.
- Altering the semantics or structure for reasons of community culture -- in particular, preserving legacy markup for the sake of existing processing or author familiarity.

In short, these accommodations can result in variant content models. The resulting XML document types define one possible expression of a fundamental discourse model.

§ Modelling inheritance and composition relationships for types

To summarize the preceding analysis, the goals of the DITA community for more flexible design can be met through more precise control of substitution, addition of properties, subdivision of containment, and tolerance for variance in the XML expression. Identifying the type relationships between the elements, attributes, and content models provided by general and specialized (often pragmatic) document types calls for a model that is independent of and simpler than the schemas for the document types.

The model for the type hierarchy (the IS-A relationships) can be a straightforward tree[3]. The following example gives a simplified view of some basic discourse types and some specialized types for task content:

```
DiscourseFragmentType
    NullType
    TextType
    ValueType
        IDType
    DiscourseUnitType
        PhraseType
        BlockType
            ParagraphType
            TitleType
        ListType
            OrderedListType
                TaskStepsType
            UnorderedListType
        ListItemType
            TaskStepType
        DivisionType
            SectionType
                TaskContextType
        BodyType
```

The model for the composition relationships (the PART-OF relationships) poses more of a challenge. To express the relationship of the specialized composition to the base composition, we must be able to address parts of the composition structure. One possible approach for simplifying this problem would be, first, to define a content option as a union of content types. The following example defines some typical content options:

```
PhraseOption:       TextType | PhraseType
BlockPhraseOption:  PhraseOption | BlockType | ListType
```

The model could then define composition relationships as assertions of cardinality on either a content type or content option. A content type can only appear once (whether explicitly or in a content option) within a composition definition. While this constraint might seem severe, XML content models with mixed text in DTD or with sequential or regular repeating elements conform to this restriction. Because the content models for the base DITA elements conform, existing DITA specializations can also be modelled in this way.

The following example (using UML [UML] notation for cardinality) defines the composition of a simple section consisting of an optional id, an optional title, and block, list, phrase, or text content as well as the composition of a specialized section describing task context that removes the optional title:

```
SectionType:        0..1 IDType
                    0..1 TitleType
                    0..* BlockPhraseOption

TaskContextType:    0..1 IDType
                    0..* BlockPhraseOption
```

The purpose of this model is not to provide a new schema language for validating document instances or to capture all of the information provided by a schema language but, instead, to identify type relationships. In particular, there's no need to model the sequence of an element's content model or the distinction between elements and attributes.

This approach identifies specialization by adding new content types, content options, and composition definitions to the model. The composition definition for a specialized content type must be a variant on the composition definition of the base content type. A variant composition definition could differ from the base definition in the following ways:

- Narrow the range of the cardinality of any composed item within the bounds in the base composition definition.

  In the example, the specialized section composes zero titles.
- Replace any composed item with a semantic subset of the composed item.

  A composed content type can be replaced with a single specialized content type or with a content option that includes several specialized content types. In the example, a specialized title could be substituted for the base title.

  Similarly, a composed content option can be replaced with a single content type (or specialization thereof) from the option or with a new option that lists a subset of the content types (or specializations thereof). In the example, the content option for block, list, and phrase text could be replaced with the block type.
- Replace any composed item with multiple composed items that constitute a semantic subset and that have an aggregate cardinality that's the same or narrower than the original cardinality.

  For example, an unordered list composed of zero or more list items could be specialized as a product features list with zero or more critical features and zero or more important features.
- Add a composed item.

  In the example, the task context could add a product identifier as part of the context. On generalization, composed items introduced through extension by addition would be hidden.
- Replace a composed item with a proxy content type to subdivide the content.

  The composition definition for the proxy content type must be a valid replacement for the composed item. For example, a product features list could introduce a product module proxy that groups the subset of critical and important features for each product module. On generalization, the proxy composed item would be hidden but not its contents.

  The text content type can be proxied, which allows text to be grouped in units that are hidden on generalization without hiding the textual content.

This model could be represented in OWL, which offers some significant processing options.

§ The XML expression of the type relationships model

To realize the type relationship model, an XML document does the following:

- Selects an attribute or element representation for content types that should be manifest in the XML vocabulary.

  A content type cannot be represented as an attribute if it has cardinality larger than 1 in any composition definition within the model.
- Assigns names to the content types.

  A content type might have multiple named elements in an XML document type (for instance, to support legacy associations).
- Defines element or attribute groups (in XML Schema terms) for the content options.
- Represents the composition definitions as either attribute value definitions or as element content models, imposing a sequence on the composed items.

A content type and a specialized type might have the same name in different XML vocabularies. For instance, some DITA adopters would like to change the composition definition of the paragraph type, either to exclude phrase and text content or to add metadata values. Such adopters would be able to define a specialized paragraph type with the appropriate composition definition but still give their content type the element name "p" within their document type.

§ Annotating designs with type relationships

To declare the type relationships for the model, a more expressive mechanism than architectural attributes (which provide simple values) is desirable. Given the representation of other aspects of design in XML, the natural approach is to represent type relationships in XML as well.

In many cases, the schema for the XML document type has a close parallel with the type model. Thus, it's important for clarity and maintainability to be able to write the type declaration close to the schema. The existing DITA practice of declaring type ancestry with architectural attributes provides a precedent, in decoupling the type declaration from the schema declaration but maintaining the type declaration close to the schema declarations. In addition, a goal is to extract the type declaration where possible from the schema notation using a process sensitive to the schema language. At one level, this approach leverages the schema declarations as a shorthand. At another, we can blur the distinction between the type and the element as the XML expression of the type for the typical case but recognize the distinction when we need the precision.

Because the type declarations can annotate a schema, let's call the vocabulary for type declaration the Design Annotation Specialization Language or DASL. DASL can follow the RDF principle of combining the namespace and the element or attribute name to produce a globally unique identifier for assertions about the element or attribute.

Some potential core statements for DASL follow:

dasl:type	Identifies the type within the model that corresponds to the element or attribute. As a shorthand, this statement can be omitted to use the element identifier as the type identifier.
dasl:baseType	Identifies the general type for the type corresponding to the element or attribute. If the general type is declared in some other resource available for processing, this statement can be omitted. Where the base type is equivalent to an element type (that is, where the shorthand for dasl:type has been used to declare the base type), this statement identifies that element. Thus, using the shorthand, the DITA `steps` element could identify the DITA `ol` element as a base type. The shorthand base type for an element could be an attribute and visa versa (with the exception of the root element of the discourse object). A shorthand base type could be provided by the same document type module.
dasl:option	Defines a content option as a union of content types or other content options for use in defining content composition. In XML Schema, this statement might annotate an element or attribute group.
dasl:composition	Defines content composition for the type as a list of composition items and, in the context of the XML document type, declares the sequence of the composition items. If the content composition is defined in another resource or if the content composition can be determined from the declaration of the content model in the schema language, this statement can be omitted.
dasl:compositionItem	Identifies a composed content type or content option and defines the cardinality for the composed item. The statement can also identifies the composed item as an unchanged item, a replacement, an addition, or a proxy for a composed item within the base content composition. In the case of a proxy, the statement can declare textual delimiters to use for the proxied content on generalization. It might be possible for the statement to identify a composed item as virtual within the specialized content composition, in effect, flattening a base container element. For instance, a definition list element might provide a virtual list item that implicitly groups the definition term and data elements (as in the XHTML document type). The statement would have to provide a grouping instruction to assign the appropriate part of the specialized content

model to the base content position during generalization. The grouping instruction might resemble Schematron abstract patterns.

It might be possible for the statement to specify fixed content for a composition item. The fixed content should be inserted into a document instance during generalization.

§ Embedding type annotations in schemas

The DASL declarations of type relationships would be possible to embed in the schema definition as annotations. Each annotation applies to the element or attribute whose schema definition contains the DASL declaration.

For instance, here is a skeleton of a Relax NG annotation:

```
<rng:element name="ol">
  <dasl:design
        xmlns:dasl="http://ibm.com/experimental/dasl/design"
        xmlns:discourse="http://ibm.com/experimental/dasl/discourse"
        xmlns:topic="http://oasis-open.org/dita/2005/topic">
    <dasl:type href="discourse:OrderedListType"/>
    ... other DASL design declarations about the type ...
  </dasl:design>
  ... Relax NG element and attribute content definition ...
</rng:element>
```

Similarly, here is a skeleton of an XML Schema annotation:

```
<xs:complexType name="ol.class">
  <xs:annotation>
    <xs:appinfo source="http://ibm.com/experimental/dasl/design"
          xmlns:dasl="http://ibm.com/experimental/dasl/design"
          xmlns:discourse="http://ibm.com/experimental/dasl/discourse"
          xmlns:topic="http://oasis-open.org/dita/2005/topic">
      <dasl:type href="discourse:OrderedListType"/>
      ... other DASL design declarations about the type ...
    </xs:appinfo>
  </xs:annotation>
  ... XML Schema element and attribute content definition ...
</xs:complexType>
```

The DASL declarations of type relationships could also be maintained outside of the schema definition, which would be useful for DTD definitions, which have no standard annotation mechanism. In addition, this approach would be useful where the schema definition is maintained by someone other than the type relationship annotator. The schema construct annotated by the definition could be declared explicitly as in the following example:

```
<dasl:types
        xmlns:dasl="http://ibm.com/experimental/dasl/design"
        xmlns:discourse="http://ibm.com/experimental/dasl/discourse"
        xmlns:topic="http://oasis-open.org/dita/2005/topic">
    <dasl:element href="topic:ol">
        <dasl:type href="discourse:OrderedListType"/>
        ... other DASL design declarations about the type ...
    </dasl:element>
    ...
</dasl:types>
```

A designer could choose to maintain the type relationships in the embedded form in one schema language and generate an external DASL representation for use with other schema languages.

Using the type relationships and a base schema definition, it should be possible to generate a specialized schema definition. The specialized schema would preserve the same attribute and element assignments and content sequences as the base schema. This capability would allow the designer to work by specifying the deltas on a base design, which would greatly reduce the maintenance effort on specialized designs.

§ Representing and locating the model for processing

From the DASL statements and the annotated schema, the type hierarchy can be expressed naturally as an XML tree:

```
<dasl:types
      xmlns:dasl="http://ibm.com/experimental/dasl"
      xmlns:discourse="http://ibm.com/experimental/dasl/discourse"
      xmlns:topic="http://oasis-open.org/dita/2005/topic"
      xmlns:task="http://oasis-open.org/dita/2005/task">
  <dasl:type id="discourse:ListType">
    <dasl:type id="discourse:OrderedListType">
      <dasl:element name="html:ol">
```

```
      ...
    </dasl:element>
    <dasl:element name="topic:ol">
      ...
    </dasl:element>
    <dasl:type id="task:TaskStepsType">
      <dasl:element name="task:steps">
        ...
      </dasl:element>
    </dasl:type>
  </dasl:type>
  <dasl:type id="discourse:UnorderedListType">
    <dasl:element name="topic:ul">
      ...
    </dasl:element>
  </dasl:type>
  ...
 </dasl:type>
 ...
</dasl:types>
```

Processing can climb the tree to traverse the type hierarchy from specialized to general types. The composition declarations can be represented in similar structures that are convenient for processing.

In the same way that a document instance can use an attribute to point to a schema definition for the document instance, the root element for a discourse object could point to the assembled type declarations:

```
<task:task
      xmlns:topic="http://oasis-open.org/dita/2005/topic"
      xmlns:task="http://oasis-open.org/dita/2005/task"
      xmlns:dasl="http://ibm.com/experimental/dasl"
      dasl:typeref="http://ibm.com/experimental/dasl/task/type.xml"
      ...>
```

For authoring convenience, this architectural type reference attribute could be set by default in the schema language.

§ Processing discourse objects against types

The type relationships aren't used to validate the XML expression, which remains the task of a schema language processor. Instead, the type relationships are used to:

- Match processing rules with a typed discourse object
- Compare two typed discourse objects
- Generalize and respecialize the XML expression of a typed discourse object
- In general, interpret the semantics of a typed discourse object

In particular, for base processing to match specialized instances, the processor must read the type hierarchy. A processor could read the external type hierarchy using the architectural type reference attribute. As an alternative for more efficient repeated processing, a preprocess might normalize the document instance by wrapping the content and the type hierarchy in a container element:

```
<dasl:typedContent
      xmlns:dasl="http://ibm.com/experimental/dasl"
      xmlns:discourse="http://ibm.com/experimental/dasl/discourse"
      xmlns:topic="http://oasis-open.org/dita/2005/topic"
      xmlns:task="http://oasis-open.org/dita/2005/task">
  <dasl:types xml:id="types">
    <dasl:type id="discourse:ListType">
    ...
  </dasl:types>
  <task:task dasl:typeref="#types" ...>
    ...
  </task:task>
</dasl:typedContent>
```

To enable convenient processing in XSLT, a library might define a key on the type identifier and provide an isType() function and other, complementary functions to use in matching instances (whether using the declared type or a specialized type). For example, the following idiom matches a title in any XML representation, whether of the base type or a specialized type and even whether an element or attribute:

```
<xsl:template match="node()[dasl:isType(.,'discourse:TitleType')]">
```

§ Generalizing discourse objects

A type system requires that instances of a specialized type can be treated as instances of the base type (sometimes known as casting). The previous section indicated how base processing might match

specialized instances (in XSLT, with an isType() library function). It can also be useful to serialize a specialized discourse object as an XML expression of the base type. For instance, the discourse object might be exchanged with someone who is not adopting the specialized type, or the specialized type might be retired.

The input to the generalization process would consist of the input discourse object, the type model for the input, the type model for the output, and an identification of the target document type. The generalization process would traverse the discourse object, comparing the two type hierarchies to find the lowest point of agreement and expressing the content in the appropriate XML representation. In particular, the generalization process must make the following changes:

- Renaming substitutions.
- Hiding the start and end but not the content of subdivision containers.
- Hiding the entire branch for added content.
- Changing the form of attributes and elements.
- Rearranging the content sequence.
- Inserting fixed content.

As noted previously, content can be hidden in processing instructions. The generalized serialization of the instance must be possible to respecialize. The process could use the processing instructions embedded in the instance and the type declarations for the document type to restore the original form of the instance.

§ Generic discourse types and non-DITA specializations

As hinted in the previous examples, the type hierarchy shouldn't stop with the existing DITA general types. Instead, the type hierarchy can support a much broader range of instances by providing a more general declaration of standard discourse types such as list, table, block, and in-line phrase. Because the standard discourse types underlay many document types, it may be possible to recognize some existing document types with close similarity to DITA as specializations and thus share a unified type hierarchy. As always with a shared type hierarchy, the advantages would include interoperability in general and shared processing in particular.

For instance, compare the basic representation of a unit of discourse in XHTML and DITA:

Table 10

XHTML	DITA

<pre><html>
  <head>
  <title>Editing</title>
  <meta
  name="publisher"
  content="ABC"/>
  </head>
  <body>
 	
   - Execute emacs.
  Emacs displays.
 	
   - Open the file.
 	
  </body>
</html></pre>
	<pre><topic>
	  <title>Editing</title>
	  <prolog>
	  <publisher>ABC</publisher>
	  </prolog>
	  <body>
	
	   - Execute emacs.
	  Emacs displays.
	
	   - Open the file.
	
	  </body>
	</topic></pre>

The differences in the two markup languages reflect legitimate differences in their primary goals:

XHTML — The discourse object participates in a distributed hypertext system, so the markup first provides the browser with the properties about the discourse (the head) and then with the discourse itself (the body).

DITA — The discourse object is aggregated by reference into many different navigable information sets, so the markup distinguishes the referenceable label (the title) from the properties (the prolog) and discourse (the body).

The fundamental model of a discourse object, however, the same in both markup languages. The two languages merely express the model in different ways. That is, the same mechanisms for expressing substitution, addition, subdivision, and variation between DITA types can also serve to identify XHTML and the base DITA topic as specializations of a more general, shared discourse type.

For a sample, here is the DASL declaration to express the commonality of the ordered list type in XHTML and DITA:

```
<dasl:types
      xmlns:dasl="http://ibm.com/experimental/dasl"
      xmlns:discourse="http://ibm.com/experimental/dasl/discourse"
      xmlns:html="http://www.w3.org/1999/xhtml"
      xmlns:topic="http://oasis-open.org/dita/2005/topic">
  <dasl:type id="discourse:ListType">
    <dasl:type id="discourse:OrderedListType">
      <dasl:element name="html:ol">
        ...
      </dasl:element>
      <dasl:element name="topic:ol">
        ...
      </dasl:element>
    </dasl:type>
    ...
  </dasl:type>
  ...
</dasl:types>
```

Because most Wiki [Wiki] markup languages are a subset of XHTML, Wiki text can be seen as a non-XML serialization of a very simple discourse object:

```
= Editing
1.  Execute emacs.
    Emacs displays.
2.  Open the file.
```

The fundamental discourse models of the DocBook `section` element and DITA topic are also similar enough to raise the possibility of a common type ancestry.

While recognizing common discourse models has the potential for considerable benefit, the overall goal remains realizing the benefits of a type hierarchy through disciplined specialization of types rather than general interoperability between arbitrary document types. The plausibility of incorporating an existing document type through specialization into the type hierarchy depends entirely on the degree to which the existing document type reflects a common type model. For instance, XBRL[XBRL] clearly represents financial data rather than discourse.

§ Summary

This paper has presented a strategy for improving the capabilities of DITA extensibility. In particular, the paper has argued the benefits of a unified type hierarchy, the importance of containment changes for specialization of discourse, and the need to decouple the declaration of type relationships from the schema declarations that validate the XML expression of those types. We plan to submit this strategy for the consideration of the OASIS DITA Technical Committee. Discussion and refinement of the strategy by interested parties there and generally is welcome and important.

Notes

1. It's worth noting that Object-Oriented languages have more recently incorporated notions of extension by substitution. For instance, the Java feature of *generics* is a type substitution mechanism.
2. From the perspective of subdivision, the most general form of any discourse object might be a purely textual object with no markup but embedded newlines, spaces, and other textual delimiters.
3. An issue for investigation is whether it might be desirable to express a complex semantic through multiple base type relationships, though possibly with a single IS-A parent.

Acknowledgements

I am especially indebted to Michael Priestley for generous, thought-provoking conversation and correspondence about specialization. Additional thanks go to the DITA OASIS Technical Committee, to Indi Liepa, Sirpa Ruokangas, and their colleagues at Nokia as well as to Dave Schell, Don Day, Eric Sirois, John Hunt, Nancy Harrison, and other colleagues at IBM for the discussions that helped stimulate this paper.

Bibliography

[DITA Architecture 2005] *OASIS DITA Architectural Specification*, Michael Priestley, editor, OASIS Committee Draft 01 First Edition, 17 February 2005, **http://xml.coverpages.org/DITA-CD11428-ArchSpec.pdf**

[DITA Language 2005] *OASIS DITA Language Specification*, Michael Priestley, editor, OASIS Committee Draft 01 First Edition, 17 February 2005, **http://xml.coverpages.org/DITA-CD11428-LangSpec.pdf**

[DTLL] Jeni Tennison, *Datatype Library Language (DTLL)*, **http://www.jenitennison.com/datatypes/DTLL.html**

[Dubin and Birnbaum 2004] David Dubin and David J. Birnbaum, *Interpretation Beyond Markup*, paper presented at Extreme Markup Languages 2004, Montreal, August 2004, **http://www.mulberrytech.com/Extreme/Proceedings/html/2004/Dubin01/EML2004Dubin01.html**

[HyTime] *ISO/IEC 10744:1997: Information processing - Hypermedia/Time-based Structuring Language (HyTime)*, second ed. International Organization for Standardization, Geneva, May 1997, appendix A.3 Architectural Form Definition Requirements, **http://www.y12.doe.gov/sgml/wg8/docs/n1920/html/clause-A.3.html**

[Ogbuji 2004] Uche Ogbuji, *Discover the flexibility of Schematron abstract patterns*, DeveloperWorks, 8 Oct 2004, **http://www-106.ibm.com/developerworks/xml/library/x-stron.html**

[OOP] Wegrzanowski et al, *Object-oriented programming*, Wikipedia, 2005, **http://en.wikipedia.org/wiki/Object-oriented_programming**

[OWL] *OWL Web Ontology Language Overview*, Deborah L. McGuinness and Frank van Harmelen, editors, W3C Recommendation, 10 February 2004, **http://www.w3.org/TR/owl-features/**

[Rahtz et al 2004] Sebastian Rahtz, Norman Walsh, and Lou Burnard, *A unified model for text markup: TEI, Docbook, and beyond*, paper presented at XML Europe 2004, Amsterdam, April 2004, **http://www.idealliance.org/papers/dx_xmle04/papers/03-08-01/03-08-01.html**

[Renear et al. 2002] Allen Renear, David Dubin, C. M. Sperberg-McQueen, and Claus Huitfeldt, *Towards a Semantics for XML Markup*, paper presented at DocEng 2002, 8 November 2002, McLean, Virginia, **http://wam.inrialpes.fr/people/roisin/mw2004/Renear.pdf**

[Sasaki 2004] Felix Sasaki, *Secondary Information Structuring - A Methodology for the Vertical Interrelation of Information Resources*, paper presented at Extreme Markup Languages 2004, Montreal, August 2004, **http://www.mulberrytech.com/Extreme/Proceedings/html/2004/Sasaki01/EML2004Sasaki01.html**

[UML] OMG, *Unified Modeling Language Specification*, Version 1.5 Recommendation, 3 March 2001, **http://www.uml.org/**

[Wiki] Sunir Shah et al, *WikiMarkupStandard*, MeatballWiki, 11 April 2005, **http://www.usemod.com/cgi-bin/mb.pl?WikiMarkupStandard**

[XBRL] Phillip Engel et al,*Extensible Business Reporting Language (XBRL)*, 2.1 Recommendation, 25 April 2005,**http://xbrl.org/SpecRecommendations/**

[XML Schema 2004] *XML Schema Part 1: Structures*, Henry S. Thompson, David Beech, Murray Maloney, and Noah Mendelsohn, editors, W3C Recommendation Second Edition, 28 October 2004,**http://www.w3.org/TR/xmlschema-1/**

The Author

Erik Hennum
Information Architect
IBM
78 St. Mary's Avenue
San Francisco
CA
94112
ehennum@us.ibm.com

Erik Hennum works on the design and implementation of User Assistance for the IBM Storage Systems Group. For DITA, he has helped shape the principles of domain specialization. He participates in the OASIS DITA Technical Committee as a member.

Extreme Markup Languages 2005®
Montréal, Québec, August 1-5, 2005
This paper was formatted from XML source via XSL
by Mulberry Technologies, Inc.