ODF Modularization toward Document 2.0

Hisashi Miyashita, Daisuke Sato,
Hironobu Takagi, and Chieko Asakawa
IBM Research, Tokyo Research Laboratory

1. Introduction

For many purposes related to collaboration and liberation of information, Office documents are now aggressively used. Therefore, ODF should be convenient for wide range of developers to create situational applications with less effort. However, the specification of ODF is designed as a huge monolithic schema. That leads to two serious problems: 1) poor programmability and 2) lack of interoperability. First, since ODF is monolithic, developers must understand the whole of the specification to create ODF based applications. The whole ODF specification is so large and complex that supporting all the functionalities needs high development and maintenance cost. Second, since supporting all the ODF functionalities is very difficult, developers tend to support only a part of the functions. This may cause interoperability problems since systems using ODF for communication do not know what functionalities the target systems support.

Modularization can be a remedy for this situation, which is a standard technique to divide a huge schema. For example, XHTML™ Modularization 1.1 decomposes a large set of XHTML functionalities into about 30 modules, each of which is easy to mange and implement. Developers of XHTML can choose the required modules to fit their own purpose. By using XHTML Modularization, XHTML Basic is defined as a minimal set of modules that many Web clients such as portable devices can easily support.

Likewise, ODF should be decomposed into fine-grained modules. Each of them should be easy to understand and implement for users to make and maintain wide ranges of Office applications with less cost.

2. Issues

Along the line with that motivation, we attempt to modularize the ODF specification, and notice that the namespaces in ODF are appropriate units for modularization. The original ODF specification has 21 namespaces. The 14 namespaces of them are defined by the ODF specification and the 4 of them come from the other sources such as W3C specification and the 3 of them are used for compatibility with other XML vocabularies. That means 7 (3 + 4) namespaces are relatively well modularized since all of them fit with the other namespaces.

We examined the dependencies of the 14 ODF original namespaces by analyzing the ODF RELAX NG schema We extracted the element dependencies and categorized them by namespaces from the schema. In Figure 1, we show the element dependency graph of ODF namespaces.

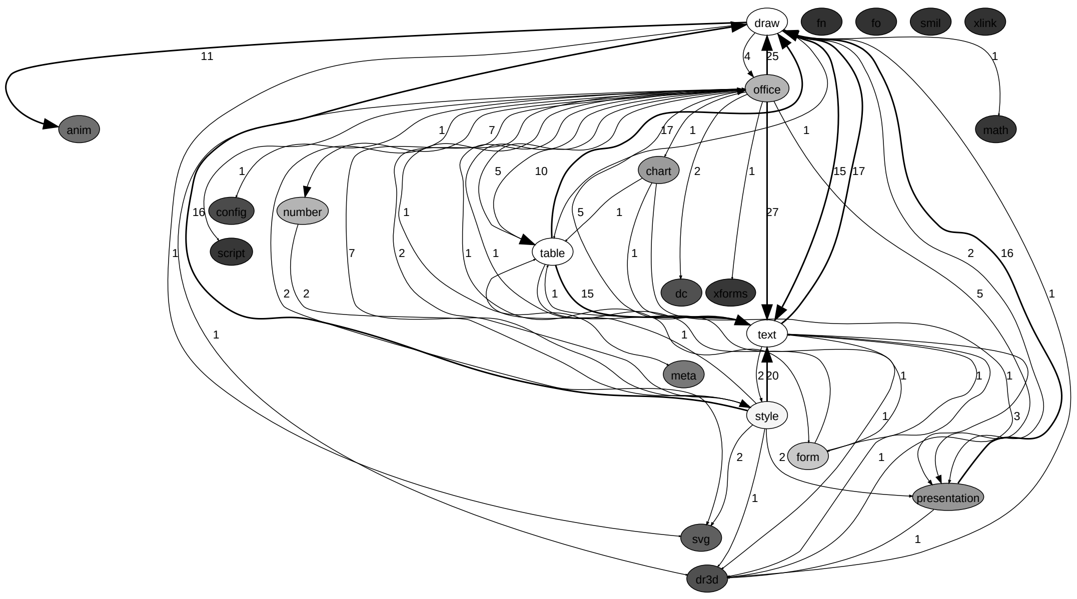

Figure 1: The dependency graph of ODF namespaces by elements. This graph is automatically plotted by Graphviz. Each vertex represents an ODF namespace. Vertices filled with lighter color means they have more element definitions. Each edge denotes dependencies of elements and the adjacent number means the number of source elements depended on target namespace.

As we can tell from the graph, these namespaces have many mutual dependencies and cannot be defined as distinct modules. For example, “form” namespace naturally uses “text” namespace since text is essential to describe forms. However, since “text” namespace also refers to “form” namespace, “text” and “form” namespaces are mutually dependent as follows.

```
<element name="text:database-name">
    <ref name="common-field-database-table"/>
    <text/>
</element>
<define name="common-field-database-table">
    <ref name="common-field-database-table-attlist"/>
```

```
        <ref name="common-field-database-name"/>
    </define>
    <define name="common-field-database-name" combine="choice">
        <optional>
            <attribute name="text:database-name">
                <ref name="string"/>
            </attribute>
        </optional>
    </define>
    <define name="common-field-database-name" combine="choice">
        <ref name="form-connection-resource"/>
    </define>
    <define name="form-connection-resource">
        <element name="form:connection-resource">
            <attribute name="xlink:href">
                <ref name="anyURI"/>
            </attribute>
            <empty/>
        </element>
    </define>
```

As the above schema definitions taken from ODF schema show, text:database-name refers to form:connection-data. Therefore, “text” namespace is dependent of “form” namespace, which means we cannot “text” namespace as a module without “form” namespace.

We found the mutual dependencies discussed so far are mostly caused by some nonessential dependencies between various modules. For example, “style” namespace refers to office:binary-data as follows.

```
    <define name="style-background-image">
        <optional>
            <element name="style:background-image">
                <ref name="style-background-image-attlist"/>
                <choice>
                    <ref name="common-draw-data-attlist"/>
                    <ref name="office-binary-data"/>
                    <empty/>
                </choice>
            </element>
        </optional>
    </define>
```

In this example, style:background-image element refers to office-binary-data, which is defined in “office” namespace as follows.

```
    <define name="office-binary-data">
        <element name="office:binary-data">
            <ref name="base64Binary"/>
        </element>
    </define>
```

Such kind of binary data is not specific to the core feature of “office” document but a generic data type. Actually, office-binary-data definition is referred by three namespaces, “draw”, “style” and “text.” Only by defining such a generic data type, those three namespaces depend on “office” namespace. That means if we use the features provided by “text” namespace, we have to refer at least “style” and “draw” namespaces because the features depend on these namespaces as well.

2.1 Dependencies by Elements

We carefully examined the dependencies in the schema of ODF 1.1 to specify what prevents the modularization. We categorized the dependencies into the following four cases.

E-A) Essential dependencies.
The dependencies are essential for ODF. Without these, ODF does not function well.

E-B) Backward dependencies to “office” namespace.
Since “office” namespace contains the document element, any dependency to “office” namespace is harmful for modularization. One exception is that draw:object refers to office:document. This dependency is essential because draw:object can contain any ODF document as its child.

E-C) Dependencies to “style” namespace.
“style” namespace refers to many other namespaces such as “presentation”, “draw” and “text” to annotate objects with styles. Thus, any dependency to “style” namespace causes lots of dependencies to such namespaces.

E-D) Cross dependencies to the major namespaces but not necessary ones.
The major namespaces such as “presentation”, “chart”, and “form” namespaces classifies the major divisions of ODF usages, namely, presentations, charts for spreadsheet, and forms. If “text” namespace refers to “presentation” namespace, only using “text” in a document also requires “presentation” namespace. Thus, such dependencies may prevent modularization.

Dependencies applicable to Cases E-B, E-C, and E-D are what we call “inappropriate dependencies.” In Figure 2, we denote the inappropriate dependencies in ODF by the dotted lines. As we can tell from this figure, by removing these inappropriate dependencies, we can greatly reduce mutual dependencies, which prevent modularization, among the ODF namespaces. Eventually, each namespace can be a good candidate for a module.

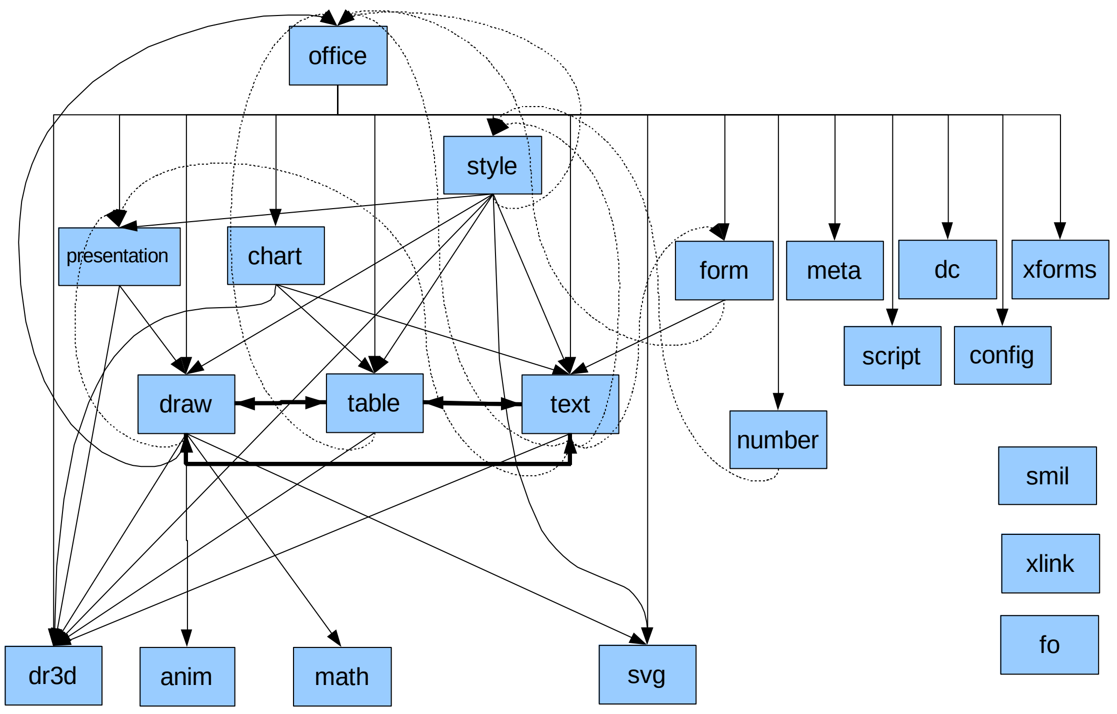

Figure 2 The dependencies among the ODF namespaces. The dotted lines represent inappropriate dependencies.

We also list up the details of the inappropriate dependencies in Table 1.

Element Name	Referrer	Case	Description
office:annotation	table,text	E-B	This specifies an OpenDocument annotation. The annotation's text is contained in <text:p> and <text:list> elements. This elements annotates display , position, size, style, text anchor, caption points and so on.
office:binary-data	style, text, draw	E-B	A container element for binary data in Base 64.
office:change-info	table, text	E-B	Meta-data for change tracking is contained inside an <office:change-info> element. It contains the author and creation date of a tracked change, as well as an optional comment.
office:dde-source	table, text	E-B	This contains DDE connection data.
office:event-listeners	table, text, form, draw	E-B	A container element for event elements associated with an object
office:forms	style, table, draw, presentation	E-B	A container for user interface controls which a user interacts with.
style:text-properties	text, number	E-C	It conveys various information on styles can be stored in attributes such as fo:font-variant and fo:text-transform.
style:list-level-properties	text	E-C	It conveys various information on list-level styles can be stored in attributes such as fo:text-align and text:space-before..
style:map	number	E-C	It specifies the mapping to another style. Possible attributes are style:condition, style:apply-style-name, and style:base-cell-address.

presentation:animations	draw	E-D	A container element for animation effects, which refers to variaous "anim" elements. "draw" namespace refers to this by draw:page element.
presentation:notes	draw,style	E-D	This element contains presentation notes consisting of a preview of the drawing page and additional graphic shapes. "draw" namespace refers to this by draw:page element "style" namespace refers to this by style:master-page element.
presentation:header	text	E-D	This element specifies a header field "text" namespace refers to this because paragraph-content allows it.
presentation:footer	text	E-D	This element specifies a footer field. "text" namespace refers to this because paragraph-content allows it.
presentation:date-time	text	E-D	This element specifies a date and time field. "text" namespace refers to this because paragraph-content allows it.
form:connection-resource	text	E-D	This element specifies the source database by XLink. "text" namespace refers to this through common-field-database-table.

Table 1 A list of elements making inappropriate dependencies.

2.2 Dependencies by Attributes

In ODF 1.1, there is a number of global attributes (806 global attributes). That is, these attributes are specified as qualified names such as “office:name” so that the other elements can reuse them. The global attributes in ODF make many complicated dependencies among the namespaces. In Figure 3, we show the namespace dependencies by attributes.

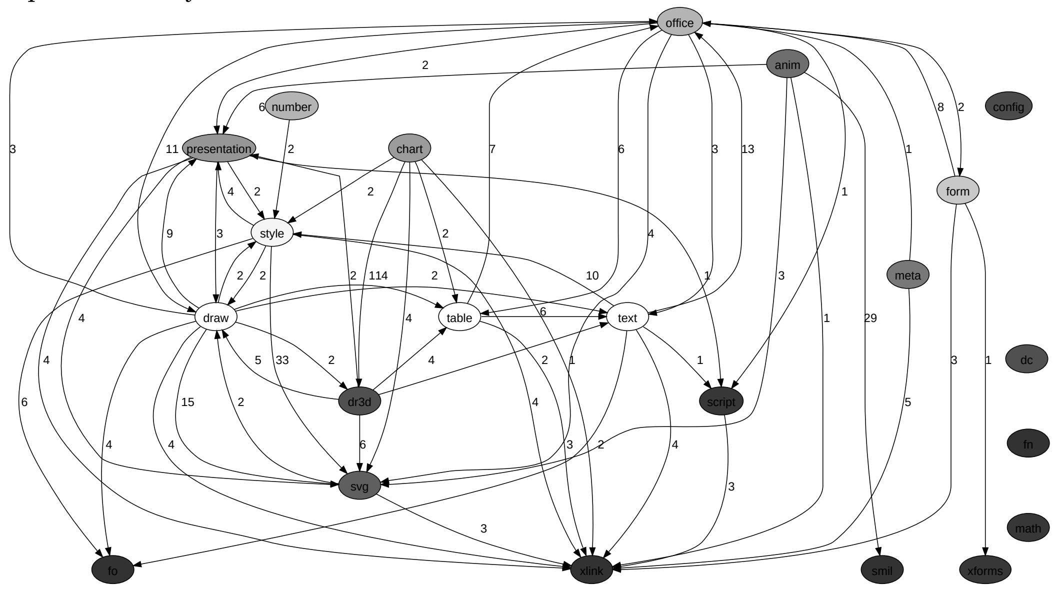

Figure 3 The dependency graph of namespace by attributes.

The dependencies by attributes are much more complicated than those by elements. This is because almost all of the ODF attributes are defined globally not locally.

We examined the dependencies by global attributes and classified them into 4 categories.

A-A) The attributes defined by another official specification such as XLink.

108 attributes are in this category. "svg" (SVG), "fo" (XSL-FO), "smil" (SMIL), "xforms" (XForms), "fn" (XPath Functions), "dc" (Dublin Core), "math" (MathML) and "xlink" (XLink) namespaces contains the vocabularies defined by the other specification than ODF. Therefore, the vocabularies defined in these namespaces must be independent of other ODF constructs.

A-B) The attributes referred only by the namespace defining it but not applicable to this category.

600 global attributes in ODF are in this category. Every attribute in this category is used only by the elements in the namespace that the attribute belong to. For example, anim:id is used only by the elements defined in "anim" namespace. These attributes should have been defined locally in order to prevent other namespaces from referring to these attributes.

A-C) The attributes referred by only one namespace (only one) but not referred by the namespace defining it.

33 attributes are applicable to this category. Every attribute in this category is NOT used by any elements in the namespace that the attribute belong to. Instead, the elements in the other namespace refer to it. For example, style:leader-char is not used by any elements in "style" namespace. It is used by text:index-entry-tab-stop element in "text" namespace. And in any other namespace, there is no element using it. Such attributes in this category should have been defined locally or in the referring namespaces. In this example, style:leader-char attribute should have been defined locally or as text:leader-char attribute.

A-D) Other global attributes.

The rest of the attributes (65 attributes) are in this category. These are used as truly global attributes.

For modularization, we have to reconsider the attributes in Categories A-C and A-D, which form dependencies between namespaces. In Tables 2 and 3, we list up all the attributes in Categories A-C and A-D, respectively.

Attribute	**Referer**
draw:opacity	style
draw:shape-id	presentation
form:apply-design-mode	office
form:automatic-focus	office
office:server-map	draw
office:target-frame	form
presentation:class	draw
presentation:group-id	anim
presentation:master-element	anim
presentation:node-type	anim

presentation:placeholder	draw
presentation:preset-class	anim
presentation:preset-id	anim
presentation:preset-sub-type	anim
presentation:user-transformed	draw
style:leader-char	text
style:legend-expansion	chart
style:legend-expansion-aspect-ratio	chart
style:num-format	text
style:num-letter-sync	text
style:num-prefix	text
style:num-suffix	text
style:rel-height	draw
style:volatile	number
table:cell-range	chart
table:structure-protected	office
text:first-row-end-column	table
text:first-row-start-column	table
text:global	office
text:last-row-end-column	table
text:last-row-start-column	table
text:paragraph-style-name	table
text:use-soft-page-breaks	office

Table 2 A List of Attributes in Category A-C.

Attribute	**Referer**
dr3d:ambient-color	dr3d chart
dr3d:distance	dr3d chart
dr3d:focal-length	dr3d chart
dr3d:lighting-mode	dr3d chart
dr3d:projection	draw dr3d chart
dr3d:shade-mode	draw dr3d chart
dr3d:shadow-slant	dr3d chart
dr3d:transform	dr3d chart
dr3d:vpn	dr3d chart
dr3d:vrp	dr3d chart
dr3d:vup	dr3d chart
draw:caption-id	draw dr3d
draw:caption-point-x	office draw
draw:caption-point-y	office draw
draw:class-names	office draw dr3d
draw:color	presentation draw
draw:corner-radius	office draw
draw:display-name	svg draw

draw:id	office draw dr3d
draw:layer	office draw dr3d
draw:name	svg office draw
draw:style-name	style presentation office draw dr3d
draw:text-style-name	office draw
draw:transform	office draw
draw:z-index	office draw dr3d
office:automatic-update	text office
office:boolean-value	text table form
office:currency	text table form
office:date-value	text table form
office:dde-application	text office
office:dde-item	text office
office:dde-topic	text office
office:name	text office draw
office:string-value	text table form
office:target-frame-name	text meta draw
office:time-value	text table form
office:title	text draw
office:value-type	text table form
office:value	text table form
presentation:class-names	office draw dr3d
presentation:presentation-page-layout-name	style draw
presentation:style-name	office draw dr3d
presentation:use-date-time-name	style presentation draw
presentation:use-footer-name	style presentation draw
presentation:use-header-name	style presentation draw
script:event-name	script presentation
script:language	text script office
style:data-style-name	text style presentation
style:display-name	text style
style:name	text style number
style:page-layout-name	style presentation
style:position	text style
style:rel-width	style draw
style:type	text style
table:cell-range-address	table chart
table:end-cell-address	office draw dr3d
table:end-x	office draw dr3d
table:end-y	office draw dr3d
table:protection-key	table office
table:table-background	office draw dr3d
text:anchor-page-number	office draw dr3d
text:anchor-type	office draw dr3d
text:id	text draw

text:name	text table
text:style-name	text table

Table 3 A List of Attributes in Category A-D.

Owing to a lot of global attributes in Categories A-C and A-D, we are not able to simply modularize ODF schema by namespaces.

2.3 Inconsistent Styles of Schema Definitions

Schema rewriting is an essential technique to modularize schemas. We have to extend, reuse, and modify original schemas to make distinct modules. The ODF schema is, however, not organized enough to easily handle them. The problems we found are threefold: 1) the styles of contents models are inconsistent; 2) the naming conventions are not consistent; and 3) the schema excessively uses redefinition by RELAX NG *combine* feature.

As for 1), inconsistent styles of content models are harmful for schema extension. For example, office:styles element is defined as follows.

```
<define name="office-styles">
  <optional>
    <element name="office:styles">
      <interleave>
        <ref name="styles"/>
        <zeroOrMore><ref name="style-default-style"/></zeroOrMore>
        <optional><ref name="text-outline-style"/></optional>
        <zeroOrMore><refname="text-notes-configuration"/></zeroOrMore>
        <optional><ref name="text-bibliography-configuration"/></optional>
                  <!-- ….omitted…-->
      </interleave>
    </element>
 </optional>
</define>
```

In this example, <ref name="styles"/> is referred as a single pattern. In other words, it is not specified with any extra patterns such as zeroOrMore and optional. Meanwhile, the second content in this element is specified with zeroOrMore (<zeroOrMore><ref name="style-default-style"/></zeroOrMore>). Therefore, readers of this schema may well think that the content referred by <ref name="styles"/> cannot be repeated or omitted. However, "styles" is actually defined as follows.

```
      <define name="styles">
        <interleave>
          <zeroOrMore><ref name="style-style"/></zeroOrMore>
          <zeroOrMore><ref name="text-list-style"/></zeroOrMore>
              <!—omitted -->
          <zeroOrMore><ref name="number-boolean-style"/></zeroOrMore>
        <zeroOrMore><ref name="number-text-style"/></zeroOrMore>
    </interleave>
</define>
```

By reading this definition, readers can understand the contents in styles can be repeated. It is known that by unifying the schema styles into one, we can avoid such confusion.

As for 2), inconsistent naming conventions of ODF schema make maintenance work hard. For example, there exist two styles, namely "*-attlist" and "*-attrs", to specify attribute lists in ODF schema. "*-content" names go with the case as well.

As for 3), many names in ODF schema are excessively redefined. For example, style-graphic-properties-attlist is redefined 116 times in the single ODF schema. Such a style is quite hard to read since we do not know the final result until we read all the definitions of the name.

3. Proposal

As we have seen so far, the current ODF has many problems for modularization. In particular, many dependencies among namespaces are major obstacles for modularization but removing such dependencies without careful consideration may cause incompatibility. In order to avoid these problems, we propose a step-by-step solution for ODF modularization as a form of roadmap.

3.1 Roadmap

Since ODF is a very large format, changing it gives a considerable impact on many implementations and developers. Especially when modularization brings compatibility issues, we have to carefully step forward in well-ordered way. Here we show a possible roadmap to well modularized ODF specification.

The first stage toward modularization is that we separate the dependencies that cause mutual dependencies between namespaces. The second stage is that we should unify naming conventions, schema styles. The third stage is that we should reallocate the namespaces of some elements and attributes in order for each namespace to represent each module. The forth stage is that we should reorganize modules into fine-grained ones for better usability. We show the impacts of the changes by these stages in Table 4. In this table, impacts are classified into three types as follows

(I-1) Changes in ODF Schema file catalogue.

By the stage involving Impact (I-1), we have to reallocate ODF Schema into one or more files. Although in the current ODF specification, the schema is stored in a single file, some parts in the schema will be moved into other files.

(I-2) Changes in ODF Schema definitions and ODF specification

By the stage involving Impact (I-2), we have to change some definitions in ODF Schema, which will cause some modifications in ODF specification since the description in the specification heavily depends on schema definitions.

(I-3) ODF document instance incompatibility

By the stage involving Impact (I-3), compatibility of ODF document instances may not be kept. In other words, the document valid by the old schema may not be valid by the new schema after the stage is done. This is the most serious impact in ODF modularization. Therefore we should carefully step forward if the change has this impact.

Impact / Stage	(I-1) Changes in ODF Schema file catalogue	(I-2) Changes in Schema definitions and ODF specification	(I-3) ODF document instance incompatibility
Stage 1	Yes	No (except for adding some extension points)	No
Stage 2	No	Yes	No
Stage 3	Possibly Yes	Yes	Possibly Yes
Stage 4	Yes	Yes	Possibly Yes

Table 4 The Impacts of the Changes by Modularization.

Next, we explain the details of these stages.

Stage 1) Separating Dependencies

As we have explained in Sections 2.1 and 2.2, ODF schema has many dependencies between namespaces that discourage modularization. In this stage, we separate such dependencies into distinct modules. If the users want to use such dependencies, they only have to load the corresponding module. After this stage is completed, users can choose modules for their own purpose with less extra modules. Since this stage only adds some extension points to ODF schema, possible impacts at this stage are so small, and the necessary changes in ODF specification will be small as well.

Stage 2) Unifying Naming Conventions and Schema Styles

For the further schema modification, the current ODF schema has considerable problems in defined names and styles. In this stage, we unify inconsistent names and styles. For example, all the attribute lists should be named *-attlist, and all the content model should be named *-content, and so on. In addition, we should unify schema style into Garden of Eden style, which has advantages on schema extensibility and readability. These modifications may introduce some incompatibility in ODF schema because some definitions in RELAX NG will be changed. Therefore, the descriptions in ODF specification have to be updated. However, after this stage is complete, users can easily read and extend ODF schemas.

Stage 3) Reallocating Namespaces

The namespaces in the current ODF Schema are not always appropriate for better modularization. In this stage, we should reallocate the functions of ODF into well-chosen namespaces. Since this change will cause incompatibility problems, we should carefully consider many issues around ODF (for example, implementations such as OpenOffice.org) as well.

Stage 4) For Better Modularization

Modules should be fine-grained if possible since fine-grained modules are more consumable for many purposes. After this stage is complete, each ODF module is so small that we can compose the modules to just fit with our requirement.

3.2 Stage 1 --- Separating Dependencies

Overview

At this stage, we separate the dependencies in ODF schema, and then form basic modules based on the namespaces. By considering the ODF namespaces, we propose the following 11 major modules are adequate for our objectives.

1. “common” module as a fundamental library
2. “office” as a container module
3. “text” as a text module
4. “style” as a style module
5. “presentation” as a presentation module
6. “chart” as a chart module
7. “draw” as a drawing module
8. “table” as a table module
9. “form” as a form module
10. “property” module
11. “connection” module

We show a schematic diagram of ODF modularization at this stage in Figure 4. In this diagram, each box with relief denotes a major module and boxes in a major module are submodules that depend on the major module. And arrows represent dependencies between modules. Notice that we omit arrows between a submodule and a major module. When a major module depends on another module through its submodule, the dependency is effective only if the submodule is loaded. For example, “table” module depends on “text” module if and only if “table-text” module is loaded. Hereafter, we call such module “glue”. And 8 official modules, “svg”, “xlink”, “fo”, “dc”, “smil”, “fn”, “xforms” and “math”, are separately defined and they should be maintained with considering the original specifications. We do not consider “dr3d”, “number”, “anim”, “config”, and “script” modules here since the dependencies involving them are so simple that we can deal with them in a similar way.

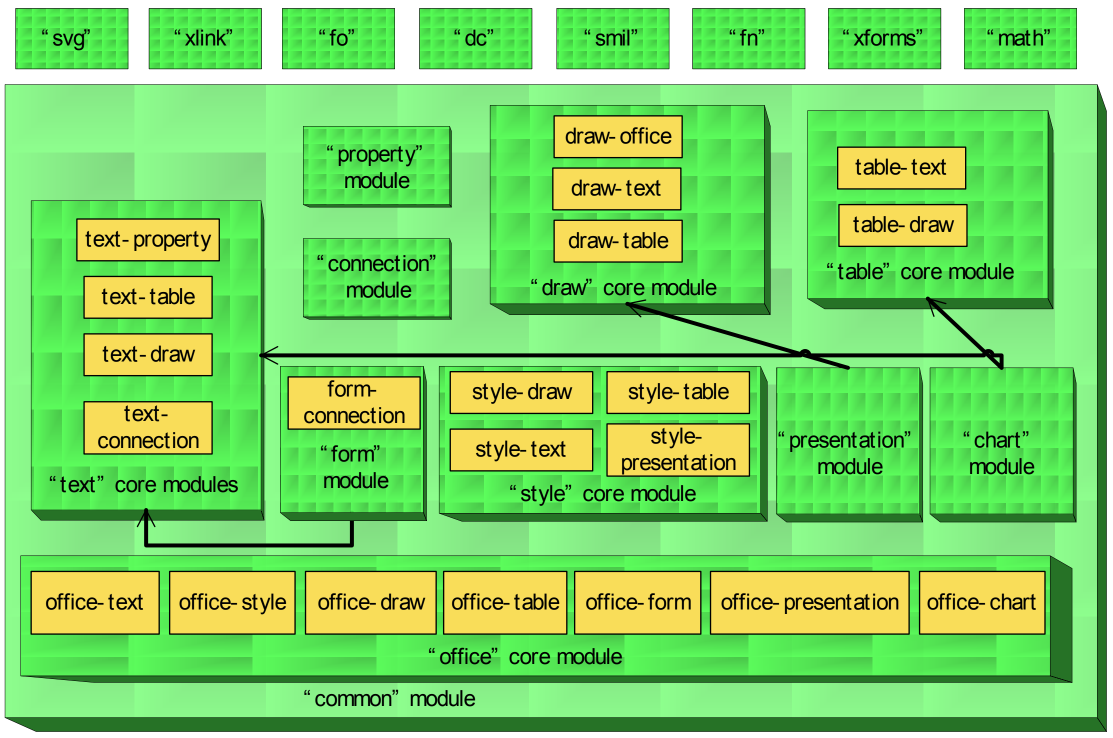

Figure 4 A Schematic Modularization Diagram.

One of the goals at this stage is enabling us to make a simple text processing system by choosing the smallest set of modules for it. For example, by choosing common, "office" core, "text" core, office-text modules, we can make the simplest profile for such text processing as shown in Figure 5. Although it cannot use any styles, drawings, nor tables, it still process texts by ODF.

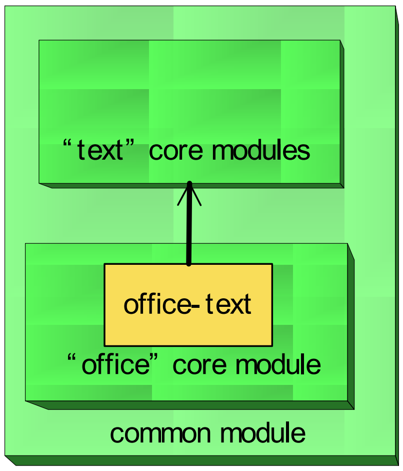

Figure 5 An Example of Module Conformation

Our Approach

If we simply create modules by the ODF namespaces, these modules are tightly coupled with each other as we have seen so far in Sections 2.1 and 2.2. At this point, we have two alternatives to remove the dependencies unfavorable for modularization: 1) moving the elements and attributes making those dependencies to other namespaces; or 2) isolating these elements and attributes to distinct modules with keeping their own namespaces. Each alternative has pros and cons. Although changing the namespace of an element or an attribute breaks compatibility of documents, we can recognize modules only by looking at the namespace of an element or an attribute.

However, we think Option 2) is favorable since at this early stage we should give first priority to compatibility of documents. For this option, we create three new modules, “common”, “property”, and “connection” modules. Simply put, “common” module contains definitions of elements, attributes, and datatypes referred by many modules; “property” module contains many properties such as text properties and list level properties; and “connection” module provides the definitions on database connection, which may be used by “form” and “text” modules. By following this direction, the elements in Category E-B) in Table 1 are accommodated in “common” module; style:text-properties and style:list-level-properties are accommodated in “property” module; and form:connection-resource is put in “connection” module.

Design of Glue

Glue plays a role of bridge between two modules. For example, text-conneciton glue injects a dependency between “text” and “connection” modules. This would looks like the followings.

```
<?xml version="1.0" encoding="UTF-8"?>
<grammar xmlns="http://relaxng.org/ns/structure/1.0">
  <define name="connection-resource.extra" combine="interleave">
    <ref name="form-connection-resource"/>
  </define>
</grammar>
```

where connection-resource.extra is a new extension point introduced in common-field-database-name in “text” module, which is originally defined as follows.

```
<define name="common-field-database-name">
  <choice>
    <optional>
      <attribute name="text:database-name">
        <ref name="string"/>
      </attribute>
    </optional>
    <ref name="form-connection-resource"/>
  </choice>
</define>
```

Since this definition directly depends on form-connection-resource, we change it as follows to remove the dependency with introducing connection-resource.extra.

```
<define name="common-field-database-name">
  <choice>
```

```
      <optional>
        <attribute name="text:database-name">
          <ref name="string"/>
        </attribute>
      </optional>
      <ref name="connection-resource.extra"/>
    </choice>
  </define>
```

After this modification, “text” module does not directly depend on form-connection-resource defining form:connection-resource element. By loading text-connection glue, “text” module can use it since connection-resource.extra is redefined.

By moving the dependencies by the elements in Categories E-C and E-D into appropriate glue, we can remove the dependencies hindering modularization from the major modules.

Global Attributes

As we have seen in Section 2.2, the attributes in Categories A-C) and A-D) are problematic for modularization. Although it is the best way to redesign them to reduce the complicated dependencies, modifying any definitions of these attributes would break compatibility of ODF instances. Therefore, at this stage, we should only move these attributes to common module. Because any attribute has a datatype as its child and does not have any children, moving them into common module does not make any dependencies to common module. Other attributes in Categories A-A) and A-B) should be left unmoved since they do not prevent modularization.

3.3 Stage 2 --- Unifying Naming Conventions and Schema Styles

Overview

In the specification of ODF, ODF Schema plays an important role. It actually defines and classifies the vocabularies of ODF. Therefore, keeping ODF schema clean and easy to handle is deeply important also for modularization. If we can extend ODF schema without changing the original version, the other external module will work well with ODF without changing it. At this stage, we focus on unifying inconsistent naming conventions and schema styles in ODF schema. This stage involves Impact I-2 described in Section 3.1 since we have to change some defined names and content models in the schema. That leads updating many descriptions in ODF specification since the specification heavily depends on the schema.

Naming Conventions

ODF schema has few naming conventions in the defined names unlike the W3C XML specification [XMLSpecGuide]. In the W3C XML specification, parameter entities (like defined names in RELAX NG) strictly follow the following typologies.

- *.att
  It is used for any definition of attribute(s).
- *.class
  It is used for any content model
- *.mix
  It is used for free mixtures (repeatable OR) referred by some content model.
- local.*
  It is used for extension points.
- *.mdl
  It is used for content model fragments (not free mixtures) common or customizable.

Unified naming conventions are helpful for schema users to understand and extend the schema. Thus, we should do the same thing for ODF schema for further modularization and extension. We propose the following naming conventions for ODF schema

- *.attlist
  It is for definitions of attribute(s).
- *.class
  It is for content models.
- *.element
  It is for element definitions
- *.extra
  It is for extension points.
- *.model
  It is for content any model fragments.

Note that all the conventions use a period "." to specify the suffices since any periods are not used for the current ODF schema.

Schema Style

XML schema design patterns are a useful technique that makes schemas readable and extensible by unifying styles of schema definitions. The most common XML schema design patterns are Russian Doll, Salami Slice, Venetian Blind, and Garden of Eden [DP]. In Table ?, we list up these 4 styles.

Design Pattern	Description	Pros.	Cons.
Russian Doll	Only a document element is defined globally, and others are defined locally.	Simple.	Can reuse the entire construct or nothing.
Salami Slice	Define all the elements in a document.	Can reuse schema per element	Cannot extend any content models.

Venetian Blind	Define all the content models in a document.	Can reuse or extend content models in a schema.	-Sometimes Complicated. -Cannot reuse elements.
Garden of Eden	Define both all the elements and the contents models in a document.	Can reuse and extends elements and contents models.	Sometimes Complicated.

However, the current ODF schema does not follow a consistent schema style as we discussed in Section 2.3. We propose "Garden of Eden" style is favorable for ODF schema. According to our experiment of rewriting the current schema to "Garden of Eden" style, 63 definitions can be reduced to the same definition. That means "Garden of Eden" style simplify ODF schema with allowing maximum extensibility and reusability.

3.4 Stage 3 --- Reallocating Namespaces

Namespace is an important feature in XML for modularization since namespace is vital for two purposes: 1) multiple vocabularies and 2) structured extension. The first one, multiple vocabularies, is essential for modularization. By using namespace, we can accommodate multiple vocabularies in a single document without conflicting among these. The second one, structured extension, is essential for schema extension. By giving another namespace than those of ODF schema, users can allow their own extensions and put them in the existing documents without confusion.

Therefore, giving a unique namespace to each module in ODF modularization is a necessary step for the future. However, at Stage 1, some elements and attributes are allocated to distinct modules even though they have the same namespace. At this stage, we reallocate namespaces for the ODF modules.

Unfortunately, reallocating namespaces may involve document instance level incompatibilities since the elements or attributes having different namespaces are considered different vocabularies without any special specification such as markup compatibility in OOXML.

3.5 Stage 4 --- For Better Modularization

Since this stage, the design of modularization is decided by the organization of the current ODF namespace mainly for keeping compatibility. At this stage, however, we redesign ODF modularization for more reusable, robust, and extensible one. We propose the following aspects to be considered at this stage.

- Fine-granularity
  Large and versatile modules are not adequate to use for many purposes. On the contrary, fine-grained modules can be fitted for different kind of purposes by appropriately choosing such modules.

- Understandability
  Easy module should be so understandable that developers use it with less effort. Uniformity, Simplicity and Consistency in modules are essential for understandability.

- Extensibility
  Each module should be extensible enough to allow user's customization and endure for a long time. Deliberate definitions of content models and namespaces are essential for extensibility.

- Composability
  Modules should be as much composable as they can be for a wide range of usages. Inappropriate dependencies among modules prevent us from composing them.

Although in this paper, we do not give a concrete design of better modularization of ODF, we should give a good modularization design to ODF specification since this step will be important in the long run.

4 Concluding Remarks

ODF Modularization is an important step for securing better programmability and interoperability. We found, however, the current ODF specification is not designed to be modularized mainly due to the complicated mutual dependencies. We propose step-by-step solutions by 4 stages for modularization. At Stage 1, we concentrate on separating dependencies that prevent modularization by introducing glue, which conveys bridges between two modules. By this operation, we can decompose 11 major modules and it should be stressed that the impact by this change is so small that ODF specification is not required to be updated drastically. At Stage 2, we unify the naming conventions by following typical XML specifications such as the W3C XML spec. And we adopt "Garden of Eden" schema design pattern to ODF schema. These contribute extensibility and readability of ODF schema. At Stage 3, we reallocate a unique namespace to each module for programs to easily handle the modules by looking at their namespaces. Finally, at Stage 4, we redesign the modules to attain fine-granularity, understandability, extensibility, and composability. Since standardization is essential to achieve these steps, we request many feedbacks for this proposal to build a consensus among the parties interested in ODF.

5 References

[DP] Ayub Khan and Marina Sum, Introducing Design Patterns in XML Schemas, 2006, http://developers.sun.com/prodtech/javatools/jsenterprise/nb_enterprise_pack/reference/techart/design_patterns.html

[XMLSpecGuide] Eve Maler, Guide to the W3C XML Specification ("XMLspec") DTD, Version 2.1, http://www.w3.org/XML/1998/06/xmlspec-report-v21.htm