CDISC ODM White Paper:
Vendor extensions to the CDISC ODM DTD

By Donald Kacher, Oracle Corporation

10 October 2001

This white paper describes how to create and use extensions to the CDISC ODM Version 1.1 DTD.

While the ODM DTD attempts to provide elements and attributes sufficient to transmit most information about the conduct of a clinical trial, it may be missing elements or attributes needed for various special cases, or for transmission of idiosyncratic information between two databases maintained by the same application. For this reason, CDISC permits the extension of the DTD, under the rules defined here.

An extended DTD is one that is a superset of the base DTD. It adds (but never removes) elements or attributes that are needed to make transmitted documents complete and useful to the recipient. An extended DTD may be used where a document compliant with the base DTD would have to be supplemented with information sent by another channel before it could be useful.

This document has two parts. The first defines the rules for managing extensions to DTDs. The second gives a more extended discussion of how extensions to DTDs work.

1. Rules governing the creation and use of extensions to the ODM DTD

1.1 CDISC activities related to DTD extension

1.1.1 CDISC defines extensible and non-extensible DTD elements

CDISC determines which elements in the base DTD shall not be extensible. Initially, this list consists of the element ODM.

1.1.2 CDISC generates an extensible version of the DTD

When CDISC has determined that a new version of the ODM DTD is ready to publish, it passes the DTD through the extensibility translator. Then it publishes the URL of the resulting DTD, with content rules replaced by Parameter Entities. This is referred to as the base DTD. The filename of the base DTD file must include its version number.

1.2 How an organization creates an extended DTD

An organization may want to specify an extension to the base DTD either for documents it sends, or for documents it receives. This will allow it to include additional information not directly

supported within the base CDISC DTD. In either case, the organization composes an extended DTD. An extended DTD is a file that takes this form:

```
<!-- Overrides of attribute declarations present in the base DTD -->

  <!-- Overrides of element content rule declarations from the base
   |   DTD, in the form
   |     <!ENTITY % element-name.content "modified-content-rule" >
   -->

  <!-- Define and use an entity reference to the base DTD -->
  <!ENTITY % baseDTD SYSTEM "URL-of-base-DTD">
  %baseDTD;

  <!-- New element declarations -->

  <!-- New attribute declarations -->
```

Extensions must follow these rules:

1. Modification of element declarations is permitted only for those elements for which CDISC has extracted the content rule into a Parameter Entity declaration.
2. A replacement content rule for an element must include all children that are declared in the base DTD content rule for the element. New elements may be added; none may be removed.
3. An override of the attributes of an element must include all attributes declared for the element in the base DTD. New attributes may be added; none may be removed.
4. The organization must place the extended DTD file on an externally-visible Web site. The filename of the extended DTD must include the version number of the base DTD it references.

1.3 How to use an extended DTD in an ODM document

An extended document is one that includes XML entities (representing attributes or elements) that are declared in an extended DTD.

1.3.1 Receiving extended documents

If an organization wants to receive extended documents, it is that organization's responsibility to write code to parse and interpret the extended entities. An organization is entitled to disregard data in extended elements and attributes. An organization is entitled to receive from the sender a stripped version of the document that excludes the extended elements and attributes.

1.3.2 Sending extended documents

If an organization wants to send an extended document, it must do these things:

- In the DOCTYPE declaration, reference the extended DTD to be used. This must take the form of the URL of an extended DTD. The extended DTD may be on the organization's web site, or on that of some other organization.
- Pass the document through a validator, to confirm that the document is valid with respect to the extended DTD.
- Use a extension-removal filter to generate a version of the document that does not include any extended elements.
- Send the extended document to the recipient.
- If requested, send the non-extended version of the document to the recipient.

2. Background on extending a DTD

This section gives some background and tutorial information on extending a DTD.

There are two possible approaches to writing an extended DTD:

The first approach is to take a complete copy of the base DTD, and hack it, adding new elements and attributes, and modifying existing elements and attributes. The result is a new DTD against which XML documents can be validated.

The second approach is to construct a new, "effective", DTD comprised of the extensions you want to add, plus an inclusion of the base DTD, in which the declarations of everything else reside. You include one DTD inside of another by using an External Parameter Entity (PE), like this:

```
<!ENTITY % baseDTD SYSTEM "URI-of-base-dtd">
%baseDTD;
```

New elements and new attributes are then easy to add; you just define them in the new, extended, DTD. Modifications are a bit trickier, but manageable. XML parsers follow a rule that, if there is more than one declaration of a particular attribute, the first declaration is the one that is used. So the extended DTD can modify an attribute that is already declared in the base DTD by declaring it with new characteristics before including the base DTD.

Regrettably, the rule does not apply to duplicate element declarations; an XML parser will consider two declarations of the same element within a DTD to be an error. But, luckily, the first-instance-wins rule does apply to Internal Parameter Entities, and they can be used to get around the restriction on duplicate element declarations. The method is this: in the base DTD, for each element that you want to permit modification of, you extract the content rule into an Internal PE declaration, and then replace the content rule of the Element declaration with a reference to the corresponding entity. For example, replace

```
<!ELEMENT AuditRecord
   (UserRef, LocationRef, DateTimeStamp,
    ReasonForChange?, SourceID?)>
```

with

```
<!ENTITY % AuditRecord.content
   "UserRef, LocationRef, DateTimeStamp,
    ReasonForChange?, SourceID?" >
<!ELEMENT  AuditRecord (%AuditRecord.content;)>
```

in the base DTD. Then, in the extended DTD, before including the base DTD, provide the extension version of the content rule PE. Since this appears first, it is used instead of the one in the base DTD, and the parser validates against the extended version of the element content rule.

Here, then, is what an extended DTD file under the second approach would look like:

```
<!-- Overrides of existing attribute declarations -->

<!-- Overrides of content rule entities declarations from
 |  the base DTD
 +-->

<!-- Define and use an entity reference to the base DTD -->
<!ENTITY % baseDTD SYSTEM "URI-of-base-dtd">
%baseDTD;

<!-- New element declarations -->

<!-- New attribute declarations -->
```

The new element and attribute declarations could precede the inclusion of the base DTD; it would make no difference. But placing the External PE declaration and reference between modification and creation makes it easier to see which is which.

The XML document that is to be validated against the extended DTD names it, rather than the base DTD, in its Document Type declaration. Since PEs and attribute declarations do follow the rule that the first one wins, the extended version of them gets selected instead of the base version. And the newly declared elements and attributes are simply treated as part of the DTD. Everything else, however, is as it was in the base DTD.

The second method is clearly better:

- You have complete control over what elements can be modified, because only those elements for which content rule is parameterized can be altered
- The extension contains only what's different, so it's easy to see how it differs from the base
- You can change what the base PEs point to without needing to update all of the extensions (though you will probably want to eventually)
- You can use an extended DTD to constrain documents. For instance, you can override an attributes declaration which is CDATA in the base DTD with an enumerated list in the extended DTD.

The cost is relatively low: we have to convert the content rule of each element in the base DTD to a PE followed by a reference to the PE in the Element declaration. This is a simple mechanical task that can be scripted.

Here is an example. We start with a DTD named base.dtd, with these contents:

[1] base.dtd

```
<!-- This is base.dtd -->
<!ELEMENT  families (family*)>
<!ELEMENT family ( mom*, dad*, child*, address*)>
<!ATTLIST family name CDATA #REQUIRED>
<!ATTLIST family partyAffiliation CDATA #IMPLIED>
<!ELEMENT mom (#PCDATA)>
<!ELEMENT dad (#PCDATA)>
<!ELEMENT child (#PCDATA)>
<!ATTLIST child birthday CDATA #IMPLIED>
```

Here is an XML document, family.xml, that declares base.dtd to be its DTD. This file parses successfully against the base DTD.

[2] family.xml

```
<?xml version="1.0" standalone="no" ?>
<!DOCTYPE families SYSTEM "base.dtd" >

<!--
  | families.xml. Uses base.dtd as its dtd.
  |
  +-->

<families>
  <family name="Smith" partyAffiliation="democratic">
    <mom> Alice </mom>
    <dad> John </dad>
    <child birthday="1990-08-02" > Susie </child>
    <child birthday="1995-10-15" > Mark </child>
  </family>
</families>
```

Here is a copy of the base DTD, with content rules extracted to PEs. It is functionally identical to base.dtd. This file is called
base_p.dtd.

[3] base_p.dtd

```
<!--
  | This is base_p.dtd. It replaces each element's content
  | rule with a parameter entity (PE). The PE, in turn
  | contains the base content rule
  | for the element. This can be modified by an extension
  | PE with the same name that supplies a different version
```

```
    | of the content rule, either directly, or by naming a
    | file that contains it.
    |
    +-->

<!--ELEMENT  families (family*)-->
<!ENTITY % families.content " family* " >
<!ELEMENT  families (%families.content;)>

<!ENTITY % family.content "mom*, dad*, child*, address*" >
<!ELEMENT family ( %family.content; )>

<!ENTITY % mom.content "#PCDATA" >
<!ELEMENT mom (%mom.content;)>

<!ENTITY % dad.content "#PCDATA" >
<!ELEMENT dad (%dad.content;)>

<!ENTITY % child.content "#PCDATA" >
<!ELEMENT child (%child.content;)>

<!ATTLIST family name CDATA #REQUIRED>
<!ATTLIST family partyAffiliation CDATA #IMPLIED>
<!ATTLIST child birthday CDATA #IMPLIED>
```

If we make one change to family.xml (show in bold below), we can use base_p.dtd as the DTD against which to validate. This gives us the file families_p.xml:

[4] families_p.xml

```
<?xml version="1.0" standalone="no" ?>
<!DOCTYPE families SYSTEM "base_p.dtd" >

<!--
  | families_p1.xml. Uses base_p.dtd as its dtd.
  | This dtd has a PE for each element's content rule.
  +-->

<families>
  <family name="Smith" partyAffiliation="Green">
    <mom> Alice </mom>
    <dad> John </dad>
    <child birthday="1990-08-02" > Susie </child>
    <child birthday="1995-10-15" > Mark </child>
  </family>
</families>
```

This also parses successfully.

Now, suppose that we want to extend the base dtd, in these ways:

- Add a new element <pet>, with attributes name and animalType.
- Declare that new animalType attribute has an enumerated set of allowed values, consisting of cat, dog, and hamster.
- Add a new attribute, favoriteColor, to the existing child element.
- Override the attribute declaration for partyAffiliation, so that instead of accepting arbitrary CDATA, it has an enumerated set of values.
- Override the base declaration of <family>, in these ways:
    - Allow at most one instance each of mom and dad
    - Allow a family to have zero or more pets

Here is an extended DTD, named extend_base1.dtd, that does these things, by extending the base DTD:

[5] extend_base1.dtd

```
<!--
 | This is extend_base1.dtd.
 | It creates new elements and attributes, and overrides
 | other attributes and element content rules.
 | Then it references the base DTD for the remaining
 | element and attribute definitions.
 +-->

<!-- Overrides of existing attribute declarations -->
<!ATTLIST family partyAffiliation
   (Democrat|Republican) #IMPLIED>

<!-- Overrides of entities from the base DTD -->
<!ENTITY % family.content
   "mom?, dad?, child*, address?, pet*" >

<!-- Define and use an entity reference to the base DTD -->
<!ENTITY % baseDTD SYSTEM "base_p.dtd">
%baseDTD;

<!-- New element declarations -->
<!ELEMENT pet EMPTY>

<!-- New attribute declarations -->
<!ATTLIST child favoriteColor CDATA #IMPLIED >
<!ATTLIST pet name CDATA #REQUIRED>
<!ATTLIST pet animalType (cat|dog|hamster) #IMPLIED >
```

The declaration of the External PE base DTD, and the subsequent reference to it, have the effect of incorporating the base DTD into the overall effective DTD. But order counts, so the PE declaration for family.content that precedes the inclusion of the base DTD is the one that goes into the final effective DTD. Similarly, the override version of the partyAffiliation attribute, since it precedes the declaration in the base DTD, is the one that gets incorporated into the final, effective DTD.

Now, we can send an XML document, families_ep1.xml, that uses this extended DTD. Changes from families.xml are shown in bold.

[6] families_ep1.xml

```
<?xml version="1.0" standalone="no" ?>
<!DOCTYPE families SYSTEM "extend_base1.dtd" >
<!--
 | families_ep1.xml.
 | Calls extended DTD extend_base1.dtd, to pick up
 | extension of definition of family content rule.
 +-->

<families>
  <family name="Smith" partyAffiliation="Democrat">
    <mom> Alice </mom>
    <dad> John </dad>
    <child birthday="1990-08-02" favoriteColor="green" >
      Susie
    </child>
    <child birthday="1995-10-15" > Mark </child>
    <pet animalType="cat" name="Frisky" />
  </family>
</families>
```

Conclusions

The best way to enable extensions to the DTD is by applying a simple mechanical translation to the base DTD and publishing the translated DTD. The extensibility translator's job is this:

for each element for which CDISC wants to enable extensions {
generate a Parameter Entity, with name equal to element-name.cr;
set the Parameter Entity body's contents equal to the element's content rule;
replace the element's content rule with a reference to the corresponding Parameter Entity;
}

If you want to use an extended DTD, you construct a new DTD comprised of the extensions you want to add, plus an inclusion of the base DTD, in which the declarations of everything else reside. You include one DTD inside of another by using an External Parameter Entity.

If we follow the convention that extensions can only extend, rather than modify, the base DTD, then any XML document that conforms to the extension DTD will also conform to the base DTD, once you strip out those elements that are defined only in the extension portion of the DTD. Under this convention, it is possible to write a filter that takes an extended XML document and gives back an XML document that contains only base DTD elements and attributes.