Model-Based XML Editor Generation

Jong-Myung Choi **Soo-Lyul Oh** **Dong-Soon Ahn** **Jong-Hwa Kim** **Kyung-Woo Park**
Han-Suk Choi[†] **Hea-Sang Shin**

Dept. of Computer Eng., Mokpo National Univ., Korea
Dept. of Multimedia Eng., Mokpo National Univ., Korea[†]
{jmchoi, syoh, dsahn, kimjh, kwpark, chs[†], sea}@mokpo.ac.kr

Abstract - *As XML is becoming more widely used, the demand for user-friendly XML editors is increasing. In the perspective of software engineering, the development of user-friendly editors comprises considerable portions of overall development costs. In order to reduce the costs, we introduce how model-based user interface generation techniques are applied to generating form-based XML editors. In our approach, we consider a DTD as a document consisting of sequence, choice, and repetition, and we describe the document in Document Decomposition Graph (DDG). By applying the presentation rules to DDG, we can generate an XML editor automatically. Our approach will reduce the costs for developing the user interfaces for XML applications, and the generated XML editors will help users in writing documents correctly and easily.*

Keywords: XML editor, form-based interface, model-based

1 Introduction

After becoming the standard for describing data and document, XML has come to be widely used in various areas. For example, it is used for describing office documents [1], graphic data [2], protocols [3], and so on. As it is widely used, the opportunity that experts and end users have to write XML documents is increasing. However, it is difficult for end users to write XML documents correctly due to the some problems. First, it is required for end users to have knowledge of XML syntax. For example, they should know that a start tag must have a matching end tag, or the attribute value should be quoted. Second, they should have knowledge of the overall structure of the document and its vocabularies. In addition, they should also know the name and meaning of the vocabularies.

Many general-purpose XML editors have been developed to help users writing documents correctly and easily. Most of these editors are classified into two groups: tree-based or syntax-directed. These general-purpose editors show the list of available elements and attributes and allow the user to select one of them, and after writing the document, the editors check well-formedness and validity of the document. These general-purpose XML editors are useful for users, but these tools have some problems. First of all, it is hard for end users to learn and use because the interface is not user-friendly. Second, although the editors show the list of valid elements and attributes, end users normally have no idea of their meanings. Third, these editors don't provide semantic information or context help for a specific XML document. Finally, it is very hard to find syntactic or semantic errors in documents. Some intelligent editors check syntax errors automatically, but they cannot find the semantic errors. Therefore, in order to find semantic errors, end users have to browse the whole document.

Most of the currently used office applications allow users to handle data by providing GUIs that consist of menus, buttons, and form style interfaces. The user friendly GUIs allow end users to handle or write application specific data without any knowledge of applications. For example, they don't need to know about data format, data structure, or vocabularies. After considering the data used in office applications, we found that the data of most applications are data-centric and they are easily handled with form-based GUIs.

Since XML are used for applications, if the data for the applications are data-centric, the form-based XML editor will be perfect for end users. These user-friendly GUIs will reduce errors, and allow end-users to make XML documents more easily. For example, the deploy tool of J2EE SDK [4] provides a form-based interface, and it interacts with users to generate an XML document for EJB (Enterprise JavaBean). It allows users to make a document without knowledge of its structure or vocabularies.

In this paper, in order to provide end users user-friendly editors, we introduce a new method to automatically generate form-based XML editors, and an XML editor generator called XED. XED generates XML editors, and it also allows users to customize the XML editors according to their own style. Furthermore, during the customization, it allows users to add tool-tips and context help to the editor. The context help is very useful for users to write documents correctly.

The rest of this paper is divided into 4 sections. In the next section, we show how to model a DTD as DDG, and then we introduce the presentation model. We then discuss system architecture and comparisons with other editors. In the last section, we draw some preliminary conclusions.

2 Task Model

User interface development requires considerable costs, so that there has been much research on these fields. Model based interface development has been received attention because it can reduce the costs by generating user interfaces automatically. In model based interface development, the first thing to be done is task analysis [5]. A task means an activity that results in a significant state change in a given contextual situation [6]. The task analysis is performed through decomposing the task into sub-tasks until they cannot be decomposed. Several methods can be used for the analysis: dataflow analysis, event-driven analysis, and constraint-based approaches [6].

DTD defines the structure and the vocabularies of an XML application, and the application has sequences, choices, repetitions, and options of elements. We can see an XML application as the tree that has the application's root element as its root node, and its leaf nodes are related with data that are filled when editing the document. Therefore, if we see the filling data into the leaf nodes as "data input events", then the event-driven task analysis is easily applied to the XML editor. Furthermore, the task analysis is easily represented with DDG (Document Decomposition Graph), a variation graph of EDG (Event Decomposition Graph) [7]. The DDG is an AND/OR graph, where nodes represent task completion events (e.g., the termination of an input, the validation of a form), and the leaf nodes mean the completion of data input events. The DTD is defined as Definition 1.

Definition 1. Document Decomposition Graph (DDG)

DDG = <V, A> is an event graph for XML documents, where $V = \{ X_1, X_2, .., X_n \}$ and $A = \{ (X_i, X_j) \mid X_i, X_j \in V \}$. A node, X_i, in DDG means an element, an attribute, or a member of a set DS = { SEQ, OR, ?, +, * }. □

When an element's contents consist of child elements, the child elements are grouped by SEQ or OR. SEQ is the sequence in element contents, and OR is the choice in element contents.

Lemma 1. SEQ and OR Group

An element definition that has SEQ or OR can be rewritten into two production rules: one which produces SEQ or OR groups, and the other which produces the child elements. □

For example, by applying the lemma 1, the element definition <!ELEMENT A (B_1 | B_2 | .. | B_n)> can be transformed into two production rules as follows:

A -> A_OR - production rule 1

A_OR -> B_1 | B_2 | .. | B_n - production rule 2

For building DDG, the right-hand side of a production rule becomes the child nodes of the left-hand side. Therefore, the above element definition generates a sub-tree as illustrated in Figure 1.

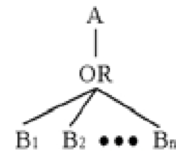

Figure 1. Group

Lemma 2. Repetition and Option

The repetition or option that is applied to elements or groups is transformed into the parent node of those elements or groups. □

For example, the element definition <!ELEMENT A (B_1, B_2?, .. , B_n)*> is divided into the following production rules by applying lemma 1 and lemma 2.

A -> A_0_MORE - production rule 3

A_0_MORE -> A_SEQ - production rule 4

A_SEQ -> B_1, B_2_?, .. , B_n - production rule 5

B_2_? -> B_2 - production rule 6

The productions illustrated above are represented as the sub-tree in Figure 2.

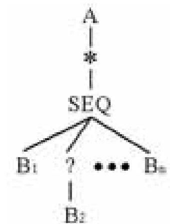

Figure 2. Repetition and Option

The #PCDATA, EMPTY, ANY, and attribute of DTD become the leaf nodes in DDG, and they are directly related to the data input events. As a result, the elements

with child elements or a member of DS become the intermediary nodes in DDG.

When designing XML applications, the choice between elements and attributes is up to the designer's preference. Some prefer attributes, but others like elements. In addition, attributes and elements can be used interchangeably.

Lemma 3. Element with Attributes

The attributes can be transformed into child elements with #PCDATA contents. In addition, the attributes with the #IMPLIED property are converted into the optional elements. □

For example, assume that an element, A, is defined with its attributes, as shown in Figure 3. In particular, an attribute, b_2, has the #IMPLIED property.

```
<!ELEMENT A  ( B1, B2, .. , Bn )*>
<!ATTLIST   A
        b1  CDATA   #REQUIRED
        b2  CDATA   #IMPLIED
        .....
        bn  CDATA   #REQUIRED>
```

Figure 3. Element with Attributes

The element A and its attributes are transformed into element definitions as shown in Figure 4. The attributes are converted to A's child elements, and b_2 is converted to an optional element because of its #IMPLIED property.

```
<!ELEMENT A ((A_b1, A_b2?, .. , A_bn),
                    (B1, B2, .. , Bn )*)>
<!ELEMENT A_b1  (#PCDATA)>
 ......
<!ELEMENT A_bn  (#PCDATA)>
```

Figure 4. Element without Attributes

Lemma 4. Leaf Nodes with Attributes

When an element has #PCDATA contents and attributes, then the attributes are transformed into its child elements, and new elements with #PCDATA contents is added. □

For example, the element, A, in Figure 5 has #PCDATA contents and attributes.

```
<!ELEMENT A  ( #PCDATA ) >
<!ATTLIST   A
        b1  CDATA   #REQUIRED
        b2  CDATA   #IMPLIED
        .....
        bn  CDATA   #REQUIRED>
```

Figure 5. Terminal Node with Attributes

The element, A, in Figure 5 is transformed according to lemma 4, and it is converted into the element in Figure 6. As shown in Figure 6, the new element A_0 is added, and the attributes also are converted into the child elements of A.

```
<!ELEMENT A (A0, A_b1, A_b2?,.., A_bn)>
<!ELEMENT A0    (#PCDATA)>
<!ELEMENT A_b1  (#PCDATA)>
 ......
<!ELEMENT A_bn  (#PCDATA)>
```

Figure 6. Element without Attributes

Algorithm 1 performs the transformation from attributes to elements.

Algorithm 1. Attribute2Element

```
A : the current element
B : the set of attributes defined in A
isRequired(b) : it returns true if attribute b has the
                #REQUIRED property, otherwise false.
BEGIN
  switch(childType(A))
    case PCDATA:
      transform A's contents into "<!ELEMENT A ( A0 )> "
     define new element "A0" with (#PCDATA) contents
    case ANY:
      transform A's contents into "<!ELEMENT A ( A0 )> "
      define new element "A0" with ANY contents
  end-switch
  foreach  bi ∈ B
      define new element "A_bi" with (#PCDATA) contents
      if ( isRequired(bi) == true )
        insert "A_bi" name into A's content using SEQ
      else
        insert "A_bi" name into A's content with option
          property using SEQ
      end-if
   end-foreach
END
```

Using definition 1 and lemmas 1-4, we can transform an XML DTD into a DDG rather easily. Let us take an example. The DTD in Figure 7 is an example XML

application for a name card document. The namecard element uses SEQ and contains name, company, and contact information. The company address element has an optional element, named country, and users can choose one from phone, cellphone, or email for the contact information.

```
<!ELEMENT namecard  (name, company, contact)>
<!ELEMENT name      (#PCDATA)>
<!ELEMENT company  (name, address)>
<!ELEMENT address (country?,city,street,zipcode)>
<!ELEMENT country   (#PCDATA)>
<!ELEMENT city      (#PCDATA)>
<!ELEMENT street    (#PCDATA)>
<!ELEMENT zipcode   (#PCDATA)>
<!ELEMENT contact   (phone | cellphone | email)+>
<!ELEMENT phone    (#PCDATA)>
<!ELEMENT cellphone (#PCDATA)>
<!ELEMENT email    (#PCDATA)>
```

Figure 7. DTD for Name Card

Applying the lemmas, the namecard DTD is transformed into the DDG in Figure 8. As shown in Figure 8, the elements with #PCDATA are represented as leaf nodes, and the elements with child elements are represented as intermediary nodes. The root element is represented as the root node in DDG.

Figure 8. The DDG for namecard

Transforming a DTD into a DDG, there may be cycles. When already defined elements are used several times, the indirect cycles occur. To address this problem, in DDG, once an element definition used, it is checked, and if a checked element is found, then the one previously defined one is reused. Similarly, DTDs are reused using the entity reference mechanism. For example, users may want to construct a business card book DTD using the predefined name card DTD. Figure 9 shows the cardbook DTD, which reuses the predefined elements and attributes.

```
<!ENTITY  % cardref  SYSTEM "namecard.dtd">
<!ELEMENT cardbook (owner, data)>
<!ELEMENT owner    (name, contact)>
<!ELEMENT data      (namecard)*>
%cardref;
```

Figure 9. DTD for Name Card Book

The reused or predefined elements can be checked, and they are represented as the gray-filled rectangle nodes in the DDG. Figure 10 shows the DDG for the business card book. The name, contact, and namecard elements are reused and filled with gray color.

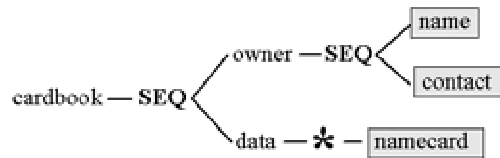

Figure 10. The DDG for cardbook

The DDG can be easily built from DTD, and it supports modularity and reuse of the predefined DDG components.

3 Presentation Model

Presentation model specifies the appearance of the user interface with widgets. This model applies interface-mapping rules to the task model to generate XML editors automatically.

The leaf nodes of the DDG are related to the input of real data, and so they must be represented by "simple components". We call the components that interact with users and get a unit of data simple components. For example, text field, combo-box, or checkbox are typical examples of simple components.

The intermediary nodes of the DDG are used for grouping other nodes, and they are mapped to "group components". The group components usually classify GUI components into logically closely related GUI components. The examples of group components are panel and tabbed-pane. The group component may contain other group components or simple components. Some intermediary nodes are mapped to special simple components such as tree and table. In another cases, the intermediary nodes are mapped to windows or dialogues. We call the window or dialog a PU (presentation unit), and represents a logically independent data unit.

The presentation model has hierarchical structure, and Figure 11 shows the structure. We can consider that an XML document can be represented with PUs, and each one also has several group components or simple components. In addition, each group component also can contain other group components or simple components.

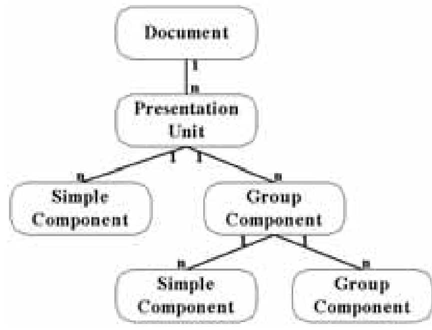

Figure 11. Hierarchy of Presentation Model

The DTD components can be mapped to various GUI components. We classified them according to their data types and structures and the mapping rules between the DTD components and the GUI components. Table 1 shows the mapping rules.

Table 1. GUI Mapping Rules

DTD Structures	GUI Components
#PCDATA	text field, text area, combo box, tree node, table cell, slider, spinner, gauge
Mixed contents	text area
ANY	text area
Option	checkbox
Element with child	tree, table, group component (panel, tab-pane, etc)
Group	group component (panel, tab-pane, etc)
ENTITY, ENTITIES	file dialog, text field
Enumeration attribute	text field, combo box, list
ID, IDREF, IDREFS, NMTOKEN, CDATA, NMTOKENS, NOTATION	text field

The DDG and presentation model can be combined with the help of user interface designers, and it produces a decorated DDG (DDDG). The DDDG is a graph that adds the presentation model information to each node in DDG.

Definition 2. DDDG

DDDG = <DDG, UI, PM>, where UI is the set of possible user interface components, and PM is the mapping rules between DDG nodes and UI components. PM can be defined as PM = $\{ (<X_i, X_j, .., X_n>, U_k) \mid X_i, X_j, .., X_n \in V, U_k \in UI \}$. □

The nodes of DDDG have the presentation model information for the DDG nodes. Therefore, the root node has a window, the intermediary nodes have group components, and the leaf nodes have the simple components information. Figure 12 shows an example of DDDG for the namecard DDG.

Figure 12. DDDG for namecard DDG

The XED builds the DDDG by interacting with user interface designers. However, XED helps the designers by determining the possible widgets for each DDG node and suggesting showing them. Algorithm 2 determines the possible widgets from DDG nodes.

Algorithm 2: DetermineUI

```
child(x) : the set of child elements of node x
height(x) : the height of node x. If height(y) = 0, where y
            means a leaf node.
is_root(x) : whether the node x is root element or not
type(x) : it returns the DTD component of node x
att_type(x) : the data type of attribute node x
node : current node
BEGIN
switch (type(node)) :
  case ELEMENT:
    if(height(e) == 0)
      UI_COMPONENT = TEXTFIELD
    else
      if(is_root(node))
        UI_COMPONENT = WINDOW
      else
        UI_COMPONENT = GROUP_PANEL
      end-if
    end-if
  case ATTRIBUTE:
    switch(attr_type(node))
      case ENUM:
        UI_COMPOENT = COMBOBOX
      default:
        UI_COMPONENT = TEXTFIELD
    end-switch
  case ? :
    ∀e, e ∈ child(node),
    if (height(e) == 0)
      UI_COMPONENT= CHECKBOX & TEXTFIELD
```

```
        else
          UICOMPONENT=CHECKBOX&GROUP_PANEL
        end-if
    case + :
    case * :
        if (child(node) == SEQ)
            ∀e, e ∈ child(child(node))
            if (height(e) == 0)
                UI_COMPONENT = TABLE or MULTIPANEL
            else
                UI_COMPONENT = MULTIPANEL
            end-if
        else if (child(node) == OR)
            ∀e, e ∈ child(child(node))
            if (height(e) == 0)
                UI_COMPONENT = TEXTAREA or TABLE
            else
                UI_COMPONENT = WINDOW
            end-if
        end-if
end-switch
END
```

After determining the UI and building the DDDG, the XED can rather easily generate XML editors. It uses the recursive-descent parsing method to generate editors. Algorithm 3 shows the method. The XED defines the functions for each XML element and attribute of the DTD. Then, the XED invokes the "root( )" function for the root node of the DDDG.

Algorithm 3: XML Editor Generation

```
function UI de_i ( de_i ) {
    UI ui = determinUI()
    foreach e_i ∈ child(de_i)
        ui_e_i = e_i();
        ui.add (ui_e_i )
    end-foreach
}
BEGIN
foreach de_i ∈ DE
    define function de_i
end-foreach
call the function matching the root node in DDG
END
```

4 System Architecture

The XED receives a DTD document as its input, and it generates the DDG through task analysis. After that, the generated DDG is transformed into the DDDG with the presentation model information, and the XED generates a draft XML editor from the DDDG. Finally, user interface designers or end users may customize the editor for their styles. They may change or modify the layout or graphical properties by direct manipulation. Figure 13 shows the XED's architectures.

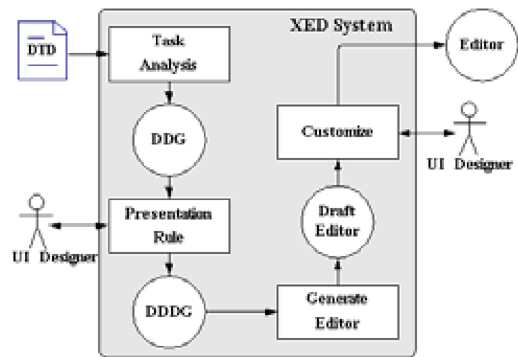

Figure 13. Architecture of XED

The XED runtime consists of three parts: DDG area, work area, and possible widgets area. Figure 14 shows the runtime XED. The left side is the DDG area, and the top area is the possible widget area and shows the possible widgets for the currently selected node of the DDG. Finally, the right side is the work area, and it shows the basic shape of the generated XML editor. Users can modify the layout of the components of the editor by direct manipulation. It also allows users to add context helps to components through use of the editor. For example, they can add help messages to each group component, and tool-tips to each simple component. The context help and tool-tips will guide users to write documents correctly.

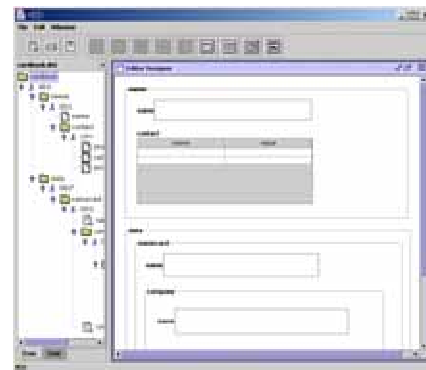

Figure 14. XED Runtime

The XED runtime generates XML editors. Figure 15 shows the generated XML editor for the cardbook DTD. The each part of the DTD is mapped into the widget. For example, the contact element is represented as the table with two columns: name and value. The name column consists of combo box, and it shows the possible child elements of the contact element.

Figure 15. CardBook : The Generated Editor

We surveyed 25 participants in order to test whether the form-based XML editor is easy to learn and use. The participants were novices in XML, and we tested three XML editors – XMLSpy [8], SDE [9], and our automatically generated XML editor, CardBook. We asked participants which one was the easiest to learn and use. About 56% of them answered that the CardBook was the easiest to use, and 70% answered that it is the easiest to learn. Table 2 shows the results of the survey.

Table 2. Result of Survey

	XMLSpy	SDE	CardBook
Easy to use	40%	4%	56%
Easy to learn	28%	2%	70%

We also examined the XML files that they wrote with the editors, and we found that 20% of the XML files that were written in XMLSpy and SDE had semantic errors. For example, they put the friend's name in the company name element. It is because they were confused by the same tag names. In addition, we also found that some of elements have empty contents. They missed some elements to fill in. It is because the XMLSpy and SDE have grid-style interfaces. However, there are relatively few syntactic errors because the editors check well-formedness and validity.

5 Conclusions

After becoming the standard, XML has bean widely used in all fields of industry. Now, the necessity for end users to use XML documents is increasing. However, the XML editors currently being used are rather hard for end users to use. Currently, office applications have form-based GUIs, and users use those applications without noticing how the data are stored and what format is used. XML editors should be as easy to use as these office applications.

In this paper, we introduced a new method for generating form-based editors for data centric XML documents. The XML editor generator receives a DTD document, and it generates an XML editor applying a model-based user interface development method. It transforms the DTD into the DDG. The DDG decomposes the document structure into sub-structures and primitive data elements, and makes a list of possible widgets for each node of the DDG. After that, the DDG is transformed into the DDDG with interaction with users. Users choose the proper widgets from the possible widgets for each node of DDG. Finally, the XED generates the form-based XML editor from the DDDG.

The form-based XML editor has several advantages over existing editors. First, it is easy to learn and use. According to survey, the form-based editor is much easier than the commercial products. Second, it reduces the syntactic and semantic errors in XML documents. The form-based XML editors will help end users in write XML documents easily and correctly.

6 References

[1] Michael Brauer, et al., Open Document Format for Office Applications v1.0, OASIS Standard, May, 2005.

[2] Scalable Vector Graphics, http://www.w3.org/Graphics/SVG/.

[3] Simple Object Access Protocol, http://www.w3.org/TR/SOAP/.

[4] J2EE SDK, available at http://java.sun.com/j2ee/.

[5] Jean Vanderdonckt, "Using Data Flow Diagrams for Supporting Task Models", *DSV-IS'98*, June, 1998.

[6] Francois Bodart, et al., "Key Activities for a Development Methodology of Interactive Applications", *Critical Issues in User Interface Systems Engineering*, Springer-Verlag, 1996.

[7] Michael F. Kleyn and Indranil Chakravarty, "EDGE - A Graph Based Tool for Specifying Interaction", *UIST'89*, pp. 1-14, 1989.

[8] Kyung H. Shin and Chae W. Yoo, "A Multiple-View Structured XML Editor Generation", *SIGPL*, KISS, 2001.

[9] XML Spy, http://www.xmlspy.com/.