XML 문서에 자동 전파하는 XML 스키마 변경 접근법

나 영 국†

요 약

XML은 자기 기술적인 특징이 있기 때문에 구조를 한정하기 위해서 DTD 또는 XML 스키마를 사용한다. XML 스키마가 아직 권고안에 불과하지만 DTD는 XML 언어가 아니고 표현에도 한계가 있기 때문에 XML 스키마의 사용이 보편화 될 것이다. XML 문서의 구조와 데이터는 XML 스키마의 잘못된 디자인, 어플리케이션의 새로운 요구 등의 여러 복합적인 이유로 변할 수가 있다. 이에 우리는 XML 스키마 진화(schema evolution)을 분석하여 wils화의 기능을 실현할 수 있는 최소한의 연산자들을 제안한다. 이러한 스키마 진화 연산자들은 XML 스키마에 순응하는 XML 문서의 수가 많을 경우, XSE가 없다면 불가능한 XML 스키마와 XML 문서의 수정을 가능하게 한다. 더욱이 연산자들은 자동적으로 XML 스키마에 등록되어 있는 모든 XML 문서의 수정위치를 찾아주고 수정 후에도 XML 문서를 well-formed 상태가 아닌 valid 상태를 유지시키는 기능을 포함하고 있다. 이 논문은 XML 스키마를 체계적으로 수정하는 첫 번째 시도이며 XML의 수정에 필요한 대부분의 연산을 제공한다. 이 논문의 연구는 XML 문서의 구조뿐만 아니라 데이터까지 쉽고 정확하게 수정하는 것을 도와주기 때문에 XML 어플리케이션의 개발과 유지를 위해 꼭 필요한 작업이다.

키워드 : XML, XML 스키마, XML 스키마 진화, XML 스키마 변경, XML 스키마 진화 관리, XML 스키마 수정, XML 문서

XML Schema Evolution Approach Assuring the Automatic Propagation to XML Documents

Young-Gook Ra†

ABSTRACT

XML has the characteristics of self-describing and uses DTD or XML schema in order to constraint its structure. Even though the XML schema is only at the stage of recommendation yet, it will be prevalently used because DTD is not itself XML and has the limitation on the expression power. The structure defined by the XML schema as well as the data of the XML documents can vary due to complex reasons. Those reasons are errors in the XML schema design, new requirements due to new applications, etc. Thus, we propose XML schema evolution operators that are extracted from the analysis of the XML schema updates. These schema evolution operators enable the XML schema updates that would have been impossible without supporting tools if there are a large number of XML documents complying the XML schema. In addition, these operators includes the function of automatically finding the update place in the XML documents which are registered to the XSE system, and maintaining the XML documents valid to the XML schema rather than merely well-formed. This paper is the first attempt to update XML schemas of the XML documents and provides the comprehensive set of schema updating operations. Our work is necessary for the XML application development and maintenance in that it helps to update the structure of the XML documents as well as the data in the easy and precise manner.

Key Words : XML, XML Schema, XML Schema Evolution, XML Schema Evolution Management, XML Schema Update, Schema Update Propagation, XML Document

1. 서 론

최근 인터넷이 보편화되고 웹이 급성장함에 따라서 많은 양의 데이터가 웹을 통해서 교환되고 있고, 대부분의 필요한 정보들은 웹을 통해서 얻을 수 있게 되었다. 하지만 웹상에 존재하는 데이터에는 구조 정보가 없어 그 의미를 파악하기 힘들고 데이터들이 대부분 표현에만 국한된 HTML로 작성되어 있어서 실제로 우리가 원하는 정확한 정보만을 찾기에는 한계가 있다. 이러한 한계를 극복하기 위해 웹 문서 내에 그 문서의 구조 정보를 같이 포함할 수 있게 하는 XML(eXtensible Markup Language)[1]이 HTML의 대안으로 등장하게 되었다. XML은 자기 기술적인 특징을 갖고 있으므로 많은 응용 프로그램에서는 XML 문서의 구조를 한정하기 위해서 DTD(Document Type Definitions) 또는 XML 스키마[2, 3]을 사용한다. 이 때 XML 문서의 구조와 데이터는 XML

† 정 회 원 : 서울시립대학교 전기전자컴퓨터학부 부교수
논문접수 : 2006년 2월 13일, 심사완료 : 2006년 6월 12일

스키마의 잘못된 디자인, XML 문서에 새로운 내용의 추가, 어플리케이션의 새로운 요구 등의 여러 복합적인 이유로 XML 스키마 및 이에 순응하는 XML 문서는 변할 필요성이 있다.

전통적으로 스키마 진화에 대한 연구는 객체지향 데이터베이스 및 관계데이터베이스에서 많이 이루어졌으며 이러한 데이터베이스 시스템에서의 스키마 변경은 스키마 자체의 변경뿐만 아니라 이 스키마에 순응하는 문서의 변경까지 이루어져야 한다. 이와 마찬가지로 XML 스키마의 변경 또한 XML 스키마뿐만 아니라 스키마에 순응하는 모든 XML 문서 역시 변경해 주어야 한다. XML 스키마의 구조가 복잡하고 더욱이 XML 스키마에 순응하는 XML 문서의 수가 많을 경우, XML Schema의 수정은 현실적으로 불가능한 작업이 될 수 있다. 이점은 앞으로 XML 어플리케이션의 개발을 방해하는 요인이 될 것이다. XML의 사용이 많아짐에 따라서 XML 스키마를 수정하는 전문적인 도구의 필요성이 커지고 있다.

사실, XML 문서의 수정과 DTD의 수정에 대한 연구가 몇몇 진행되었다. DOEM[4]는 XML 문서 뿐만 아니라 준-구조(semi-structured) 데이터[5, 6]의 변경을(update) 제안한다. 하지만 단지 데이터의 변화만을 다루고 스키마의 변화를 다루지는 않는다. EXcelon[7]은 XPath[8]을 사용하여 XML 스키마뿐만 아니라 이 스키마에 순응하는 XML 문서의 삽입(Insertion)과 삭제(Deletion)를 제공한다. 이 연산들은 강력하고 실제로 가장 많이 사용되는 연산들이기 때문에 XML 데이터베이스 시스템 수정에 있어서 큰 역할을 한다. 하지만 이동(Movement), 변화(Change)같은 연산들을 지원하지 않기 때문에 XML 수정에 있어서 다소 한정적이다. XEM[9]는 XML 문서뿐만 아니라 스키마까지 수정을 지원하며 XML 수정에 필요한 대부분의 연산을 제공한다. 하지만 스키마로써 DTD를 사용하기 때문에 XPath, XLink, XSLT (Extensible Stylesheet Language Transformation)[10, 11] 같은 XML의 기술을 스키마에 응용하여 사용할 수 없을 뿐 아니라 데이터 타입이 한정적이라는 단점이 있다.

우리는 XML 스키마와 그 스키마에 순응하는 XML 문서의 수정을 지원하며 XML 수정에 필요한 대부분의 기능을 조합해낼 수 있는 기본 연산자 집합을 제안한다. 이 논문은 XML 스키마 또는 DTD 수정 연구 분야에서 아래와 같은 공헌을 한다.

(1) 스키마 수정이 순응하는 XML 문서로 전파되는 2가지 방법을 완전하게 연구했다. 하나는 사용자 개입(intervention) 모드이고 다른 하나는 일괄처리(batch) 모드이다. XML 스키마의 일부분이 수정될 때 XML 스키마에 순응하는 모든 XML 문서까지 변화가 전파된다. 이 경우에, XSE는 자동적으로 XML 스키마에 등록되어 있는 모든 XML 문서의 수정 위치를 찾아주고 수정 후에도 XML 문서를 well-formed 상태가 아닌 valid 상태를 유지시킨다.

(2) 이 연산자 집합들은 XML 스키마를 수정하는 최초의 시도이며 XML 스키마의 수정에 필요한 대부분의 기능을 조합해낼 수 있다.

(3) 이 연산자들을 구현 한다면 GUI를 지원하여 복잡한 구조의 XML 스키마를 쉽게 이해할 수 있게 하며 등록 기능을 추가함으로써 XML 스키마와 XML 문서의 관리를 용이하게 할 수 있을 것이다.

이 논문은 다음과 같이 구성되었다. 연구의 개괄적인 접근 방식은 다음 장에서 상세하게 설명하겠다. XML 스키마 수정에 필요한 연산들은 3장에서 살펴보고 4장에서는 스키마 진화에 대한 관련 연구를 살펴보겠다. 마지막으로 5장에서는 이 논문의 결론을 살펴보면서 논문을 마치겠다.

2. 개괄적인 접근 방식

프로젝트 관리 시스템의 간단한 예를 사용해서 XML 스키마 진화의 유용성과 개요에 대해 설명하겠다. (그림 1)과 같이, 프로젝트 관리 시스템의 XML 스키마의 파일명을 "University.xsd"라고 하고 이에 순응하는 XML 문서들을 "University1.xml, University2.xml....,Universityn.xml"라고 하자. 우리는 (그림 1)(a)에서 (그림 2)(a)로 스키마를 변화시키려고 한다. 즉, (그림 1)(a)의 University.xsd에서 "Professor" 요소 및 자식 노드들을 "Project" 요소의 자식 노드에서 "Dept" 요소의 자식 노드로 이동시키려고 한다. 또 이 스키마에 순응하는 XML 문서도 변화한 스키마에 순응하도록 (그림 1)(b)를 수정된 XML 스키마에 순응하도록 (그림 2)(b)와 같이 수정하고자 한다. 우리는 아래와 같이 XML 스키마 및 XML 문서를 이동시키는 연산 'Move element'을 제안한다.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:element name="University">
  <xs:complexType>
   <xs:sequence>
    <xs:element name="Dept" maxOccurs="2" minOccurs="0">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="Project" maxOccurs="2 minOccurs="0">
        <xs:complexType>
         <xs:sequence>
          <xs:element name="Professor">
           <xs:complexType>
            <xs:sequence>
             <xs:element name="Student" type="xs:string"/>
            </xs:sequence>
            <xs:attribute name="Prof_Name" type="xs:string"/>
           </xs:complexType>
          </xs:element>
         </xs:sequence>
        </xs:complexType>
       </xs:element>
       <xs:attribute name="Proj_Name" type="xs:string"/>
      </xs:sequence>
      <xs:attribute name="Dept_Name" type="xs:string"/>
     </xs:complexType>
    </xs:element>
   </xs:sequence>
  </xs:complexType>
 </xs:element>
</xs:schema>
```

(a) University.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<University>
xmlns:xs="http://www.w3.org/2001/10/XMLSchema-instance"
xs:noNamespaceSchemaLocation="University.xsd">
  <Dept Dept_Name="EEC">
    <Project Proj_Name="Research1">
      <Professor Prof_Name="Lee K.">
       <Student>Kent Tamura</Student>
      </Professor>
    </Project>
    <Project Proj_Name="Research2">
      <Professor Prof_Name="Lee K.">
       <Student>Kim J.</Student>
      </Professor>
    </Project>
  </Dept>
  <Dept Dept_Name="Arch">
    <Project Proj_Name="Research3">
      <Professor Name="Hang T.M.">
        <Student>Jin S.H.</Student>
      </Professor>
    </Project>
  </Dept>
</University>
```

(b)University1.xml

(그림 1) 수정전의 XML 스키마 및 순응하는 XML 문서 moveEle (Professor, string, University/Dept/Project/Professor, null, University/Dept, 1, null)

이 연산은 XML Schema를 먼저 수정한다. 경로가 'University/Dept/Project/Professor' 이고 이름이 'Professor'인 요소 및 이 요소의 자식 요소를 경로가 'University/Dept'인 'Dept' 요소의 '1'번째 자식 요소로 이동한다. 이 때 이동전 부모요소인 'Project' 요소는 복잡타입에서 단순타입으로 바뀌기 때문에 데이터 타입으로 'string'을 지정해준다. 만약 'Project' 요소가 복합 모델 속성을 가지고 있다면 복합 모델 속성이 삭제된다. 4번째와 7번째 파라미터는 이동 후 부모요소의 타입이 변할 때 사용되므로 이 경우 'null'로 지정해준다.

"moveEle" 연산은 다음으로 XML 스키마에 순응하는 XML 문서들을 수정한다. 이 연산은 professor와 그 자식 요소들을 project 요소들의 자식에서 그 상위인 dept 요소들의 자식으로 이동시킨다. 이때 동일한 professor가 여러 project의 자식으로 있을 수 있다. 이것은 한 professor가 여러 project들에 참여할 수 있음을 의미한다. "moveEle" 연산에 의해 professor와 project의 상관성이 제거되면, 중복된 professor 요소들은 dept 자식으로 하나의 요소가 된다. (그림 1) (b) 에서, <professor prof_name = "Lee K."> 요소는 <project proj_name = "research1"> 요소와 <project proj_name = "research2"> 요소의 자식이다. 이는 "Lee K." professor가 "research1"과 "research2" 프로젝트에 관련되어 있음을 나타낸다. "moveEle" 연산에 의해서 project와 professor의 상관성이 없어지면 두 프로젝트의 자식으로 있는 두개의 "Lee K." 요소는 (그림 2) (b)에서처럼 "EEC" dept 요소의 하나의 자식 요소가 된다. "moveEle" 연산에 의해 professor 요소가 이동하면 이의 자식인 student 요소도 함께 이동한다. 위에서 보듯이 중복된 professor 요소가 이 연산에 의해 하나의 professor 요소로 표현될 때 중복된 professor 요소의 자식으로 있던 여러 student 요소는 이 하나의 professor 요소의 자식으로 모아진다. (그림 1) (b) 의 student "Kent Tamura"와 "Kim J."는 두개의 "Lee K." professor 요소의 자식이다. 이 두 'student' 요소는 professor 요소가 하나의 요소로 통합될 때 (그림 2) (b)에서 보듯이 하나의 professor "Lee K."의 두 개의 자식 요소로 이동한다. dept 요소와 관계된 student 요소의 관련성은 수정 후에도 유지됨을 주의하자.

이 'moveEle' 연산을 XML Spy등의 편집기를 이용해서 수작업을 통해서 수행하는 것은 매우 복잡한 작업이다. 만약 위와 같은 예를 수작업을 통해서 수행한다면 아래와 같은 순서로 수행해야 할 것이다. (1) (그림 1) (b)에서 "EEC" dept 요소의 후손 요소들 중에서 professor 요소들을 전부 찾아내고; (2) professor 요소의 중복을 제거하고 (그림 2) (b)와 같이 "EEC" dept 요소의 자식으로 이동하고; (3) 각각의 'professor' 요소의 자식 요소인 'student' 요소들을 찾고; (4) 동일한 professor의 자식인 student 요소들을 모아서 해당 'professor' 요소의 자식으로 이동시켜야 한다. 이 작업은 동일한 'dept' 요소의 'project', 'professor', 'student' 수들이 증가하면 더욱 복잡해 질것이다. 만약 이 작업을 수작업으로 한다면 시간이 많이 걸릴 뿐만 아니라 에러 없이 정확한 수정을 하기에는 어려움이 있다. 이 문제는 수정해야 하는 XML 문서가 수백 혹은 수천 개라면 더욱 어려워진다.

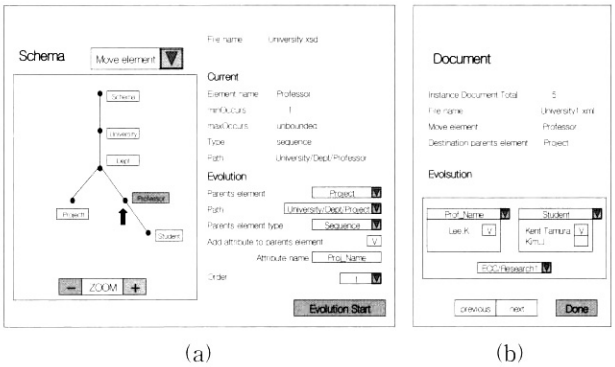
이 논문은 스키마 수정에 기본 연산자들의 사용을 보여주기 위하여 (그림 3)과 같은 GUI를 가진 가상의 시스템을 제안한다. (그림 3) (a)에서, 우리는 위의 풀다운 메뉴에서 "move element"를 선택하고, 그 아래의 캔버스에서 "Professor" 노드를 마우스를 사용해서 선택한다. (그림 3) (a)의 오른쪽 아래에는 Evolution 부분이 있다. 여기의 1 번째 풀다운 메뉴에서 "professor" 요소의 이동 목적지의 부모 요소가 되는 "project"를 선택한다. 2 번째 풀다운 메뉴에서 이동 목적지 요소의 경로 "university/dept/project"를 선택한다. "professor" 요소가 이동함으로써 이동 목적지의 부모인 "project" 요소가 단순 타입에서 복잡 타입으로 바뀌므로 3 번째 풀다운 메뉴에서 복잡 타입의 세 가지 내용 모델 중 하나인 "sequence"를 선택해준다. 바로 밑의 'add attribute to parent element'는 이동 목적지 부모 요소가 단순 타입 일 때의 데이터 값을 저장할 수 있는 속성을 더할 것 인지를 지정하고 그 속성 이름으로 'proj_name'을 입력한다. 다음으로 'Evolution start' 버튼을 선택하면 XML 스키마의 수정이 완성되고 바뀐 스키마에 순응하는 XML 문서를 수정할 수 있는 창을 보여 준다.

(그림 3) (b) 에서 'Instance Document Total'은 (그림 3) (a)의 XML 스키마에 순응하는 XML 문서의 총 수를 나타낸다. 아래의 'File name'은 현재 수정하는 문서의 이름, 'University1.xml'을 나타낸다. 이동 할 요소의 이름 "Professor" 와 이동 목적지의 부모 요소 이름 "Project"를 사용자에게 보여준다. 아래의 'Evolution' 에서는 이동할 요소와 목적지 부모 요소가 어떻게 관련되어 있는지를 보여준다. 먼저 "Dept" 요소와 "Project" 요소의 속성 데이터 값을 "EEC/Research1"로 선택하면 "EEC" dept에 속한 "professor" 요소를 사용자에게 보여준다. 이 경우, 요소는 "prof_name" 속성의 값에 의해서 구별된다. (그림 3) (b)에서,

"research1" project 요소의 자식이 될 수 있는 "professor" 요소의 속성 값을 보여준다. 우리의 예에서는 "Lee K." 이다. "Lee K."를 선택하면 "Lee K."의 자식의 될 수 있고 (즉, "Lee K."의 지도학생이며) "research1" project 요소의 후손이 될 수 있는 (즉, "Research1" project에 참여하는) "student" 요소를 사용자에게 보여준다. 우리의 예에서는 "Kent Tamura", "Kim J."를 보여준다. 사용자가 "Kent Tamura"를 선택하면 (그림 3) (b)와 같이 이 요소가 이름이 "Lee K." 인 Professor 요소의 자식 요소로 이동한다. 다른 XML 문서도 'Previous' 또는 'Next'를 선택해서 유사하게 수정할 수 있다.

3. XML 스키마 변경 연산자

본 논문에서는 19개의 XML 스키마 진화 연산들을 <표 1>과 같이 제안한다. 요소를 삽입, 삭제, 이동, 변화, 합병, 분해하는 각각의 연산들과 요소들의 집단으로 이루어진 그룹(group)을 삽입, 삭제, 변화시키는 각각의 연산들이 있다. 또한 속성을 삽입, 삭제, 변화시키는 각각의 연산과 연산들의 집단으로 이루어진 속성그룹(attributegroup)을 삽입, 삭제, 변화시키는 각각의 연산들로 구성된다. 본 논문에서는 "addEle", "dropEle", "moveEle", "mergeEle", "partitionEle", "changeAtt" 6개의 연산을 설명하겠다.



(a) (b)

(그림 3) XSE의 사용자 인터페이스 (Move element)

3.1 Move element

문법 : moveEle(n, pt, p, dt, d, f, da)

의미 : 이 연산은 이름이 'n'인 요소를 위치 'p'에서 위치 'd'인 목적지 요소의 'f'번째 자식으로 이동시킨다. 'n' 요소가 자식 요소를 포함 할 경우에는 자식 요소도 함께 이동된다. 파라미터 'p'와 'd'는 'n' 요소의 원래 경로와 목적지 경로를 각각 나타낸다. 파라미터 'pt'는 'n' 요소의 이동에 의해서 부모 요소가 복잡타입에서 단순타입으로 바뀌는 경우에 사용한다.

XML 스키마에서 'n' 요소를 'p'에서 'd'로 이동 시킨 후에, 순응하는 XML 문서는 'n' 요소에 해당하는 문서 요소 (nE_i)를 'p'에 해당하는 문서 경로에서 'd'에 해당하는 문서 경로에 있는 'd'의 마지막 요소에 해당하는 문서 요소 (tE_j)의 'f'번째 자식요소로

〈표 1〉 XSE에서 지원하는 연산들의 집합

연산	설명
addEle(n, pt, s1, s2, nt, p, f)	이름이 'n'인 요소를 위치 'p'의 'f' 번째 자식으로 추가한다.
dropEle(n, pt, p)	이름이 'n'이고 경로(path)가 'p'인 요소를 삭제한다.
moveEle(n, pt, p, dt, d, f)	이름이 'n'인 요소를 경로 'p'에서 새로운 위치 'd'의 'f'번째 자식으로 이동시킨다.
changeEle(n, n1,pt, s1, s2, nt, p,)	이름이 'n'이고 경로가 'p'인 요소를 새로운 이름 'n2', 카디널러티 's1', 's2', 데이터 타입 'nt', 복잡타입(complexType) 'pt'로 변화시킨다.
mergeEle(n1, n2,...,ni, n, s1, s2, t, p)	이름이 각각 'n1', 'n2',...',ni'인 element를 이름이 'n'인 하나의 element로 합병한다.
partitionEle(n, n1, n2,...,ni, p)	이름이 'n'인 요소를 이름이 각각 'n1', 'n2',...',ni'인 여러 개의 요소로 분리시킨다.
addGro(n)	이름이 'n'인 그룹을 추가한다.
addGroRef(n, p)	이름이 'n'인 그룹 참조(group ref)를 경로 'p'에 추가한다.
dropGro(n)	이름이 'n'인 그룹을 삭제한다.
dropGroRef(n, p)	이름이 'n'인 그룹 참조를 경로 'p'에서 삭제한다.
changeGro(n, n1)	이름이 'n'인 그룹을 새로운 이름 'n1'으로 변화시킨다.
addAtt(n, nt, p)	이름이 'n'이고 데이터 타입이 'nt'인 속성을 경로 'p'에 추가한다.
dropAtt(n, p)	이름이 'n'인 속성을 경로 'p'에서 삭제한다.
changeAtt(n, n1, nt, u, p)	이름이 'n'이고 경로가 'p'인 속성을 새로운 이름 'n2', 카디널러티 's1', 's2', 데이터 타입'nt', 출현횟수 'u'로 변화시킨다.
addAttGro(n)	이름이 'n'인 속성그룹을 추가한다.
addAttGroRef(n, p)	이름이 'n'인 속성그룹 참조(attributegroup ref)를 경로 'p'에서 추가한다.
dropAttGro(n)	이름이 'n'인 속성그룹을 삭제한다.
dropAttGroRef(n, p)	이름이 'n'인 속성그룹 참조를 경로 'p'에서 제거한다.
changeAttGro(n, n1)	이름이 'n'인 속성그룹의 이름을 'n1'으로 변화시킨다.

이동시켜주어야 한다. XML 문서는 요소의 이동시킬 때 요소를 자동으로 이동시키는 경우와 사용자 개입에 의해서 이동시키는 2가지 경우가 있다. 먼저 요소를 자동으로 이동시키는 경우는 'd'의 마지막 요소가 'n' 요소의 조상인 경우이다. 이 경우, nE_i는 자동적으로 원래의 XML 문서의 유일한 조상인 tE_j의 직접적인 자식이 된다. 두 번째 경우는, 사용자가 이동할 각각의 nEi 의 부모 요소가 될 tE_j를 결정해야 한다. 이것은 가상 시스템의 GUI를 이용해서 수정할 수 있다.

요소를 이동시키는 경우는 4가지가 있다. 먼저 'n' 요소의 이동 전 부모요소가 리프 노드로 바뀌고 이동 후의 부모노드가 단순타입에서 복잡타입으로 바꾸는 경우이다. 이 경우, 파라미터 'pt'에 데이터 타입을 지정해주고 파라미터 'dt'에 복잡타입의 3가지 내용

모델 중 하나를 지정해준다. 2번째 경우로, 'n'요소의 이동 전 부모 노드가 복잡타입을 유지하고 이동 목적지 부모 노드도 복잡타입을 유지하게 하는 경우이다. 이 경우, 파라미터 'pt'에 'null' 을 지정해주고 파라미터 'dt'에 'null'을 지정해준다. 3번째 경우로, 'n'요소의 이동전 부모 노드가 리프 노드로 바뀌고 이동목적지 부모 노드가 복잡타입을 유지하는 방법이다. 이 경우, 파라미터 'pt'에 데이터 타입을 지정해주고 파라미터 'dt'에 'null'을 지정해준다. 마지막으로, 'n'요소의 이동전 부모 노드가 복잡타입을 유지하고 이동 후의 부모노드가 단순타입에서 복잡타입으로 바꾸는 경우이다. 이 경우, 파라미터 'pt'에 'null'을 지정해주며 파라미터 'dt'에 복잡타입의 3가지 내용 모델 중 하나를 지정해준다.

Move element 알고리즘 'Move element' 연산의 알고리즘은 (그림 4)와 같다. 먼저 'moveElemFiles'은 XML 스키마와 XML 문서 파일들을 DOMs (Document Object Model) 로 변환하며 각각의 파일들을 schemaDom과 xmlDomi으로 바꾸어준다. 다음으로, 'moveEle' 는 schemaDom에서 이동하는 요소의 위치를 나타내는 경로 'p'의 마지막 노드를 얻고 요소가 이동할 목적지의 위치를 나타내는 경로 'd'의 마지막 노드를 얻는다. 'move element' 는 연산의 의미에 맞게 XML 스키마의 DOMs를 변화시키는 'move ElemSchema'를 부른다. XML 문서의 DOMs는 'getDocumentList' 에 의해서 순차적으로 접근되며 각 문서 DOM은 스키마의 변화에 따르는 'moveElemDocument'에 의해서 변화한다.

```
/* n : 이동 할 요소 이름          pt : n의 부모 요소 타입
p : n의 경로                     dt : 이동 목적지 부모 요소의 타입
d : 이동 목적지 부모 요소의 경로
f : 'n' 요소가 이동 목적지의 몇 번째 자식으로 이동하는지를 나타내는 순서
da : 이동 목적지 부모 요소가 복잡타입으로 바뀔 때 추가되는 속성 이름  */

moveElemFiles (schemaFile, xmlFile1,....,xmlFile n)
schemaDom = parse(schemaFile)
   for(i=1, n)  {
      xmlDoc i = parse(xmlFile i)
      moveElem(p, d, schemaDom, xmlDoc i)
}

moveEle(n, pt, p, dt, d, f, da, schemaDom) {
      // 경로 p로부터 이동할 요소 nE을 구함
    nE = getLastElemFromPath(p, schemaDom);
      // 경로 d로부터 이동 목적지 부모 요소 tE를 구함
    tE = getLastElemFromPath(d, schemaDom);
      // XML 스키마에서 요소를 이동함
    moveElemSchema(nE, tE, pt, dt, f, da);
      // XML 문서에서 요소를 이동함
    docList = getDocumentList(xmlDom1, xmlDom2,....xmlDom n);
        for (i=1; i=n; i++)  {
          moveElemDocument(p,d,xmlDoc i);      XML 문서를 수정함
        }
}
moveElemSchema(nE, tE, pt, dt, f, da) {
    if nE의 부모 요소가 리프 노드가 되면
        then 부모 요소를 'pt'를 갖는 단순타입으로 변화시킴;
    if tE가 리프 노드이면
        then tE를 'dt'를 갖는 복잡타입으로 변화시킴   and
        'da' 속성을 이 요소에 추가함;

        nE을 tE의 f번째 자식 요소로 이동시킴;
}
```

```
moveElemDocument(nE, tE) {
      // nE의 인스턴스 요소들의 배열
   nEInstanceArray = {nE1, nE2,....,nEp};
      // tE의 인스턴스 요소들의 배열
   tEInstanceArray = {tE1, tE2,....tEq};
      // nE의 자식 요소들의 인스타스 요소들 배열
   chnEInstanceArray = {nE11, nE12,...nE1k};
      // nE에 해당하는 문서 요소를 찾음
   nEInstanceArray = getInstancesFromSchemaNode(nE);
      // tE에 해당하는 문서 요소를 찾음
   tEInstanceArray = getInstancesFromSchemaNode(tE);
      // EinstanceArray 요소들의 자식 요소를 찾음
   chnEInstanceArray = getIchildrenNodes(nEinstanceArray);

   if nEi 및 자식 요소가 자신의 조상 요소로 이동하면 then nEi 및 자식 소를
     tEj의 자식으로 이동시킴;
   else if {
        if tE가 leaf 노드라면
        then tEj의 데이터 값을 'da' 속성에 이동시킨 후 속성 데이터 값과 nE 값
             을 보여 준다;
        else if tEj의 데이터 값 또는 속성 데이터 값과 nEi 데이터 값을 보여 준다;
         // p=size(tEinstanceArray)
        for (i=1; i=p; i++)
             nEm의 부모를 어떤 tEi로 할지를 사용자에게 결정 하게 한다;

        if nE가 자식 요소가 있다면
        then nEi의 속성 데이터 값과 자식노드 nEim의 데이터 값을 보여 준다
             for (i=1; i=m; i++)
                  nEim의 부모를 어떤 nEj로 할지를 사용자에게 결정하게 한다.
      }
}
```

(그림 4) 'Move element'의 알고리즘

'moveElemSchema'는 'nE' 노드를 목적지 부모 요소인 'tE'의 자식으로 이동시키며 이때 필요한 추가적인 파라미터는 'pt', 'dt', 'f' 그리고 'da'이다. 이것은 'nE'를 'tE'의 'f'번째 자식으로 이동시킴으로써 'schemaDOM'을 변화시킨다. 이 때 'nE'의 이동전 부모 요소가 리프 노드로 변화한다면 부모 요소의 데이터 타입은 'pt'로 변화한다. 또 'tE'가 리프노드에서 복잡타입으로 변화하면 속성 'da'가 'tE'의 데이터 값을 보호하기 위해서 추가된다.

'moveElemDocument'는 파라미터 'nE'와 'tE'를 취하며 스키마 DOM의 nE 노드와 연관된 문서 DOM의 노드를 찾는다. 이 때 'getInstancesFromSchemaNode'를 사용해서 노드를 찾아서 'nEInstanceArray'에 저장한다. 또한, 스키마 노드 'tE'와 연관된 문서 DOM의 노드들은 찾아서 'tEInstanceArray'에 저장한다. 같은 방식으로 'nEInstanceArray'의 자식 노드를 'getChildren Nodes'를 사용해서 찾은 후에 'chnEInstanceArray'에 저장한다. 만약 'nE'가 자신의 조상노드의 자식으로 이동한다면 사용자 개입 없이 자동으로 이동이 이루어진다. 'nEi' 노드의 데이터 값이 같다면 구별되는 하나의 노드만이 이동하고 나머지는 삭제된다. 이것의 예는 2장에 있다. 그리고 'nEim'도 자신의 부모노드와 함께 이동한다.

만약 'nE'가 자신의 조상의 자식으로 이동하지 않는다면 'nEi'와 'tEj' 사이의 관계는 사용자가 개입해야 한다. 먼저 'tE'가 리프노드라면 XSE는 속성 'da'에 'tEj'의 데이터 값을 저장한다. 다음으로 'tEj'의 데이터 값 또는 'da'의 속성 값과 'nEi'의 데이터 값을 보여준다. 사용자는 XSE를 통해서 'tEj'와 'nEi'의 관계가 맞게 'nEi'를 'tEj'의 자식으로 직접 이동시켜주어야 한다. 만약 'nEi'가 자식 요소 'nEim'을 갖고 있다면, XSE는 이 자식 요소의 데이터 값을

'tE_i'의 데이터 값과 함께 보여준다. 사용자는 역시 'nE_i'와 'nE_{lm}'의 관계가 맞도록 'nE_{lm}'을 'nE_i'의 자식으로 직접 이동시켜주어야 한다.

3.2 Add element

문법 : addEle(n, pt, s1, s2, nt, p, f, da)

의미 : 이 연산은 이름이 'n'인 요소를 위치 'p'에 위치한 요소의 'f'번째 자식으로 요소를 추가한다. 파라미터 'p'는 추가할 이름이 'n'인 요소의 경로이다. 파라미터 'nt'는 추가할 요소의 데이터 타입을 나타내며 요소가 리프 노드로 추가될 때 지정해준다. 만약 요소가 리프 노드에 추가되지 않는다면 파라미터 'nt'에 'null'을 지정해준다. 파라미터 'pt'는 추가하는 'n' 요소의 부모 요소가 단순타입에서 복잡타입으로 바뀌는 경우에 사용하며 복잡타입의 내용모델 3가지 중 하나를 입력한다. 이 때 부모 요소가 복잡타입을 유지하는 경우에는 'null' 을 지정해 준다. 파라미터 's1'과 's2'는 추가하는 요소의 카디널러티를 나타낸다. 파라미터 'da'는 'n' 요소의 부모 요소가 단순타입에서 복잡타입으로 바뀔 때 이 요소의 데이터 값들을 보존하기 위해 이름이 'da'인 속성을 추가해 준다.

XML 스키마에서 'n' 요소를 경로 'p'에 추가시킨 이후에, 순응하는 XML 문서에도 요소가 자동으로 추가된다. 즉, 'n' 요소에 해당하는 문서 요소 (nE_i)를 위치 'p'에 해당하는 문서 경로에 위치한 요소의 'f'번째 자식으로 카디널러티의 최소 값만큼 추가한다.(minOccurs가 0일 경우에는 하나의 요소가 추가된다)

XML 문서에 요소를 추가하는 경우가 2가지 있다. 'addEle'는 사용자 개입을 요구할 수도 있고 자동으로 수행 할 수도 있다. 첫 번째 경우는, XML 문서의 요소가 리프 노드로써 추가되는 경우에는 사용자가 XSE가 제공하는 인터페이스를 이용해서 데이터를 직접 넣어주어야 한다. 두 번째 경우는 요소를 XML 문서의 중간 (inner) 노드로 추가되는 경우에는 사용자 개입 없이 자동으로 요소의 추가가 이루어진다.

Add element 알고리즘 'Add element'연산의 알고리즘은 (그림 5)와 같다. 'addElemSchema'는 이름이 'n'인 'nE'노드를 'tE'의 자식노드로 추가하며 이때 필요한 추가적인 파라미터는 'nt', 'pt', 's1', 's2', 'f' 그리고 'da'이다. 이것은 'nE'를 'tE'의 'f'번째 자식으로 추가시킴으로써 'schemaDOM'을 변화시킨다. 이 때 'tE' 요소가 리프 노드라면 파라미터 'pt'를 갖는 복잡타입으로 바뀌고 데이터 값을 보존하기 위해서 'da' 속성을 추가시킨다.

'addElemDocument'는 파라미터 'tE'를 취하며 스키마 DOM의 nE 노드와 연관된 문서 DOM의 노드를 찾는다. 이 때 'getInstances FromSchemaNode'를 사용해서 노드를 찾고 찾은 노드를 'tEInstanceArray'에 저장한다. 'tEInstanceArray'의 각 노드 'tE_j'를 부모로 하는 새로운 노드 'nE_i'를 추가한다. 만약 'tE_j'를 리프 노드로써 추가한다면 'nE_i'의 데이터 값을 사용자에게 추가하도록 요구한다. 이 때 'nE_i'가 리프 노드에서 중간 노드로 바뀐다면 'tE_j'의 데이터 값을 'da' 속성 값으로 이동시킨다. 다음으로 'nE_i'를 'tE_j'의 f번째 자식부터 's1'수만큼 자동으로 추가시킨다.

```
// nt : 추가할 요소의 타입   s1 : 요소의 최소 인스턴스 개수
// s2 : 요소의 최대 인스턴스 개수

addEle(n, pt, s1, s2, nt, p, f, da, schemaDom) {
      // 경로 p로부터 이동할 요소의 부모 tE을 구함
    nE = getLastElemFromPath(p, schemaDom);
      // XML 스키마에서 요소를 추가함
    moveElemSchema(nE, tE, nt, pt, s1, s2, f, da);
      // XML 문서에서 요소를 이동함
    docList = getDocumentList(xmlDom1, xmlDom2,...,xmlDomn);
        for (i=1; i=n; i++) {
          addElemDocument(p, xmlDoci);  // XML 문서를 수정함
        }
}

addElemSchema(nE, tE, nt, pt, s1, s2, f, da) {
    if tE가 리프 노드이면
    then tE를 'pt'타입을 갖는 복잡타입으로 바꾸고 'da' 속성을 추가시킴;
    if nE가 중간 노드로 추가되면
    then nE는 복잡타입이 된다;

      's1', 's2'인 nE를 tE의 f번째 자식 요소로 추가시킴;
}
```

```
addElemDocument(tE, xmlDoci) {
    // tE의 인스턴스 요소들의 배열
  tEInstanceArray = {tE1, tE2,...,tEq};
    // nE에 해당하는 문서 요소를 찾음
  tEInstanceArray = getInstancesFromSchemaNode(tE);
    // q=size(tEInstanceArray)
  for (j=1; j=q; j++) {
    if nE가 리프노드라면
    then {
      if tE가 리프노드라면
      then tEj의 데이터 값을 'da' 속성으로 이동시킴;

      nEi의 데이터 값을 사용자에게 추가하기를 요구함;
      }

    nEi를 tEj의 f번째 자식부터 's1'만큼 추가시킴;
    }
}
```

(그림 5) 'Add element'의 알고리즘

3.3 Drop element

문법 : dropEle(n, pt, p, da)

의미 : 이 연산은 이름이 'n'이고 경로가 'p'인 요소를 삭제한다. 파라미터 'pt'는 이름이 'n' 인 요소의 부모 요소가 'n' 요소의 삭제로 인해서 복잡타입에서 단순타입으로 바뀌는 경우에 부모요소의 타입을 지정해준다. 'n' 요소의 부모 요소가 복잡타입을 유지하는 경우에는 'null'을 지정한다. 파라미터 'da'는 'n' 요소의 부모 요소가 복잡타입에서 단순타입으로 바뀔 때 이름이 'da'인 속성을 삭제하며 속성의 데이터 값은 'n' 요소의 부모 요소의 데이터 값으로 이동하게 한다. 'n' 요소의 부모 요소가 복잡타입을 유지하는 경우에는 'null'을 지정해 준다.

XML 스키마에서 위치가 'p' 이고 이름이 'n' 인 요소를 삭제시킨 후에, 순응하는 XML 문서에서 경로 'p'에 해당하는 문서 경로에

```
dropEle(n, pt, p, schemaDom) {
    // 경로 p로부터 삭제할 요소 nE를 구함
    nE = getLastElemFromPath(p, schemaDom);
    // XML 스키마에서 요소를 삭제함
    dropElemSchema(nE, pt,);
    // XML 문서에서 요소를 삭제함
    docList = getDocumentList(xmlDom1, xmlDom2,...,xmlDomn);
        for (i=1; i=n; i++) {
          dropElemDocument(p, xmlDoci);      XML 문서를 수정함
        }
}

dropElemSchema(nE, pt, da, schemaDom) {
    if nE의 부모 요소가 리프 노드가 된다면
    then nE의 부모 요소의 'da' 속성을 삭제하고 데이터 타입이
        'pt'인 요소로 변화시킴;

    nE를 삭제함;
}
```

```
dropElemDocument(nE, xmlDoc) {
    // nE의 인스턴스 요소들의 배열
  nEInstanceArray = {nE1, nE2,...,nEq};
    // nE에 해당하는 문서 요소를 찾음
  nEInstanceArray = getInstancesFromSchemaNode(nE);
    // q=size(nEinstanceArray)
  for (i=1; i=q; i++) {
    if nEi의 부모 요소가 리프 노드가 된다면
    then nEi의 부모 요소의 속성을 삭제하고 속성 값을 요소 데이터
        값으로 이동시킴;

    nEi를 삭제함;
    }
}
```

(그림 6) 'Drop element' 의 알고리즘

위치하고 'n' 요소에 해당하는 문서 요소 (nE_i)를 삭제해야 한다. 만약 삭제되는 요소의 부모 요소가 리프 노드가 된다면 연산에서 지정한 이 요소의 속성 데이터 값이 요소의 데이터 값으로 자동으로 이동하게 된다.

Drop element 알고리즘 'Drop element' 연산의 알고리즘은 (그림 6)과 같다. 'dropElemSchema'는 'schemaDom'에서 경로 'p'에 위치한 'nE' 노드를 삭제하며 이때 필요한 추가적인 파라미터는 'pt'이다. 이 때 'tE' 요소의 부모 노드가 리프 노드로 변한다면 이름이 'da'인 속성을 삭제하고 단순타입 'pt'를 갖는 요소로 변화시킨다.

'dropElemDocument'는 파라미터 'nE'를 취하며 스키마 DOM의 'nE' 노드와 연관된 노드를 문서 Dom 'xmlDoc$_i$'에서 찾는다. 이 때 'getInstancesFromSchemaNode'를 사용해서 노드를 찾고 찾은 노드를 'nEInstanceArray'에 저장한다. 'nEInstanceArray'의 각 노드 'nE_i'에 대하여 만약 'nE_i'의 부모 노드가 리프 노드로 변한다면 이름이 'da'인 속성을 삭제하고 속성 값을 부모 요소의 데이터 값으로 이동시킨다. 다음으로 'nE_i'를 삭제한다.

3.4 Merge element

의미 : mergeEle(n_1, n_2,...,n_m, n, p)

문법 : 이 연산은 각각의 이름이 'n_1', 'n_2',...,'n_m'이고 부모 요소의 위치가 'p'인 'm'개의 요소들을 이름이 'n'인 하나의 요소로 합병한다. 이름이 각각 'n_1', 'n_2',...,'n_m'인 요소들은 리프 노드에 위치하며 동일한 부모 노드를 갖는 형제 노드로 구성되어 있고 카디낼러티 데이터 타입이 동일하며 이름이 'n'인 요소의 데이터 타입은 'n_1', 'n_2',...,'n_m'와 같고 카디낼러티 max(n_1, n_2,...,n_m)로 지정한다.

XML 스키마에서 부모요소의 경로가 'p'이고 이름이 'n_1', 'n_2',...,'n_m'인 'm'개의 요소들을 이름이 'n'인 하나의 요소로 합병시킨 이후, 순응하는 XML 문서도 수정시켜주어야 한다. 새로운 요소 'n'에 해당하는 문서 요소들의 데이터 값은 'n_1', 'n_2',...,'n_m' 요소들의 데이터 값을 순서대로 합병한 것과 같고 'n'에 해당하는 문서 요소들의 카디낼러티 'n_1, n_2,...,n_m'의 카디낼러티 최대 값이다.

Merge element 알고리즘 'Merge element' 연산의 알고리즘은 (그림 7)과 같다. 'MergeEle'에서 'getLastElemFromPath'를 이용하여 'n_1', 'n_2',...,'n_m'에 해당하는 schemaDom 노드 'rE_1', 'rE_2',...,'rE_m'를 구한다. 'mergeElemSchema'는 'rE_1', 'rE_2',...,'rE_m' 노드를 'nE' 노드로 합병한다. 즉, 'rE_1', 'rE_2',...,'rE_m'를 삭제하고 'nE'를 생성함으로써 'schemaDOM'을 변화시킨다.

'mergeElemDocument'는 파라미터 'rE_1', 'rE_2',...,'rE_m', 'n_E', 'xmlDoc$_i$'를 취하며 스키마 DOM의 'rE_1', 'rE_2',...,'rE_m' 노드에 해당하는 노드를 xmlDoc$_i$에서 찾는다. 그리고 찾은 'rE_1, rE_2,...,rE_m'의 인스턴스 요소들을 'nEInstanceArray$_i$' ($1 \le i \le m$)에 저장한다. 'nEInstanceArray$_i$[j]' 인덱스 j 각각에 대하여 ΣnEinstance Array$_i$[j]를 구하여 nEValue$_j$에 저장하고 새로운 노드 nE_j를 생성한다. 그리고 'nEValue$_j$'를 'nE_j'의 데이터 요소의 데이터 값으로 이동시킨다.

```
mergeEle(n1, n2,...,nm, n, s1, s2, nt, p, schemaDom) {
    // 부모 요소의 경로 p로부터 합병할 요소 rE1, rE2,...,rEm를 구함
    rEi = getLastElemFromPath(p, schemaDom);
    // XML 스키마에서 rE1, rE2,...,rEm를 nE로 합병함
    mergeElemSchema(rE1, rE2,...,rEm, n, s1,s2, nt);
    // XML 문서에서 요소를 합병함
    docList = getDocumentList(xmlDom1, xmlDom2,...,xmlDomn);
        XML 문서를 수정함
        for (i=1; i=p; i++) {
          mergeElemDocument(rE1, rE2,...,rEm, nE, xmlDoci);
        }
}

mergeElemSchema(rE1, rE2,...,rEm, nE) {
    rE1, rE2,...,rEm를 삭제하고 nE를 생성함;
}
```

```
mergeElemDocument(rE1, rE2,...,rEm, nE, xmlDoci) {
    for (i=1; i=m; i++)
        rE1, rE2,...,rEm의 인스턴스 요소들을 찾아서 저장
    nEInstanceArray = getInstancesFromSchemaNode(rEi);
        n=max(size(rEinstanceArrayi))
    for (i=1; i=n; i++) {
        새로운 노드 nEi를 생성;
        nEInstanceArrayi[j] (i≤m)의 값들을 합병해서 nEValuej로 저장;
        nEValuej를 nEj의 데이터 요소의 데이터 값으로 이동 시킨다;
        }
}
```

(그림 7) 'Merge element' 의 알고리즘

3.5 Partition element

문법 : partitionEle(n, n_1, n_2,...,n_m, p)

의미 : 이 연산은 경로가 'p'이고 이름이 'n' 인 요소를 이름이 각각 'n_1', 'n_2',...,'n_m' 인 'm' 개의 요소로 분할한다. 'n'은 분할할 요소의 이름이며 'n_1', 'n_2',...,'n_m'는 각각 분할되어 생기는 요소의 이름이며 분할하려고 하는 개수 'm' 만큼 파라미터를 입력한다. 이 때 요소의 분할은 리프 요소만으로 국한된다. 또 분할되는 각각의 요소들의 데이터 타입, 카디널러티 'n' 요소로와 같다.

XML 스키마에서 경로가 'p' 이고 이름이 'n' 인 요소를 이름이 각각 'n_1', 'n_2',...,'n_m'인 'm' 개의 요소로 분할한 후에, 순응하는 XML 문서를 수정해 주어야 한다. 즉, 경로 'p' 에 해당하는 문서 경로에 위치한 이름이 'n' 인 요소에 해당하는 문서 요소들을 이름이 'n_1', 'n_2',...,'n_i'인 문서 요소로 분할시켜주어야 한다. 이 경우 이름이 'n'인 요소에 해당하는 문서 요소들의 데이터 값은 삭제되며 이름이 'n_1', 'n_2',...,'n_m'인 문서 요소의 데이터 값은 XSE 시스템을 이용해서 사용자가 직접 지정해 주어야 한다.

Partition element 알고리즘 이 연산자의 알고리즘의 3.4절의 Merge 연산자의 역함수이므로 반대 기능을 수행한다.

3.6 Change Attribute

문법 : changeAtt(n, n1, pt, u, p)

의미 : 이 연산은 위치가 'p' 이고 이름이 'n' 인 속성을 새로운 이름 'n1', 새로운 데이터 타입 'pt', 새로운 출현 횟수 'u'로 변화시켜 준다. 파라미터 'u' 는 속성의 출현 횟수인 use를 나타내며 'required', 'optional', 'prohibited' 중 하나의 값을 선택해준다. 파라미터 'p' 는 이름이 'n' 인 속성을 소유한 요소의 경로를 나타낸다. 만약 파라미터 'n1', 'pt', 'u' 중에 변경을 원하지 않는 파라미터는 'null'값을 지정해준다.

XML 스키마의 속성을 변화시켜준 후 순응하는 XML 문서의 속성 역시 변경시켜 주어야 한다. 즉 경로 'p' 에 해당하는 문서 경로에서 'n' 속성에 해당하는 문서 속성의 이름을 'n'에서 'n1'으로 변경시켜준다. 이 때 만약 데이터 타입 'pt' 가 변경될 경우에는 XSE가 제공하는 인터페이스를 이용해서 데이터 값을 사용자가 직접 넣어준다. 또한 출현 횟수 use가 "required" 또는 "optional"에서 "prohibited"로 변경될 경우에는 XML 문서에서 속성이 삭제되며 "prohibited"에서 "required" 또는 "optional" 로 변경될 경우에는 속성이 생성된다.

Change attribute 알고리즘 'Change attribute' 연산의 알고리즘은 (그림 9)와 같다. 'changeAtt'에서 변화시킬 속성에 해당하는 노드를 'schemaDoc'에서 찾는다. 'changeAttSchema'는 'tA' 속성을 이름이 'n1', 데이터 타입 'pt', 'use', 'u'로 변환시킴으로써 'schemaDOM'을 변화시킨다. 이때 필요한 추가적인 파라미터는 'n1', 'pt', 'u'이다.

'changeAttDocument'는 파라미터 'tA', '$xmlDoc_i$'을 취하며 스키마 DOM의 'tA' 속성에 해당하는 DOM의 '$xmlDoc_i$'에서 찾는다. 이 때 'getInstancesFromSchemaNode'를 사용해서 속성을 찾고 'tEInstanceArray'에 저장한다. 'tAInstanceArray'의 각 노드 'tA_i'에 대하여 만약 'tA_i'의 속성의 데이터 타입 변화가 없다면 사용자 개입 없이 자동으로 'tA_i'를 새로운 이름 'n1'인 속성으로 변화시키고 'tA_i'의 속성의 데이터 타입이 변한다면 'tA_i'의 데이터 값이 삭제되고 이 요소의 데이터 값을 사용자에게 입력하도록 요구한다. 다음으로 'tA_i'를 새로운 이름 'n1'인 속성으로 변화시킨다. 이때 만약 파라미터 'u'가 'prohibit'이라면 'tA_i'를 삭제한다.

```
changeAtt(n, n1, pt, u, p, schemaDom) {
      // 경로 p로부터 변화시킬 속성 tA를 구함
   tA   getLastAttFromPath(p, schemaDom);
     // XML 스키마에서 속성을 변화시킴
   changeAttSchema(n, tA, pt, u);
     // XML 문서에서 속성을 변화시킴
   docList = getDocumentList(xmlDom1, xmlDom2,...,xmlDomn);
       for (i=1; n) {
          changeAttDocument(tA, xmlDoc i);  // XML 문서를 수정함
       }


changeAttSchema(n1, tA, pt, u) {
    속성 tA를 이름이 'n', 데이터 타입 'pt', use 'u'로 변환;
}
```

```
changeAttDocument(tA, xmlDoc i) {
      // tA의 인스턴스 속성들의 배열
  tAInstanceArray = {tA1, tA2,...tAn};
     // tA에 해당하는 문서 요소를 찾음
  tAinstanceArray = getInstancesFromSchemaNode(tA);

     for (i=1; i<q; i++) {
       if 파라미터 'u'가 required 또는 optional이라면
       then {
          if tA의 타입이 변화하면
          then 속성의 데이터 값을 사용자에게 요구함;
          tAi의 이름을 'n' 변화시킴;
       }
       else if 파라미터 'u'가 prohibit이라면
       then tA를 삭제
     }
}
```

(그림 9) 'Change attribute' 의 알고리즘

4. 관련 연구

XML이 웹의 문서 교환 표준이 되어 가면서 XML 분야에서 많은 연구가 이루어지고 있으며 특히 XML 진화(Evolution) 분야의 관심이 점점 높아지고 있다. 많은 전통적인 데이터베이스 시스템은 스키마 진화[13, 14] 뿐만 아니라 스키마의 수정에 따르는 인스턴스의 수정에도 많은 관심을 가져왔다. 대부분의 RDB (Relation Database) 또는 OODB(Object-Oriented Data base)를 위한 상업적 데이터베이스 시스템은 단순한 진화 프리미티브들을 사용하여 스키마의 재구성(re-structuring)을 지원한다. OODBS (Object Oriented Database System) [15]를 위한 연구는 더욱 복잡한 스키마 진화 연산 을 지원하는 것에 초점을 맞추고 있다. 이들은 사용자가 몇몇의 프리미티브들을 함께 사용하여 전통적인 데이터베이스 시스템의 스키마 진화보다 더 높은 수준의 스키마

진화를 할 수 있게 지원하지만 모든 변화를 지원하지는 못한다.

어떤 도구들은 XML 문서뿐만 아니라 DTD의 수정을 지원한다. 예를 들어, EXcelon은 DTD와 XML 문서의 수정이 가능한 시스템으로 XPath를 사용하여 삽입 (Insertions)과 (Deletions)를 제공한다. 이 시스템은 스키마의 수정을 지원한다는 점에서 기존의 시스템보다 기능면에서 강력하지만 이동 (Movement), 변화 (Change) 같은 수정에 필요한 연산들을 지원하지 않기 때문에 사용자가 원하는 수준의 XML 수정을 지원하지 못한다. XML Evolution Management (XEM)는 진화 프리미티브들을 제공해서 DTD를 수정할 수 있도록 지원한다. 더욱이 그 동안 무시되었던 스키마와 스키마에 순응하는 데이터의 일관성을 유지하기 위해서 DTD에 순응하는 XML 문서의 수정 역시 지원해 준다. 즉 DTD 관리모듈과 XML 문서 관리모듈을 따로 구성해주어서 진화 프리미티브가 실행될 때, DTD 모듈에 의해서 DTD의 수정이 이루어지게 하고 XML 문서 관리모듈을 통해서 변화한 DTD에 순응할 수 있도록 기존의 XML 문서를 진화 시켜줌으로써 일관성을 유지시키고 있다. 이 시스템은 DTD 및 XML 문서의 수정에 필요한 대부분의 진화 프리미티브들을 제공하기 때문에 실질적으로 XML 진화 가능하게 하며 수정한 XML 문서를 valid한 문서 상태를 유지하게 해준다. 하지만 XML 스키마의 수정을 지원하지는 않는다.

지금까지 진행된 연구는 XML 문서의 수정만을 지원하거나 XML 문서와 DTD의 수정만을 지원한다. 반면 우리가 제안하는 XSE는 XML 진화 분야에서 그동안 연구되지 않았던 XML 스키마를 수정하는 최초의 시도로써, XML 스키마와 XML 문서의 수정을 지원한다. 더욱이 사용자가 원하는 수준의 수정을 가능하게 하는 대부분의 연산들을 제공한다. 또한 XSE는 XML 스키마의 수정시 이 스키마에 순응하는 XML 문서 레벨까지 일괄 처리 모드 또는 개입 모드로 전파된다. 일괄처리 모드에서는 XML 스키마와 이에 순응하는 모든 XML 문서의 수정을 자동으로 처리하며 개입 모드에서는 XML 스키마를 수정한 후 순응하는 XML 문서에 대하여 시스템이 자동적으로 찾아주는 위치에 사용자가 XSE에서 제공하는 인터페이스를 이용해서 데이터를 입력해 주어야 한다.

5. 결 론

XML의 사용이 증대함에 따라서 XML의 자유로운 수정이 요구되고 있으며 몇몇 XML 관리 도구와 데이터베이스 시스템들이 XML 수정을 지원하고 있다. 하지만 이들은 한정된 프리미티브들을 지원하고 있으며 스키마를 무시하거나 스키마로써 DTD를 사용하고 있다. 현재 XML을 사용하는 대부분의 어플리케이션이 DTD를 사용하고 있지만 XML 문법을 사용하지 않으며 데이터 타입이 한정적이라는 단점 때문에 가까운 미래에는 XML 스키마의 사용이 일반화 될 것이다. 이러한 이유로 우리는 XML 스키마를 수정뿐만 아니라 순응하는 XML 문서의 수정까지 지원하는 연구에 집중하였다.

XSE는 XML 스키마를 수정하는 최초의 시도이다. XSE는 XML 스키마의 수정이후에도 XML 문서가 valid 상태를 유지하게 한다. 사용자 개입은 XML 스키마에서 XML 문서로 전파 시 필요에 따라 요청된다. 그리고 XML 스키마 수정 연산들이 어떻게 사용되는지를 대표적인 예를 이용해서 보여줌으로써 XML 스키마와 순응하는 모든 인스턴스 문서의 요소 및 속성의 수정을 보여주었다.

참 고 문 헌

[1] W3C. Extensible Markup Language (XML) 1.0, 2nd Edition-W3C Recommendation6-Octorber-2000. http://www.w3.org/TR/REC-xml, 2000.

[2] W3C. XML Schema - W3C Proposed Recommendation 6-October-2000. http://www.w3.org/XML/Schema, 2001.

[3] D. Lee and W. Chu, "Comparative analysis of six XML schema languages." SIGMOD Record, 29(3), pp.76-87, 2000.

[4] S. Chawathe, S. Abiteboul, and J. Widom. Representing and Querying Changes in Semistructured Data. In ICKE, pages 4-43, February 1998.

[5] Tamino: Technical description. www.softwareag.com/tamino/technical/desciption.html.

[6] Oracle Technologies Network. Oracle8i. http://www.oracle.com/database/oracle9i, 2002.

[7] Object Design. Excelon Data Integration Server. Http://www.odi.com/excelon, 1999.

[8] J. Clark and S. DeRose, "XML path language (XPATH) recommendation", http://www.w3.org/TR/1999/FEC-xpath - 19991116, 1999.

[9] Hong Su, Diane K. Kramer, and ElKe A. Rundensteiner. XML Evolution Management. In Computer Science Technical Report, 2002.

[10] W3C XSL Working Group, "XSL transformations(XSLT)." Http://www.w3.org/TR/xslt/.

[11] G.Bex, S. Maneth, and F. Neven, "A formal model for an expressive fragment of XSLT." Proc. DOOD, pp.1137-1151, 2000.

[12] N. Alon, T. Milo, F. Neven, D. Duciu, and V. Vianu, "XML with data values: typechecking revisited." Proc. ACM PODS, 2001.

[13] J. Banerjee, W. Kim, H. F. Kim, and H. F. Korth. Semantics and Implementation if Schema Evolution in Object-Oriented Database. SIGMOD, page 311-322, 1987.

[14] R. Zicari. A Framework for O2 Schema Updates. In 7th IEEE Int. Conf. on Data Engineering, pages 146-183, April 1991.

[15] P. Breche. Advanced Primitives for Changing Schemas of Object Databases. In CAISE, pages 476-495, 1996.

[16] Paul, Cotten, "Implementing XQuery." SIGMOD conference, 2002.

[17] Infozone Group. Lexus. http://www.infozoon-group.org/lexusDocs/html/wd-lexus.html, 2000.

[18] A. Deutsch, M. f. Fernandez, and D. Suciu. Storing

Semi-structured Data with STORED. In SIGMOD Conference, pages 431-442, 1999.

[19] J. Shanmugasundaram, K. Tufte, C. Zhang, G. he, D. J. DeWitt, and J. F. Naughton, Relational Database for Querying XML Documents: Limitations and Opportunities. In VLDB, pages 302-214, 1999.

[20] Oracle Technologies Network, "Oracle8i." http://www.oracle.com/database/oracle8i, 2000

[21] IBM Software. DB2 XML Extender. 2000.

[22] S. Chawathe. Describing and Manipulating XML DATA. In IEEE Data Engineering Bulletin 22(3), pages 3-9, 1999.

[23] Igor Tatarinov, Zachary G. Ives, Alon Y. Halevy, and Daniel S. Weld. Updating XML. In ACM SIGMOD, 2001.

나 영 국

e-mail : ygra@uos.ac.kr

1987년 서울대학교 전자공학과(학사)

1989년 The Penn. State Univ. 컴퓨터공학과(석사)

1996년 The Univ. of Michigan (Ann Arbor) Comp Sci.(박사)

1997년~1999년 삼성SDS 책임

1999년~2002년 국립한경대학교 컴퓨터공학과 조교수

2002년~현재 서울시립대학교 전기전자컴퓨터학부 부교수

관심분야 : 데이터베이스, 데이터베이스 응용 프로그램 저작도구