논문 2011-48CI-1-11

XML 문서에 대한 편집스크립트 생성 알고리즘

(An Algorithm Generating Edit Scripts for XML Documents)

이 석 균*

(Suk Kyoon Lee)

요 약

최근 XML문서의 변화탐지가 중요한 연구 분야로 등장하고 있지만 변화탐지의 결과, 즉 편집스크립트에 대한 연구는 아직 초보적인 수준이다. 본 연구에서는 XML 문서의 변화탐지 알고리즘인 X-treeDiff+의 편집스크립트 생성알고리즘인 X-treeESgen을 제시한다. 이는 실행시간 O(n)의 성능을 가지면서 기존의 다른 알고리즘과 달리 삽입, 삭제, 갱신의 기본연산 이외에 이동 및 복사 연산을 지원한다. 또한 편집스크립트 생성과정이 변화탐지의 대응과정과 독립적으로 설계되어 있어 다양한 튜닝이 가능한 특징이 있다.

Abstract

While detecting changes in XML documents has emerged as a major research area, the level of research on edit scripts, which are the result of the change detection, is not far from satisfactory. In this paper, we present X-treeESgen, the algorithm generating edit scripts used in X-treeDiff+, a change detection alogrithm for XML documents. X-treeESgen shows the time complexity of O(n), and support move and copy operations in addition to the basic set that is update, insert, and delete operations. Note that no other change detection algorithm provide all the operations. Also, since the process of generating edit scripts in X-treeESgen is designed independent of the process of matching, various tuning techniques may be applied.

Keywords: XML, diff, edit script, change detection

I. 서 론

diff는 텍스트 파일의 차이를 탐지하는 도구로 문서의 버전관리, 문서병합 등 많은 응용분야에서 활용되고 있다. 최근 XML 문서의 사용이 급속히 확산되고 있으나, XML 문서의 변화(차이) 탐지에 대한 연구와 이의 활용 연구들은 아직 만족스러운 수준에 이르지 못하다[1]. 특히 XML문서의 변화를 나타내는 편집스크립트에 관한 연구는 초보적인 수준이다. 본 논문에서는 XML 편집스크립트에 대한 기초 연구로 XML 문서의 변화 탐지 결과로 제시되는 편집스크립트의 생성 알고리즘 X-treeESgen에 대해 소개한다.

트리구조 문서에 대한 변화 탐지 연구는 70년 후반부터 진행되었다[2~5]. 초기 연구들이 최소 비용의 편집스크립트의 생성 등에 관심을 가졌으나 대부분의 문제가 NP-Hard로[6], 실시간 처리에는 적합하지 않았다. 최근에는 주로 실시간의 응용을 위한 알고리즘들이 연구되었다[7~15]. 이중 MH-Diff[9], X-Diff[11]는 비순서 트리 모델에 대한 변화탐지 알고리즘들이고, XML 문서에 적용 가능한 순서트리(ordered tree) 모델의 알고리즘들로는 XyDiff[7], X-treeDiff+[10], XMDiff[12], DiffXML[13], DeltaXML[14~15]이 대표적이다. 이들 중 X-Diff, XMDiff, DeltaXML은 트리구조 문서에 대한 변화탐지 알고리즘들은 편집연산으로 기본연산들 즉 삽입, 삭제, 갱신연산들만 지원한다. 이 경우 대응 및 편집연산 생

* 정회원-교신저자, 단국대학교 공과대학 컴퓨터학부
(Division of Computer Science and Engineering, Dankook University)

※ 이 연구는 2009년도 단국대학교 대학연구비로 지원으로 연구되었음.
접수일자: 2010년9월10일, 수정완료일: 2010년12월30일

성 알고리즘의 난이도가 높지 않으나 사용자의 편의성은 떨어지고 편집스크립트의 크기는 커진다.

순서트리 모델이면서 기본연산 외에 이동연산을 지원하는 알고리즘으로는 XyDiff, DiffXML과 X-treeDiff+가 있는데, XyDiff는 XML 문서들의 웨어하우스 구축을 위한 알고리즘으로 실행시간이 트리의 노드 수 n에 대해 O(n*log(n))의 성능을 보인다. DiffXML은 삽입, 삭제연산이 노드 단위로 이루어지며 e를 삽입, 삭제, 이동된 노드 수라 할 때 실행시간이 O(n*e + e2)의 비용을 갖는다. 이들 변화탐지 알고리즘의 편집스크립트의 생성은 대응 알고리즘에 종속되어 있다. X-treeDiff+는 해킹에 의한 웹 페이지의 변조 공격을 실시간 탐지하기 위한 시스템에 사용되었던 X-treeDiff[16]을 확장하여 편집스크립트 생성 기능을 갖으며 대응의 질을 높인 알고리즘이다. 이는 실행시간이 O(n)의 성능을 보인다.

본 논문에서는 X-tree Diff+의 편집스크립트 생성 알고리즘 X-treeESgen을 소개한다. X-treeDiff+는 순서트리 모델로 XML 문서에 적용가능하고 X-treeESgen을 통해 기본연산 외에 이동연산과 복사연산까지 제공하는 유일한 변화탐지 시스템이다. 기본 연산이외에 이동연산과 복사연산을 지원할 경우, 대응 과정 및 편집스크립트의 생성 과정의 알고리즘의 난이도는 상당히 높아진다. X-treeESgen은 다른 알고리즘들과는 달리 대응과정과 편집스크립트 생성과정이 상대적으로 독립되어 있어 전체적인 이해가 쉽고 응용분야의 요구에 따라 다양한 변화를 도입할 수 있다.

논문의 구성은 다음과 같다. Ⅱ장에서는 X-tree Diff+의 소개와 예제를 통해 대응과정을 설명하고 Ⅲ장에서 X-treeESgen 알고리즘을 제시하고 Ⅳ장에서는 이의 성능 분석 및 실험 결과를, 그리고 Ⅴ장에서 결론 및 미래 연구를 설명한다.

Ⅱ. X-treeDiff+의 개요

X-treeDiff+는 비교 대상의 두 XML 문서를 각각 X-tree로 변환하고 두 X-tree들 사이의 대응을 생성하고 이로부터 편집 스크립트를 추출한다. 이때 비교 대상의 두 XML 문서를 원본문서와 결과문서라 한다.

1. X-tree의 구조 및 X-treeDiff+의 대응과정

X-treeDiff+의 첫 단계는 두 XML문서를 파싱하여

```
<SECTION>
 <P ParaShape="1" Style="0">
  <TEXT CharShape="1">Pretty </TEXT>
  <TEXT CharShape="1">Korean </TEXT>
 </P>
 <P ParaShape="1" Style="0">
  <TEXT CharShape="1">XML format</TEXT>
 </P>
 <P ParaShape="1" Style="0">
  <PICTURE Id="lds1436.gif"/>
  <CHART Id="23e3r3g.gif"/>
 </P>
</SECTION>
```

그림 1. 간단한 XML 문서
Fig. 1. A simple XML document.

X-tree를 생성한다. X-tree는 X-tree 노드들로 구성되는데 X-tree 노드는 트리구조 유지를 위한 필드들과 Type, Value, AttMap, M_Node, M_Type, Op, Opord 등의 필드들로 구성된다. Type은 노드가 엘리먼트 또는 텍스트 노드인지를 구별하고, Value는 엘리먼트 명 또는 텍스트 값을 저장하며, AttMap은 엘리먼트 노드의 경우 이에 속한 속성들을 속성 명을 키로 속성 값을 값으로 저장하는 해시 맵이다. 한편 M_Node는 대응된 노드의 포인터를, M_Type은 대응유형을, Op는 편집유형을, Opord는 편집연산의 실행순서를 저장한다. 한편, X-tree에서 nid(노드 식별자) 개념을 사용하는데 이는 XML의 XPath의 개념이다.

그림 1에 XML 문서가 예시되는데 이는 한글의 XML 표현형식인 HWPML[17]을 단순화한 예제이다. 그림 1은 원본문서, 그림 2는 결과문서로 가정하며 이들에 대한 X-tree의 표현을 각각 원본트리, 결과트리로 호칭한다.

그림 1의 <SECTION>, <P>, <TEXT>, <PICTURE>, <CHART>는 각각 문서의 섹션, 문단, 문자열, 그림, 차트를 나타내는 엘리먼트들이고 ParaShape, Style, CharShape은 문단 서식, 스타일 식별자, 문자 서식을 나타내는 속성들이다. 가령 '섹션의 두 번째 문단의 문자열' 노드는 /SECTION[0]/P[1]/TEXT[0]으로 나타내며, "XML format"의 텍스트 노드를 nid로 표현하면 /SECTION[0]/P[1]/TEXT[0]/text()[0]이다. 이때 text()는 텍스트 노드를 나타내고 [0]은 텍스트 노드 중 첫 번째를 의미한다. 앞으로 nid를 표시할 때는 표기의 단순화를 위해 SECTION,

TEXT, text(), PICTURE, CHART는 각각 S, T, t, PIC, CHA로 표기한다. 따라서 위의 텍스트 노드는 /S[0]/P[1]/T[0]/t[0]로 표현된다.

X-tree 노드에는 대응과정의 효율성을 위해 nMD와 tMD 필드들을 포함한다. nMD 필드는 두 노드 간의 비교를 목적으로 노드의 데이터(엘리먼트 명과 속성 정보)에 MD4해시함수를 적용한 결과 값이 저장된다. tMD 필드는 두 서브트리 간의 비교를 목적으로 단말 노드의 경우는 nMD 필드값이 저장되며 비 단말 노드는 자신 노드의 nMD값과 모든 자식 노드들의 tMD값에 대한 MD4 해시값이 저장된다. 따라서 두 노드가 동일한지 여부는 nMD 값의 비교를, 두 서브트리가 동일한지는 tMD 값의 비교를 통해 확인할 수 있다.

두 문서에 대해 X-tree 생성 후 원본트리와 결과트리는 대응과정을 통해 모든 노드들은 대응유형을 갖게 된다. 대응유형의 리스트가 표 1에 제시된다.

BY_tMD는 동일한 tMD 값을 갖는 노드들 간의 1-1 대응을 의미하는데 tMD의 특성 때문에 이들의 자손들도 동일하게 1-1 대응한다. 이들 자손들의 대응유형은 BY_tMD_P라 하여 따로 구분한다. tMD는 다르지만 nMD가 같은 노드들 간의 대응유형은 BY_nMD라 표현하고, nMD는 다르지만 동일한 엘리먼트 노드간의 대응유형은 BY_LABEL, 그리고 텍스트 값이 서로 다른 텍스트 노드 간의 대응유형은 BY_TEXT로 표현한다. 한편 동일 tMD 값을 갖는 노드가 결과문서에 다수 존재할 때, 하나의 노드는 원본문서 노드와 BY_tMD 대응시키지만 나머지들은 BY_tMD_COPY 대응시킨다. BY_tMD_COPY_P는 BY_tMD_P와 유사하게 정의된다.

원본트리와 결과트리가 생성된 후, 5단계의 대응과정이 시작된다. 1단계에서는 tMD 값의 비교를 통해 1-1 대응되는 동일 서브트리의 쌍을 찾아 대응시키고 2단계에서는 이들 대응을 루트로 확대하고 3단계에서는 루트로부터 깊이우선탐색을 수행하면서 대응이 결정되지 않은 노드의 쌍을 대응시킨다. 4단계에서는 대응된 노드쌍에 대해 대응의 튜닝 작업을 수행한다. 5단계에서는 아직 대응되지 않은 노드들 중 tMD 값이 같은 노드들에 대해 대응을 시킨다.

2,3단계에서 대응의 확장 시 각 조건에 따라 BY_nMD, BY_LABEL, BY_tMD 등 각 대응유형이 사용된다. 모든 노드가 대응 전에는 대응유형(M_Type)이 NOT_MATCHED 값으로 초기화되어 있어 대응의 모든 단계가 끝난 후에는 두 X-tree의 모든 노드들은 표1의 대응 유형 중 하나를 갖게 된다. 대응과정은 2.2절에서 예제를 통해 설명한다.

표 1. 대응 유형의 리스트.
Table 1. List of Matching Type.

대응유형	의미
NOT_MATCHED	대응이 되지 않은 노드의 대응유형.
BY_tMD	동일한 해시 값으로 1-1 대응된 서브트리의 루트 간의 대응유형.
BY_tMD_P	BY_tMD 대응된 노드의 자손노드들의 대응유형.
BY_tMD_COPY	동일한 해시 값으로 1-n 대응된 서브트리의 루트 간의 대응유형.
BY_tMD_COPY_P	BY_tMD_COPY 대응된 노드들의 자손노드들의 대응유형.
BY_nMD	동일한 해시 값으로 대응된 노드의 대응유형.
BY_LABEL	동일한 엘리먼트 명으로 대응된 노드 간의 대응유형.
BY_TEXT	텍스트 노드 간의 대응으로 서로 동일하지 않을 경우의 대응유형.

2. 예제를 통한 대응과정 각 노드의 대응유형 설명

그림 2에서는 원본문서를 수정해서 생성된 결과문서를 제시한다. 이 두 문서를 통해 대응과정 속에서 각 노드들의 대응유형이 어떻게 결정되는지를 설명한다. 표2에서는 원본트리의 각 노드에 대해 대응된 결과트리의 노드와 대응유형을 제시하고 있다.

모든 노드들이 NOT_MATCHED로 초기화된 상태에

```
<SECTION PageBreak="true">
 <P ParaShape="1" Style="0">
  <TEXT CharShape="1">Pretty </TEXT>
  <TEXT CharShape="1">XML format</TEXT>
  <CHART Id="23e3r3g.gif"/>
 </P>
 <P Style="1">
  <TEXT CharShape="1">Pretty </TEXT>
  <TEXT CharShape="1">Korean Text</TEXT>
 </P>
 <P ParaShape="1" Style="0">
  <TEXT CharShape="1">A line added.</TEXT>
 </P>
</SECTION>
```

그림 2. XML 결과문서
Fig. 2. A simple XML target document.

서 1단계에서는 1-1 대응 관계의 동일 서브트리를 대응시키는데, 원본트리의 /S[0]/P[1]/T[0]이 결과트리의 /S[0]/P[0]/T[1]이 BY_tMD로 대응(대응7)되고, (대응10)의 (/S[0]/P[2]/CHA[0], /S[0]/P[0]/CHA[0])이 마찬가지로 처리된다. 이들의 자식노드들은 (대응6)과 같이 BY_tMD_P으로 대응된다. 원본의 /S[0]/P[0]/T[0]도 결과의 /S[0]/P[0]/T[0], /S[0]/P[1]/T[0]과 동일하나 1-1 대응관계가 아니어서 이 시점에서는 대응되지 않는다.

2단계에서는 1단계에서 대응되었던 노드의 부모 노드들(/S[0]/P[1], /S[0]/P[0])에 대해 대응시도가 이루어지는데 엘리먼트의 이름과 속성 값들이 동일하므로 BY_nMD 대응이 이루어진다(대응8). 다시 루트(/S[0], /S[0])도 역시 BY_nMD 대응이 된다(대응12).

3단계의 하향식 대응과정에서는 원본의 루트의 대응되지 않은 자식노드들 {/S[0]/P[0], /S[0]/P[2]}와 결과 루트의 대응되지 않은 자식노드들{/S[0]/P[1], /S[0]/P[2]} 간의 대응이 이루어지는데 이때 자식의 수 등의 휴리스틱이 사용된다. 본 예제의 (대응5)와 (대응11)에서는 BY_LABEL 대응이 이루어진다. 그 후 (대응5)의 하향 확장으로 (대응2)와 (대응4)가 이루어지며, (대응2)는 BY_tMD로, 그리고 이로부터 (대응1)의 대응도 BY_tMD_P로 생성된다. BY_nMD로 대응된 (대응4)로부터 하향대응이 이루어지는데 텍스트노드가 서로 다른 값이므로 BY_TEXT의 대응 (대응3)이 이루어진다. 한편, (대응11)으로부터는 더 이상 자식노드의 대응은 이루어지지 않는다.

표 2. 예제문서의 노드들의 대응유형
Table 2. Matching Type of nodes in example documents.

대응 번호	원본트리의 노드	결과트리의 노드	대응	
			단계	유형
1	/S[0]/P[0]/T[0]/t[0]	/S[0]/P[1]/T[0]/t[0]	3	BY_tMD_P
2	/S[0]/P[0]/T[0]	/S[0]/P[1]/T[0]	3	BY_tMD
3	/S[0]/P[0]/T[1]/t[0]	/S[0]/P[1]/T[1]/t[0]	3	BY_TEXT
4	/S[0]/P[0]/T[1]	/S[0]/P[1]/T[1]	3	BY_nMD
5	/S[0]/P[0]	/S[0]/P[1]	3	BY_LABEL
6	/S[0]/P[1]/T[0]/t[0]	/S[0]/P[0]/T[1]/t[0]	1	BY_tMD_P
7	/S[0]/P[1]/T[0]	/S[0]/P[0]/T[1]	1	BY_tMD
8	/S[0]/P[1]	/S[0]/P[0]	2	BY_nMD
9	/S[0]/P[2]/PIC[0]			NOT_MATCHED
10	/S[0]/P[2]/CHA[0]	/S[0]/P[0]/CHA[0]	1	BY_tMD
11	/S[0]/P[2]	/S[0]/P[2]	3	BY_LABEL
12	/S[0]	/S[0]	2	BY_nMD
13		/S[0]/P[2]/T[0]/t[0]		NOT_MATCHED
14		/S[0]/P[2]/T[0]/		NOT_MATCHED
15	/S[0]/P[0]/T[0]/t[0]	/S[0]/P[0]/T[0]/t[0]	5	By_tMD_COPY_P
16	/S[0]/P[0]/T[0]	/S[0]/P[0]/T[0]	5	By_tMD_COPY

본 예제에서 4단계의 튜닝과정은 적용되지 않는다. 5단계에서는 이때까지 대응되지 않았던 동일 tMD를 갖는 서브트리들에 대한 대응이 이루어진다. (대응16)가 이에 해당되는데 원본의 /S[0]/P[0]/T[0]이 이미 (대응2)에서 BY_tMD 대응이 되어있어 결과의 /S[0]/P[0]/T[0] 노드는 원본의 /S[0]/P[0]/T[0]에 대해 BY_tMD_COPY 대응시키며 (대응15)도 유사하게 처리된다. (대응16)의 원본 노드는 (대응2)에서 대응되어 있어 BY_tMD_COPY는 결과 노드에만 적용되는 연산유형이다. BY_tMD_COPY_P도 유사하다.

III. X-treeDiff+의 편집 스크립트 생성 기법

1. X-treeDiff+의 편집연산

X-treeDiff+에서는 엘리먼트 또는 텍스트 노드의 추가, 삭제, 이동, 복사 연산으로 INSERT, DELETE, MOVE, COPY가, 그리고 텍스트노드의 갱신을 나타내기 위해 UPDATE_TEXT가 지원된다. 속성의 삽입, 삭제, 갱신 연산으로 INSERT_ATTR, DELETE_ATTR, UPDATE_ATTR이 제공된다. 본 절에서는 앞의 원본문서와 결과문서에 대해 X-treeDiff+가 생성한 편집스크립트의 예제를 통해 각 연산의 의미를 설명한다.

속성 관련 연산들과 텍스트 갱신 연산부터 설명한다. 편집연산은 XML의 엘리먼트로 표현되며 snid는 원본문서의 편집 대상 노드의 nid, 즉 XPath를 의미한다. INSERT_ATTR은 원본문서의 첫 번째 SECTION 노드에 PageBreak라는 속성에 true라는 값을 추가하는 연산이고, UPDATE_TEXT 엘리먼트는 첫 번째 문단의 문자열을 나타내는 텍스트 노드의 값을 기존의 "Korean "에서 "Korean Text"로 갱신하는 연산이다.

<INSERT_ATTR snid="/S[0]" attrName="PageBreak" attrValue="true"/>
<DELETE_ATTR snid="/S[0]/P[0]" attrName="ParaShape" />
<UPDATE_ATTR snid="/S[0]/P[0]" attrName="Style" attrValue="1"/>
<UPDATE_TEXT snid="/S[0]/P[0]/T[1]/t[0]" tv="Korean Text"/>

위의 연산들은 트리의 구조는 그대로 두며 단지 값만 변화시킨다. 이러한 연산들을 '값-변환 연산'이라한다. 한편, 삭제, 이동, 복사, 삽입 연산들은 트리의 구조를 변화시킨다. 아래의 DELETE는 원본의 세 번째 문단의 PIC[0]을 삭제한다. 이동, 삽입, 복사연산의 tnid는 이동, 삽입, 복사될 장소의 노드, tpos는 tnid가 나타내는 노드의 몇 번째 자식노드로 이동, 삽입, 복사될 지를 표현한다. INSERT는 세 번째 문단에 첫 번째 자식 노드로 "A line added"를 포함한 <TEXT> 노드를 추가한다. 한편, 이동연산은 MOVE와 LOCAL_MOVE로 구분되는데 MOVE는 이동 시 동일하지 않은 부모 간에 이동이고 LOCAL_MOVE는 동일한 부모 내에서의 이동, 즉 자식노드들 간의 위치 변화를 의미한다. 사용자 한테는 차이가 없지만 생성 알고리즘에서는 중요한 차이가 있어 일단 구분한다.

```
<MOVE snid="/S[0]/P[2]/CHA[0]" tnid="/S[0]/P[0]" tpos="2" opord="3"/>
<DELETE snid="/S[0]/P[2]/PIC[0]" />
<LOCAL_MOVE snid="/S[0]/P[1]" tnid="/S[0]" tpos="0" opord="1"/>
<COPY snid="/S[0]/P[0]/T[0]" tnid="/S[0]/P[0]" tpos="0" opord="2"/>
<INSERT tnid="/S[0]/P[2]" tpos="0" opord="4">
  <TEXT CharShape="1">A line added.</TEXT>
</INSERT>
```

X-treeDiff+에서의 이동연산의 의미는 일반적인 해석과는 달리 '잘라내기'와 '붙여넣기'의 복합연산의 의미로 정의된다. '잘라내기'와 '붙여넣기'가 연속적인 작업으로 이루어지면 일반적인 이동의 의미인데, 위의 LOCAL_MOVE가 이 경우로 두 번째 문단을 SECTION의 첫 번째 문단 위치로 이동하고 있다. 한편, 위의 MOVE 연산은 두 작업이 연속적으로 이루어지지 않는 예인데 이를 연속적으로 실행할 경우 엉뚱한 곳에 이동시키게 된다. 위의 MOVE 이하의 연산들은 트리를 축소시키다가 일정시점 이후 확장시키는 구조로 구성된다. 이때 축소 국면의 연산들, 이들을 '트리-축소 연산'이라고 하는데 이들은 DELETE, MOVE의 '잘라내기', LOCAL_MOVE의 '잘라내기' 연산들로 구성되며 편집스크립트의 연산 순서대로 실행된다.

확장 국면의 연산들은 '트리-확장 연산'이라 하며, 이들은 INSERT, MOVE의 '붙여넣기', LOCAL_MOVE의 '붙여넣기', COPY 연산들로 구성되며 편집스크립트 내의 순서와 관계없이 편집연산의 opord 순서대로 실행한다. 따라서 위의 편집스크립트의 실행 순서는 네 개의 '값 변환연산들', MOVE의 '잘라내기', DELETE, LOCAL_MOVE의 '잘라내기', 그리고 확장 국면의 연산들로 LOCAL_MOVE의 '붙여넣기', COPY, MOVE의 '붙여넣기', 그리고 INSERT로 구성된다. 원본문서에 대해 이 순서로 실행되면 결과문서가 생성된다.

X-treeDiff+에서의 편집스크립트는 트리 단위의 삭제/삽입 모드와 노드 단위의 삭제/삽입 모드에서 생성가능하다. 위에 예는 트리 단위의 삭제/삽입에서 생성된 편집스크립트로 사용에는 편리하나 편집연산 생성 시의 알고리즘이 좀 복잡하다. 본 논문에서는 두 모드가 핵심 알고리즘이 동일하기 때문에 노드 단위의 삭제/삽입 모드를 전제로 편집 스크립트 생성 알고리즘을 설명한다. 위의 편집스크립트를 노드 단위의 삭제/삽입 모드로 다시 표현하면 다음과 같다. INSERT는 다음과 같이 두 연산으로 표현되며 DELETE는 삭제대상이 하나의 노드여서 아무런 변화가 없다.

```
<INSERT tnid="/S[0]/P[2]" tpos="0" opord="4">
  <TEXT CharShape="1"></TEXT>
</INSERT>
<INSERT tnid="/S[0]/P[2]/T[0]" tpos="0" opord="5">
  A line added.
</INSERT>
```

2. X-tree 노드의 편집유형 결정 알고리즘

X-treeESgen은 모든 노드의 편집유형 결정 부분과 편집스크립트 추출 부분으로 구성된다. 본 절에서는 표2에 제시된 예제의 노드의 대응유형들로부터 각 노드의 편집유형의 결정과정을 설명한다. 표 3의 노드의 설정 가능한 편집유형 리스트를 소개하면, Op 필드는 NOT_ASSIGNED로 초기화된 후 대응유형 정보를 기반으로 노드에 적용할 편집유형이 설정된다. NOP는 원본과 결과트리 간에 변화가 없는 노드들의 편집유형이며, OP_UPD_A_MOV는 복합 편집유형으로 대상 노드에 속성 관련 연산과 이동연산이 적용됨을 의미한다. 편집유형 결정은 네 단계에 걸쳐 이루어진다.

1단계에서는 원본트리의 노드들에 대해 표3의 OP_UPD_T, OP_UPD_A 편집유형의 설정 여부를 결정하는데, 그림 3에 알고리즘이 제시된다. 원본트리의 노

표 3. 편집유형 리스트.
Table 3. List of edit types.

편집유형	의미
NOT_ASSIGNED	정해지지 않았음.
OP_L_MOV	노드(서브트리)가 지역이동됨.
OP_MOV	노드(서브트리)가 이동됨.
NOP	편집연산(즉 변화)이 필요없음.
OP_COPY	노드(서브트리)가 복사됨.
OP_DEL	노드(서브트리)가 삽입됨.
OP_INS	노드(서브트리)가 삭제됨.
OP_UPD_T	텍스트가 갱신됨.
OP_UPD_A	노드의 속성들이 삽입/삭제/갱신됨.
OP_UPD_A__MOV	OP_UPD_A와 OP_MOV의 복합 편집유형
OP_UPD_A__L_MOV	OP_UPD_A와 OP_L_MOV의 복합 편집유형
OP_UPD_T__L_MOV	텍스트값이 갱신된 된 노드가 이동될 경우의 편집유형

드들을 DFS(Depth-First-Search) 순으로 방문하면서, 현 노드가 텍스트 노드이면서 대응유형이 BY_TEXT일 때 노드의 연산유형을 OP_UPD_T로, 엘리먼트 노드이면서 대응유형이 BY_LABEL일 때는 OP_UPD_A로 결정한다. ListOfOldXtreeNodesByDFS는 원본트리의 노드들을 DFS 순으로 저장한 리스트이고 노드 n에 대해 m은 n의 대응노드로 가정한다. 1단계를 논문의 예제에 적용하면, 표2의 (대응3)에 해당하는 노드 쌍은 OP_UPD_T로, (대응5)와 (대응11)의 노드들의 Op는 OP_UPD_A로 결정된다.

2단계에서는 원본트리의 각 노드에 대해 OP_DEL, OP_MOV, OP_UPD_A__MOV 편집유형의 적용 여부가 결정된다. 그림 4의 알고리즘과 같이 원본트리를 DFS 역순으로 방문하면서 현 노드 n이 대응되지 않은 경우 OP_DEL로 설정하며, 만일 n이 대응(대응 노드 m)은 되었지만 n과 m의 부모가 서로 대응되지 않은 경우(즉, n.Parent.M_node <> m.Parent)에는 OP_MOV를 설정한다. 이때 노드의 방문순서가 중요한데, 가령 이동 대상의 서브트리가 일부 자손노드들이 삭제된 후 이동될 경우, 이의 처리를 위해서는 반드시 DFS 역순으로 진

```
Foreach n in ListOfOldXtreeNodesByDFS
 if n.Type = Text AND n.M_Type = BY_TEXT then
   n.Op = m.Op = OP_UPD_T;
 if n.Type = Element AND n.M_Type = BY_LABEL then
   n.Op = m.Op  = OP_UPD_A;
End Foreach
```

그림 3. 편집유형 결정 1단계.
Fig. 3. Step 1 of edit type determinationt.

```
Foreach n in ListOfOldXtreeNodesByReverseDFS
  if n.M_Type = NOT_MATCHED then
    n.Op = OP_DEL; continue;
  if n.M_Type in [By_tMD, By_nMD, By_LABLEL] then
    if n과 m의 부모가 존재하나 서로 대응되지 않을 때 then
      if n.Op = NOT_ASSIGNED then
        n.Op = m.Op = OP_MOV;
      else /* n.Op가 OP_UPD_A인 경우 */
        n.Op = m.Op = OP_UPD_A__MOV;
    SetNOPtoDescendentsWithNA(n);
End Foreach
```

그림 4. 편집유형 결정 2단계.
Fig. 4. Step 2 of edit type determinationt.

행해야 한다. ListOfOldXtreeNodesByReverseDFS는 원본트리의 노드들을 DFS 역순으로 저장한 리스트를 의미한다. SetNOPtoDescendentsWithNA(n)은 n의 자손노드들과 m의 자손노드들 중 Op가 NOT_ASSIGNED인 경우에만 이를 NOP로 설정하고 재귀호출을 하는 함수이다. (대응9)의 노드는 OP_DEL으로, 그리고 (대응10)의 노드 쌍은 OP_MOV로 설정되는 예이다.

3단계에서는 동일 부모의 자식들 간의 이동을 의미하는 OP_L_MOV의 편집유형을 결정한다. 이의 알고리즘이 그림 5에 제시되는데, OP_MOV와의 차이점은 한 자식노드의 이동이 다른 자식들의 순서에 영향을 끼치므로 동일 부모의 OP_L_MOV 후보 노드들 중 어느 자식 노드의 편집유형을 OP_L_MOV로 결정하는가가 알고리즘의 핵심이다.

GetCandidateListForL_Move는 매개변수로 주어지는 노드의 자식노드 리스트를 입력받아 자식노드들 중 다음 조건을 만족하는 노드들의 리스트를 반환한다. 자식노드들 중 임의의 노드 r의 대응노드가 존재하고 그 대응노드의 부모노드와 노드 r의 부모노드가 서로 대응될 때 r을 반환 노드 리스트에 포함한다.

ListForL_MOV는 반환 리스트를 저장하는 변수이고, SelectNodesForL_MOV는 반환된 노드 리스트에 대해 지역 이동 대상의 노드들을 선별하는 함수로 선별결과를 반환해서 ResultList에 저장한다. 이때 Longest Common Subsequence Algorithm[18] 등 다양한 알고리즘들이 사용될 수 있으나, X-treeDiff+에서는 자식 노드 리스트의 길이가 p일 때 O(p)의 실행비용인 휴리스틱 알고리즘[16]이 사용된다. (대응8)의 노드 쌍은 OP_L_MOV로 설정되는 예이다.

그림 6에서와 같이 4단계에서는 결과트리의 노드들

```
Foreach n in ListOfNewXtreeNodesByReverseDFS
  ListForL_MOV = GetCandidateListForL_Move(n.Children)
  ResultList = SelectNodesForL_MOV(ListForL_MOV)
  Foreach p in ResultList // q는 p의 대응노드로 가정
    if p.Op = NOT_ASSIGNED then
      p.Op = q.Op = OP_L_MOV;
    elsif p.Op = OP_UPD_A then
      p.Op = q.Op = OP_UPD_A_L_MOV;
    else /* p.Op가 OP_UPD_T의 경우 */
      p.Op = q.Op = OP_UPD_T_L_MOV;.
    SetNOPtoDecendentsWithNA(p);
End Foreach
```

그림 5. 편집유형 결정 3단계.
Fig. 5. Step 3 of edit type determinationt.

```
InsOpOrder = 0;
Foreach n in ListOfNewXtreeNodesByDFS
  if n.M_Type = NOT_MATCHED then
    n.Op = OP_INS;  n.Opord = ++InsOpOrder;
  if n.M_Type = BY_tMD_COPY then
    n.Op = OP_COPY; n.Opord = ++InsOpOrder;
  if n.M_Type = BY_tMD_COPY_P then
    n.Op = NOP;
  if n.Op in [OP_MOV, OP_L_MOV, OP_UPD_A_MOV,
        OP_UPD_A_L_MOV, OP_UPD_T_L_MOV] then
    n.Opord = ++InsOpOrder;
End Foreach
```

그림 6. 편집유형 결정 4단계.
Fig. 6. Step 4 of edit type determinationt.

을 DFS순으로 방문하면서 편집유형을 설정한다. 노드 n이 대응되지 않을 경우, 원본에는 대응노드가 없으므로 OP_INS로, n의 대응유형이 BY_tMD_COPY의 경우는 OP_COPY로, n의 대응유형이 BY_tMD_COPY_P인 경우는 n의 조상노드가 OP_COPY로 설정된 경우이므로 NOP으로 설정한다.

2.2절에서 설명했던 것처럼 '값-변환 연산'들과 '트리-축소 연산'들은 편집스크립트 내의 순서대로 진행된다. 그러나 '트리-확장 연산'들의 순서는 Opord 속성 값의 오름차순 순서로 진행된다. 따라서 이들에 대해서는 연산 순서를 결정해야 하는데 이를 위해 InsOpOrder 변수가 사용되어 노드 n의 Opord 필드에 순서를 지정한다. 앞의 예제에 대해 그림 6의 알고리즘을 적용하면 /S[0]/P[0]/T[0] 노드가 BY_tMD_COPY로 대응(대응16)되어 있어 Op필드에 OP_COPY으로, 그 자식노드(대응15)는 NOP로, 그리고 (대응13,14)의 결과 노드들은 각각 OP_INS로 설정한다.

3. 편집스크립트 추출 알고리즘

편집스크립트 추출 알고리즘은 그림 7에 제시된다. 이는 갱신관련 연산부터 시작하는데, 원본트리를 DFS순으로 방문하면서 편집연산들을 생성한다. 이때 processOP_UPD_A(n)는 노드 n의 속성 해시맵(n.AttMap)과 대응노드 m의 속성 해시맵(m.AttMap)에 대해 n.AttMap에 없는 속성이 m.AttMap에 존재하면 INSERT_ATTR을, 그리고 n.AttMap에 있는 속성이 m.AttMap에 없으면 DELETE_ATTR를 생성한다. 만일 속성이 양쪽에 다 있으나 속성 값이 서로 다른 경우에는 UPDATE_ATTR를 생성한다. 갱신관련 연산 추출 후 원본트리를 DFS 역순으로 방문하면서 DELETE, MOVE, LOCAL_MOVE를 생성한다. 따라서 이때 생성된 연산들의 순서는 원본 트리의 DFS 역순과 동일하나, 이들 연산의 '트리-확장 연산' 부분은 노드의 Opord 값을 기준으로 실행순서가 결정된다. 끝으로 삽입관련 연산들이, 즉 INSERT와 COPY이 생성된다.

본 논문에서는 3.1절의 예제와 같은 XML 형식의 편집연산의 세부 생성 방법을 설명하지 않는다. 대신 각 편집연산에 사용되는 속성이 무엇이며 어떤 값이 사용되는가를 설명한다. 갱신관련 연산들과 DELETE, MOVE,

```
Foreach n in ListOfOldXtreeNodesByDFS
  if n.Op in [OP_UPD_T, OP_UPD_T_L_MOV] then
    UPDATE_TEXT 생성;
  if n.Op in [OP_UPD_A, OP_UPD_A_MOV,
                OP_UPD_A_L_MOV] then
    processOP_UPD_A(n);
End Foreach // 갱신관련 연산생성
Foreach n in ListOfOldXtreeNodesByReverseDFS
  if n.Op = OP_DEL then
    DELETE 생성;
  if n.Op in [OP_MOV, OP_UPD_A_MOV] then
    MOVE 생성;
  if n.Op in [OP_L_MOVE, OP_UPD_A_L_MOV,
                OP_UPD_T_L_MOV] then
    LOCAL_MOVE 생성;
End Foreach // 삭제관련 연산생성
Foreach n in ListOfNewXtreeNodesByDFS
  if n.Op = OP_COPY then
    COPY 생성;
  if n.Op = OP_INS then
    INSERT 생성;
End Foreach // 삽입관련 연산생성
```

그림 7. 편집스크립트 추출 알고리즘.
Fig. 7. Extraction algorithm of edit script.

LOCAL_MOVE, COPY에는 snid 속성이 사용되는데 이는 원본트리의 노드 n의 경로가 사용된다. INSERT의 tnid에는 결과트리의 n의 부모 노드 경로가 사용되고 tpos는 n이 그 부모노드에서 몇 번째 자식노드인지, 즉 자식 순서를 나타낸다. MOVE, LOCAL_MOVE, COPY의 tnid는 결과 트리의 노드 m(n의 대응노드)의 부모 경로, tpos는 m의 자식 순서를 나타낸다.

Ⅳ. 편집스크립트 생성 알고리즘의 분석

앞 절의 편집스크립트 생성 알고리즘에서는 두 X-tree들의 정적인 대응관계, 즉 노드들의 대응유형들로부터 동적으로 적용될 편집연산들을 추출하는 방법을 설명했다. 본 절에서는 이의 실행시간에 대한 성능분석과 알고리즘의 적합성(soundness)에 대해 논의한다. 참고로 n이 두 문서의 총 노드 수의 합이라 할 때 X-treeDiff+에서 X-tree의 생성과 대응과정까지의 실행시간은 O(n)의 성능을 보인다[10].

편집스크립트 생성 알고리즘은 3.2절과 3.3절의 두 서브 알고리즘으로 구성되는데, 그림7의 편집스크립트 추출 알고리즘에는 편집연산의 세부 생성 과정이 소개되어있지 않으나 이는 해당 노드에 속한 정보로부터 문자스트링을 만드는 작업이므로 비용이 O(1)이고 따라서 그림7의 알고리즘의 실행시간은 O(n)의 성능을 갖는다.

한편 X-tree 노드의 편집유형 결정 알고리즘의 1단계와 4단계는 트리 방문 비용이 지배적으로 각각 O(n)이다. 한편, 2단계와 3단계의 경우 SetNOPtoDescendentsWithNA(n)는 자손노드들 중 Op필드가 NOT_ASSIGNED경우에만 재귀 호출되는 함수이고 이 단계에서는 트리를 DFS 역순으로 방문하면서 처리하기 때문에 이 함수로 인해 자손노드들의 Op 필드가 두 번 이상 설정되는 경우는 없다. 이로 인한 비용은 O(n)이 된다. 3단계 알고리즘에서도 트리를 DFS역순으로 방문하면서 GetCandidateListForL_Move함수와 SelectNodesForL_MOV함수가 호출하는데 트리 노드들이 이 함수 호출에 한번 이상 참여하지 않으며 SelectNodesForL_MOV의 호출 비용이 O(p)이나 p는 트리 노드 수에 비교하면 상수로 간주될 수 있어, 전체적으로 실행 비용이 O(n)의 성능을 갖게 된다. 따라서 편집스크립트 생성 알고리즘의 전체의 실행시간은 O(n)의 성능을 갖게 된다.

다음에서는 원본문서에 편집스크립트 적용 시 결과문서와 동일한 문서가 생성되는지, 즉 X-treeESgen의 적합성에 대해 논의한다. 이론적인 증명 대신 증명에 필요한 아이디어를 제시한다.

X-treeESgen의 편집스크립트의 연산들은 '값-변화 연산들', '트리-축소 연산들', 그리고 '트리-확장 연산들'의 세 유형으로 구성되는데, 간단히 이들을 각각 갱신작업, 삭제작업, 삽입작업으로 표현한다. 편집스크립트에 속한 모든 연산들은 snid나 tnid 속성을 포함하는데 snid는 원본문서의 노드를, tnid는 결과문서의 노드를 의미한다. 원본문서에 편집스크립트를 적용함이 변환과정을 통해 결과문서를 생성한다고 할 때, snid는 갱신작업의 대상 노드이거나 삭제작업의 대상노드를, 그리고 tnid는 삽입작업의 대상노드를 의미한다. 그런데 갱신작업은 원본의 구조 변화 없이 대상노드의 값만을 변화시키고 있어 갱신작업의 실행 후에도 다른 후속 편집연산에는 아무런 영향을 끼치지 않는다.

삭제작업은 원본문서에 대해 일련의 삭제를 진행하는데 진행 순서가 DFS의 역순으로 진행되므로 먼저 실행된 삭제의 연산 결과가 나중에 실행될 삭제의 연산의 snid에 영향을 주지 않는다. 따라서 DFS 역순으로 진행되는 일련의 삭제는 아무런 문제없이 실행 가능하며 모든 삭제작업이 완료된 후의 문서의 노드들 중에 결과문서의 노드와 대응되지 않는 노드는 더 이상 존재하지 않는다. 이는 대응과정에서 삭제작업으로 지정된 내용은 결과문서에 존재하지 않는 부분들이기 때문이다. 한편 갱신 및 삭제작업이 완료된 후의 문서를 문서 D라 하고 이에 대한 삽입작업을 고려하자. D와 결과문서 사이의 대응(차이)의 내용은 D에 적용할 삽입작업의 내용이다. 삽입작업의 내용이 각 연산을 통해 제대로 생성

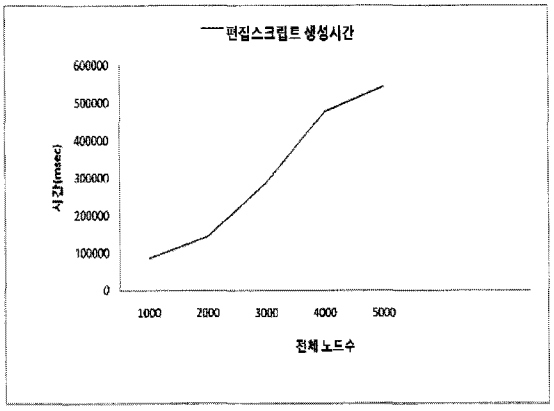

그림 8. 노드수에 따른 편집스크립트 생성시간
Fig. 8. Generation time of edit script based on the number of nodes.

되는지가 핵심인데 삽입작업의 진행이 D에 대해 결과 문서의 DFS 순으로 이루어지므로 문서 즉 트리의 위쪽 부분부터 차이 내용을 순차적으로 채워나가게 된다. 즉 삽입작업의 임의의 연산의 실행 시점에 그 연산의 tnid에 해당하는 노드가 D에 존재하게 된다.

다음에서는 X-treeESgen의 실험결과를 소개한다. 편집스크립트를 원본문서에 적용하는 프로그램 ApplyES를 구현 한 후 다양한 크기의 혼글 문서를 XML형식으로 저장한 파일들의 20개의 쌍에 대해 다음의 실험을 수행하였다. 임의의 원본문서 source, 결과문서 target에 대해 X-treeDiff+의 실행결과 반환되는 편집스크립트 es라 할 때, ApplyES를 통해 source에 es를 적용한 결과문서 newtarget을 생성하고 다시 target과 newtarget에 대해 X-treeDiff+를 적용하여 반환되는 편집스크립트를 분석한다. 이때 최종 편집스크립트가 공집합이면 X-treeESgen이 적합한 편집스크립트를 생성했다고 판정한다. 위의 실험 결과는 모두 공집합의 편집스크립트를 반환한다. 이 이외에도 테스트 목적의 XML 문서들, MS Word 문서들 등 다양한 테스트가 진행 중인데 현재까지 만족스러운 결과를 반환하고 있다. 위의 혼글 문서들은 전체 노드 수에서 다소 간의 차이는 있지만 전체를 5개의 그룹(1000, 2000, ..., 5000)으로 구분해서 각 그룹 별 평균 시간을 계산해보면 노드 수에 실행시간이 선형 형태를 보이는 것을 알 수 있다.

V. 결론 및 미래연구

XML기반의 문서의 버전관리, 문서 병합 등의 분야에서 편집스크립트의 사용은 필수적이나 이에 대한 연구는 아직 초보적인 상태이다. 최근 많은 학술논문에서 XyDiff, DiffXML 등 XML 문서의 변화 탐지 알고리즘들이 제안되었지만 기본연산이외에 이동 및 복사연산을 지원하는 편집스크립트 생성 알고리즘으로는 제안된 X-treeESgen이 유일하다. X-treeESgen은 X-treeDiff+에서 사용하고 있는 편집스크립트 생성 알고리즘 X-treeESgen으로 문서의 노드 수를 n이라 할 때 실행시간 O(n)의 성능을 가지고 있다.

X-treeESgen은 트리 매칭을 통해 생성된 두 문서 간의 정적인 대응 관계로부터 원본 문서에 동적으로 적용할 수 있는 편집스크립트 즉 일련의 편집 연산들을 생성하는데, 편집연산의 생성 과정을 문서의 대응 과정으로부터 독립적으로 구성하여 이해가 쉽고 다양한 변화를 도입할 수 있으며, 또한 노드의 대응정보만 제공되면 다른 대응 알고리즘에 대해서도 쉽게 수정하여 적용할 수 있는 특징이 있다.

현재의 X-treeESgen에서의 이동연산은 사실은 '잘라내기'와 '붙여넣기'로 일반 사용자들의 사용에는 불편할 수 있어 이를 개선한 알고리즘 개발 중에 있다. 현재는 편집스크립트의 병합 기법, X-treeDiff+를 사용하여 문서의 버전관리, 프로그램 복제 탐지[19] 등에 대한 연구가 진행 중이며 추후 편집스크립트 기반의 버전 관리 시스템의 프로토타입을 구축할 예정이다.

참 고 문 헌

[1] S. Ronnau, J. Scheffczyk, and U. Borghoff, "Towards XML Version Control of Office Documents," In Proc. of ACE Symposium on Document Engineering, pp. 10-19, Nov. 2005.

[2] R. Wagner and M. Fischer, "The string-to-string correction problem," *Journal of the ACM*, 21, pp.168-173, 1974.

[3] K. Tai, "The tree-to-tree correction problem," Journal of the ACM, 26(3), pp.422-433, July 1979.

[4] S. Selkow, "The tree-to-tree editing problem," Information Processing Letters, 6, 1977.

[5] E. W. Myers, "An O(ND) Difference Algorithm and Its Variations," Algorithmica, 1(2), pp.251-266, 1986.

[6] S. Chawathe and H. Molina, "Meaningful Change Detection in Structured Data," In SIGMOD '97, pp.26-37, 1997.

[7] G. Cobéna, S. Abiteboul and A. Marian, "Detecting Changes in XML Documents," The 18th ICDE, 2002.

[8] K. Zhang and D. Shasha, "Simple fast algorithms for the editing distance between trees and related problems," SIAM Journal of Computing, 18(6), pp.1245-1262, 1989.

[9] S. Chawathe and H. G. Molina, "Meaningful Change Detection in Structured Data," In SIGMOD '97, pp.26-37, 1997.

[10] S. Lee and D. Kim, "X-treeDiff+: Efficient Change Detection Algorithm in XML Documents," LNCS 4096, pp.1037-1046, 2006.

[11] Y. Wang, D. DeWitt, J. Cai, "X-Diff: An Effective Change Detection Algorithm for XML Documents," in Proc. of ICDE, pp519-530, Mar., 2003.

[12] S. Chawathe, "Comparing Hierarchical Data in External Memory," Proc. of VLDB, Sept. 1999.
[13] diffxml, http://diffxml.sourceforge.net/
[14] R. Fontaine, "Change Control for XML:Do it right," In Proc. of XML Europe 2003.
[15] DeltaXML, http://www.deltaxml.com
[16] 김동아, "XML 문서에 대한 변화 탐지 및 관리," 단국대학교 전산통계학과 박사학위논문, pp.1-111, 2005.
[17] 한글과 컴퓨터, http://www.hancom.co.kr/
[18] S. Chawathe, A. Garcia-Molina, and J Widom, "Change Detection in hierarchically structured information," In Proc. of SIGMOD, pp493-504, 1996.
[19] 이석균, "X-treeDiff+기반의 프로그램 복제 탐지," 전자공학회논문지, 제47권 CI편, 4호, 2010년 7월.

저 자 소 개

이 석 균(정회원)
대한전자공학회논문지,
제47권 CI편 제 4호 참조