DOI: 10.3745/KIPSTD.2009.16-D.4.497

삽입/삭제 편집연산 기반의 XML 문서 병합

이 석 균†

요 약

오피스 및 과학 분야의 문서 작업 등에서 XML의 사용이 보편화되고 협업이 요구됨에 따라 효과적인 XML 문서 병합 방법이 필요하다. 이에 대한 해결 방안으로 본 논문에서는 동일 원본 문서에 대한 다수 사용자의 편집 작업들의 병합을 위한 이론적 틀을 제시한다. 문서들의 병합 시 문서 자체를 병합하는 기존의 방법들과는 달리, 사용자의 편집 작업을 원본 문서에 적용되는 일련의 편집 연산들, 즉 편집 스크립트로 표현하고 다수 사용자의 편집 스크립트들을 병합하고 원본 문서에 적용하여 문서의 병합 효과를 이루고자 한다. 이를 위해 삽입과 삭제연산으로 구성된 편집스크립트를 전제로 정적 편집 스크립트, 편집 스크립트의 간섭 및 충돌 등의 개념들을 정의하고 편집 스크립트들의 충돌 조건과 병합 시 편집 스크립트 조정기법을 제안한다. 이 방법은 분산 환경에서 네트워크 부하를 줄이며 각 편집 작업의 의미가 보존되어 버전관리에 효과적이다.

키워드 : 협업적 편집, 문서 병합, XML, X-treeDiff+, Diff, 패치, 버전관리

Merging XML Documents Based on Insertion/Deletion Edit Operations

Suk Kyoon Lee†

ABSTRACT

The method of effectively merging XML documents becomes necessary, as the use of XML is popular and the collaborative editing is required in the areas such as office documents and scientific documents editing works. As a solution to this problem, in this paper we present a theoretical framework for merging individual editing works by muli-users to a same source document. Different from existing approaches which merge documents themselves when they are merged, we represent editing works with a series of edit operations applied to a source document, which is called a edit script, merge those edit scripts by multi-users, and apply the merged one to the source document so that we can achieve the same effect of merging documents. In order to do this, assuming edit scripts based on insertion and deletion edit operations, we define notions such as static edit scripts, the intervention between edit scripts and the conflict between the ones, then propose the conflict conditions between edit scripts and the method of adjusting edit scripts when merged. This approach is effective in reducing network overhead in distributed environments and also in version management systems because of preserving the semantics of individual editing works.

Keywords : Collaborative Editing, Merging Documents, XML, X-treeDiff+, Diff, Patch, Version Management

1. 서 론

오피스 업무들은 종종 다수의 협력 작업을 통해 이루어진다. 문서 공동 작업, 프로젝트 구현 등의 협업적 작업(collaborative work)의 효과적인 처리를 위해서는 동일 문서의 복사본들에 대해 병렬적으로 수행되는 편집 작업들을 하나의 문서로 병합하는 기술이 중요하다. OpenOffice, Microsoft Office, 훈글 등의 오피스 소프트웨어에서는 파일의 표현 형식으로 XML 파일을 지원하는 등 XML의 사용은 과학 분야를 포함한 대부분의 분야에서 증가 추세에 있어서[1, 2, 19], 본 논문에서는 XML 기반의 협업적 문서 편집을 위한 문서의 병합 기법에 대해 소개한다.

책의 공동 저술, 팀 차원의 보고서 작성, 프로그래밍 작업은 협업이 필요한 분야로 병렬적으로 편집된 내용의 병합에 있어서 일관성, 편의성 및 효율성은 업무에 중대한 영향을 끼친다[14, 20, 21]. 병합은 편집된 문서들 사이의 동일한 부분들을 대응시키고 대응되지 않은 부분, 문서 간의 차이를 합침으로 이루어지는데 이 과정에 DTD, 또는 도메인의 특성을 반영한 규칙(rule)들의 집합이 사용되곤 한다. 이때 문서간의 차이는 원본 문서에 대해 각각 갱신된 부분이다. 문

※ 이 연구는 2007학년도 단국대학교 연구년 연구비 지원으로 연구되었음.
† 종신회원 : 단국대학교 컴퓨터학부 교수
논문접수 : 2009년 2월 25일
수 정 일 : 1차 2009년 4월 20일, 2차 2009년 5월 22일
심사완료 : 2009년 5월 22일

서 간의 대응(matching) 기술은 문서 간의 차이의 발견 및 문서의 병합에 기초가 되는 중요한 연구 분야다[3-10]. Balasubramaniam은 네트워크 연결이 불완전한 분산 환경을 전제로 파일 동기화도구(file synchronizer)라는 개념을 통해 분산 복제된 파일 시스템간의 대응, 차이(갱신)의 발견 및 병합 과정 전체에 대한 주요 이슈들 및 프레임워크를 제시했다[13].

텍스트 문서 간의 차이와 병합을 위해 흔히 diff와 patch가 사용되는데, diff3는 하나의 원본 텍스트 파일과 이로부터 수정된 두 개의 텍스트 파일들 사이의 비교와 병합을 수행한다[12, 18]. diff3와 변종들은 CVS[16]와 Subversion[17] 등의 버전 관리 시스템에서 널리 사용된다[11].

트리구조의 문서의 병합에 대한 연구들은 2000년대 초기부터 이루어졌다[14, 15, 22]. Manger는 자동차 산업의 제어장치 소프트웨어의 기능 문서화를 위한 SGML/XML 문서의 병합 알고리즘 σ을 제시했다[22]. σ는 동일 DTD의 문서들을 위한 병합 알고리즘으로 DTD의 구조에 따라 문서들을 병합한다. Fontaine은 두 문서의 병합을 수행하는 기존의 2-way 병합 알고리즘과 달리 원본 문서가 존재할 경우 이로부터 독립적으로 편집된 두 파일에 대해 병합을 수행하는 3-way 병합 알고리즘을 최초로 제시했는데 이때 DeltaXML을 통해 문서들 간의 차이를 발견하고 이를 기반으로 병합을 수행했다[15]. 한편 트리구조 문서의 병합은 트리구조의 복잡성으로 인해 라인 단위의 텍스트 문서의 병합과 같은 일반화된 병합기법의 제안이 쉽지 않다. Lindholm은 다양한 병합의 상황들을 나타내는 use cases를 기반으로 병합 규칙을 제시하고 이를 기초로 3-way 병합 기법과 충돌 조건을 제안했다[14]. 이들 병합 기법의 공통점은 XML 문서들을 직접 비교하고 차이를 발견하고 병합하는 상태(state) 기반의 알고리즘이다.

본 논문에서는 기존의 상태 기반의 병합 기법들과는 달리 편집 스크립트 기반의 새로운 병합 기법을 제안한다. 원본 문서에 대한 독립적인 편집 작업들을, 원본 문서와 편집된 문서 간의 차이가 아니라, 각각 일련의 편집연산들, 즉 편집 스크립트들로 표현하고 이들 편집 스크립트들을 병합한 후 원본문서에 적용하여 병합 효과를 내고자 한다. 이때 편집 스크립트로 편집 작업을 표현하면 문서의 상태의 차이에서는 쉽게 알 수 없는 편집 작업의 의도가 파악되므로 병합 과정에서 상황에 적합한 처리가 가능하며, 편집 스크립트 기반의 버전 관리는 편집 이력에 대한 질의가 편리하며 분산 환경에서 변화 내용을 전파하는 데 효율적이다. 최근 XML 문서에 대한 편집 스크립트 기반의 버전 관리의 중요성, 그리고 XML 병합 기법이 강조되고 있다[2, 19].

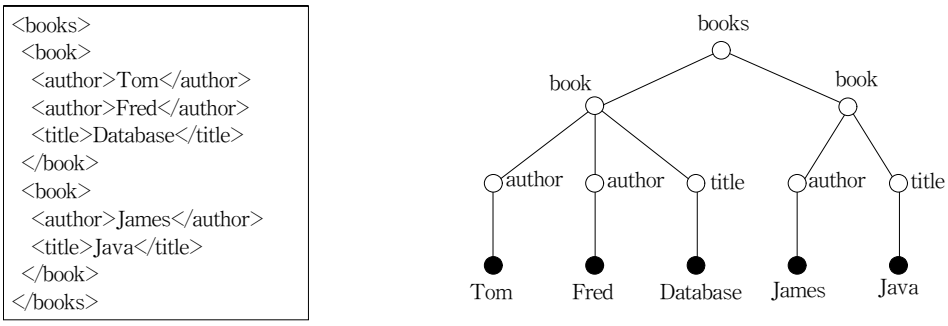
본 논문에서는 XML 문서들의 병합 시 DTD나 Schema는 고려하지 않는다. 원본 XML 문서에 대한 편집 스크립트들을 병합하고 이를 원본 문서에 적용하는 문서 병합 모델을 제안한다. 우선 병합에 필요한 각종 개념들, 즉 정적 편집 스크립트, 병합 시 간섭 및 충돌 개념을 정의하고 병합 시 간섭이 발생할 경우의 편집 스크립트의 조정 방법을 제시한다. 그리고 충돌에 관한 지정된 조건을 만족하는 두 편집 스크립트들에 병합 시 편집 스크립트의 조정 방법을 사용하면 병합 순서에 관계없이 동일한 결과를 생성함을 증명한다. 편집 스크립트, 즉 원본문서와 수정된 문서의 변화(차이)의 계산은 X-treeDiff+[4]에 의해 생성된다고 가정한다. 본 논문의 구성은 다음과 같다. 2절에서는 X-treeDiff+에서의 문서 표현인 X-tree와 편집연산들에 대해 소개하며, 3절에서는 편집 스크립트 기반의 문서 병합 모델을 제시한다. 이때 정적 편집 스크립트, 편집 스크립트 간의 간섭 및 조정 기법, 그리고 충돌 개념과 이들에 기초한 병합의 특성들을 설명한다. 끝으로 4절에서 결론과 미래 연구에 대해 언급한다.

2. 기본 개념

X-treeDiff+[4]는 웹페이지 변조(defacement)의 실시간 탐지를 위한 변화탐지 시스템에 사용된 기법으로 문서의 노드 수를 n이라할 때 O(n)의 성능을 보이는 해싱기반의 알고리즘이다. X-treeDiff+에서는 비교 대상의 두 XML 문서를 각각 X-tree로 변환하고 두 X-tree들 사이의 대응을 생성하고 이로부터 편집 스크립트를 추출한다. 본 절에서는 X-treeDiff+에서 사용되는 트리 구조인 X-tree에서의 노드 식별방법과 본 논문에서 사용될 편집연산에 대해 소개한다.

X-tree 내에서 노드는 nID(node identifier)를 통해 식별된다. 노드 n의 nID는 루트노드로부터 노드 n까지의 경로를 나타내는데, 이는 점으로 구분되는 일련의 레이블과 인덱스의 쌍으로 $.label_1[index_1].label_2[index_2]....label_k[index_k]$의 구조를 갖는다. 이때 *label*은 노드의 엘리먼트 명이나 텍스트 값을 의미하며 *index*는 형제노드들 중 동일한 *label*이 있는 경우 이를 구별하기 위해 사용된다. 형제 노드들 중 동일 *label*의 노드가 하나 밖에 없는 경우 *index*는 0의 값을 가지며 m개의 노드가 존재하는 경우, 좌측 노드부터 *index*는 0, 1, ..., m-1 값을 갖는다.

(그림 1)의 (a)는 책의 리스트에 대한 XML 문서의 간단한 예이고 이는 (그림 1)(b)에서 트리구조로 표시되는데, 흰색 원은 엘리먼트 노드를, 검은 원은 텍스트 노드를 의미한다. nID 개념을 (그림 1)의 예를 통해 설명하면, 문서 T_0에서 첫 번째 책의 두 번째 저자 엘리먼트의 nID는 .books[0].book[0].author[1]로, 두 번째 책의 책 제목은 .books[0].book[1].title[0].Java[0]로 표현된다. 이때 텍스트 노드는 텍스트 값을 nID에 표시한다. X-treeDiff+에서는 두 XML문서에 대해 1단계에서는 ID 속성과 서브트리의 해싱 값을 통해 동일 서브트리의 쌍을 대응시키고 2단계에서는 이들 대응을 루트로 확대하고 3단계에서는 루트로부터 깊이우선탐색을 수행하면서 대응이 결정되지 않은 노드의 쌍을 대응시킨다. 4단계에서는 대응된 노드 쌍에 대해 대응의 튜닝 작업을 수행한다. X-treeDiff+는 대응의 결과를 기반으로 차이를 나타내는 편집연산들, 즉 편집 스크립트를 추출한다.



(a) XML 문서의 예: T_0 (b) XML 문서 T_0의 트리 표현

(그림 1) 간단한 XML 문서의 예

X-treeDiff+는 삽입, 삭제, 갱신, 이동 그리고 복사연산을 지원하는데[4], 편집 스크립트의 추출 시, 삭제연산은 원본 문서에는 있으나 결과 문서에 없는 노드나 서브트리가 존재할 때, 삽입연산과 복사연산은 결과 문서에 새로운 노드나 서브트리가 추가될 때, 갱신노드는 텍스트 노드의 값이 변경되었을 때 생성되며 이동연산은 삭제와 삽입연산의 결합 형태로 생성된다[23]. 그런데 갱신연산과 이동연산은 삭제연산과 삽입연산으로, 그리고 복사연산은 삽입연산으로 대체할 수 있어 본 논문에서는 분석의 편의를 위해 삽입연산, 삭제연산만을 고려한다. 삽입연산과 삭제연산의 정의는 <표 1>에 제시되어있으며, 노드 n은 노드의 nID로 삽입연산의 노드 n은 삽입될 내용의 부모의 노드의 nID, 삭제연산의 n은 삭제될 내용의 루트노드의 nID를 나타낸다.

(그림 2)의 문서 T_1과 T_2에 대해 X-treeDiff+가 생성한 편집 스크립트 ES_1이, 그리고 문서 T_1과 T_3에 대해 생성된 편집 스크립트 ES_2가 다음에 제시되어있다.

ES_1 = [D(.a[0].b[0].d[0]), I("<i/>", .a[0].c[0].f[0], 2)]

ES_2 = [I("<j/>", .a[0].b[0].e[0], 2)]

참고로 문서 T_1에 ES_1을 적용하면 T_2가, 그리고 ES_2를 적용하면 T_3이 생성됨을 알 수 있다.

〈표 1〉 편집 연산

편집연산	연산 의미
삽입연산 I(c, n, k)	삽입내용 c를 노드 n의 k번째 자식노드로 삽입.
삭제연산 D(n)	노드 n을 루트로 하는 서브트리의 내용을 삭제.

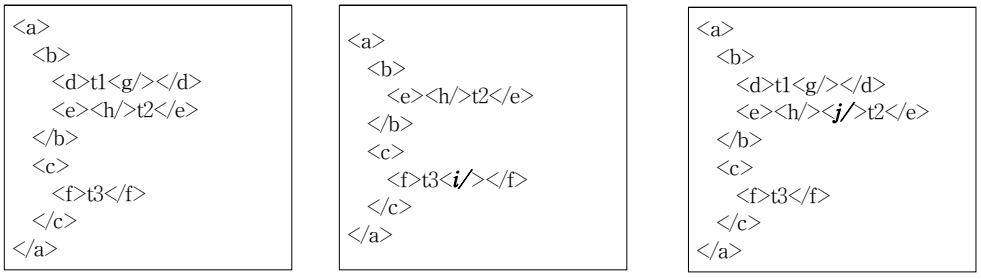
3. 편집 스크립트 기반의 XML 문서 병합 모델

본 절에서는 편집 스크립트들의 병합 그리고 이를 원본문서에 적용하여 문서 병합 기능을 수행하는 병합 모델에 대해 설명한다.

3.1 기본 용어 및 개념 정의

사용자의 편집 작업은 일련의 편집 연산들, 즉 편집 스크립트로 표현될 수 있다. 그러나 사용자에 의한 직접적인 편집 스크립트는 편집과정의 실수 및 수정, 단순 작업의 반복, 작업의 비효율적인 구성 등으로 인해 종종 길고 의미가 명확하지 않아 이를 문서의 병합에 사용하는 것은 적절하지 않다. 따라서 본 논문에서는 원본문서와 사용자의 편집 작업으로 생성된 결과문서에 대해 X-treeDiff+가 생성한 편집 스크립트를 사용한다. 기본 개념들을 소개하면,

[정의 1] 문서 T에 대한 편집 연산 op의 ***적용***과 ***적용가능***은 다음과 같이 정의한다. 삽입연산 I(c, n, k)는 문서 T에 노드 n이 존재하고 n의 자식노드가 k-1개 이상이면 I(c, n, k)는 T에 대해 적용가능, 삭제연산 D(n)는 문서 T에 노드 n이 존재하면 D(n)는 T에 대해 *적용가능*하다고 한다. 문서 T에 대한 편집 연산 op의 *적용*을 $op(T)$와 같은 함수 형식으로 표현하며, 문서 T에 편집연산 op가 적용가능하면 $op(T)$는 op가 적용된 문서를 반환하고 적용불가능하면 '정의되지 않음' 의미의 상수 *undef*를 반환한다. 한편, 문서 T에 대해 n개의 편집연산의 리스트([op_1, op_2, ..., op_{n-1}, op_n])로 구성된 편집 스크립트 ES의 ***적용***과 ***적용가능***은 다음과



(a) 원본문서 T_1 (b) 편집된 문서 T_2 (c) 편집된 문서 T_3

(그림 2) XML 문서의 예

같이 정의된다. op_1이 문서 T에 적용가능하고 모든 $i(0<i\leq n)$에 대해 처음부터 i-1번째까지의 편집연산들이 적용된 결과문서(즉, $op_{i-1}(op_{i-2}(...op_1(T)))$에 대해 op_i이 적용가능하면 ES는 T에 적용가능하다고 한다. T에 대해 ES의 적용은 $ES(T)$로 표시되며 이는 $op_n(op_{n-1}(...op_1(T))$로 정의되며 적용불가능한 경우에는 *undef*를 반환한다.

편집연산의 적용가능은 편집연산의 정의로부터 도출된다. (그림 2)의 문서 T_1에 대해 D(.a[0].b[0].d[0])는 적용가능하지만 I("<i/>", .a[0].c[0].f[0], 3)은 적용불가능하다. ES_1과 편집 스크립트 [D(.a[0].b[0].d[0]), I("<k/>", .a[0].b[0], 2)]는 각각 T_1에 대해 적용가능하나, 편집 스크립트 [D(.a[0].b[0].d[0]), I("<k/>", .a[0].b[0].d[0], 1)]은 T_1에 대해 적용불가능하다. I("<k/>", .a[0].b[0].d[0], 1)가 D(.a[0].b[0].d[0])(T_1)에 적용불가능하기 때문이다. 한편, 편집 스크립트의 각 편집연산의 대상 노드의 결정 시점에 따라 편집스크립트는 다음과 같이 구분된다.

[정의 2] 편집 스크립트 ES에 속한 임의의 편집연산 op의 대상 노드가 ES에 속한 op이전의 편집연산들의 실행 결과에 의해 결정될 때, 편집연산 op의 대상노드는 *동적으로 결정*된다고 하고 op의 대상 노드가 ES에 속한 op이전의 편집연산들의 실행결과에 관계없이 동일할 때 op의 대상노드는 *정적으로 결정*된다고 한다. 편집 스크립트에 속한 모든 편집연산의 대상 노드들이 원본문서에 대해 정적으로 결정될 때 이 편집 스크립트는 원본문서에 대해 ***정적 편집 스크립트***라 하고, 편집 스크립트에 대상노드가 동적으로 결정되는 편집연산이 존재할 때 이 편집 스크립트를 ***동적 편집 스크립트***라고 한다.

정적 편집 스크립트에 속한 편집연산의 nID는 편집 스크립트의 실행 전과 편집 스크립트 실행 중에도 동일한 노드를 가리킨다. T_1에 대해 ES_1와 ES_2는 각각 정적 편집 스크립트이며 편집 스크립트 [I("<m>t4</m>", .a[0], 1), I("<n/>", .a[0].m[0], 1)]은 동적 편집 스크립트이다. 사용자들은 직전의 편집결과를 보면서 다음 작업을 진행하므로 사용자들에 의한 편집 스크립트들은 동적일 가능성이 높다. 다음에서는 편집 스크립트의 병합의 정의를 소개한다.

[정의 3] 편집 스크립트 $ES_i([op_1^i, op_2^i, ..., op_{n-1}^i, op_n^i])$와 $ES_j([op_1^j, op_2^j,..., op_{m-1}^j, op_m^j])$에 대해 ***편집 스크립트의 병합***(concatenation) $ES_i \oplus ES_j$은 두 스크립트를 연결한 리스트 $[op_1^i, op_2^i, ..., op_{n-1}^i, op_n^i, op_1^j, op_2^j,..., op_{m-1}^j, op_m^j]$를 반환한다. 병합된 편집 스크립트 $ES_i \oplus ES_j$를 문서 T에 적용한 결과는 $ES_i \oplus ES_j(T)$로 표현되며 이의 의미는 $ES_j(ES_i(T))$으로 정의한다.

병합의 의미에서 알 수 있듯이 병합의 순서대로 편집 스크립트들이 실행되므로 독립적으로 실행될 때의 의도와는 다른 결과들이 발생할 수 있다. 그러나 정적 편집 스크립트들을 가정하면 각 편집연산들의 대상노드를 알 수 있기 때문에 편집 스크립트들을 실행하지 않고서도 분석이 가능하다. 따라서 본 논문에서는 정적 편집 스크립트들에 대한 병합을 고려한다.

3.2 병합 시 편집 스크립트의 간섭

편집 스크립트 간의 간섭 개념을 설명하고 간섭이 없을 경우 정적 편집 스크립트의 병합의 특성을 분석한다. 우선 편집연산의 간섭 개념부터 소개하면,

[정의 4] op_x와 op_y는 문서 T에 각각 적용가능한 편집연산이다. op_x의 실행 후 결과문서 $op_x(T)$에 대해 op_y의 nID가 나타내는 노드가, (i) 존재하지 않거나 T에 대한 op_y의 대상 노드와 같지 않을 때, 또는 (ii) T에 대한 op_y의 대상 노드와 같은 경우 다음의 (a) 또는 (b)를 만족할 때, op_x는 op_y를 ***간섭한다***고 한다. (a)는 op_x가 삭제연산, op_y가 삽입연산인 경우 op_x가 op_y에 의해 삽입되는 노드의 왼쪽 형제노드를 삭제하는 경우를 의미하고, (b)는 op_x와 op_y가 삽입연산으로 op_x가 op_y와 같은 부모노드에 삽입하나 그 삽입되는 위치가 op_y와보다 빠르거나 같은 위치일 경우를 의미한다. 한편, 조건(i)과 (ii)를 만족하지 않을 때 op_x가 op_y를 간섭하지 않는다고 한다.

조건(i)은 op_y가 삽입연산이든 삭제연산이든 관계없이 적용되는 조건으로 편집연산의 대상노드가 삭제되거나 변경되는 경우를 의미하며 조건(ii)는 op_y가 삽입연산일 경우, 삽입될 부모 노드는 동일하지만 자식으로서의 삽입되는 위치가 영향 받는 경우를 의미한다.

[정리 1] 문서 T와 이에 각각 적용가능한 편집연산 op_x와 op_y에 대해 op_x와 op_y가 서로 간섭하지 않으면 이들은 어떤 순서로 실행해도 결과는 동일하다, 즉 $[op_x, op_y](T) = [op_y, op_x](T)$.

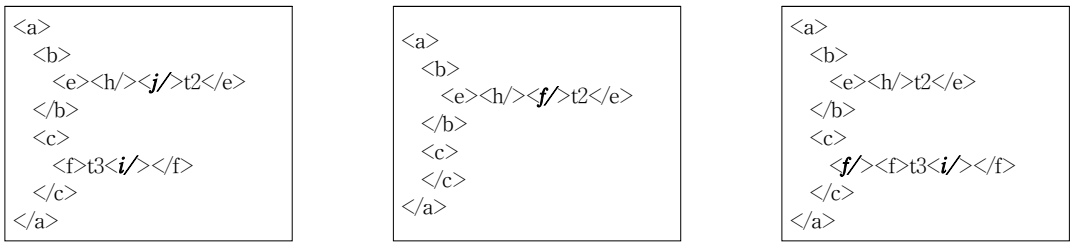
[증명] op를 나중에 실행되는 편집연산이라 하면, op가 삭제연산일 때, 먼저 실행된 편집연산의 실행 결과에 대한 편집연산 op의 대상노드는 T에서 대상노드와 동일하고 op가 삽입연산일 경우도 자식으로서의 상대적 삽입위치도 동일하므로 $[op_x, op_y](T) = [op_y, op_x](T)$이다.

[정의 5] 문서 T와 이에 각각 적용가능한 편집 스크립트 ES_i와 ES_j에 대해, ES_j에 속한 편집연산을 간섭하는 편집연산이 ES_i에 존재한다면 ES_i는 ES_j를 ***간섭한다***고 하고 ES_j에 속한 편집연산을 간섭하는 어떤 편집연산도 ES_i에 존재하지 않으면 ES_i는 ES_j를 간섭하지 않는다고 한다.

[정리 2] 문서 T와 이에 각각 적용가능한 정적 편집 스크립트 ES_i와 ES_j에 대해 ES_i가 ES_j를 간섭하지 않고 ES_j가 ES_i를 간섭하지 않으면 $ES_i \oplus ES_j(T) = ES_j \oplus ES_i(T)$이다.

[증명] ES_j에 속한 임의의 편집연산 op_y를 고려하자. ES_i의 어떤 편집연산도 op_y를 간섭하지 않고 ES_j는 정적 편집 스크립트이므로 op_y의 ES_i의(T)에 대한 대상노드는 T에서의 대상노드와 같으며 대상노드에 대한 실행효과도 동일하다. 정적 편집스크립트의 특성으로 ES_j에 속한 모든 편집연산들에도 같은 내용이 적용된다. 또한 ES_j가 ES_i를 간섭하는 반대방향도 마찬가지로 적용되므로 병합 순서에 관계없이 실행결과는 같다.

T_1에 대해 ES_1과 ES_2는 서로 간섭하지 않으며 $ES_1 \oplus ES_2(T_1)$은 (그림 3)에 제시되며 $ES_2 \oplus ES_1(T_1)$과 동일하다. T_1에 대해 ES_1과 ES_3 = [D(.a[0].c[0].f[0]), I("<j/>", .a[0].b[0].e[0],2)]



(a) $ES_1 \oplus ES_2(T_1)$의 경우 (b) $ES_1 \oplus ES_3(T_1)$의 경우 (c) $ES_1 \oplus ES_4(T_1)$의 경우

(그림 3) 병합된 편집 스크립트의 실행 예

의 경우, ES_1은 ES_3을 간섭하지 않으나, ES_3의 D(.a[0].c[0].f[0])은 ES_1의 삽입연산의 대상노드(.a[0].c[0].f[0])를 삭제하므로 ES_3은 ES_1을 간섭한다. $ES_1 \oplus ES_3(T_1)$의 결과는 (그림 3)의 (b)에 제시되며 ES_1의 삽입 결과가 ES_3에 의해 삭제되었음을 보인다.

T_1에 대해 ES_1과 ES_4 =[I("<f/>",.a[0].c[0],1)]의 경우, ES_1은 ES_4를 간섭하지 않으며 $ES_1 \oplus ES_4(T_1)$은 (그림 3)의 (c)에 제시되어있다. 그러나 $ES_4(T_1)$에 대한 ES_1의 삽입연산 I("<i/>",.a[0].c[0].f[0],2)는 ES_4의 삽입연산으로 인한 변화로 적용불가능하게 되어 ES_4는 ES_1을 간섭한다. 그러나 ES_4의 ES_1에 대한 간섭의 성격은 ES_3의 ES_1에 대한 간섭과는 다르다. ES_3의 ES_1에 대한 간섭은 의도가 서로 충돌하여 발생하고 ES_4의 ES_1에 대한 간섭은 ES_4의 실행으로 인한 변화로 ES_1의 일부 편집 연산이 원래의 의도대로 실행되지 않으므로 발생한다.

3.3 편집 스크립트의 병합 시 조정 기법

ES_4의 ES_1에 대한 간섭은 ES_4의 삽입연산으로 인해 ES_1의 삽입연산의 대상노드(T_1에서 .a[0].c[0].f[0])가 $ES_4(T_1)$에서는 .a[0].c[0].f[1]로 표현되기 때문이다. 따라서 ES_4와 ES_1의 병합 시 ES_1의 삽입연산의 nID를 .a[0].c[0].f[0]에서 .a[0].c[0].f[1]로 수정하여 T_1에 적용하면 (그림 3)의 (c)와 같은 결과를 생성한다. 본 절에서는 이와 같이 편집 스크립트의 병합 시, 먼저 실행될 편집 스크립트의 영향을 고려해서 나중에 실행될 편집 스크립트의 편집연산들의 대상노드의 식별체계(nID)를 조정하는 방법을 설명한다. 우선 필요한 함수들을 정의한다.

[정의 6] n이 원본 문서의 임의 노드를 가리키는 nID라 할 때 length(n)은 n에 대해 경로의 길이, 즉 레이블과 인덱스 쌍의 수를 반환하고 label(n,i)과 index(n,i)는 n에 대해 루트로부터 각각 i번째 레이블과 i번째 인덱스를 반환한다. prefix(n, i)는 nID n의 앞부분, 즉 루트부터 i번째까지의 레이블과 인덱스의 쌍까지를 반환한다. 한편, 삽입연산 I(x,n,k)에 대해 nID(I(x,n,k))은 대상 문서에 대한 삽입연산, 즉 x를 노드 n의 k번째 자식으로 삽입하는 연산이 실행될 때 x의 루트에 해당하는 노드의 nID를 반환하고, IncrIdx(n, k, num)는 노드 n의 k번째 인덱스를 num의 값만큼 증가시키는 함수이다.

(그림 2)의 T_0의 예에 대해 length(.books[0].book[1].title[0].Java[0]) = 4이며, label(.books[0].book[0].author[0], 2) = 'book', index(.books[0].book[1].title[0], 3) = 0이 된다. 같은 문서에 대해 prefix함수와 nID 함수의 예는 다음과 같다. prefix(.books[0].book[1].title[0], 2) = .books[0].book[1]이며, nID(I('<author>David</author>',.books[0].book[0],2)) = .books[0].book[0].author[1]이다. 한편, 가령 IncrIdx(.books[0].book[1].title[0], 3, 1)은 노드의 nID를 .books[0].book[1].title[**1**]로 변경시킨다.

편집연산 op_x가 먼저 실행될 때 대해 편집연산 op_y가 조정이 필요할 수 있다. <표 2>는 조정일 필요한 조건과 조정내용을 설명한다. 유형1은 삭제연산 op_x가 삭제연산 op_y에 영향을 끼쳐 op_y의 nID에 조정이 필요한 경우로 (그림 1)의 문서 T_0에 대해 op_x와 op_y가 각각 D(.books[0].book[0])와 D(.books[0].book[1].author[0])일 때 op_y의 nID는 .books[0].book[**0**].author[0]으로 조정되어야 한다. 유형2와 유형3은 삭제연산 op_x가 삽입연산 op_y에 영향을 주는 경우다. 유형2는 삭제연산 op_x가 동일부모의 자식에 대한 삽입연산 op_y에 영향을 주는 것으로 삽입연산의 위치(즉 k')을 감소시켜야 하는 경우이다. 유형3은 op_y가 삽입연산이란 점을 제외하고는 유형1과 거의 같은 경우이다. 예를 들면, D(.books[0].book[0])에 대해 I("<publisher>SAMS</publisher>",.books[0].book[1], 3)을 고려하면, 삽입연산의 nID의 조정(즉, .books[0].book[**0**]으로의 변화)이 필요함을 알 수 있다.

유형4는 삽입연산 op_x에 대한 삭제연산 op_y의 조정의 필요한 경우로 예를 들어 설명하자. 삽입연산 op_x, I("<book><title>C++</title></book>",.books[0], 1)는 새로운 book 정보를 첫 번째 엘리먼트로 추가한다. 이때 삭제연산 op_y, D(.books[0].book[0].author[1])는 원래의 의도와 다른 삭제를 수행하므로, 삭제연산의 nID의 조정(즉, .books[0].book[**1**].author[1]으로의 변화)이 필요하다. 유형5는 동일부모에 대한 삽입연산들의 경우, 삽입연산 op_y의 삽입 위치(k')의 조정이 필요한 경우를 설명하며, 유형6은 유형4와 거의 삭제연산이 삽입연산으로 바뀐 것을 제외하고는 거의 비슷하다. 유형4의 예의 삽입연산 op_x에 대해 유형3의 삽입연산 op_y를 고려하면, 삽입연산 op_y의 nID의 조정(즉, .books[0].book[**2**]으로의 변화)이 필요하다.

<표 2>의 조정조건 및 조정방법을 기반으로 편집연산의 실행으로 인한 조정결과와 편집 스크립트의 실행으로 인한 조정결과를 다음과 같이 정의한다.

[정의 7] 편집연산 op_x와 op_y에 대해 함수 **Adj(op_x, op_y)**

〈표 2〉 편집연산 op_x에 op_y의 병합 시 op_y의 조정 조건 및 조정 내용

op_x	op_y	유형	조 건	op_y의 조정 내용
D(n)	D(n')	1	$l_n \le$ length(n') $\wedge$ prefix(n, l_n-1) = prefix(n', l_n-1) $\wedge$ label(n, l_n) = label(n', l_n) $\wedge$ index(n, l_n) < index(n', l_n) (단, l_n = length(n))	IncrIdx(n', l_n, -1)
	I(x', n', k')	2	m' is right-sibling(n) (단, m'=nID(I(x',n',k)))	k' = k'- 1
		3	$l_n \le$ length(n') $\wedge$ prefix(n, l_n-1) = prefix(n', l_n-1) $\wedge$ label(n, l_n) = label(n', l_n) $\wedge$ index(n, l_n) < index(n', l_n) (단, l_n = length(n))	IncrIdx(n', l_n, -1)
I(x, n, k)	D(n')	4	$l_m \le$ length(n') $\wedge$ prefix(m, l_m-1) = prefix(n', l_m-1) $\wedge$ label(m, l_m) = label(n', l_m) $\wedge$ index(m, l_m) < index(n', l_m) (단, m = nID(I(x, n, k))이고 l_m = length(m))	IncrIdx(n', l_m, 1)
	I(_,n', k')	5	$n = n' \wedge k < k'$	k' = k'+ 1
		6	$l_m \le$ length(n') $\wedge$ n = prefix(n', l_m-1) $\wedge$ label(m, l_m) = label(n', l_m) $\wedge$ index(m, l_m) < index(n', l_m) (단, m = nID(I(x, n, k))이고 l_m = length(m))	IncrIdx(n', l_m, 1)

는 op_x와 op_y가 <표 2>의 조건을 만족할 때 op_y에 대해 'op_y의 조정내용'이 반영된 편집연산을 반환하고 조건을 만족하지 않으면 op_y를 그대로 반환한다고 정의된다. 편집 스크립트ES_i와 편집연산op_x에 대해 **Adj(ES_i, op_y)**는 다음과 같이 확장 정의된다.

Adj([], op_y) = op_y

Adj([op_x|L], op_y) = Adj(L, Adj(op_x, op_y))

[정리 3] 문서 T에 각각 적용가능한 편집연산 op_x와 op_y에 대해, op_x(T)에 대한 Adj(op_x, op_y)의 대상노드가 존재하면 이는 T에 대한 op_y의 대상노드와 동일하다.

[증명] 하나의 노드 p의 삭제 또는 삽입이 기존 노드 q의 nID의 변화를 유발하는 경우는 삭제될 또는 삽입될 p의 우측의 형제들 중 q 또는 q의 조상이 있는 경우일 뿐이다. <표 2>의 유형1, 유형2, 유형3은 p의 삭제에 대한, 그리고 유형4, 유형5, 유형6은 p의 삽입에 대한 모든 경우들에 대한 조건과 조정내용이 제시되어있다. 그런데 T에 대한 op_x의 실행은 op_y의 대상노드를 삭제하는 경우(즉, 대상노드가 존재하지 않음.), 아니면 op_y에 아무런 영향을 주지 않거나 op_y의 대상노드의 nID의 조정이 필요한 경우로 구분된다. 이때 op_x(T)에 대한 Adj(op_x, op_y)의 대상노드는 T에 대한 op_y의 대상노드와 동일하다..

[정리 4] 문서 T에 각각 적용 가능한 정적 편집 스크립트 ES_i와 편집연산 op_y에 대해, 문서 ES_i(T)에 대한 조정된 편집연산 Adj(ES_i, op_y)의 대상노드가 존재하면 이는 T에 대한 op_y의 대상노드와 동일하다.

[증명] ES_i의 모든 편집연산의 대상노드는 정적으로 결정되므로 각 편집연산에 대해 정리3을 귀납적(inductively)으로 적용하면 증명된다.

[정의 8] 편집 스크립트 ES_i과 ES_j에 대해 함수 AdjES(ES_i, ES_j)는 다음과 같이 정의된다.

AdjES(ES_i, []) = []

AdjES(ES_i, [op_y|L]) = [Adj(ES_i, op_y)]⊕AdjES(ES_i, L)

[정리 5] 문서 T에 각각 적용가능한 정적 편집 스크립트 ES_i과 정적 편집 스크립트 ES_j에 대해 ES_i(T)에 대한 조정된 편집 스크립트 AdjES(ES_i, ES_j)의 대상노드들이 존재하면 이들은 T에 대한 ES_j의 대상노드들과 각각 동일하다.

[증명] ES_j의 각 편집연산의 대상노드가 정적으로 결정되므로 ES_j의 각 편집연산 op_y에 대해 정리4를 귀납적으로 적용하면 증명된다.

3.4 충돌 개념과 조정 기법에 기반한 편집 스크립트 병합의 특성 분석

정리5는 정적 편집 스크립트의 병합 시, 임의의 편집 스크립트가 나중에 실행되더라도 먼저 실행될 편집 스크립트의 의한 영향을 고려해서 나중에 실행될 편집 스크립트의 편집연산들을 조정하면 원래의 의도대로 실행할 수 있다는 의미이다. 그러나 이때 조정된 편집연산들의 대상노드들이 존재하지 않을 수 있다. 이는 두 편집 스크립트의 원래의 의도가 서로 충돌하여 이들을 병합할 수 없는 경우로 3.2절의 T_1에 대한 ES_3과 ES_1의 병합의 경우가 이에 해당된다. 이를 위해 충돌 개념을 설명하고자 한다. 우선 Conflict 함수와 편집연산 간의 충돌 개념을 정의한다.

[정의 9] 집연산 op_x와 op_y에 대해 함수 Conflict(op_x, op_y)와 편집연산 간의 충돌은 다음과 같이 정의된다.

Conflict(D(n), D(n')) = if n'=descendant-or-self(n) then true else false

Conflict(D(n), I(_,n',_)) = if n'=descendant-or-self(n) then true else false

Conflict(I(_,_,_), D(_)) = false

Conflict(I(_,_,_), I(_,_,_)) = false

이때 op_x와 op_y에 대해 Conflict(op_x, op_y) = true이면 op_y는 op_x에 충돌한다고 하고 Conflict(op_x, op_y) = false이면 op_y는 op_x에 충돌하지 않는다고 한다. (단, descendant-or-self(n)는 n의 자신이나 그의 자손 노드들을 반환하는 함수로 가정한다.)

[정의 10] 편집연산 op_x와 op_y가 각각 삽입연산 I(_,n,k), I(_,n',k')라고 하자. op_x와 op_y가 동일부모에 대해 같은 자식 위치에 삽입하고자 할 때(즉, if $n = n' \wedge k = k'$) op_x와 op_y는 서로 삽입-충돌한다고 하고 그렇지 않으면 op_x와 op_y는 서로 삽입-충돌하지 않는다고 한다.

op_y가 op_x에 충돌함은 op_x가 op_y의 대상 노드를 삭제하는 경우로 op_x의 실행 시 op_y의 실행이 불가능하게 된다. op_x와

op_y가 서로 삽입-충돌함은 두 연산이 서로 같은 위치에 삽입하고자 하는 경우로 실행은 가능하나 원래의 의도대로 실행이 불가능한 경우를 의미한다.

[정리 6] 문서 T와 이에 각각 적용 가능한 편집연산 op_x와 op_y에 대해 (i) Conflict(op_x, op_y) = true이면 $op_x(T)$에 Adj(op_x, op_y)는 적용불가능하다. (ii) Conflict(op_x, op_y) = false이면 $op_x(T)$에 Adj(op_x, op_y)는 적용가능하다.

[증명] (i) Conflict(op_x, op_y) = true이면 op_y의 대상노드가 op_x에 의해 삭제되는 경우로 정의7에 의해 Adj(op_x, op_y)는 op_y를 그대로 반환하므로 Adj(op_x, op_y)는 $op_x(T)$에 적용불가능하다. (ii) Conflict(op_x, op_y) = false이면 op_y의 대상노드가 op_x에 의해 삭제되지 않는다. 정리3에 의해 $op_x(T)$에 대한 Adj(op_x, op_y)의 대상노드는 T에 대한 op_y의 대상노드와 동일하므로 적용가능하다.

[정의 11] 문서 T와 이에 각각 적용 가능한 정적 편집 스크립트 ES_i와 편집연산 op_y에 대해 Conflict(op_x,op_y) = true인 op_x가 ES_i에 존재하면 op_y는 ES_i에 충돌한다고 하고 Conflict(op_x, op_y) = true인 op_x가 ES_i에 존재하지 않으면 op_y는 ES_i에 충돌하지 않는다고 한다.

[정리 7] 문서 T와 이에 각각 적용 가능한 정적 편집 스크립트 ES_i와 편집연산 op_y에 대해, (i) op_y가 ES_i에 충돌하면 Adj(ES_i, op_y)는 $ES_i(T)$에 적용 불가능하다. 한편, (ii) op_y가 ES_i에 충돌하지 않으면 Adj(ES_i, op_y)는 $ES_i(T)$에 적용가능하다.

[증명] op_y가 ES_i에 충돌하면 Conflict(op_x,op_y) = true인 op_x가 ES_i에 존재한다. ES_i가 정적 편집스크립트이므로 이에 속한 각각의 편집연산에 대해 정리6을 적용하면 (i)이 증명된다. (ii)는 Conflict(op_x,op_y) = true인 op_x가 ES_i에 존재하지 않는 경우로, 정리4를 적용하면 (ii)이 증명된다.

앞의 정의9-11과 정리6-7은 다음의 정의12와 정리8-9를 위한 준비 단계이다. 본 논문의 핵심 정의와 정리는 다음에 제시된다.

[정의 12] 문서 T와 이에 각각 적용 가능한 정적 편집 스크립트 ES_i와 ES_j에 대해, ES_i에 충돌하는 편집연산이 ES_j에 존재하면 ES_j는 ES_i에 충돌하고 ES_i에 충돌하는 어떤 편집연산도 ES_j에 존재하지 않으면 ES_j는 ES_i에 충돌하지 않는다고 한다. ES_j가 ES_i에 충돌하지 않으며 또한 ES_i도 ES_j에 충돌하지 않으면 ES_i와 ES_j는 서로 충돌하지 않는다고 한다.

[정리 8] 문서 T와 이에 각각 적용가능한 정적 편집 스크립트 ES_i와 ES_j에 대해 (i) ES_j가 ES_i에 충돌하면 AdjES(ES_i, ES_j)는 $ES_i(T)$에 적용 불가능하다. 한편, (ii) ES_j가 ES_i에 충돌하지 않으면 AdjES(ES_i, ES_j)는 $ES_i(T)$에 적용가능하다.

[증명] ES_j가 ES_i에 충돌하면 ES_i에 충돌하는 편집연산 op_y가 ES_j에 존재한다. op_y의 대상노드는 정적으로 결정되므로 정리7에 의해 Adj(ES_i, op_y)는 $ES_i(T)$에 적용 불가능하다. 따라서 (i)이 증명됨. (ii)는 ES_i에 충돌하는 어떤 편집연산도 ES_j에 존재하지 않는 경우로 정적 편집스크립트인 ES_j의 모든 편집연산에 정리7을 적용하면 증명된다.

[정리 9] 문서 T와 이에 각각 적용가능한 정적 편집 스크립트 ES_i와 ES_j가 서로 충돌하지 않고 ES_i의 어떤 삽입연산도 ES_j의 어떤 삽입연산과 서로 삽입-충돌하지 않으면, AdjES(ES_i, ES_j)(ES_i(T)) == AdjES(ES_j, ES_i)(ES_j(T))이다.

(증명) ES_i와 ES_j가 서로 충돌하지 않으므로 정리8에 의해 AdjES(ES_i, ES_j)는 $ES_i(T)$에 적용가능하고 또한 AdjES(ES_j, ES_i)는 $ES_j(T)$에 적용가능하다. 이는 대상노드들이 존재한다는 의미로 정리5에 의해 이들 조정된 편집연산들의 대상노드들은 각각 원본문서 T에 대한 조정전의 편집연산들의 대상노드들과 동일하다. 따라서 조정된 편집연산들은 원래의 의도대로 실행가능하다. 그러나 삽입-충돌되는 두 편집연산은 실행 순서에 따라 다른 결과를 생성하므로 삽입-충돌되는 경우만 제외한다면 AdjES(ES_i, ES_j)($ES_i(T)$) == AdjES(ES_j, ES_i)($ES_j(T)$)이다.

3.5 편집 스크립트 병합 기반의 문서의 병합 기법 소개 및 분석

3.4절의 정리9는 편집 스크립트의 병합을 통한 문서 병합 기법의 핵심으로 본 절에서는 이를 기반으로 문서 병합 기법의 개요를 설명한다. 원본문서 T에 대해 독립적으로 편집이 이루어진 두 문서들 T_a와 T_b에 문서 병합은 다음 순서대로 진행된다.

(1) T와 T_a에 대한 편집 스크립트 ES_i, 그리고 T와 T_b에 대한 편집 스크립트 ES_j를 생성한 후 ES_i와 ES_j가 정적 편집 스크립트인지 검사한다. 아닐 경우는 병합 작업을 중단한다.

(2) ES_i와 ES_j가 서로 충돌하는지 검사한다. 서로 충돌하지 않을 경우 임의의 순서로 병합한다. 병합된 편집 스크립트는 $ES_i \oplus$AdjES(ES_i, ES_j) 또는 $ES_j \oplus$AdjES(ES_j, ES_i)이다. 충돌 시에는 병합 과정을 중단한다.

(3) 위의 병합된 편집 스크립트를 문서 T에 적용하여 결과문서를 생성한다.

위의 절차대로 문서를 병합할 경우, ES_i와 ES_j가 서로 삽입-충돌하지 않으면 어떤 순서로 병합하든지 결과는 동일하다. 그러나 삽입-충돌은 대개의 경우, 규칙을 정하거나 사용자의 선택을 통해 처리 방법을 결정할 수 있다.

본 논문의 목적은 정적 편집 스트립트들 간의 충돌 조건 정의와 병합 시 조정기법을 제안하여 편집 스크립트 기반의 문서 병합 기법에 대한 이론 기반을 제시하는 것으로 현재 본 논문에서는 위의 단계 (2)에 대한 정당성을 제공하고 있으며 완전한 문서 병합 시스템의 구현까지는 상당히 많은 추가적인 연구가 필요하다. 따라서 제안하는 병합 기법의 전체적인 시스템의 구조와 알고리즘에 대한 상세 소개 대신 (그림 4)와 같은 단계 (2)에 대한 개괄적 알고리즘과 이에 대한 성능 분석을 제시한다.

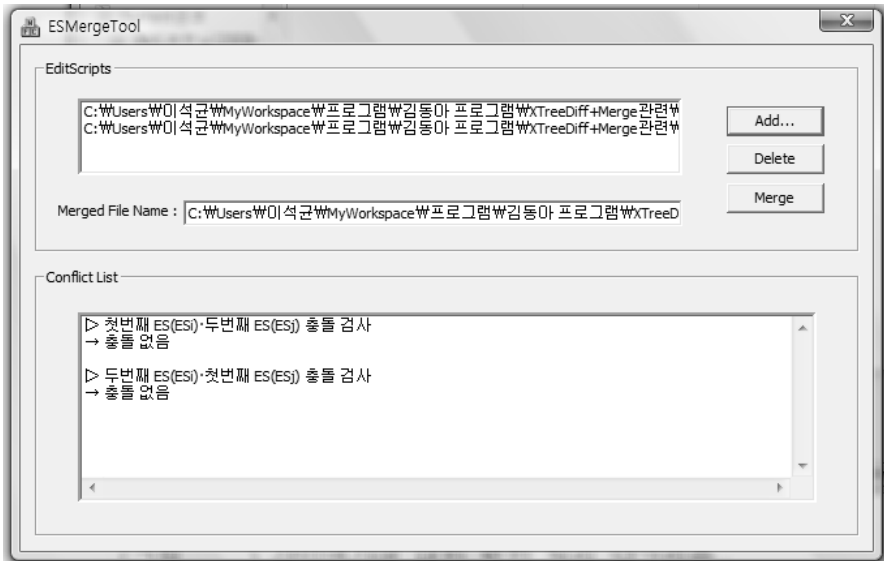
편집 스크립트 ES_i와 ES_j에 대한 충돌 검사는 Conflict(op_x, op_y)의 함수에 기반하고 있는데 이 함수는 각 대상노드의 경로의 길이에 영향 받는다. 편집 스크립트에 속한 편

```
for opai ∈ ESa {          /* 충돌 검사 부분: for루프의 iterating 과정을 묘사 */
  for opbj ∈ ESb {
    if Conflict(opai, opbj) or Conflict(opbj, opai) then
      if opai == I(_,n,k) and opbj == I(_,n′,k′) and n == n′ and k == k′ then
        report opai와 opbj는 서로 삽입 충돌함;   return NULL;
      else
        report opai와 opbj는 충돌함;   return NULL;
}}                                    /* 역방향의 충돌 검사 부분 생략 */
EStemp = ESb;                         /*    AdjES(ESi, ESj) 부분   */
for opbj ∈ EStemp { optemo = opbj;
                    for opai ∈ ESa { optemo = Adj(opai, optemo); }
                  replace opbj in EStemp with optemo              }
return ESa ⊕ EStemp
```

(그림 4) 단계 (2)의 알고리즘

집 연산의 대상 노드들의 경로의 길이의 최대값을 h라 하면, Conflict(op_x, op_y) 함수의 비용은 O(h)이며 편집 스크립트의 편집 연산의 수를 n이라 할 때 충돌 검사는 for 루프를 통해 양 방향으로 진행하므로 할 때 그 비용은 $O(2n^2h)$가 된다. AdjES(ES_i, ES_j)이 조정 부분으로 이에 구현에 필요한 정의6에서 언급한 모든 함수들의 비용은 O(h)이므로 AdjES(ES_i, ES_j)의 비용은 $O(n^2h)$이다. 따라서 단계 (2)의 비용은 $O(n^2h)$이다. 위의 단계 (2)의 알고리즘은 MS Visual Studio 2005의 MFC의 프로그램으로 구현되었고 본 논문의 예제들을 포함한 다양한 경우에 실험하여 이론적 증명에 대해 추가적인 검증을 수행하였다. 실행 화면이 (그림 5)에 제시되었다.

편집 스크립트의 충돌 검사와 병합에 대한 기존의 연구가 존재하지 않아 상태 기반의 병합에 관한 기존 연구들[14, 15, 22]의 결과와 비교 분석한다. 이들은 트리 diff 알고리즘으로 문서 간의 차이를 생성하고 이 차이를 직접 원본 문서에 병합하는 기법들로 비교 결과가 <표 3>에 제시되었다.



(그림 5) 프로토타입 시스템 실행 예

〈표 3〉 기존 병합 기법과의 비교 분석

	상태 기반 병합 기법들	제안된 편집 스크립트 기반의 병합 방법
병합 시 필요 정보	원본 문서, 수정된 문서들	편집 스크립트들
처리 방법/시간	원본 문서와 수정된 문서들에 대해 차이를 찾고 병합하고, 병합 시간은 문서의 병합으로 길다.	편집 스크립트들에 대한 병합을 수행하고 원본 문서에 적용함. 편집 스크립트의 병합으로 처리 시간은 짧다.
충돌 정의	도메인 지식 기반, 경험적 정의	경험적이 아닌 원칙에 기반한 충돌 정의
협업적 편집 적용의 편의성	쉽지 않음.	편리함.

이들 상태 기반의 병합 기법은 스크립트들의 병합에 비해 비용이 크고 충돌의 정의가 경험적이며, 협업적 편집 작업에 사용하기는 쉽지 않다. 그러나 본 논문의 제안 방법이 <표 3>에서의 장점에도 불구하고 현 상태에서는 정적 편집 스크립트를 전제로 하고 있어 실용적인 한계가 있다.

4. 결론 및 미래 연구

본 논문에서는 협업적 편집 작업을 위한 편집 스크립트 기반의 XML 문서의 병합에 필요한 주요 개념들, 즉 정적 편집 스크립트, 편집 스크립트의 간섭, 편집 스크립트의 충돌 등을 정의하고 이들 간의 다양한 특성들을 분석 및 도출하였다. 이들에 기초한 본 논문에서 제안된 병합 기법은 병합 시 대상 문서들에 대해 직접 병합 작업을 수행하는 기존의 XML 문서의 병합기법들과는 달리, 원본문서에 대한 편집 작업들, 즉 편집 스크립트들을 병합하고 병합된 편집 스크립트를 원본 문서에 적용하여 문서를 생성하게 된다.

본 논문의 병합 기법은 동일 문서에 대해 편집된 문서들의 병합 시, 편집된 문서들이 없더라도 편집 스크립트들만 있으면 병합 작업을 수행할 수 있다는 장점이 있다. 따라서 편집 작업이 보다 효율적으로 진행될 수 있을 뿐 아니라, 변화 과정의 이력정보도 유지할 수 있다는 장점이 있다. 또한 편집 작업이 여러 사이트에 분산되어 진행되는 환경에서는 네트워크 부하를 줄이는 효과도 발생한다.

본 논문에서는 편집 스크립트 기반의 문서 병합의 이론적 기틀을 구축하고 있다. 아직 편집 스크립트 병합을 위해 정적편집 스크립트, 삽입연산과 삭제연산을 전제로 하고 있는 등 많은 한계가 있다. 그러나 현재 X-treeDiff+가 정적 편집 스크립트를 생성하도록 대응된 X-tree로부터 편집 스크립트 추출 알고리즘을 개발하고 있으며, 병합된 편집 스크립트들을 다시 병합할 수 있도록 임의의 편집 스크립트를 정적편집 스크립트로 변화시키는 알고리즘에 대해 연구 중에 있다. 이들은 추후 논문으로 발표할 예정이며. 이들을 기반으로 버전 관리 시스템의 프로토타입 개발이 계획 중에 있다. 또한 편집연산을 갱신(update), 이동(move), 복사(copy) 등의 편집연산으로 범위를 확장할 예정이다.

참 고 문 헌

[1] M. Perry, and D. Agawral, “Collaborative editing within the pervasive collaborative computing environment,” Lawrence Berkeley National Laboratory, LBNL 53769, 2003.

[2] A. Marian, S. Abitebouol, G. Cobena, and L. Mignet, “Change-Centric Management of Versions in an XML Warehouse,” in Proc. of the 27th VLDB Conf., Italy, 2001.

[3] Curbera and D.A. Epstein, “Fast Difference and Update of XML Documents,” XTech ’99, San Jose, March, 1999.

[4] S.K. Lee and D.A. Kim, “X-treeDiff+: Efficient Change Detection Algorithm in XML Documents,” Lecture Notes in Computer Science(LNCS 4096), pp.1037-1046, 2006.

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

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

[7] 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.

[8] S. Chawathe and H. G. Molina, “Meaningful Change Detection in Structured Data,” In SIGMOD ’97, pp.26-37, 1997.

[9] S. Lu, “A tree-to-tree distance and its application to cluster analysis,” IEEE TPAMI, 1(2), pp.219-224, 1979.

[10] S. M. Selkow, “The tree-to-tree editing problem,” Information Processing Letters, 6, pp.184-186, 1977.

[11] S. Khanna, K.Kunal, and B.C. Pierce, “A Formal Investigation of Diff3,” FSTTCS 2007: Foundations of Software Technology and Theoretical Computer Science, Lecture Notes in Computer Science(LNCS 4855), pp.485-496, 2007.

[12] M.K. Johnson, “Diff, pathch, and friends,” Linux Journal, Aug. 1996.

[13] S. Balasubramaniam, and B.C. Pierce, “What is a file synchronizer?,” *Proc. of Fourth Annual ACM/IEEE Int. Conf. on Mobile Computing and Networking (MobiCom'98)*, pp. 98-108, Oct., 1998.

[14] T. Lindholm, “A Three-way Merge for XML Documents,” *Proc. of the 2004 ACM symp. on Document engineering*, pp.1-10, Milwaukee, Wis. Oct., 2004.

[15] R.L. Fontaine, “Merging XML files: a new approach providing intelligent merge of XML data sets,” *Proc. of the XML Europe 2002*, Barcelona, May, 2002.

[16] CVS, Concurrent Versions System, http://www.cvshome.org or http://ximbiot.com/cvs/

[17] SubVersion, Source Version Control System, http://subversion.tigris.org/

[18] R. Smith, GNU diff3 (1988) Version 2.8.1, April 2002; distributed with GNU diffutils package.

[19] S. Ronnau, J. Scheffczyk, and U. Borghoff, “Towards XML Version Control of Office Documents,” Proc. of the fifth ACM symposium on Document Engineering, Bristol, U.K. 2005

[20] S. Ronnau, C. Pauli, and U. Borghoff, “Merging changes in XML documents using reliable context fingerprints,” Proc. of the eighth ACM symposium on Document Engineering, pp.52-61, 2008.

[21] C.-L. Ignat, and M. Norrie, “Flexible collaboration over XML documents,” In Proc. of 3rd Int. Conf on Cooperative Desgin, Visualization, and Engineering(CDVE) pp.267-274, 2006 LNCS 4101.

[22] G.W. Manger, “A Generic Algorithm for Merging SGMS/XML-Instnaces,” In Proc. of XML Europe 2001, Berlin, Germany.
[23] 김동아, “XML 문서에 대한 변화 탐지 및 관리,” 단국대학교 전산통계학과 박사학위논문, 2005.

이 석 균

e-mail : sklee@dankook.ac.kr
1982년 서울대학교 경제학과(학사)
1990년 Computer Science, University of Iowa(석사)
1993년 Computer Science, University of Iowa(박사)
1993년~2007년 세종대학교 정보처리학과 교수
1997년~현 재 단국대학교 컴퓨터학부 교수
관심분야 : 데이터베이스, 실시간 스케줄링, 데이터베이스를 위한 시각 질의어, XML 및 웹 응용, 트리 diff, 트리구조 문서에 대한 버전 관리 등