소프트웨어 다이어그램의 변경 내용 검출 기법

한종대[O] 박근덕 우치수
서울대학교 컴퓨터공학부
{elvenwht[O], dean, wuchisu}@selab.snu.ac.kr

A Technique for Detecting Changes in Software Diagrams

Jongdae Han[O], Geunduk Park, Chisu Wu
School of Computer Science and Engineering, Seoul National University

요 약

XML을 이용하면 다이어그램에서 변경 내용을 검출하는 문제는 트리 구조에서 변경 내용을 검출하는 문제로 전환할 수 있다. 트리 구조의 여러 버전(version) 간에 형제(sibling) 노드의 순서가 유지되지 않는 조건하에서, 트리 구조의 변경 내용을 검출하는 문제는 NP-complete임이 증명되어 있다. 그러므로 본 논문에서는, 트리 연산에 몇 가지 제한을 부가하여, 빠른 시간 내에 최적에 가까운 변경 내용을 검출하는 방법을 제안한다.

1. 서론

소프트웨어 개발 과정에서 시스템의 이해도를 높이고, 개발자들 사이의 의사 소통을 원활히 하기 위한 목적으로, 시스템의 정적 구조나 동작 방식을 다이어그램(diagram)을 사용하여 표현한다. 최근에는 도구간에 다이어그램의 공유하거나 교환을 쉽게 하기 위하여, XML(eXtensive Markup Language)[1]을 이용하여 다이어그램의 형식을 표준화 하려는 연구가 활발히 이루어지고 있다. UXF(UML eXchange Format)[2], XMI(XML Metadata Interchange)[3], EMF(Eclipse Modeling Framework)[4]등은 UML 다이어그램을 XML로 표현하는 방식을 제안하였다.

이렇게 다이어그램의 형식을 표준화하면, CASE도구에 관계없이 변경 내용을 검출하는 일반적인 방법을 제시하는 것이 가능해진다. 즉, XML 문서는 트리 구조로 변환이 가능하기 때문에, 다이어그램의 변경을 검출하는 문제에 트리 구조의 변경을 검출하는 방법을 적용할 수 있다.

트리 구조에 대한 변경 검출 문제는 편집 간격 (edit distance) 문제[5,6]로 알려져 있다. 편집 간격 문제는 두 가지 종류로 나누어지는데, 버전 간에 형제(sibling) 노드의 순서가 유지되는 경우(ordered)와 유지되지 않는 경우(unordered)이다.

본 논문에서 고려하는 다이어그램의 변경 검출 문제는 순서가 유지되지 않는 경우인데, 이러한 문제는 NP-complete임이 증명되어 있다. 그러므로 본 논문에서는, 트리 연산에 몇 가지 제한을 가하여 빠른 시간 내에 최적에 가까운 변경 사항을 검출하는 방법을 제안한다.

2. 소프트웨어 다이어그램의 XML 변환

EMF나 XMI등에 의해 UML 다이어그램이 XML로 변환되는 경우, 그 형태는 <그림 1>과 같다.

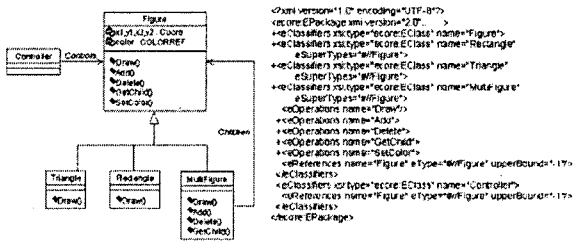

그림 1. UML 다이어그램의 XML 변환

전체적인 개체 관계는 XML 엘리먼트(element)의 트리 관계로 변환되며, 이에 따라 다이어그램상의 변경사항이 XML에 적용된다. XML에 적용된 다이어그램상의 변경연산을 크게 4가지로 나누면 다음과 같다.

- *inserted* : 과거에 없던 개체가 삽입됨
- *deleted* : 과거에 있던 개체가 삭제됨
- *updated* : 개체의 값이 변경되었으나 트리 구조에는 변화가 없음
- *moved* : 한 부모 아래에서의 자신의 위치가 변화함

<그림 2>에서 위 4가지 변경 연산의 예를 찾아볼 수 있다.

3. 관련 연구

Unix에서 파일을 비교하는 알고리즘인 *diff*[7]는 기본적으로 선형적인 텍스트 파일의 비교를 위해 제안된 기법임으로, XML 문서처럼 계층적으로 구조화된 문서를 다루는 데는 적합하지 않다.

트리 구조는 계층적 정보를 표현하는데 흔히 사용되는 유용한 자료 구조로, 변경 검출에 대한 많은 연구들이 이루어져 왔다. 이러한 기존 연구들 중에서 순서가 유지되는 경우에는 효율적인

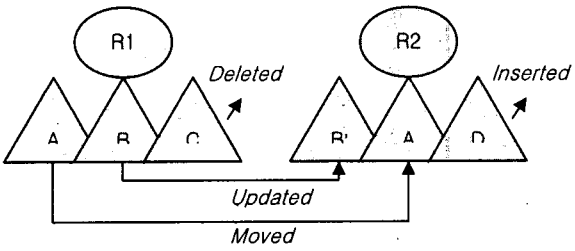

그림 2. 변경연산의 예

알고리즘이 제안되어 있지만, 순서가 유지되지 않는 경우는 NP-complete임이 증명되어 있다[5]. 따라서 순서가 유지되지 않는 편집 간격 문제는 P=NP가 아니라면, PTAS(Polynomial Time Approximation Schemes)이 존재하지 않는다.

따라서, 변경사항 검출의 품질을 일부 포기하는 대신 실용적인 시간 내에 실행되는 방법들이 제안되었는데, BULD diff[5]가 이러한 방법의 대표적인 예이다. BULD diff는 전체 트리의 일부가 최대로 일치되는 위치로부터 전체 트리로 일치성을 검증해 나가는 방식을 채택하고 있다(Bottom Up, Lazy Down).

4. XML 변경 검출기 (Change Detector)

이 논문에서 제안하고 구현하고자 하는 변경 검출기의 특징은 크게 세 가지로 정리할 수 있다.

- XML의 트리 구조를 최대한 활용하기 위해 해시테이블을 이용해 하부 트리로의 전파를 생략한다. 특히 해시값을 연산할 때 노드의 경로를 해시값에 포함시켜 해시테이블의 충돌(conflict)을 최소화하도록 한다.
- 동일한 부모 아래에서의 *moved* 만을 지원한다. 변경검출 알고리즘을 간소화 한다.
- XML 트리의 속성(attribute) 노드도 엘리먼트 노드와 동일하게 취급하여, 속성 노드의 변경 검출을 지원한다.

전체 트리 내에서의 효율적인 *moved*를 고려할 경우, 지나치게 많은 시간이 소요되고, 알고리즘에 예외적인 처리를 많이 포함해야 하는 단점이 있다. 또한 소프트웨어 다이어그램에서는 부모가 다른 노드간의 이동 연산은 *deleted* 후 *inserted* 연산으로 간주하는 것이 더 합리적이다.

4.1. 자료구조

XML 변경 검출기를 통과하는 동안 최초의 소프트웨어 다이어그램은 다양한 형태의 자료구조로 변형된다. 최초의 변형은 소프트웨어 다이어그램을 XML로 변형하는 도구에 의해 이루어지며, 본 논문에서는 이를 위해 EMF를 사용하였다. 이 결과로 생성된 XML문서는 JCP의 DOM 규약에 의해 트리 구조로 변경되며, 이에는 Apache 프로젝트의 Xerces 구현체를 이용하였다.

DOM 문서는 원본 XML의 속성을 모두 담고 있지만 실제 구현에 있어서 불필요한 속성을 제외하고 최대한 작은 공간만을 사용하고자 보다 단순한 트리를 사용하였다. 각 노드는 실제 노드의 정보 외에 자식의 *moved*를 검사하기 위한 해시값 배열을 가지고 있다. 두 문서에 대한 트리 이외에 변경사항 검출을 위해 원 문서의 노드 값을 담은 해시테이블을 사용하고 있다.

두번째 트리는 최종적으로 상호 비교과정을 거쳐 변경사항을 산출하고 결과를 출력하기 위한 트리로 변환된다. 이 마지막 트리는 XML의 각 엘리먼트를 노드로 갖는다.

4.2. 알고리즘

본 논문에서의 처리과정에 대한 의사 알고리즘은 다음과 같다.

*기본적인 변경연산 4가지 외에 구현을 위한 추가적인 변경연산 2가지가 다음과 같이 추가되었다.

- *Not modified* : 자신을 포함한 하부 트리에 변화 없음
- *Sub modified* : 자신은 변화가 없으나, 하부 트리에 변화 있음

Phase 1

XML 문서 *v1, v2*를 DOM 트리로 변환한다

DOM 트리 *v1*의 각 노드에 대해서

다음의 해시함수 *h(n), S(n)*을 실행한다

노드의 값을 S, 그 자식 노드의 집합을 C(n)이라 할 때

$$S(n) = \sum_{}^{length(S)} S[i]^i + ModulatedPath$$

$$h(n) = S(n) + \sum_{C \in C(n)} i * h(C)$$

S(n), h(n)을 key로, 각 노드를 value로 갖는 해시테이블을 작성한다

Phase 2

*v2*의 각 노드에 대해

h(v2)의 값을 해시테이블에서 찾는다

존재할 경우 종료한다

존재하지 않을 경우

S(v2)의 값을 해시테이블에서 찾는다

존재할 경우 노드의 status를 *sub modified*로 둔다

존재하지 않을 경우 노드의 status를 변경한다

자식 노드들에 대해 Phase 2를 적용한다

위의 결과를 토대로 *moved*검출 테이블을 작성한다

S(v1)에 해당하는 노드의 *moved*검출 테이블과 비교해

SequentialDetect 알고리즘을 적용한다

*v1*의 각 노드에 대해

*not modified 혹은 updated*로 설정된 경우 종료한다

아래의 두 경우 작업 수행 후 자식들 역시 검증한다

설정되지 않은 경우 status를 deleted로 변경한다

sub modified로 설정된 경우 상태는 그대로 둔다

SequentialDetect 알고리즘

인덱스와 값 (i, v)를 한 원소로 갖는 두 큐 T1, T2에 대해

T1로부터 원소 a1를 인출한다

T2로부터 원소 a2를 인출한다

a1=a2일 경우 인덱스를 비교한다

인덱스가 다를 경우 *moved*로 표시한다

a1, a2를 소거한다

a1≠a2일 경우 a1을 임시 큐 T3에 삽입한다

새로운 a1에 대해 비교를 반복한다

T1의 원소가 모두 소진된 경우 T3을 T1로 대체하고, T2의 남은 원소의 인덱스를 초기화한다

위의 작업을 T1, T2가 모두 소진될 때 까지 반복한다

Status 변경 정책

*v1*의 노드이면서 *v2*에서의 검출에 의해 *not modified, moved, updated*로 설정되지 않은 노드 -> *deleted*

*v2*의 노드로 그 해시값 h(n)이 해시테이블에 없는 경우 S(n) 역시 해시테이블에 없는 경우 -> *inserted*

leaf 노드이며, *v1*에 위치하는 부모에 값이 다른 자식이 있는 경우 -> *updated*

* *v2*에 대한 status 변경은 v1의 해당 노드에도 영향을 미친다

Phase 3

v1의 각 노드에 대해

XML 엘리먼트의 형태로 트리를 정돈한다

만약 노드의 status가 *deleted*이면 변경사항으로 출력한다

v2의 각 노드에 대해

XML 엘리먼트의 형태로 트리를 정돈한다

만약 노드의 status가 *inserted, moved, updated*이면 변경사항으로 정리한다.

이 알고리즘에서는 2가지 해시값을 사용하고 있는데, 이는 두 해시값이 다른 의미로 사용되기 때문이다. S(n)은 루트로부터 자신에게 이르는 경로와 자신의 값에 대한 해시 합으로 이루어지며 노드의 존재 여부를 가리는 데 사용된다. 반면 h(n)은 자신의 하부 트리에서 변화가 일어났는지를 포함하는 해시값으로, h(n)이 해시테이블에 존재한다는 것은 그 하부 트리에는 변화가 일어나지 않았다는 것을 의미한다. 따라서 불필요한 부분에 대한 검증을 피함으로써 전체 실행시간을 단축할 수 있다. Worst case에서는 실행시간이 그다지 단축되지 않으나, 전체 다이어그램에 대비했을 때 변화된 부분이 크지 않은 경우 실행시간이 크게 줄어든다.

SequentialDetect 알고리즘은 동일한 부모 하에서의 *moved*검출을 위한 것이다. 기본적으로 동일한 크기에 동집합 원소를 가진 배열에 대해 사용하는 것을 가정하고 있으며, BULD diff등에서 사용하는 LCS 기반 알고리즘에 비해 공간을 작게 사용한다. 실행시간에 있어서 worst case에 대해 검증할 경우 $O(n^2)$이나 비교적 유사한 배열에 대해 검증을 행할 경우 실행시간이 빠르게 감소하여 O(n)에 가까운 시간 내에 실행될 수 있다. 또한 NP-Complete문제인 *moved*검출의 최적해는 찾아낼 수 없으나 몇몇 경우를 제외하면 2회의 실행으로 최적해를 찾아낼 수 있다. 이로 인해 소프트웨어 다이어그램의 변경사항 검출을 위한 알고리즘으로서는 적당한 품질과 실행시간 모두를 만족시키고 있다.

본 논문에서는 이 알고리즘을 Java에 기반하여 구현하였다. 사용자에게 GUI를 통해 트리 구조를 눈으로 볼 수 있도록 구현하였으며 각 노드의 경로 형태로 정리된 변경사항 목록을 한다. 실 사용 예는 <그림 3>과 같다.

5. 결론 및 향후 과제

본 논문에서 XML로 표현된 다이어그램에 대해 변경 내용을 검출하는 방법을 제안하였다. 본 논문에서 제한한 방식은 *moved* 연산일 경우 동일한 부모 노드 내에서의 *moved*만을 검출하는 것으로 제한하였다. 이러한 제한은 소프트웨어 다이어그램처럼 부모가 다른 노드 간의 이동 연산이 의미가 없는 경우 타당한 제한이다.

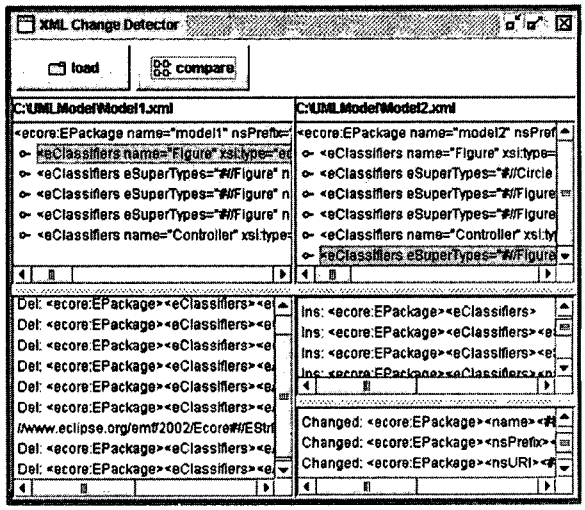

그림 3. XML 변경 검출기의 실행예

또한 본 논문의 알고리즘은 실용적인 실행 시간 우선적으로 고려했기 때문에, *moved* 검출에 있어 최적해를 찾아내지 못하는 경우가 있다. 따라서 본 논문에서 제안한 방법으로 검출한 변경사항이 최적의 변경사항과 얼마만큼의 차이가 있는 지 실험을 통해 평가해 볼 필요가 있다.

그밖에 검출된 변경 사항을 작은 공간에 효율적으로 저장하는 방법과 저장된 변경 사항을 바탕으로 특정 버전을 빠르게 찾을 수 있는 방법에 대한 연구 역시 본 논문의 결과물을 보다 향상시키기 위해 필요하다.

6. 참고 문헌

1. Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler. Extensible Markup Language. Recommendation http://www.w3.org/XML

2. Junichi Suzuki, Yoshikazu Yamamoto. Making UML models exchangeable over the internet with XML: UXF approach. *Proceedings of the First International Conference on the Unified Modeling Language* 1998; 65-74.

3. Object Management Group. *OMG-XML Metadata Interchange (XMI) Specification v1.2.* January 2002.

4. Eclipse Project. http://www.alphaworks.ibm.com/eclipse.

5. G. Cobena, S. Abiteboul, A. Marian. Detecting changes in XML documents. *Proceedings of the 18th International Conference on Data Engineering.* 2002.

6. S. Chawathe, H. Garcia-Molina. Meaningful change detection in structured data. *In SIGMOD* 1997; 26(2):26-37.

7. F. Vokolos and P. Frankl, Pythia: a regression test selection tool based on text differencing, *Int' l conference on reliability, Quality and Safety of Software Intensive Systems*, May 1997.