소프트웨어 객체의 버전 관리를 위한 연산 히스토리 모델

노정규†

요 약

소프트웨어 문서는 논리적인 객체와 객체간의 관계로 이루어진 구조를 가지고 있으며 그 구조가 빈번하게 변경된다. 기존의 소프트웨어 객체 버전 관리에서는 한 객체의 변경이 불필요하게 다른 객체로 전파되는 문제를 가지고 있다. 본 논문에서는 소프트웨어 편집 과정에서 객체에 적용된 연산의 히스토리를 이용하여 소프트웨어 객체의 버전을 효율적으로 관리할 수 있는 모델을 제안하였다. 소프트웨어 객체 편집 과정에서 객체에 적용된 연산은 연산 히스토리에 기록되고 버전 저장과 검색에 이용된다. 객체의 버전은 연산 히스토리를 이용한 델타에 의해 저장되고 검색되므로 체크인 과정에서 델타 추출을 위한 비교 과정이 필요 없다. 또, 이 모델은 객체의 생성, 삭제, 변경 연산뿐만 아니라 객체의 이동 연산을 지원함으로써 효율적으로 객체 구조의 변경을 관리할 수 있다.

An Operation History Model for Version Management of Software Objects

Jungkyu Rho†

ABSTRACT

Software documents consist of a number of objects and relationships between them, and structure of documents can be changed frequently. In the existing software version management models, changes in one object may be propagated to other objects unnecessarily. In this paper, we propose an efficient version management model for software objects based on history of operations applied to software objects. Operations applied to objects are recorded in the operation history, and those are used to retrieve versions of a document. Because versions of objects are stored and retrieved using the operation delta, it is not required to compare versions of a document to extract delta during check-in process. In addition, it can manage changes of structure of objects efficiently because it supports not only object creation, deletion, and update operation but also object move operation.

Keywords : Version management, Fine-grained object management

1. 서 론

소프트웨어 개발 과정에서 생성되는 문서는 논리적인 객체들과 그 객체들 간의 복잡한 관계로 구성된다. 소프트웨어 문서 개발 과정에서 여러 단계의 수정이 필요하고 여러 개의 버전이 생성된다. 그런데 소프트웨어 문서를 파일단위로 관리할 경우 소프트웨어 문서 내부에 존재하는 논리적인 구조를 표현할 수 없기 때문에 소프트웨어 문서를 미세 단위(fine-grained) 객체로 관리

† 정 회 원: 서경대학교 컴퓨터과학과 전임강사
논문접수: 2003년 12월 14일, 심사완료: 2004년 1월 15일

하는 모델이 제안되었다. 기존의 미세 단위 소프트웨어 객체 저장을 위한 모델로는 Orm[1], HiP[2] 등이 있으며 객체지향데이터베이스 시스템인 O_2[3]도 고려될 수 있다.

소프트웨어 문서를 미세 단위 객체로 관리하는 것은 문서 내에 포함된 논리적인 객체를 직접 표현할 수 있으므로 객체의 구조와 객체간의 관계를 명시적으로 관리할 수 있다는 장점이 있다. 반면 수많은 객체의 버전을 관리하여야 하므로 효율적인 객체 관리 방법이 필요하고 객체의 버전이 포괄적(generic) 형상을 지원하지 않을 경우 객체간의 부모 관계, 또는 참조 관계를 통하여 한 객체의 변경이 다른 객체들로 불필요하게 전파될 수 있다는 문제를 가지고 있다.

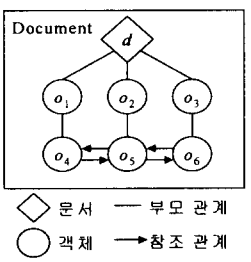
본 논문에서는 미세 단위 소프트웨어 문서의 편집 과정에서 적용되는 연산(operation)을 이용하여 객체의 버전을 저장하고 검색할 수 있는 모델을 제시하였다. 이 모델에서는 편집 과정에서 적용된 연산을 이용하여 객체의 버전을 저장하는데, 자식 객체의 생성이나 삭제가 일어나도 그 결과가 간단한 연산으로 기록되므로 객체의 구조가 자주 변하는 경우에 적합하다. 객체에 적용될 수 있는 연산은 객체의 생성(create) 및 삭제(delete)와 객체 속성의 변경(update)뿐만 아니라 구조적인 소프트웨어 문서의 편집 시 빈번히 사용되는 잘라내어 붙여넣기(cut and paste) 연산도 포함한다. 이 모델은 객체의 변경을 불필요하게 전파하지 않으면서도 객체의 버전을 효율적으로 관리할 수 있다는 특징이 있다.

본 논문은 다음과 같이 구성된다. 2절에서는 소프트웨어 객체 모델과 연산 히스토리를 소개하고 3절에서는 연산 히스토리를 이용한 버전 저장 및 검색 방법을 설명하고 기존의 소프트웨어 객체 버전 모델과의 비교를 통하여 본 모델을 평가한다. 4절에서는 본 모델의 구현에 대하여 설명하고 5절에서 결론을 맺는다.

2. 소프트웨어 객체 모델

2.1. 소프트웨어 객체 모델

본 연구에서 사용하는 객체 모델은 문서(document), 객체(object)로 구성되고 객체 간에는 여러 가지 관계(relationship)가 존재할 수 있다. 예를 들어 프로그램 실행문(statement), 루프(loop), 지역 변수, 다이어그램의 노드(node) 및 에지(edge) 등은 기본 객체가 될 수 있다. 여러 개의 클래스와 함수를 포함하고 있는 소스 코드 파일이나 다이어그램은 문서의 예이다. 관계는 객체간의 연관 관계 또는 의존 관계를 나타낸다. 예를 들어 다이어그램의 어떤 노드와 에지가 연결되어 있을 경우 노드 객체와 에지 객체 간에는 관계가 존재한다. <그림 1>은 본 연구에서 사용하는 미세 단위 객체로 표현한 소프트웨어 문서이다. 즉 문서는 여러 개의 객체로 구성되어 있으며 각 객체는 자식 객체를 가질 수 있고 객체 간에는 참조 관계가 존재할 수 있다.



<그림 1> 소프트웨어 문서의 객체 구조

본 연구에서 사용하는 객체 모델에서는 객체의 존재성과 별도로 가시성(visibility)을 정의하여 가시성이 있는 객체와 관계만 문서에 포함되는 것으로 한다. 즉 부모 객체로부터 도달 가능한 객체와 관계는 문서에 포함되고, 실제로 존재하지만 도달 불가능한 객체나 관계는 문서의 내용에 포함하지 않는다. 이를 위하여 항상 가시성이 참인 가상 루트 객체 *vroot*가 존재한다고 가정하며 다음과 같이 객체와 관계의 가시성을 정의한다.

정의 1 O가 객체의 집합일 때 부모 관계 P는 $O \times O$의 부분 집합인 O에서의 이항관계이고 $<x, y>$가 P의 원소이면 객체의 계층구조에서 객체 y가 객체 x의 부모임을 나타낸다.

정의 2 조상 관계 P^*는 $O \times O$의 부분 집합인 O에서의 유사 순서이고 객체의 계층구조에서 조상 관계를 나타낸다.

$$P^* = \{ \langle x, y\rangle \mid \langle x, y\rangle \in P \vee \exists z (\langle x, z\rangle \in P^* \wedge \langle z, y\rangle \in P^*) \}$$

정의 3 O가 객체의 집합일 때 참조 관계 R은 $O \times O$의 부분 집합인 O에서의 이항관계이고 객체 x가 객체 y를 참조할 때 $\langle x, y\rangle$는 R의 원소이다.

정의 4 객체 x의 가시성 $V(x)$는 $V(y)$가 참이고 $\langle x, y\rangle$가 P의 원소인 객체 y가 존재할 때 참이다. 그리고 그 역도 성립한다. 관계 $\langle x, y\rangle$의 가시성 $V(\langle x, y\rangle)$는 $V(x)$와 $V(y)$가 참이고 $\langle x, y\rangle$가 R의 원소일 때 참이다. 그리고 그 역도 성립한다. 즉 객체의 가시성과 관계의 가시성은 다음과 같이 정의된다.

$$V(x) \Leftrightarrow \exists y (V(y) \wedge \langle x, y\rangle \in P)$$
$$V(\langle x, y\rangle) \Leftrightarrow [V(x) \wedge V(y) \wedge \langle x, y\rangle \in R]$$

2.2. 연산 히스토리

본 연구에서는 소프트웨어 객체가 주로 다이어그램 편집기나 소스 코드 편집기 등을 이용하여 문서를 수정된다는 특징과 수정 작업은 객체의 생성, 삭제, 변경 등 연산을 통하여 이루어진다는 특징을 활용한다. 즉 문서의 수정을 위하여 편집기에서 적용된 연산을 연산 히스토리에 기록하여 이를 편집기에서의 되돌리기(undo) 기능을 위하여 사용할 뿐만 아니라 문서의 버전을 저장하고 검색하는데 사용한다.

편집기에서 객체를 변경할 경우 객체의 변경과 동시에 적용된 연산이 일시적 연산 히스토리에 기록된다. 일시적 연산 히스토리는 편집한 내용을 저장할 때 편집한 내용을 영구적 객체에 반영하는데 사용된다. 영구적 객체를 수정하기 위하여 영구적 객체에 적용된 연산은 영구적 연산 히스토리에 기록되고 이는 버전 검색을 위하여 사용된다.

연산 히스토리는 객체 연산과 관계 연산으로 구성된다. 객체 연산은 객체 y의 자식 객체로서 속성 v를 가지는 객체 x를 생성하는 연산 $c(y, x, v)$, 객체 y의 자식 객체 x를 삭제하는 연산 $d(y, x)$, 객체 x의 속성 변경 연산 $u(x, v)$, y가 부모인 객체 x를 z가 새로운 부모가 되도록 이동하는 연산 $m(y, z, x)$이 있다. 관계 연산은 관계 $\langle x, y\rangle$의 생성 연산 $cr(x, y)$의 관계 $\langle x, y\rangle$의 삭제 연산 $dr(x, y)$가 있다.

각 연산의 선 조건 과 후 조건은 <표 1>과 같다. 즉 선 조건이 만족되어야 연산을 수행할 수 있고 연산을 수행한 후에는 후 조건이 만족된다는 의미이다.

<표 1> 연산의 선 조건과 후 조건

선 조건	연산	후 조건
$V(y) \wedge \neg E(x)$	$c(y, x, v)$	$V(y) \wedge \langle x, y\rangle \in P \wedge x.value = v$
$V(y) \wedge \langle x, y\rangle \in P$	$d(y, x)$	$V(y) \wedge \langle x, y\rangle \notin P$
$V(y) \wedge V(z) \wedge \langle x, y\rangle \in P$	$m(y, z, x)$	$V(y) \wedge \langle x, y\rangle \notin P \wedge \langle V(z) \wedge \langle x, z\rangle \in P$
$V(x)$	$u(x, v)$	$V(x) \wedge x.value = v$
$V(x) \wedge V(y) \wedge \langle x, y\rangle \notin R$	$cr(x, y)$	$V(x) \wedge V(y) \wedge \langle x, y\rangle \in R$
$V(x) \wedge V(y) \wedge \langle x, y\rangle \in R$	$dr(x, y)$	$V(x) \wedge V(y) \wedge \langle x, y\rangle \notin R$

2.3. 연산 흡수

편집 과정에서 적용된 연산은 일시적 연산 히스토리에 기록되는데 일시적 연산 히스토리에는 객체 생성 후 삭제, 객체 생성 후 변경, 객체 속성의 여러 번 변경 등 중복되거나 필요 없는 연산을 포함한 경우가 있다. 이 연산들은 편집과정에서의 되돌리기를 위해서는 필요하지만 문서를 저장하기 전에 제거되어야 문서를 좀 더 효율적으로 저장할 수 있다. 따라서 본 연구에서는 다음과 같은 연산 흡수 규칙(operation subsumption rule)과 최소화된 연산 히스토리(minimized operation history)를 정의하여 문서를 저장하기 전에 불필요한 연산을 제거할 수 있도록 하였다.

정의 5 $OP=< o_1,...,o_n >$은 일시적 연산 히스토리이고 일시적 연산 히스토리에 포함된 연산 $o_k(1 \le k \le n)$는 객체의 생성, 삭제, 변경, 이동 연산 또는 관계의 생성, 삭제 연산일 때, 연산 흡수 규칙 S는 다음과 같은 OP에서의 유사 순서(quasi order)이다

$S = \{ <o_i, o_j> \mid [<x, z> \in P^* \wedge o_j=d(w, z)$
$\wedge (o_i=c(y, x, v) \vee o_i=d(u, x) \vee o_i=u(x, v)$
$\vee o_i=cr(x, y) \vee o_i=cr(y, x) \vee o_i=dr(x, y)$
$\vee o_i=dr(y, x))]$ (1)
$\vee [o_j=d(z, x) \wedge (o_i=c(y, x, v)$
$\vee o_i=u(x, v) \vee o_i=cr(x, y) \vee o_i=cr(y, x)$
$\vee o_i=dr(x, y) \vee o_i=dr(y, x))]$ (2)
$\vee (o_j=c(y, x, v') \wedge o_i=u(x, v))$ (3)
$\vee (o_j=u(x, v') \wedge o_i=u(x, v) \wedge i < j)$ (4)
$\vee (o_j=dr(x, y) \wedge o_i=cr(x, y)) \}$ (5)

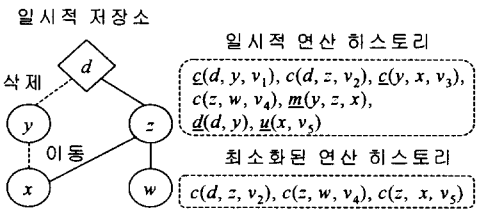
흡수 규칙 (1)은 조상 객체의 삭제 연산에 의해서 자손 객체에 대한 연산이 흡수됨을 나타낸다. 흡수 규칙 (2)는 객체의 삭제 연산에 의하여 객체에 대한 연산이 흡수됨을 나타낸다. 흡수 규칙 (3)은 생성된 후 변경된 객체에 대한 생성 연산과 변경 연산은 새로운 생성 연산으로 통합될 수 있음을 나타낸다. 흡수 규칙 (4)는 여러 개의 속성 변경 연산은 하나의 속성 변경 연산으로 통합될 수 있음을 나타낸다. 흡수 규칙 (5)는 생성되었다가 삭제된 관계의 생성 연산과 삭제 연산이 모두 제거될 수 있음을 나타낸다.

정의 6 일시적 연산 히스토리에 다음과 같은 연산 치환 규칙(operation substitution rule)을 적용한 결과를 최소화된 연산 히스토리라고 한다.

$<o_i, o_j> \in S$ 일 때

(1) $o_i = c(y, x, v)$이고 $o_j = d(y, x)$이면 o_i와 o_j, 그리고 $<p, o_j> \in S$인 모든 p를 $\varnothing$으로 치환한다.
(2) $o_i = u(x, v')$이고 $o_j = c(y, x, v)$이면 o_i를 $\varnothing$으로 치환하고 o_j를 $c(y, x, v')$로 치환한다.
(3) $o_i = u(x, v)$이고 $o_j = u(x, v')$이면 o_i를 $\varnothing$으로 치환하고 o_j를 $u(x, v')$으로 치환한다.
(4) $o_i = c(y, x, v)$이고 $o_j = m(y, z, x)$이면 o_i를 $\varnothing$으로 치환하고 o_j를 $c(z, x, v)$로 치환한다.
(5) $o_i = cr(x, y)$이고 $o_j = dr(x, y)$이면 o_i와 o_j를 $\varnothing$으로 치환한다.
(6) 이외의 경우 o_i를 $\varnothing$으로 치환한다.

<그림 2>는 문서를 편집하는 과정에서 생성된 일시적 저장소와 일시적 연산 히스토리, 중복되거나 필요 없는 연산을 제거한 최소화된 연산 히스토리를 보여준다. 문서 d를 편집하는 과정은 일시적 연산 히스토리에 기록되는데 처음 y 객체를 생성하고 z 객체를 생성하였다. 그리고 x 객체를 y 객체의 자식 객체로, w 객체를 z 객체의 자식 객체로 생성하였다. 그 후 x 객체를 z 객체 자식으로 이동하고 y 객체를 삭제하였으며 x 객체의 속성을 변경하였다. 일시적 연산 히스토리에서 밑줄이 있는 연산은 연산 치환 규칙에 의해 제거되거나 치환되는 연산을 표시한 것이다. 그 결과 최소화된 연산 히스토리를 얻을 수 있고 이는 문서의 저장에 사용된다.



<그림 2> 일시적 저장소와 연산 히스토리

3. 버전 저장 및 검색

3.1. 연산 히스토리를 이용한 버전 저장

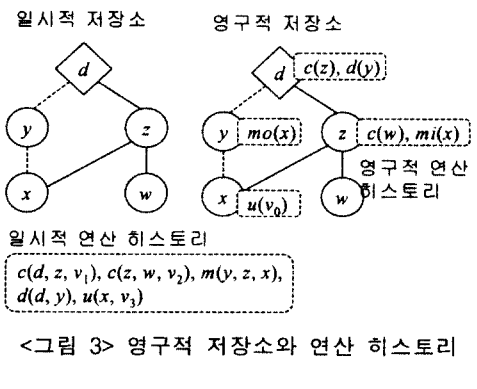
최소화된 일시적 연산 히스토리에 기록된 내용은 영구적 저장소에 문서의 버전을 저장을 위해서 사용된다. 영구적 저장소를 수정하기 위하여 적용된 영구적 연산은 영구적 연산 히스토리에 기록된다. 영구적 연산 히스토리는 문서의 버전 검색을 위한 델타로 사용된다.

영구적 연산 히스토리는 버전 검색을 위한 히스토리 탐색 시간을 줄이기 위하여 계층적 구조

를 가진다. 즉 $u(v)$, $cr(y)$, $dr(y)$는 연산이 적용된 객체에 저장되고 $c(x)$와 $d(x)$는 x의 부모 객체에, 그리고 이동 연산은 이전 부모와 새로운 부모 객체에 각각 기록된다.

객체의 속성을 변경하는 일시적 연산 $u(x, v)$가 영구적 객체에 적용될 경우 객체 x의 속성을 v로 변경하고 영구적 연산 히스토리에는 이전의 속성 v'를 가지는 역 연산 $u(v')$가 객체 x에 기록된다. 객체를 이동하는 일시적 연산 $m(y, z, x)$이 적용될 경우 이전의 부모 객체 y에는 x가 이동되어 나갔음을 나타내는 $mo(x)$ 연산이, 새로운 부모 객체 z에는 객체가 이동되어 들어왔음을 나타내는 $mi(x)$이 각각 영구적 연산 히스토리에 기록된다.

<그림 3>은 일시적 저장소에서 편집 과정을 통하여 생성된 일시적 연산 히스토리와 그를 저장한 결과인 영구적 저장소와 영구적 히스토리를 보여준다. 객체 y와 x는 기존에 존재하였으며 편집 과정에서 객체 z와 w를 생성하고 객체 x를 새로운 부모 z 아래로 이동한다. 그리고 마지막으로 객체 x의 속성을 변경한다. 편집 결과를 저장하면 영구적 저장소에 객체가 생성되고 변경되며 동시에 객체에 적용된 연산이 영구적 연산 히스토리에 기록된다.



<그림 3> 영구적 저장소와 연산 히스토리

3.2. 연산 히스토리를 이용한 버전 검색

영구적 저장소는 항상 최근 버전의 상태를 유지하고 있으며 버전 별로 구분된 영구적 연산히스토리는 이전 버전을 검색하기 위한 델타로 사용된다. 이전 버전은 객체와 관계의 가시성을 이용하여 검색된다. 다음 정의 7, 8은 i 번째 버전에서의 객체와 관계의 가시성을 정의한다. 이 가시성 정의에 따라 i 번째 버전에서 가시성이 참인 객체는 i 번째 버전에 포함되고 가시성이 거짓인 객체는 i 번째 버전에 포함되지 않는다. 관계의 경우에도 객체와 유사한 가시성 정의를 통하여 버전에서의 포함 여부를 결정한다.

정의 7 OP_{ky}가 k 번째 버전의 객체 y에 속한 영구적 연산 히스토리일 때, L_y를 객체 y의 최근 버전에서의 자식 객체의 집합, Ciy를 객체 y의 i+1번째 버전부터 최근 버전 사이에 생성되거나 이동되어 온 자식 객체의 집합, Diy를 객체 y의 i+1번째 버전부터 최근 버전 사이에 삭제되거나 이동되어 나간 자식 객체의 집합이라고 하자. $c(x)$, $d(x)$, $mo(x)$, $mi(x)$가 각각 객체 x의 생성, 삭제, 이동하여 나감, 이동하여 들어옴 연산일 때 각 집합은 다음과 같이 정의된다. 또 i번째 버전에서의 객체 x의 가시성 $Vi(x)$는 l이 최근 버전의 번호이고 $1 \le i < l$ 일 때 다음과 같이 정의된다.

$L_y=\{x \mid \langle x, y\rangle \in P\}$

$$C_{iy}=\{x \mid c(x)\in \bigcup_{k=i+1}^{l} OP_{ky} \vee mi(x)\in \bigcup_{k=i+1}^{l} OP_{ky}\}$$

$$D_{iy}=\{x \mid d(x)\in \bigcup_{k=i+1}^{l} OP_{ky} \vee mo(x)\in \bigcup_{k=i+1}^{l} OP_{ky}\}$$

$$V_i(x) \Leftrightarrow \exists y\ [\ V_i(y) \wedge x\in((\ L_y \cup D_{iy}\ ) - C_{iy})\ ]$$

즉 i번째 버전에 속한 객체를 구하기 위해서 최근 버전에 포함된 객체 중에서 i번째 버전 이후에 삭제되거나 이동되어 나간 객체를 포함시키고 i번째 버전 이후에 생성되거나 이동되어 온 객체를 제외하는 것이다.

정의 8 LR_x는 객체 x가 시작점인 관계의 끝점 객체의 집합이라고 하자. CR_{ix}는 객체 x가 시작점인 관계 중에서 i+1번째 버전부터 최근 버전 사이에 생성된 관계의 끝점 객체의 집합, DR_{ix}는 객체 x가 시작점인 관계 중에서 i+1번째 버전부터 최근 버전 사이에 삭제된 관계의 끝점 객체의 집합이다. 각각은 $cr(y)$가 관계 $\langle x, y\rangle$의 생성

연산이고 $dr(y)$가 관계 $<x, y>$의 삭제 연산일 때 다음과 같이 정의된다. 또, i번째 버전에서의 관계 $<x, y>$의 가시성 $Vi(<x, y>)$ 는 l이 최근 버전의 번호이고 $i<l$일 때 다음과 같이 정의된다.

$$LR_x = \{ y \mid <x, y> \in R \}$$

$$CR_{ix} = \{ y \mid cr(y) \in \bigcup_{k=i+1}^{l} OP_{kx} \}$$

$$DR_{ix} = \{ y \mid dr(y) \in \bigcup_{k=i+1}^{l} OP_{kx} \}$$

$$V_i(<x, y>) \Leftrightarrow [\ V_i(x) \wedge V_i(y) \wedge y \in ((LR_x \cup DR_{ix}) - CR_{ix})\ ]$$

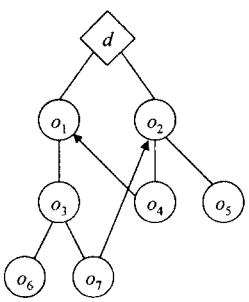
다음 함수 *visible_children*은 i번째 버전에서 객체 x의 가시 자식 객체를 구하는 알고리즘이고 함수 *visible_relation*은 i번째 버전에서 객체 x를 시작점으로 하는 가시 관계를 구하는 알고리즘이다.

```
알고리즘 visible_children
입력 : 객체와문서의 버전 번호
출력 : 버전 i에서 객체 x의 가시 자식 객체의 집합
function visible_children( x : object;
        i : revision number ) : set of objects;
begin
        children := Oix := ( Lx ∪ Dix ) - Cix;
        x.value:= value(x, i );
        for each y ∈ Oix
        children := children ∪ visible_children(y, i);
        return(children);
end { visible_children }
```

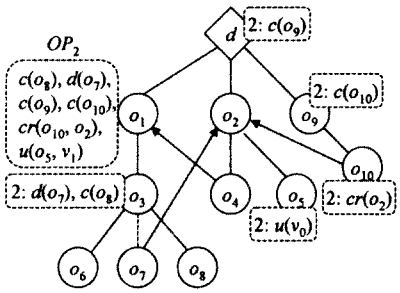
버전 i에 속한 객체의 수를 N, 객체에 적용된 연산의 수의 평균을 P라고 하자. 알고리즘 *visible_children*은 버전 i 에 속한 모든 객체를 방문하고 그 외에는 방문하지 않는다. 그리고 *visible_children*에서 가시 객체 O_{ix}를 구하는 복잡도와 알고리즘 *value*의 복잡도는 객체 x에 적용된 생성, 삭제, 변경 연산의 수에 비례한다. 알고리즘 *visible_relation*도 버전 i에 속한 모든 객체를 방문하고 *visible_children*에서 가시 관계 R_{ix}를 구하는 복잡도는 관계의 생성, 삭제 연산의 수에 비례한다. 따라서 버전 i를 검색하는 알고리즘의 복잡도는 객체를 검색하는 알고리즘과 관계를 검색하는 알고리즘의 수행 시간을 합쳐서 O(NP)가 된다.

```
알고리즘 visible_relation
입력 : 객체(x)와 검색하려는 버전(i)
출력 : 버전 i에서 객체 x의 가시 관계의 집합
function visible_relation( x : object;
        i : revision number ) : set of relations;
begin
        relation : = Rix := ( LRx ∪ DRix ) - CRix;
        for each y ∈ Rix
                relation := relation ∪ { < x, y > };
        for each y Oix relation :=
                relation ∪ visible_relation(y, i);
        return(relation);
end{ visible_relation }
```

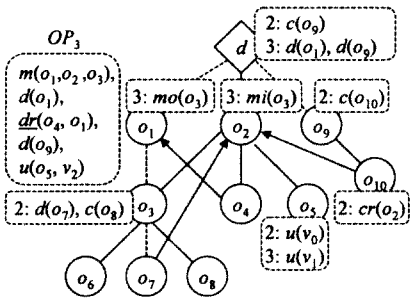
<그림 4>는 문서의 버전 1이다. 버전 1에는 영구적 연산 히스토리가 기록되지 않는다. <그림 5>은 버전 2가 생성된 결과이다. 편집 과정에서 생성된 일시적 연산 히스토리는 OP_2이며 해당되는 객체별로 영구적 연산 히스토리가 기록되었다. o_5는 속성이 변경되었는데 연산 히스토리에는 예전 속성이 기록된다. <그림 6>은 버전 3이 생성된 결과이다. 일시적 연산 히스토리는 OP_3이며 o_1의 자식 객체 o_3가 o_2의 자식 객체로 이동하는 등의 몇 가지 변경이 영구적 연산 히스토리에 기록된다. 버전 3이 생성된 후 *visible_children*과 *visible_relation* 알고리즘을 이용하여 이전 버전을 검색할 수 있다.



<그림 4> 버전 1의 영구적 객체



<그림 5> 버전 2에 적용된 연산 히스토리



<그림 6> 버전 3에 적용된 연산 히스토리

3.3. 분석

이 절에서는 각 모델과 본 연구에서 제안한 OP 모델에서 객체의 변경 연산이 적용되었을 때 필요한 저장 공간과 이전 버전을 검색하는데 필요한 시간을 비교한다. Orm 모델과 HiP 모델은 명시적인(explicit) 참조 관계를 지원하지 않는다. 그러나 비교의 편의를 위해서 명시적인 참조를 사용할 수 있도록 모델을 수정하여 생각한다.

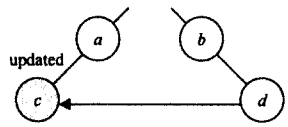
<그림 7>은 전형적인 객체 변경의 예를 보여준다. 변경된 객체 c는 객체 a의 자식 객체이고 객체 d는 c에 참조를 가지고 있다.

<그림 8>는 객체가 변경된 후의 Orm 저장소의 상태를 보여준다. Orm 모델에서는 구속적(bound) 형상만을 지원한다. 따라서 c의 변경으로 c의 객체 식별자가 변경되고 c의 부모 객체 a도 c에 대한 참조를 변경하여야 한다. 뿐만 아니라 c를 참조하는 객체 d와 그 부모객체 b도 차례로 변경되어야 한다. 이와 같은 버전 급증 현상은 부모 관계나 종속성 관계를 따라 문서 내의 모든 객체로 전파된다.

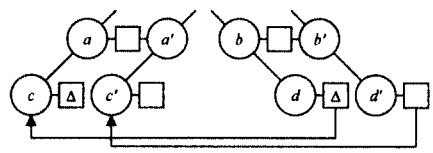
<그림 9>는 객체 변경 후의 HiP 저장소의 상태를 보여준다. HiP 모델에서는 구속적 형상의 단점인 부모 관계를 따라 버전이 전파되는 것을 보완하였다. 하지만 참조 관계를 따라서는 버전이 전파된다. 즉 객체 d의 참조가 수정되고 d의 새로운 버전이 생성되어야 한다.

<그림 10>은 객체 변경 후의 O_2 저장소의 상태를 보여준다. O_2 모델은 포괄적 객체를 지원하므로 객체의 변경에 따라 부모 관계나 종속성 관계를 따라 버전이 전파되지는 않는다. 그러나 O_2에서는 객체의 속성에 대한 델타를 제공하지 않으므로 전체 객체가 복사된다.

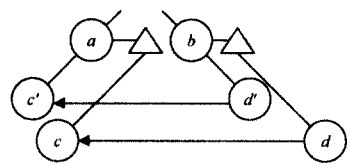
<그림 11>은 객체 변경 후의 OP 저장소의 상태를 보여준다. 객체는 복사되지 않고 직접 변경되며 변경 연산이 연산 히스토리에 기록된다. O_2와 마찬가지로 포괄적 객체를 지원하고 변경이 전파되지 않는다.



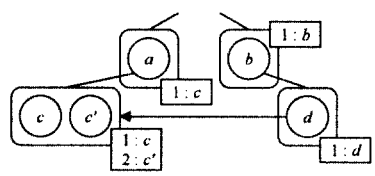
<그림 7> 객체의 변경



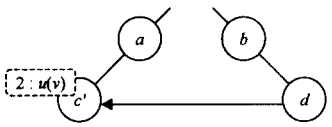
<그림 8> Orm에서 객체 변경 결과



<그림 9> HiP에서의 객체 변경 결과



<그림 10> O_2에서의 객체 변경 결과



<그림 11> *OP*에서의 객체 변경 결과

각 모델의 이전 버전 검색 시간에 대한 비교를 위해서 P_U를 객체에 적용된 변경 연산 수의 평균, P를 객체에 적용된 연산(변경, 자식 객체 생성 및 삭제, 이동) 수의 평균, N을 검색하려는 버전의 객체 수라고 하자.

Orm, HiP, OP 모델은 모두 최근 버전 상태를 유지한다. 그러므로 최근 버전 검색 시간은 모든 객체를 방문하는 시간을 고려하면 O(N)이다. 반면 O_2는 최근 버전을 검색하기 위해서는 최근 버전에 속하는 데이터베이스 버전을 구하여야 하고 이는 이전 버전의 검색 시간과 동일하다.

Orm 모델은 구조와 속성을 따로 관리하고 이전 버전의 구조를 유지한다. 따라서 이전 버전을 검색하기 위해서는 객체의 속성만 구하면 되고 그 시간은 검색하려는 버전에서 최근 버전까지 객체에 적용된 연산의 수에 비례한다. 따라서 모든 객체에 대하여 이전 버전의 상태를 구하는 시간은 P_U가 객체에 적용된 변경 연산 수의 평균이라고 할 때 $O(NP_U)$이다.

HiP와 OP의 버전 검색 알고리즘은 검색하려는 버전에 속한 객체만 방문한다. 그리고 객체의 이전 버전 상태를 검색하는 시간은 객체에 적용된 연산의 수에 비례한다. 따라서 모든 객체의 이전 버전의 상태를 구하는 시간은 P를 객체에 적용된 연산 수의 평균이라고 할 때 O(NP)이다.

O_2는 버전 스탬프를 이용하여 객체의 버전을 검색하므로 객체별로 스탬프 탐색 시간은 객체에 적용된 연산의 수에 비례하고 전체 시간은 O(NP)가 된다. <표 2>는 각 모델의 버전 검색 시간을 비교한 것이다. 각 항목은 다음과 같다.

<표 2> 버전 검색 시간

	Orm	HiP	O_2	OP
최근 버전	O(N)	O(N)	O(NP)	O(N)
이전 버전	$O(NP_U)$	O(NP)	O(NP)	O(NP)

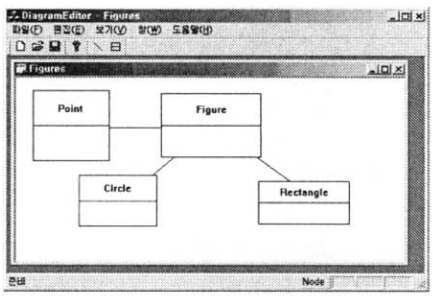
4. 구 현

본 연구에서 제시한 연산 히스토리를 이용한 객체 버전 관리 모델을 기반으로 하여 미세 단위 객체로 이루어진 소프트웨어 다이어그램 버전 관리 시스템을 구현하였다. 소프트웨어 다이어그램은 본 연구에서 제시한 객체 버전 관리에 적합한 예로서 객체들이 계층 구조를 이루고 있고 객체간에 참조 관계가 존재하며 객체의 구조가 빈번히 변경된다는 특징이 있다. 뿐만 아니라 편집 과정에서 이동 연산도 비교적 자주 일어나는 경우이다.

본 연구에서는 소프트웨어 다이어그램을 노드(node), 그래픽 노드(graphic node), 에지(edge), 집합 속성으로 구성하였다. 다이어그램은 문서로 표현되고 노드와 그래픽 노드, 에지, 집합 속성은 객체로 표현된다. 노드와 에지가 연결되어 있다는 관계는 노드 객체와 에지 객체간의 참조관계를 이용하여 표현된다. 다이어그램은 노드와 에지를 자식 객체로 가지고, 노드는 그래픽 노드와 집합 속성을 자식 객체로 가진다. 그래픽 노드는 노드를 다이어그램 상에 나타낸 것이다. 그래픽 노드는 위치, 크기, 색상과 같은 그래픽 속성을 가진다. 그래픽 노드의 존재는 부모 노드의 존재에 종속적이다. 두 그래픽 노드를 연결하는 에지는 속성을 가질 수 있다. 에지의 존재는 연결된 그래픽 노드의 존재에 종속적이다. 집합 속성은 노드의 특성을 나타내기 위하여 사용된다. 예를 들어 클래스 다이어그램에서 클래스의 속성과 오퍼레이션은 집합 속성이다. 집합 속성의 존재는 부모 노드의 존재에 종속적이다. 다이어그램은 노드, 그래픽 노드, 에지, 집합 속성 객체로 구성된 복합 객체이다.

소프트웨어 다이어그램 버전 관리 시스템의 개발 환경은 Windows 상의 Microsoft Visual C++ 6.0이다. 일시적 객체 관리는 C++ Standard Template Library(STL)를 사용하였으며 영구적 객체 관리는 Object Design사의ObjectStore PSE Pro for C++ 5.1[8] 을 사용하였다. <그림 12>는 다이어그램 버전 관리 시스템과 연동되는 다이어그램 편집기의 실행화면이다. 이 편집기에서 다

이어그램을 편집하는 과정은 일시적 연산히스토리에 기록되고 버전 저장 시 영구적 연산 히스토리로 변환되어 저장된다.



<그림 12> 연산 히스토리를 지원하는 편집기

5. 결 론

본 연구에서 제시한 소프트웨어 문서를 위한 버전 관리 모델은 미세 단위 버전 관리를 지원하는 기존의 방법에 비하여 구조적인 소프트웨어 문서의 버전을 효율적으로 저장하고 검색할 수 있다는 특징이 있다. 포괄적 객체를 지원함으로써 부모 관계나 참조 관계를 통하여 버전이 불필요하게 전파되는 것을 막을 수 있으며 효율적인 버전 검색 시간을 제공한다. 그리고 소프트웨어 문서의 편집기에서 적용되는 연산을 기록하여 버전의 저장에 사용함으로써 버전 저장 시 델타 추출을 위한 비교 시간이 필요 없으며 소프트웨어 문서가 복잡한 구조를 가질수록 더 유리한 측면이 있다. 그 외에도 기존에 고려하던 객체의 생성, 삭제, 변경 연산뿐만 아니라 객체의 이동 연산을 지원함으로써, 객체 이동시 객체를 삭제하고 새로 생성해야 하는 절차를 간소화 하였다.

참 고 문 헌

[1] B. Magnusson and U. Asklund, "Fine Grained Version Control of Configurations in COOP/Orm," *6th Int'l Workshop on Software Configuration Management, SCM-6 Selected Papers*, Mar. 1996.

[2] E. J. Choi and Y. Kwon, "An Efficient Method for Version Control of a Tree Data Structure," *Software- Practice and Experience*, vol. 27, no. 7, July 1997.

[3] F. Bancilhon, C. Delobel, and P. Kanellakis, *Building an Object-Oriented Database System, The Story of O2*, Morgan Kaufmann, 1992.

[4] G. Heidenreich, D. Kips, and M. Minas. "A New Approach to Consistency Control in Software Engineering," *Proc. of the 18th Int'l Conf. on Software Engineering*, March 1996.

[5] J. Grundy, J. Hosking, and W. B. Mugridge, "Inconsistency Management for Multiple-View Software Development Environments," *IEEE Transactions on Software Engineering*, vol. 24, no. 11, Nov. 1998.

[6] P. Lindsay, Y. Liu and O. Traynor, "A Generic Model for Fine Grained Configuration Management Including Version Control and Traceability," *Proc. of the Australian Software Engineering Conf.*, Sep. 1997.

[7] Y. J. Lin and S. P. Reiss, "Configuration Management with Logical Structures," *Proc. of the 18th Int'l Conf. on Software Engineering*, 1996.

[8] Progress Software Corporation, *ObjectStore PSE Pro for C++ API User Guide for C++*, Release 5.1, Progress Software Corporation, July 2003.

노 정 규

1991 서울대학교 계산통계학과 (이학사)
1993 서울대학교 전산과학과(이학석사)
1999 서울대학교 전산과학과(이학박사)
1999~2002 삼성전자 통신연구소 책임연구원
2002~현재 서경대학교 컴퓨터과학과 전임강사
관심분야: 소프트웨어공학, 소프트웨어개발환경
E-Mail: jkrho@skuniv.ac.kr