연산 히스토리를 이용한 소프트웨어 일관성 관리 모델

노정규†

요 약

소프트웨어 문서는 논리적인 객체와 객체간의 관계로 이루어지며 개발 과정에서 여러 버전이 생성된다. 효율적인 소프트웨어 개발을 위해서는 소프트웨어 구성 요소에 변경이 일어났을 경우 변경의 내용과 변경이 전파되어야 할 범위를 쉽게 알 수 있어야 한다. 그러나 큰 단위 소프트웨어 객체 관리에서는 변경의 내용과 전파 범위를 알기 힘들다. 따라서 논리적인 객체와 객체간의 종속성을 관리하는 미세 단위 객체 관리가 필요하다. 본 논문에서는 소프트웨어 편집 과정에서 적용되는 연산 히스토리를 이용한 미세단위 소프트웨어 일관성 관리 모델을 제안하였다. 본 논문에서는 미세 단위 일관성 관리에 대한 정형적인 모델을 제시하였으며, 일관성은 객체간의 종속성과 객체에 적용된 연산의 종류에 의해 관리되므로 불필요한 변경 전파를 피할 수 있다.

A Software Consistency Management Model using Operation History

Jungkyu Rho†

ABSTRACT

Software documents consist of a number of logical objects and relationships between them, and a lot of versions are generated during software development. When an object is changed, it is desirable to easily identify the change and the range of change propagation for efficient software development and maintenance. However, it is difficult to identify it in a coarse-grained object management model. To solve this problem, fine-grained object management is required. In this paper, I propose a consistency management model for fine-grained software objects based on operation history applied to edit software objects. This paper presents a formal model for consistency management at the fine-grained level. Consistency between documents is managed using dependency between objects and kinds of the operations applied to the objects so that unnecessary change propagation can be avoided.

1. 서 론

소프트웨어 개발 과정에서 생성되는 문서는 여러 차례 수정이 필요하고 여러 개의 버전이 생성된다. 소프트웨어 형상은 여러 문서로 구성되어 있으므로 한 문서를 변경하게 되면 다른 문서도 그 변경 내용과 일치되도록 수정하여야 한다. 그런데 소프트웨어 객체를 파일단위, 즉 큰 단위(coarse-grained) 객체로 관리할 경우 일관성 관리가 어렵다는 문제가 있다. 반면 문서 내에 포함된 논리적인 객체, 즉 미세 단위(fine-grained)를 직접 관리할 경우 문서를 여러 개의 구성 요소 객체로 이루어진 복합 객체로 취급하므로 객체의 구조와 객체간의 관계를 명시적으로 표현할

† 정 회 원: 서경대학교 컴퓨터과학과 전임강사
논문 접수 : 2002년 9월 9일, 심사완료 : 2002년 10월 12일

수 있다.

큰 단위 객체 관리에서는 불일치의 발생 여부와 불일치를 해결하는 정보가 제공되지 않는다. Make나 Heidenreich의 연구[1]는 문서간 일관성 관리를 파일 단위의 종속성을 이용하여 기술한다. 하지만 소프트웨어 문서가 개발 과정에서 변화하는 파일간의 종속성을 다루기가 힘들고 순환적 종속성을 다룰 수 없다는 문제가 있다. 이와 같은 문제가 발생하는 이유는 원래 파일간 종속성이 논리적인 미세 단위 객체간의 종속성으로부터 유도된다는데 있다. 따라서 미세 단위 종속성에 포함된 정보를 직접 관리하는 것이 더 자연스럽다고 볼 수 있다.

미세 단위 객체의 장점은 논리적인 객체와 객체간의 종속성을 명시적으로 관리함으로써 추적성(traceability) 관리에 유리하다는 것이다. 즉 변경이 일어날 경우 변경의 내용과 전파될 범위를 쉽게 알 수 있다. 반면 미세 단위 객체의 단점은 객체 관리의 부담이 크다는 것이다.

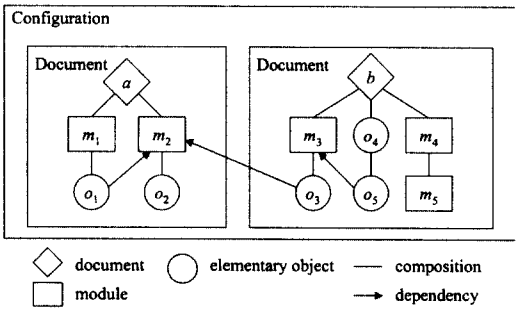
기존의 미세 단위 소프트웨어 객체의 저장 모델인 Orm[2]은 복합(composition) 관계를 제외하고는 객체간의 명시적 참조 관계를 고려하지 않는다. 그런데 명시적인 참조 관계를 유지하지 않으면 일관성 관리에서 큰 단위 객체 관리에서 제기되었던 같은 문제를 겪게 된다. 즉 불일치의 발생을 종속적인 객체에 알릴 방법이 없다.

본 논문에서는 미세 단위 소프트웨어 문서의 편집 과정에서 적용되는 연산(operation)을 이용하여 문서간의 일관성을 효율적으로 관리할 수 있는 모델을 제시하였다. 본 논문은 다음과 같이 구성된다. 2장에서는 미세 단위 소프트웨어 객체 모델과 객체 일관성 관리에 사용되는 연산 히스토리를 소개한다. 3장에서는 문서간의 종속성을 소개하고, 4장에서는 문서간 일관성을 정형적으로 정의한다. 5장에서는 새로운 버전의 생성으로 문서간 불일치가 생겼을 때 이를 해결하는 알고리즘을 설명하고 6장에서 결론을 맺는다.

2. 소프트웨어 객체 모델

2.1. 소프트웨어 객체 모델

본 논문에서 사용하는 객체 모델은 형상(configuration), 문서(document), 모듈(module), 기본 객체(elementary object)로 구성되고 객체간에는 여러 가지 관계(relation)가 존재할 수 있다. 예를 들어 프로그램 실행문(statement), 루프(loop), 지역 변수, 다이어그램의 에지(edge) 등은 기본 객체가 될 수 있다. 클래스, 함수, 전역 변수, 다이어그램의 노드(node) 등과 같이 캡슐화의 단위가 되는 객체들은 모듈로 표현될 수 있다. 여러 개의 클래스와 함수를 포함하고 있는 소스 코드 파일이나 다이어그램은 문서의 예이다. 형상은 소프트웨어 시스템을 구성하는 문서들의 집합이다. 예를 들어 하나의 시스템을 구성하는 헤더 파일들과 C 파일들은 형상이 될 수 있다. (그림 1) 은 소프트웨어 객체의 구성을 보여준다.



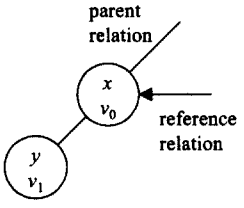
(그림 1) 소프트웨어 객체의 구성

모듈은 클래스나 함수와 같이 캡슐화의 단위가 되는 객체이다. 본 논문에서는 미세 단위 객체 관리의 복잡도를 줄이기 위하여 모듈간 종속성의 끝점은 모듈보다 큰 단위여야 한다는 제약 조건을 부여하였다. 다시 말하면 객체는 같은 모듈에 속한 객체를 제외하고는 모듈에만 의존 가능하다. 이와 같은 제약 조건을 유지하기 위해서는 클래스나 함수에서 외부에 공개되는 속성, 즉 클래스의 이름, 클래스의 공개 함수와 공개 속성 등을 모듈 자체의 속성으로 관리하여야 한다.

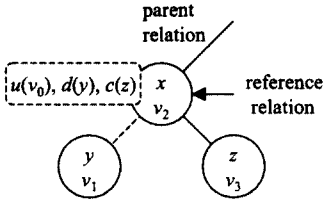
모듈 개념과 종속성에 대한 제약 조건은 소프트웨어 개발 방법의 캡슐화 개념에 부합하는 것이므로 큰 제약 조건이라고 볼 수 없으며 미세 단위 객체 관리의 복잡도를 줄일 수 있다.

2.2. 연산 히스토리

연산 히스토리 모델[3]에서는 객체를 변경할 경우 객체의 변경과 동시에 적용된 연산이 해당되는 객체의 연산 히스토리에 기록된다. 연산 히스토리는 객체 x의 생성 연산 $c(x)$, 삭제 연산 $d(x)$, 속성을 v로 변경하는 연산 $u(v)$, y로의 관계를 생성하는 연산 $cr(y)$, y로의 관계를 삭제하는 연산 $dr(y)$로 구성된다. 예를 들어 (그림 2) (가)의 객체 x의 속성을 변경하고 자식 객체 y를 삭제, 자식 객체 z를 추가한 결과는(그림 2) (나)와 같다.



(가) 처음 상태



(나) 변경 후의 상태

(그림 2) 영구적 객체의 관리

연산 히스토리는 문서의 버전 검색을 위한 델타나 버전간 일관성 관리를 위한 정보로 사용된다. 영구적 연산 히스토리는 버전 검색을 위한 히스토리 탐색 시간을 줄이기 위하여 계층적 구조를 가진다. 즉 $u(v)$, $cr(y)$, $dr(y)$는 연산이 적용된 객체에 저장되고 $c(x)$와 $d(x)$는 x의 부모 객체에 저장된다.

다음은 본 논문의 객체 모델에서 객체 간에 존재할 수 있는 관계의 정의이다.

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

정의 2 과거의 부모 관계 P'는 $O \times O$의 부분 집합인 O에서의 이항 관계이고 $<x, y>$가 P'의 원소이면 객체 y가 객체 x의 부모였고 현재는 부모가 아님을 나타낸다.

정의 3 조상 관계 P^+는 $O \times O$의 부분 집합인 O에서의 유사 순서이고 복합 객체의 계층구조에서 조상 관계를 나타낸다. 역사적 조상 관계 P^*는 $O \times O$의 부분 집합인 O에서의 유사 순서이고 현재의 조상 관계 뿐만 아니라 과거의 조상 관계도 포함한다. 즉 조상 관계와 역사적 조상 관계는 다음과 같이 정의된다.

$$P^+ = \{ <x, y> \mid <x, y> \in P \vee \exists z (<x, z> \in P^+ \wedge <z, y> \in P^+) \}$$

$$P^* = \{ <x, y> \mid <x, y> \in (P \cup P') \vee \exists z (<x, z> \in P^* \wedge <z, y> \in P^*) \}$$

정의 4 O가 객체의 집합일 때 참조 관계 R은 $O \times O$의 부분 집합인 O에서의 이항 관계이고 객체 x가 객체 y를 참조할 때 $<x, y>$는 R의 원소이다.

정의 5 O이 객체의 집합이고 M이 모듈의 집합일 때 종속 관계 D는 $O \times M$의 부분 집합이고 객체 x가 모듈 y에 종속적일 때 $<x, y>$는 D의 원소이다.

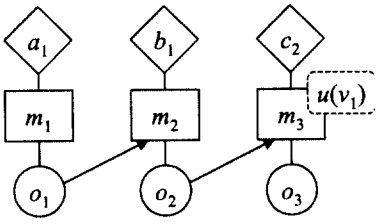
3. 문서간 종속성

본 논문에서 제시한 객체 모델에서는 객체와 모듈간의 미세 단위 종속성이 문서의 경계를 넘어서 존재할 수 있다. 만일 종속성 $<x, m>$이 존재하면 객체 x가 모듈 m에 종속적이다라고 하고 모듈 m의 변경이 객체 x의 변경을 요구할 수도 있다는 것을 의미한다. 그런데 프로그래머는 문서 단위로 편집을 하고 버전 관리 단위가 문서이기 때문에 문서간의 종속성을 관리할 필요가 있다.

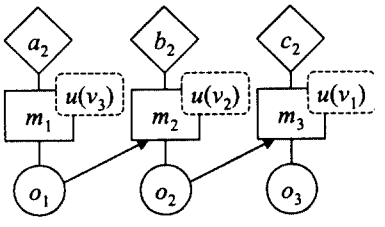
소프트웨어 문서간에는 순환적 종속성이 존재

할 수 있다. Make와 같은 규칙을 사용하는 큰 단위 객체 모델에서는 문서간의 순환적 종속성을 나타낼 수 없다. 그러나 미세 단위 객체 모델에서는 문서간 종속성이 아닌 객체와 모듈간의 종속성을 나타내기 때문에 문서간의 순환적 종속성을 표현할 수 있다.

종속성에 의해서 모듈로 변경이 전파되었을 때 그 모듈에 전파된 변경이 다른 종속성에 의해 다른 객체로 전파될 때 두 종속성을 이행적(transitive) 이라고 한다. 예를 들어 (그림 3)에서 모듈 m_3의 변경이 모듈 m_2로 전파되고 그로 인한 모듈 m_2의 변경이 모듈 m_1으로 전파될 때 두 종속성 $<o_1, m_2>$와 $<o_2, m_3>$는 이행적이다. 종속성이 이행적이지 않을 때 모듈간에도 순환적 종속성이 존재할 수 있다. 예를들어 (그림 4) Dept 클래스 모듈과 Employee 클래스 모듈간에는 순환적 종속성이 존재하지만 이행적이지 않으므로 문제가 없다.



(가) 모듈 m_3의 변경



(나) 모듈 m_2, m_1으로 전파된 변경

(그림 3) 이행적 종속성

미세 단위 객체 관리에서는 미세 단위 종속성이 문서가 버전이 변함에 따라서 추가되거나 삭제될 수 있다. 미세 단위 종속성이 변함에 따라 문서간의 종속성도 변화할 수 있기 때문에 일관성 관리 모델이 종속성의 변화를 다룰 수 있어야 한다. 본 논문에서 제시하는 일관성 관리 모델에서는 프로그래머는 미세 단위 종속성만 다루고 문서간의 종속성을 직접 다룰 필요가 없기 때문에 문서간의 종속성이 변화하는 것을 염두에 둘 필요가 없다.

불일치가 일어나면 즉시(eager) 해결하는 방법 [4]에서는 편집 과정에서 불일치가 일어나면 불일치에 대한 설명이 생성되고 종속적인 객체로 즉시 통보된다. 이와 같은 방법에서는 종속적인 객체에서 주도적인 객체로의 종속성 관계뿐만 아니라 주도적인 객체에서 종속적인 객체로의 관계도 유지하고 있어야 한다. 뿐만 아니라 반복적으로 자주 바뀌는 속성에 대한 변경 전파를 즉시 함으로써 불필요한 불일치 해결 과정을 겪을 수도 있다.

따라서 본 논문에서는 불필요한 반복적인 불일치 해결 과정을 피하기 위하여 불일치를 지연(lazy) 해결하는 방법을 채택하였다. 한 문서의 새로운 수정버전이 생성되어 잠재적 불일치가 발생하여도 종속적인 문서로 통보되지 않고 종속적인 문서에서 불일치 발생 여부를 검사할 때까지 불일치 해결이 지연된다. 불일치는 종속적인 문서에서 일관성 검사를 수행하여 변경이 전파되거나 변경 전파가 필요 없음을 확인할 때 해결된다. 예를 들어 (그림 4)의 클래스 Dept의 getEmployeeName 소속 함수는 클래스 Employee에 종속적이다. 하지만 클래스 Employee의 getDeptName 소속 함수의 시그너처가 변경된 경우에는 변경이 전파될 필요가 없다. 이와 같은 경우를 동치 버전 개념을 이용하여 해결한다.

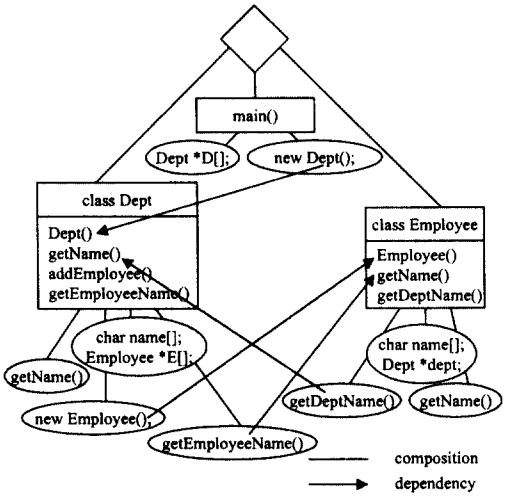
4. 일관성 정의

이 절에서는 문서의 버전간에 존재하는 종속성과 일관성을 정형적으로 정의한다. 그리고 미세 단위 종속성을 이용하여 문서간의 종속성과 일관성을 나타내는 방법을 설명한다.

정의 6 R_{ai}는 문서 a의 i번째 수정버전에서의 미세 단위 종속성 관계이다.

정의 7 P_{ai}는 문서 a의 i번째 수정버전에서의

조상 관계이다.



(그림 4) 모듈간 순환적 종속성

정의 8 P_a^*는 문서 a의 역사적 조상 관계이다.

다음은 문서의 특정 수정버전과 다른 문서와의 종속성 관계를 정의한다.

정의 9 문서 a의 i번째 수정버전 a_i는 다음과 같은 조건이 만족하면 문서 b에 종속성 관계를 가진다라고 하며 $a_i \rightarrow b$로 표현된다.

$a_i \rightarrow b \Leftrightarrow \exists x \exists y$ [$<x, y> \in R_{ai} \wedge <x, a> \in P_{ai}^* \wedge <y, b> \in P_b^*$]

정의 10 $D_{ai} = \{ b \mid a_i \rightarrow b \}$는 a_i의 종속성 집합이다.

다음은 문서의 버전간의 일관성을 정의한다.

정의 11 문서 a의 i번째 수정버전 a_i가 문서 b에 종속적이고 a_i가 b_j의 내용에 기반하여 생성되었거나 a_i와 b_j의 일관성이 검사되어 변경되지 않았으면 a_i는 b_j와 일관성 관계를 가진다라고 하며 $a_i \sim b_j$로 표현된다.

정의 12 $C_{ai} = \{ b_j \mid a_i \sim b_j \}$는 a_i의 일관성 집합이다.

정의 13 a_i의 일관성 집합에 문서 b의 j번째 수정버전 b_j가 속하고 문서 b의 최근 버전 번호가 j보다 크면 a_i와 b사이에는 잠재적 불일치가 존재한다.

정의 14 객체 x에서 모듈 m으로 종속성 $<x, m>$이 존재하고 모듈 m의 속성 p가 변경될 경우 항상 객체 x가 변경되어야 한다면 종속성 $<x, m>$은 모듈 m의 공개 속성 p와 연관성이 있다고 한다.

예를 들어 (그림 4)의 클래스 Dept의 소속 함수 getEmployee의 구현 객체는 클래스 Employee의 소속 함수 getName의 시그너처에 연관성이 있다. 즉 getName의 시그너처가 변경되면 getEmployee의 구현도 반드시 변경되어야 한다. 그러나 getEmployee의 구현 객체는 getDeptName의 시그너처에는 연관성이 없다.

정의 15 모듈 m이 문서 a의 두 버전 a_i와 a_j에 존재하고 a_i에서 a_j로의 변경이 미세 단위 종속성 $<x, m>$의 관점에서 객체 x로 전파될 필요가 없을 때 두 버전은 종속성 $<x, m>$에 관하여 동치라고 하고 $a_i \equiv^{x,m} a_j$로 표현한다.

정리 1 모듈 m이 문서 a의 두 버전 a_i와 a_j에서 모두 존재하고 다음 중 한가지 조건이 만족될 때 $a_i \equiv^{x,m} a_j$이다. 단 $i < j$ 이다.

(1) 버전 i와 j사이에 모듈 m의 영구적 연산이 존재하지 않으면 버전 i와 버전 j는 종속성 $<x, m>$에 관하여 동치이다. 즉 $\cup k=i+1 j OP_{km} = \emptyset$ 인 경우이다. 단 OP_{km} 은 모듈 m의 k번째 버전에서의 연산 히스토리이다.

(2) 버전 i와 j사이에 존재하는 모듈 m의 연산 히스토리의 합집합을 최소화한 결과 연산이 존재하지 않으면 버전 i와 버전 j는 종속성 $<x, m>$에 관하여 동치이다. 즉 $M(\cup k=i+1 j OP_{km}) = \emptyset$ 인 경우이다. 단 M은 연산 히스토리를 최소화하는 사상이다[3].

(3) 버전 i와 j사이에 존재하는 모듈 m의 연산 히스토리의 합집합을 최소화한 결과 모든 집

합 속성의 생성 및 삭제 연산과 단위 속성의 변경 연산이 종속성 <x, m>과 연관성이 없으면 버전 i와 버전 j는 종속성 <x, m>에 관하여 동치이다. 즉 모든 $c(s)$, $d(s) \in M(\cup k=i+1jOP_{km}$)에 대하여 <x, m>과 집합 속성 s는 연관성이 없고 모든 $u(v) \in M(\cup k=i+1jOP_{km}$)의 모든 (aid, val) $\in v$에 대하여 <x, m>과 단위 속성 aid도 연관성이 없는 경우이다.

(증명) 증명은 세 부분으로 나뉘어진다

(1) 모듈 m의 연산이 존재하지 않는다는 것은 모듈 m이 변경되지 않고 자식 객체가 생성되거나 삭제되지 않은 것이다. 그리고 종속성과 모듈의 정의에 의해서 종속성 <x, m>이 존재한다는 것은 객체 x가 모듈 m의 특정 공개 속성에만 종속적이라는 것을 의미한다. 따라서 a_i와 a_j는 동치이다.

(2) 일시적 연산 히스토리를 최소화한 결과는 객체의 가시성과 속성 값에 영향을 주지 않고 이 결과는 영구적 연산 히스토리로 쉽게 확장이 될 수 있다[Rho]. 따라서 최소화한 결과 연산이 존재하지 않는다는 것은 모듈 m이 변경되지 않은 것과 동일하고 a_i와 a_j는 동치이다.

(3) 객체 x는 종속성 <x, m>와 연관된 모듈 m의 공개 속성의 변경에만 종속적이다. 그런데 연관된 공개 속성이 변경되지 않았으므로 a_i와 a_j는 동치이다. *Q.E.D.*

동치 조건(1)은 문서의 버전이 바뀌었지만 종속성에 관련된 모듈은 변경되지 않은 경우이다. 동치 조건(2)는 모듈에 연산이 적용되었지만 버전간 연산 최소화 과정에 의해서 연산이 존재하지 않는 것과 마찬가지안 경우이다. 동치 조건(3)은 모듈의 변경된 부분이 종속성과 연관성 없는 경우이다. (그림 5)는 종속성 <x, m_1>에 대해 동치인 버전의 예이다. (그림 5) (가)는 버전 1이다. (그림 5) (나)의 버전 2는 모듈 m_1에는 연산 히스토리가 없으므로 모듈 m_1은 변경되지 않았고 따라서 버전 1과 동치이다. (그림 5) (다)는 집합 속성 s_1이 버전 3에서 생성되었다가 버전 4에서 삭제된 경우이다. 그런데 연산 히스토리의 최소화에 의해서 생성과 삭제 연산은 제거될 수 있으므로 버전 4는 버전 1과 동치이다. (그림 5) (라)의 버전 5는 연산 히스토리를 최소화한 후에도 변경 연산이 존재한다. 그러나 변경된 속성 aid가 종속성 <x, m_1>과 연관성이 없으므로 버전 1과 동치이다. 그런데 버전 2, 4의 경우에는 버전 1과 동치라는 것을 사람의 개입 없이도 알 수 있지만 버전 5의 경우에는 연관성이 없음을 알기 위해서는 사람의 결정이 필요하다. 이와 같은 문제를 피하기 위해서는 종속성의 대상을 모듈이 아니라 모듈의 속성으로 하는 방법이 있다. 그러나 이 경우 객체 관리가 복잡해지는 단점이 있다.

형상은 문서의 버전으로 구성된 집합이다. 하나의 형상에는 한 문서의 버전이 하나만 포함될 수 있고 버전은 여러 형상에 속할 수 있다. 형상은 정의 16과 같은 조건이 만족되면 일관성 있는 형상이라고 한다. 간단히 말해서 형상에 속한 모든 버전간에 잠재적 불일치가 없거나 잠재적 불일치가 존재하더라도 버전간에 존재하는 종속성에 대하여 동치 버전이면 일관성 있는 형상이다. 일관성 있는 형상의 정형적 정의는 다음과 같다.

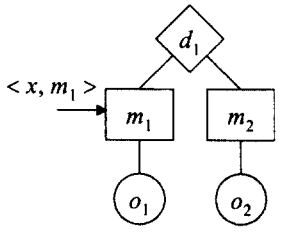
정의 16 형상 G는 다음과 같은 조건이 만족될 때 일관성 있는 형상이라고 한다.

$\forall a_i \in G$ [$\forall b \in D_{ai}$ ($\exists b_j \in G$ (bj$\in C_{ai}$)) $\vee$ $\exists b_k \in G$ ($b_j \in C_{ai}$ $\wedge$ $\forall$<x, m>$\in R_{ai}$ s.t. $x \in a_i$, $m \in b_j$, and $m \in b_k$ ($b_j \equiv^{x,m} b_k$))]

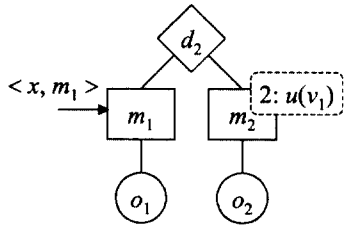
5. 불일치 해결 방법

앞 절에서는 문서간 종속성 관계와 버전간 일관성 관계, 그리고 일관성 있는 형상을 정의하였다. 이 절에서는 종속성과 일관성을 관리하는 방법과 불일치가 발생하였을 때 불일치를 해결하는 방법을 설명한다.

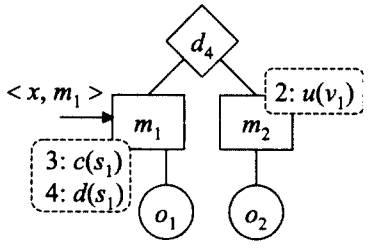
종속성과 일관성 관계를 관리하기 위하여 문서의 각 버전은 문서간 종속성 리스트와 문서의 버전간 일관성 리스트를 유지한다. 문서간 종속성 리스트와 버전간 일관성 리스트는 각각 다음과 같이 정의된다.



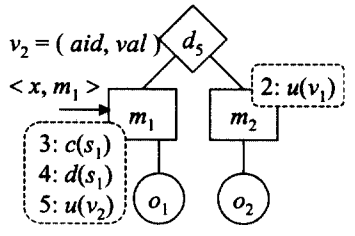
(가) 버전 1, 처음 상태



(나) 버전 2, 모듈 m_2변경 후



(다) 버전 4, 집합 속성 s_1 생성 및 삭제 후



(라) 버전 5, 속성 a 변경 후

(그림 5) 문서 d의 종속성 $< x, m_1 >$에 관한 동치 버전

정의 17 문서 a의 i번째 버전 a_i의 다른 문서와의 종속성 리스트는 다음과 같이 정의된다.

$D_{ai}[b]$ = a_i에서 문서 b로의 직접적인 미세 단위 종속성 관계의 수

정의 18 문서 a의 i번째 버전 a_i의 다른 문서의 버전과의 일관성 리스트는 다음과 같이 정의된다.

$C_{ai}[b]$ = ai와 일관성 관계를 가지는 문서 b의 버전 번호

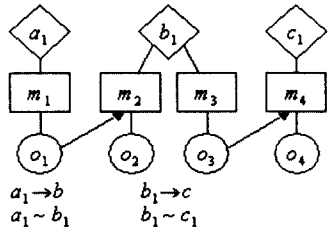
어떤 문서의 최근 버전이 그 문서가 종속적인 다른 문서와의 불일치가 발생하였는지를 알기 위해서 다음과 같은 알고리즘 consist_check를 이용한다. 함수 consist_check는 문서의 최근 수정 버전을 인자로 받아서 그 버전이 가지는 문서간 종속성을 따라 깊이 우선 탐색을 한다. 탐색 중 잠재적 불일치가 존재하면 propagate_change를 호출한다. propagate_change는 종속적인 문서와 주도적인 문서의 버전을 하나씩 인자로 받아서 변경을 전파, 즉 불일치가 발견되면 불일치를 해결한다. 변경의 전파로 종속적인 문서의 새로운 수정버전이 생성될 수도 있고 주도적인 문서의 버전이 미세 단위 종속성에 관하여 이전 버전과 동치이면 새로운 수정버전이 생성되지 않고 일관성 관계만 수정된다. 변경이 전파되어 새로운 버전이 생성되어야 하는 경우 사람의 개입이 필요하고 동치인 경우는 자동으로 해결될 수도 있다.

```
function consist_check( a : document; i : revision number ) :
revision number;
begin
  a_i.visited := true;
  changed := false;
  for each b ∈ D_ai do
  begin
    j := C_ai[b];
    if b_j.visited = false then j_1 := consist_check(b, j)
    else j_1 := the latest revision number of b;
    if j_1 > j then
      changed := changed ∨ propagate_change(a_i, b_j1)
    C_ai[b] := j_1;
  end;
  if changed then
  begin
    the latest revision number of a := i + 1;
    return(i + 1);
  end
```

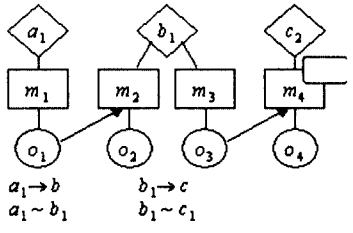
```
  else
    return(i);
end; {consist_check}
function propagate_change( a_i, b_j : revision of document ) :
boolean;
begin
  investigate the change in b_j
    and propagate it to a_i if required;
    {human intervention may be required}
  modify D_ai if required as the result of propagation;
  {human intervention may be required}
  if changed return(true)
  else return(false);
end; { propagate_change }
```

consist_check와 propagate_change에 의한 일관성 검사와 불일치 해결 과정을 간단히 설명하면 다음과 같다. 문서 a의 최근 버전 a_l의 일관성을 검사하기 위하여 $a_l \rightarrow b$ 와 같은 종속성 관계를 따라 깊이 우선 탐색을 한다. 만일 문서 b의 최근 버전 b_m이 $a_l \rightarrow c$인 문서 c가 존재하지 않으면 b_m은 일관성이 유지되어 있는 것이다. 만일 $a_l \rightarrow b$인 모든 문서 b에 대하여 $a_l \sim b_m$이면 a_l은 일관성이 유지되어 있는 것이다. 만일 b의 최근 버전이 b_{m+1}이고 a_l의 일관성 집합에는 b_m이 속했으면 잠재적 불일치가 존재하는 것이다. 따라서 b_{m+1}에서 a_l로의 변경 전파 여부를 결정하여야 한다. 변경이 전파되어야 하면 편집 과정을 거쳐서 문서 a를 수정하여 a_{l+1}이 생성되고 일관성 $a_{l+1} \sim b_{m+1}$이 추가된다. 변경이 전파될 필요가 없으면 a의 새로운 버전이 생성되지 않고 $a_l \sim b_{m+1}$이 추가된다.

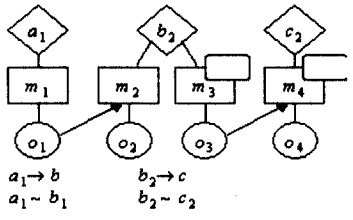
예를 들어 (그림 6) (가)는 일관성 있는 처음 상태의 형상을 보여준다. (그림 6) (나)는 문서 c의 모듈 m_4가 변경되어 새로운 버전 c_2가 생성되었지만 아직 변경이 전파되지 않은 상태이다. 그 상태에서 문서 a의 최근 버전 a_1과 다른 문서와의 불일치 발생 여부를 검사하면 깊이 우선 탐색에 의해 b_1과 c_2가 차례로 방문된다. c_2는 다른 문서로의 종속성을 가지고 있지 않으므로 일관성이 유지되어 있는 것이다. 따라서 b_1의 일관성이 검사되고 그 결과 변경이 전파되어 b_2가 생성되고 $b_2 \sim c_2$가 추가된다. 문서 a는 b_1과 b_2가 미세 단위 종속성 $<o_1, m_2>$에 관하여 동치인 경우이기 때문에 변경되지 않고 $a_1 \sim b_2$이 추가된다.



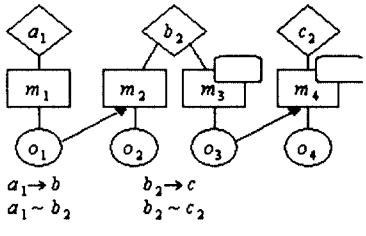
(가) 일관성 있는 처음 상태



(나) c를 변경한 후의 상태



(다) b로 변경을 전파한 후의 상태



(라) a로 변경을 전파한 후의 상태

(그림 6) 불일치 해결의 예

6. 결 론

본 논문에서는 미세단위 소프트웨어 객체의 일관성을 효율적으로 관리할 수 있는 모델을 제시하였다. 객체의 버전이 생성될 때 사용된 연산 히스토리와 객체간 종속성 관계로부터 일관성 검사를 수행할 수 있다. 문서의 새로운 버전이 생성되었을 때 큰 단위 객체 관리에서는 변경의 전파 여부를 결정하기 위하여 변경 내용을 확인하여야 한다. 본 논문에서 제시하는 모델에서는 일관성은 객체간의 종속성과 객체에 적용된 연산에 의해 관리되는데 미세 단위 객체간의 관계가 명시되어 있기 때문에 불필요한 일관성 검사를 피할 수 있고 연산 히스토리로부터 버전간의 변경 내용을 쉽게 파악하여 버전 전파 여부를 결정지을 수 있다. 또, 기존의 일관성 관리 모델에서는 문서간의 종속성이 버전에 따라 변하는 경우와 문서간 순환적 종속성이 존재하는 경우를 관리할 수 없었다. 그러나 본 논문에서 제시한 모델에서는 미세 단위 종속성의 변화에 따라 문서간 종속성이 변화하는 경우와 순환적 종속성이 존재하는 경우를 모두 관리할 수 있다.

참 고 문 헌

[1] 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.

[2] 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.

[3] J. Rho and C. Wu, "An Operation-Based Model of Version Storage and Consistency Management for Fine-Grained Software Objects", *J. of Korea Information Science Society: Software and Applications*, vol.27, No.7 July 2000.

[4] 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.

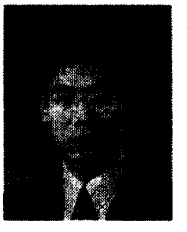
[5] E. J. Choi and Y. Kwon, "An Efficient Method for Version Control of a Tree Data Structure," *Software- Practice and Experience*, vol. 27, no. 7, July 1997.

[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] J. Estublier and R. Casallas, "Three Dimensional Versioning," *Int'l Workshop on Software Configuration Management, Selected Papers SCM-4 and SCM-5*, April 1995.

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, 데이터베이스
E-Mail: jkrho@skuniv.ac.kr