능동 객체지향 데이터베이스에서의 일관성 관리에 관한 연구

이 형 민† · 김 응 모†† · 윤 종 필†††

요 약

본 논문은 능동 객체지향 데이터베이스 (active object-oriented databases)의 인스턴스 변화에 대한 일관성을 관리하는 모델을 제시한다. 데이터베이스의 인스턴스의 변화는 사용자의 갱신에 의하여 구동되는 데, 능동데이터베이스에서는 데이터 변화에 따른 능동규칙 (active rules)의 구동 (activation)으로 말미암아 데이터 변화 효과 (side-effects)가 연속적으로 파급 (propagation)될 수 있다. 이러한 모델에서는 데이터베이스의 능동성은 보장하지만, 능동성 보장 (즉, 능동규칙 구동과 데이터 변화 효과의 연속 파급)으로 인하여 발생하는 비일관성은 아직 문제로 남게 된다. 데이터 변화 효과의 파급이 가져오는 전체 데이터베이스의 일관성 문제를 파악하고 해결하는 것이 문제이다. 뿐만 아니라, 객체지향 데이터베이스에서는 객체지향성이 내포하는 복잡한 특성인 일반화 계층으로 인하여 능동성 보장 및 일관성 관리에도 어려움이 있다는 것은 잘 알려진 사실이다.

본 논문은 능동 객체지향 데이터베이스에서 능동성을 보장할 때 유발되는 비일관성 문제를 해결하는 것이 목적이다. 일관성 관리를 위해 먼저, 무결성 제약조건 (integrity constraints)과 능동규칙을 객체지향 데이터베이스에서 이용가능하도록 표현한다. 이러한 논리적 표현 (logical representation)을 바탕으로 능동 객체지향 데이터베이스에서 발생되는 "전체 변화"를 순차적인 갱신문으로 변환시킨다. 본 논문에서 제시하는 순차적 갱신문은 (1) 데이터베이스 변화의 사전조건 (변화가 시도되어도 좋을 조건 : pre-condition), (2) 변화 효과의 전파 (연속해서 수행되어야 하는 능동규칙), (3) 보수가 필요하지 않도록 데이터 변화의 폭을 제한하는 사후조건 (변화후의 일관성을 확인하는 조건 : post-condition)을 포함하도록 한다. 본 논문의 기여도는 먼저 능동성을 객체지향 데이터베이스에 새롭게 적용하였으며, 전체 데이터베이스의 일관성을 관리하기 위하여 갱신문들이 순차적으로 모델링되는 데 있다. 결국, 이 모델링은 사용자가 제시하는 단순한 갱신이 데이터베이스내에서 능동적으로 뿐만 아니라 일관성을 유지하도록 수행될 수 있는 장점을 가지고 있다.

Consistency Management in Active Object-Oriented Database

H. M. Lee† · U. M. Kim†† · J. P. Yoon†††

ABSTRACT

This paper describes consistency management in active object-oriented databases. State changes in typical databases are initiated by users, however, in active databases, active rules are activated in response of database state changes, and further the effects of active rule activation are propagated as well. Although this manner

※이 논문은 1995년도 한국학술진흥재단의 공모과제 연구비에 의하여 연구되었음.
† 준 회 원 : LG-soft 근무
†† 정 회 원 : 성균관대학교 정보공학과
††† 정 회 원 : 숙명여자대학교 전산학과
논문접수 : 1997년 1월 6일, 심사완료 : 1997년 9월 11일

increases the activeness in databases by side-effect propagations, it may cause inconsistencies to the overall databases by those side-effect propagations. Activation of active rules and propagation of rule activation side-effects are useful in one hand to make databases active, but it come shortfalls in another hand to keep databases in consistency. In addition, object-orientation characteristics are yet another shortfall to make databases both active and consistent.

This paper aims to solve the inconsistency problem which may occur in active object-oriented databases. First, integrity constraints and active rules are formalized to be suitable to object-oriented databases. Such a logical representation can be semantically reformulated to a sequence of database updates. A sequence of database updates thus contains (1) pre-conditions for updates to be activated, (2) propagations of side-effects of active rule activations, (3) post-conditions for updates to be in consitency. The contributions of this paper contain an extension of activeness features to object-oriented databases, and modeling a sequence of updates to manage database consistencies. In this way, a single database update can be reformulated to a sequence of updates in order to make object-oriented databases active and maintain the databases consistent.

1. 서 론

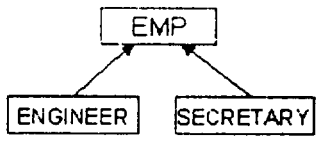
능동데이터베이스(active database)는 능동규칙이라는 메카니즘(mechanism)을 이용하여 올바른 데이터베이스의 상태(database state)를 능동적으로 변화시키는 강력한 기능을 제공한다. 이러한 메카니즘의 일부는 이미 상용 데이터베이스의 "트리거(trigger)"로 구현되기도 하였다. 다만, 능동데이터베이스에서는 트리거가 데이터의 상태를 변화시키고 이 변화된 데이터의 상태로 말미암아 또 다른 트리거가 실행되도록 모델링되는 것이다. 이 경우, 능동데이터베이스의 특성이 사용자의 간섭 없이 능동적으로 작업을 실행하는 것이기 때문에, 트리거의 연속성은 전체 데이터베이스로 파급되고 데이터베이스의 어느 한 부분이 (만약 전체가 아니라면) 일관성 (consistency)을 잃게 되는 결과를 초래하게 된다. 능동적인 트리거의 실행 결과는 데이터베이스의 일관성을 보장하여야 한다. 본 논문은 다음과 같이 두가지 문제점을 해결하려 한다: 1) 논리식(logical formula)으로 정의된 무결성 제약조건과 능동규칙을 그 각각의 의미 (semantics)에 따라 객체지향 데이터베이스 기본 연산 (예, update, insert, delete)로 변환하고, 2) 제약조건에 위반될 수 있는 부분을 미리 제한시키므로 일관성을 보장하므로 보수(repair)장치가 불필요하게 된다.

본 논문은 이미 기존의 연구[24]에서 관계형 데이터베이스를 모델로 하여 소개된 바 있는 제약 언어(constraint language)를 객체지향 데이터베이스 (object-oriented database)의 특성인 일반화 (generalization) 계층에서도 적용할 수 있도록 확장한다. 사용자의 질의 (query)는 SQL의 형식을 따르며, 이는 다시 정당한 갱신 (proper update)을 미리 보장하는데 필요한 사전조건과 갱신 후의 변화 효과의 일관성을 확인하는 데 필요한 사후조건으로 변환된다. 능동적인 데이터 변화가 연속적일 때 비일관성 (inconsistency)에 다다를 수도 있는 데, 바로 사후조건이 갱신으로 인해 유발될 수 있는 비일관성을 배제하는 조건이다. 사용자의 갱신은 데이터의 상태를 변화시키고, 이 데이터의 변화는 또 다른 능동규칙을 실행하게 한다. 이와 같은 능동적 실행은 전체 데이터베이스를 대상으로 계속 진행되는데 이 때 전체 데이터베이스의 일관성을 유지하기 위한 방법이 필요하게 된다. 만약 비일관성이 발견되었을 경우 이전 까지의 방법은 다시 사용자에게 알리거나 사용자로부터 보수방법을 기다리는 것이었다. 연구 [24]에서는 관계형 데이터베이스에서 능동규칙의 실행으로 비일관성을 보수하는 방법을 제시하였다. 이 경우 능동규칙은 주어진다고 가정하고 출발하였다. 이와는 달리, 본 논문에서는 객체지향 데이터베이스의 일반화 계층의 상속성(inheritance)하에서 이용하여 제약조건에 위배되는 부분을 미리 제한함으로써 보수가 필요없게 되는 방법을 제시하려 한다. 더 진행하기 전에 예제를 통해 문제점을 살펴보기로 하자.

1.1 동기 부여 예제

예제 1.1 어떤 회사의 사원에 대한 데이터베이스를 구축함에 있어, 다음과 같은 클래스형으로 구축하였

다고 가정한다. 각 사원의 정보는 다시 기술자와 비서에 대한 정보로 세분화된다.



(그림 1) 클래스 예제 구조
(Fig.1) Example of Class Hierarchy

각 클래스는 다음과 같은 자기 고유의 제약조건을 가진다고 가정한다.

· EMP (사원에 대한 클래스:수퍼클래스(superclass))
  −IC1:근무년수(years_of_expr)가 4년이 안되는 사원의 임금(salary)은 50K미만이다.
  −IC2:임금이 50K보다 큰 사람의 세율(tex_rate)은 0.15보다 크다.
· ENGINEER (기술자에 대한 클래스:EMP 클래스의 서브클래스(subclass))
  −IC1:"반도체" 부서(dept)에 소속된 기술자의 근무년수는 3년이상이다.
  −IC2:기술직의 임금은 40K보다 작아서는 안된다.
· SECRETARY (비서에 대한 클래스:EMP 클래스의 서브클래스(subclass))
  −IC1:비서는 반드시 여성이다.
  −IC2:비서의 타이핑 속도는 300이상이어야 한다.
  −IC3:비서의 임금은 30K보다 작아서는 안된다.

이와 같이 설정되어 있고 다음과 같이 SQL 형식으로 각 사원에 대한 임금을 10% 인상한다는 질의가 제시될 수 있다.

```
UPDATE EMP
SET salary:=salary*1.1
```

이 질의에 대한 갱신은 EMP객체에 대해서 IC1, IC2, ENGINEER객체에 대해서 IC2, SECRETARY 객체에 대해서 IC3와 연관되어 있다. 즉, 새롭게 갱신된 임금은 이 제약조건들에 대해 객체들 모두 만족하여야 하고, 그렇지 않은 객체에 대해서는 갱신을 취소하여야 한다. 또한, EMP객체의 IC2의 경우는, 새 임금에 의해서 세율의 값도 만족하는지 검사하여야 한다. □

예제 1.2 능동데이터베이스는 제약조건뿐만 아니라, 능동규칙도 가지게 된다. 위의 예제 1.1에서 보여주었던 데이터들에 대해서 각각 가지게 되는 능동규칙은 다음과 같다고 가정한다.

· EMP
  −PR1:임금이 50K이상이고 부양가족수가 3이상인 사원의 세율은 0.20이다.
  −PR2:사원은 20년이상 근무할 수 없다.
  −PR3:임금이 80K이상이면 'high_paid' 객체에 삽입된다.
· ENGINEER
  −PR1:기술자는 10년 이상 근무할 수 없다.
  −PR2:임금이 50K이상이고, 근무년수가 2년이 안되면, 근무시간은 10시간이다.
· SECRETARY
  −PR1:타이핑 속도가 350이하인 사람의 최대 임금은 45K이다.

앞의 예제 1.1에서의 갱신, 즉, 사원들의 임금을 10% 인상하라는 질의에 대해, EMP객체의 PR1, PR3, ENGINEER객체의 PR2, SECRETARY객체의 PR1이 연관된다. 새롭게 갱신된 임금이 능동규칙의 조건을 만족하는 경우, 그에 해당하는 행동을 실행하여야 한다.

본 논문은 위의 두가지 경우에 대해서 중점적으로 다루어보고자 한다.

1.2 관련 연구

능동데이터베이스는 [2, 7, 8, 17]에서 보여진 바와 같이 데이터베이스 상태를 감시(monitor)하므로, 그 상태가 바뀌면, 어떤 행동들을 트리거하기 위해서 알맞은 규칙들이 행동화되어 진다. 이러한 작업들은 사용자 간섭(user intervention)이 전혀 없는 상태에서 발생한다.

선언적 갱신(declarative update)은 곧 데이터베이스

상태의 변환과정을 나타내며, 이는 절차적인 방식(procedural specification)으로 변환이 가능하다. 이러한 갱신의 변환에 관해서는 [16, 21]에서 연구된 바 있다. 뷰(view) 갱신 문제에 관해서는 많은 연구가 이루어져 왔다. [13]은 어브덕션(abduction)의 방법을 이용하고 있는데, 이는 갱신에 관련된 제약조건 검사에 대한 부분을 갱신과 병합하여 일관성이 유지되지 않는 상태의 능동을 제한하기 위한 것이다. [6]은 사용자가 SQL의 표현으로 뷰를 정의하였을때, 뷰의 상태에 대해 생성 규칙(production rule)의 적용시의 일관성 유지에 대해 연구하였다. [10]은 원시적인 갱신 연산자(primitive update operator)를 복잡한 갱신(complex update)으로 재기술하는 방법과 뷰 갱신을 데이터베이스 갱신으로 바꾸는 방법에 대해서 연구하였고, [24]는 이와 비슷한 방법으로 뷰에서의 갱신을 연구하였다.

[3]과 [12]는 갱신을 규칙 언어(rule language)에 병합하는 연구를 하였다. [23]은 SQL로 만든 규칙 언어를 스타버스트(Starburst) 규칙 시스템에 적용하였다. [24]는 SQL의 표현으로 관계형 데이터베이스에서 규칙들과 제약조건들을 갱신에 병합하였다. 본 논문도 [24]의 방법처럼 SQL의 표현으로 규칙들과 제약조건들을 이용하되 객체지향 데이터베이스의 일반화 계층 특성을 이용하여 일관성을 관리한다.

제약조건 검사와 제약조건 위반에 대한 보수(constraint violation repair) 역시 능동데이터베이스에서의 중요한 주제들이다. 많은 연구의 결과로 제약조건들의 서술[14, 20, 22]과 제한(enforcement)[4, 7, 22]에 대한 방법들이 개발되었다. 제약조건 위반에 대한 보수 방법은 [18]에서 제안된 바 있다. 여기서는 제약조건에 대한 위반이 적절하지 못한 트랜잭션(transaction)에 의해 발생한다고 가정하고 있고, 따라서 비일관성을 생성하는 현상은 트랜잭션으로부터 제거되어야 한다고 가정하고 있다. 반대로, [24]는 제약조건 위반은 불완전한 갱신 표현에 의해 발생한다고 가정하였고, 갱신의 효과는 전파되고, 제약조건을 만족하지 않는 데이터베이스의 요소(instance)들은 고쳐져야 한다고 하였다. [5]는 생성 규칙을 비일관성 상태를 보수하는데 사용하였다. 여기서는 제약조건을 일관성을 유지하는 능동규칙으로 번역(translate)하는 방법을 보여주었다. 그러나, 이 번역 작업은 사용자의 간섭을 요구하는데, 이는 정적(static)이고, 수동적(manual)이다. 본 논문에서는 이와 같은 보수작업의 어려움을 해결하기 위하여 미리 데이터 변화에 제한을 가하도록 하였다. 즉 사후조건처리로 말미암아, 갱신에 또 다른 제한을 가하고 있다.

마지막으로, [1, 11, 19]는 전체 데이터베이스의 일관성을 유지하기 위한 갱신 전파에 대한 필요성을 주장하고 있다.

1.3 논문의 내용 소개

이 논문의 나머지 부분은 다음과 같이 구성되어 있다. 2장은 제약조건을 일차 논리(first-order logic)로 표현하고, 능동 객체지향 데이터베이스에서의 새로운 클래스(class)형을 제시하였다. 3장은 객체지향 데이터베이스의 일반화에 대해 제약조건을 사용한 갱신 확인(verification)에 대해 서술하고 있다. 여기서는 제약조건이 갱신에 어떻게 병합되는지 SQL의 형식을 빌어 표현하였다. 4장은 2장의 방법을 이용하여 능동규칙에 대해 어떻게 적용되는지를 표현하였고, 5장은 3장의 방법을 이용하여, 능동규칙에 대해서 어떻게 적용되고, 병합되는 지를 표현하였고, 변화 효과의 전파에 대해서도 다루고 있다. 마지막으로, 6장은 본 논문의 결론을 서술하고 있다.

2. 제약조건 언어

본 장에서는 제약조건의 문법과 객체지향 데이터베이스에서의 선언에 대하여 정의하려고 한다.

2.1 문 법

객체지향 데이터베이스에서 이용하는 제약조건은 다음과 같이 if~then 의 형식으로 표현될 수 있다. 이때 if조건이 만족해야 then의 조건을 테스트하고 만약 then의 조건도 만족하면 이 제약조건을 만족한다고 할 수 있다.

if *condition1*
then *condition2*

여기서 특별한 경우를 살펴보면, 만약 *if condition1* 만 선언되는 경우 (즉, then 조건이 없는 경우)는 "데

이터베이스가 그 condition1을 만족해서는 않된다"는 조건이고, 만약 *then condition2*만 선언되는 경우(즉 if 조건이 없는 경우)는 "데이터베이스가 그 condition2를 만족해야 한다"는 조건이다. 의미에 따라, 객체지향 데이터베이스에서 선언할 수 있다. 이는 다음과 같은 일차 논리의 형태로 나타낼 수 있고, 이 일차 논리화된 제약조건이 다음 절에서와 같이 각 클래스에 병합된다.

condition1 → *condition2*

여기서도 *if condition1* 은 "*condition1* →"으로, *then condition2*은 "*condition2*"로 나타낼 수 있다.

2.2 클래스형 제시

본 장에서는 아래와 같이 제약조건을 병합하여 객체지향 데이터베이스에서 클래스를 선언하려 한다.

```
class class_name {
        OID        : oid-type
        attribute (…

        ),
        constraint ( // 무결성제약조건에 관한 rule


}
```

이 클래스 구조는 기존의 클래스 구조에서 제약조건과 능동규칙을 포함하는 형태이다. 여기서, 제약조건 부분은 2.1에서 소개한 바와 같이 일차 논리로 표현하여 구성하고 있다. 이 클래스형은 4장에서 완성될 것이다. 다음의 예제를 살펴보자.

2.3 예 제

1.1절의 예제에서 소개된 데이터들의 클래스형을 구축하면 다음과 같다.

```
Class EMP {
   OID:  oid-type,
   attribute (
     name:  string(20),
     sex: { M , F },
     salary: int,
     years_of_expr: int,
     tex_rate:  float,
     num_depn: int
   ),
   constraint (
     IC1: (years_of_expr<4) →
              (salary<50K),
     IC2: (salary>50K) →
              (tex_rate>0.15)
   )
}
```

```
class ENGINEER {
   OID:  oid-type,
   inherit EMP,
   attribute (
      dept:  string(20),
      job_time_day: int
   ),
   constraint (
      IC1: (dept="반도체") →
              (years_of_exp>=3),
      IC2: (salary<40K) →
   )
}
```

```
class SECRETARY {
   OID:  oid-type,
   inherit  EMP,
   attribute (
     typing_speed: int
   ),
   constraint (
     IC1 : (sex!="F") → ,
     IC2 : (typing_speed<300) →,
     IC3 : (salary<30K) →
   )
}
```

위에서 보여주는 바와 같이, 각 클래스에 해당하는 제약조건이 기존의 속성(attribute)과 함께 클래스에 캡슐화(encapsulation)되어 있음을 알 수 있다. □

3. 데이터 변화 중의 제약조건 관리

데이터베이스는 모든 제약조건들이 전체 데이터베이스 상태에 의해 만족되어질 때, 일관성이 있다고 한다. 그러나, 데이터베이스가 갱신되어지고 그 갱신이 사용자의 간섭없이 능동적으로 일어난다면, 일관성이 깨질 수 있다. 더욱 이 갱신의 부작용이 전파되어질 때, 그 문제는 더욱 더 심각해진다. 이 장에서는

원래의 갱신에 사전/사후조건을 병합하는 갱신 언어를 객체지향 데이터베이스의 일반화에 대해 적용하여 보여준다. 이러한 조건들은 데이터베이스의 일관성을 관리하기 위해 사용될 수 있다.

3.1 갱신 언어

데이터 변화는 갱신 (수정, 추가, 삭제 등)이 일어나기 전에 만족하여야 하는 사전조건과 갱신이 끝난 후 만족하여야 하는 사후조건으로 나누어 진다 [24]. 전형적인 update-set-where구문[15]의 갱신은 where부분의 조건이 만족되어야 실행된다. 여기서, where부분의 조건은 갱신전에 만족해야 하는 조건이기 때문에, 사전조건과 같다. 하지만, 갱신 작업이 전파된다면, 데이터베이스의 일관성을 유지하기 어렵기 때문에, 사후조건을 필요로 한다 [24]. 이러한 사전조건과 사후조건을 포함하는 갱신 언어에 대한 연구는 [24]에서 관계형 데이터베이스에 대해 연구된 바 있다. 본 논문은 이러한 갱신 언어를 객체지향 데이터베이스의 일반화에 대해 적용한다.

이 일반화의 개념은 기존의 갱신 언어에 대한 확장을 요구한다. 일반화된 클래스들은 각자의 고유한 데이터들과 속성을 가지게 된다. 이러한 성질 때문에, 각 클래스는 제약조건 또한 고유하게 가지게 된다. 따라서, 갱신 언어를 일반화된 클래스에 적용할 경우, 다음과 같이 개념을 확장하여야 한다. 여기서, 우리는 R1은 R의 서브클래스이고, R2는 R1의 서브클래스로 가정하자.

3.1.1 수퍼클래스 (R)에 대해 갱신이 가해질 경우

갱신이 수퍼클래스 R에 요구되어 졌을 경우를 고려하자. 우선, 주어진 갱신에 수퍼클래스가 가지고 있는 제약조건들 중에서 해당 조건들을 이용하여 사전조건과 사후조건을 찾아내어 다음과 같은 형식으로 수퍼클래스에 적용한다.

UPDATE *R class*
SET assignments
PRECOND *R's constraints* are satisfied
POSTCOND *R's constraints* are not violated

수퍼클래스에 대한 갱신이 끝나면, 갱신의 범위를 그 수퍼클래스의 서브클래스(R1 class)로 옮긴다. 즉, 다음과 같은 형식으로 서브클래스에 적용한다.

UPDATE *R1 class*
SET assignments
PRECOND *R's constraints* are satisfied
POSTCOND *R's constraints* are not violated

같은 방법으로 이어지는 서브클래스로 범위를 옮겨간다.

UPDATE *R2 class*
SET assignments
PRECOND *R's constraints* are satisfied
POSTCOND *R's constraints* are not violated

3.1.2 임의의 서브클래스 (R1)에 대해 갱신이 가해질 경우

그 서브클래스의 모든 수퍼클래스를 찾아내어 갱신 대상 서브클래스와 수퍼클래스들의 조건들을 사전조건과 사후조건으로 분류하여 각각을 모두 논리곱하여 조건 부분을 완성한다. 완성된 조건 부분은 원래의 갱신에 병합되어 갱신이 가해진 서브클래스부터 아래로 범위를 옮겨가며 갱신작업을 한다. 단, 이때, 3.1.1절에서와 같이 조건부분은 변하지 않는다.

예를 들어, R1 class에 갱신이 가해졌을 경우는 다음과 같이 갱신 언어가 만들어진다.

UPDATE *R1 class*
SET assignments
PRECOND *R's constraints* $\wedge$ *R1's constraints* are satisfied
POSTCOND *R's constraints* $\wedge$ *R1's constraints* are not violated

같은 방법으로 이어지는 서브클래스(R2 class)로 범위를 옮겨간다.

UPDATE *R2 class*
SET assignments
PRECOND *R's constraints* $\wedge$ *R1's constraints* are satisfied
POSTCOND *R's constraints* $\wedge$ *R1's constraints* are not violated

(3) 모든 클래스 계층을 통합하였을 때의 갱신작업

이 경우는 객체지향 데이터베이스에서 SQL문 (예 SELECT R FROM R)[1]을 이용하여 갱신을 하고자 할 경우로써, 수퍼클래스에서 갱신이 가장 먼저 일어나며, 이 때의 조건 부분은 그 수퍼클래스만의 제약조건만으로 만든다. 서브클래스로 범위를 옮기면 조건 부분은 바로 전의 클래스에서 이루어졌던 갱신 언어의 조건 부분과 그 서브클래스의 사전조건과 사후조건으로 분류된 제약 조건들을 각각 논리곱 하여 조건 부분을 완성하여 갱신 작업을 하는 식으로 진행된다.

예를 들어, 클래스 계층에 대한 갱신은 다음과 같이 진행된다. 우선 수퍼클래스인 R class에 대한 갱신 작업은 다음과 같이 이루어진다.

UPDATE *R class*
SET assignments
PRECOND *R's constraints* are satisfied
POSTCOND *R's constraints* are not violated

그 다음 서브클래스인 R1 class에 대한 갱신작업은 다음과 같이 이루어진다.

UPDATE *R1 class*
SET assignments
PRECOND *R's constraints* ∧ *R1's constraints* are satisfied
POSTCOND *R's constraints* ∧ *R1's constraints* are not violated

같은 방법으로, R1 class의 서브클래스인 R2 class에 대한 갱신작업은 다음과 같다.

UPDATE *R2 class*
SET assignments
PRECOND *R's constraints* ∧ *R1's constraints* ∧ *R2's constraints* are satisfied
POSTCOND *R's constraints* ∧ *R1's constraints* ∧ *R2's constraints* are not violated

위와 같이 SQL의 표현을 이용하여 원래의 갱신을 의미적으로 풍부한(semantically-rich) 갱신의 순차로 만들었다.

3.2 제약조건을 질의 표현으로 바꾸기

이 부분은 제약조건을 SQL 질의 표현으로 어떻게 바꾸느냐에 대해 서술하고 있다. 제약조건에 대해서 데이터베이스는 단 두 가지의 상태만을 가진다: 제약조건을 '만족'하느냐(satisfy), '위반'하느냐(violate)이다. 여기에 대해 SQL의 형식으로 정의하면 다음과 같다 [24].

정의1 (제약조건 만족). p→q라는 제약조건을 만족하기 위해서는 p가 거짓(false)이 되거나, q가 참(true)이되어야 한다. □

정의2 (제약조건 위반). p→q라는 제약조건을 위반하기 위해서는 p가 참(true)이 되고, q가 거짓(false)이 되어야 한다. □

제약조건 IC는 전체 데이터베이스가 아닌 일부에 대해 만족될 수도 있고, 위반될 수도 있다. 다음과 같은 경우를 생각해 보자

IC: (LogicSet1)→(LogicSet2)
(for R1, R2, R3 class)

'(LogicSet1)→(LogicSet2)'를 다시 표현하면 '¬(LogicSet1) ∨ (LogicSet2)'가 되고, 이 질의의 결과가 공집합(empty)이라면, 그 데이터베이스는 전체가 비일관적이라는 의미이다. 이 IC를 만족하는 객체들을 선택하는 질의를 SQL의 표현으로 나타내면 다음과 같다.

SELECT *
FROM R1, R2, R3
WHERE NOT(LogicSet1) OR (LogicSet2)

1) 실제 객체관계형데이터베이스 Illustra에서 가능한 질의문임.

예제 3.1 1장의 예제에서 EMP클래스의 IC2는 임금이 50K보다 큰 사람의 세율(tex_rate)은 0.15보다 크다는 조건이다.

IC2:(salary > 50K)→(tex_rate > 0.15).

이와 같을 때, IC2를 만족하는 객체들의 집합의 표현은 다음과 같다.

```
SELECT   *
 FROM    EMP
 WHERE   NOT(salary > 50K) OR (tex_rate > 0.15)
```

반대로, IC2를 위반하는 객체들의 집합의 표현은 다음과 같다.

```
SELECT   *
FROM     EMP
WHERE    (salary > 50K) AND NOT(tex_rate > 0.15)
```

후자의 결과가 만약 공집합이라면, 데이터베이스에서 EMP클래스의 IC2에 대해 EMP클래스는 일관성있다 하겠다. □

3.3 제약조건을 사용한 갱신 일관성 관리

능동데이터베이스에 갱신 U가 가해질 때, 제약조건은 데이터베이스의 상태를 검사하여야 한다. 이 논문에서는 갱신의 부작용의 범위를 정의하는 방법에 대해서 서술한다. IC_i와 IC_j가 제약조건으로 지정되었고 IC_i는 갱신 U에 대한 데이터베이스 상태를 검사하고, IC_j는 갱신의 결과에 대한 검사를 한다고 가정하자. U가 미치지 않는 데이터베이스 객체들과 U에 의해 제약조건이 위반되는 객체들은 갱신 작업 범위 밖으로 옮겨서 작업할 수 있다. 이러한 범위의 축소작업을 병합하여 사용자가 만든 갱신 질의 언어를 다시 나타내면 다음과 같다.

```
UPDATE      object
SET         assignment in U
PRECOND EXIST   entities satisfying IC_i AND the
                condition of U
POSTCOND EXIST  entities satisfying IC_j
```

이러한 형식을 예제 1.1에 적용하면 다음과 같다.

예제 3.2 예제 1.1의 제약조건들의 논리적 표현은 다음과 같다.

- EMP
  - IC1:(years_of_exp < 4)→(salary < 50K)
  - IC2:(salary > 50K)→(tex_rate > 0.15)
- ENGINEER
  - IC1:(dept = "반도체")→(years_of_exp > = 3)
  - IC2:(salary < 40K)→
- SECRETARY
  - IC1:(sex! = "F")→
  - IC2:(typing_speed < 300)→
  - IC3:(salary < 30K)→

여기서, 다음과 같이 사원들의 임금을 10% 인상한다는 질의가 주어졌다고 가정하자.

```
UPDATE      EMP
SET         salary:=salary*1.1
```

위의 갱신작업에 대해 관련되는 제약조건들은 다음과 같다.

- EMP : IC1, IC2
- ENGINEER : IC2
- SECRETARY : IC3

각각 클래스의 객체들에 대해서 위의 제약조건들이 모두 만족되어야 그 데이터베이스의 일관성이 지켜진다. 우선 사전조건과 사후조건으로 나누면 다음과 같다:EMP클래스의 IC1의 조건은 갱신 전에 검사되어야 하고, IC2는 갱신 후에 검사되어야 한다. ENGINEER클래스의 IC2의 조건은 갱신 후에 검사되어야 한다. SECRETARY객체의 IC3의 조건은 갱신 후에 검사되어야 한다. 결국, EMP클래스의 IC1은 사전조건이 되며, EMP클래스의 IC2, ENGINEER클래스

의 IC2, SECRETARY객체의 IC3는 사후조건이 된다. □

3.3.1 EMP클래스(수퍼클래스)에 갱신이 가해지는 경우

EMP클래스에 임금 10%인상이라는 질의가 가해진다면, 갱신 언어는 다음과 같이 전개된다.

```
UPDATE   EMP
SET      salary := salary * 1.1
```

여기에 EMP클래스의 사전조건과 사후조건을 병합하면 다음과 같다. 이러한 갱신 질의로 EMP클래스에 대한 갱신 작업이 진행된다.

```
UPDATE   EMP
SET      salary := salary * 1.1
PRECOND EXIST (SELECT *
          FROM    EMP
          WHERE   NOT(years_of_exp < 4) OR
          (salary < 50K))
POSTCOND EXIST (SELECT *
          FROM    EMP
          WHERE   NOT(salary > 50K) OR
          (tex_rate > 0.15))
```

EMP클래스에 대한 갱신 작업이 끝나면, 다음과 같이 갱신 범위가 EMP클래스의 서브클래스인 ENGINEER클래스로 바뀌면서, 갱신 작업은 ENGINEER클래스에 대해 일어난다.

```
UPDATE   ENGINEER
SET      salary := salary *  1.1
PRECOND EXIST (SELECT *
          FROM          ENGINEER
          WHERE         NOT(years_of_exp < 4)
          OR (salary < 50K))
POSTCOND EXIST (SELECT *
          FROM          ENGINEER
          WHERE   NOT(salary > 50K) OR
          (tex_rate > 0.15))
```

ENGINEER클래스에 대한 갱신 작업이 끝나면, ENGINEER의 서브클래스는 더 이상 없기 때문에, 다음과 같이, 다시 수퍼클래스인 EMP클래스의 다른 서브클래스인 SECRETARY객체로 범위를 옮겨서 갱신을 실행한다.

```
UPDATE   SECRETARY
SET      salary := salary * 1.1
PRECOND EXIST (SELECT *
          FROM    SECRETARY
          WHERE   NOT(years_of_exp < 4)
          OR (salary < 50K))
POSTCOND EXIST (SELECT *
          FROM    SECRETARY
          WHERE   NOT(salary > 50K)
          OR (tex_rate > 0.15))
```

사용자가 요구한 간단한 갱신문이 전체 객체지향 데이터베이스의 일관성을 관리하기 위하여 사전/사후조건으로 검증할 수 있도록 갱신문을 모델링하였다.

3.3.2 ENGINEER클래스(서브클래스)에 갱신이 가해지는 경우

이 경우는 우선 ENGINEER클래스의 수퍼클래스인 EMP클래스의 제약조건들과 ENGINEER클래스의 제약조건들을 사전조건, 사후조건으로 분류하여 병합하여야 한다. 이 예제에서는 ENGINEER의 IC2가 사후조건이기 때문에, 사후조건에 EMP클래스의 IC2와 ENGINEER클래스의 IC2를 논리곱하여 표현한다. 갱신 작업은 ENGINEER클래스에서 시작되며, 작업 후 ENGINEER클래스의 서브클래스가 존재하지 않기 때문에, 작업은 끝난다.

```
UPDATE   ENGINEER
SET      salary := salary * 1.1
```

사용자가 요구하는 위와 같은 갱신문은 아래와 같이 변화된다.

```
UPDATE   ENGINEER
SET      salary := salary * 1.1
```

```
PRECOND EXIST (SELECT *
               FROM   ENGINEER
               WHERE  NOT(years_of_exp < 4)
               OR (salary < 50K))
POSTCOND EXIST (SELECT *
                FROM   ENGINEER
                WHERE (NOT(salary > 50K) OR
                (tex_rate > 0.15)) AND (NOT
                (salary < 40K))
```

3.3.3 (그림 1)의 클래스 계층을 모두 통합하였을 때

(그림 1)에 대한 통합 작업은 'SELECT EMP FROM EMP'라는 표현에 의해 가능하며, 갱신 작업에 대한 표현은 작업의 범위가 되는 클래스가 달라질 때마다 표현도 달라지게 된다. 다음과 같은 갱신 작업은 수퍼클래스인 EMP클래스로부터 시작된다.

```
UPDATE    SELECT EMP FROM EMP
SET       salary: = salary * 1.1
```

갱신 작업의 범위를 EMP클래스로 바꾸고, EMP클래스의 IC1을 사전조건으로, IC2를 사후조건으로 다음과 같이 완성시킨다.

```
UPDATE    EMP
SET       salary: = salary * 1.1
PRECOND EXIST (SELECT *
               FROM   EMP
               WHERE  NOT(years_of_exp < 4)
               OR (salary < 50K))
POSTCOND EXIST (SELECT *
                FROM   EMP
                WHERE NOT(salary > 50K) OR
                (tex_rate > 0.15))
```

EMP클래스에 대한 작업이 끝나면, 서브 클래스인 ENGINEER클래스에 대한 갱신 작업의 표현으로 바꾸어야 한다. 우선 범위를 ENGINEER클래스로 바꾸고, EMP클래스에서 사용하였던 사전조건과 사후조건을 그대로 유지한다. 여기에 임금 갱신에 관련되는 ENGINEER클래스의 사후조건 IC2를 기존의 사후조건에 논리곱하여 표현한다.

```
UPDATE    ENGINEER
SET       salary: = salary * 1.1
PRECOND EXIST (SELECT *
               FROM   ENGINEER
               WHERE  NOT(years_of_exp < 4)
               OR (salary < 50K))
POSTCOND EXIST (SELECT *
                FROM   ENGINEER
                WHERE  (NOT(salary > 50K)
                OR (tex_rate > 0.15)) AND (NOT
                (salary < 40K)))
```

ENGINEER클래스의 갱신 작업이 끝나고 나면, 더 이상의 서브클래스는 존재하지 않기 때문에, EMP클래스의 또 다른 서브클래스인 SECRETARY클래스로 범위를 옮긴다. ENGINEER클래스의 갱신 작업에 대한 표현과 비슷하게 진행된다. 여기서는 SECRETARY클래스의 IC3가 임금 갱신이 관련된 사후조건이기 때문에, EMP클래스 갱신 작업에 사용되었던 사후조건과 논리곱하여 다음과 같이 표현한다.

```
UPDATE    SECRECTARY
SET       salary: = salary * 1.1
PRECOND EXIST (SELECT *
               FROM   SECRETARY
               WHERE  NOT(years_of_exp < 4)
               OR (salary < 50K))
POSTCOND EXIST (SELECT *
                FROM   SECRETARY
                WHERE (NOT(salary > 50K) OR
                (tex_rate > 0.15)) AND (NOT
                (salary < 30K)))
```

사용자의 갱신 질의는 위와 같이 의미적으로 풍부한(semantically-rich) 갱신 질의로 바뀌었다. 바뀌어진 갱신 질의는 임금에 대한 검사도 이루어지지만, 그에 관련되어 세율 등에도 검사가 이루어진다.

4. 객체지향 클래스에 능동규칙 병합하기

본 장에서는 2장에서 소개된 제약조건 언어 뿐만 아니라 능동데이터베이스에서 사용되는 능동규칙을 병합하여 더 확장시킨다. 능동규칙은 일차 논리의 왼쪽 부분의 조건이 만족하면 일련의 행동들(actions)을 실행한다. 능동규칙의 행동들은 데이터베이스의 추가, 삭제, 수정 등의 갱신만을 한다고 가정한다.

4.1 문 법

능동규칙은 제약조건의 표현과 같은 일차 논리(first-order logic)이다. 제약조건과 다른 점은 일련의 데이터베이스 연산자(수정, 추가, 삭제등)들을 가진다는 것이다. 능동데이터베이스에서 이 연산자들은 규칙에 나열되어 있는 순서대로 실행된다.

능동규칙 표현에 대한 기존의 방식은 다음과 같은 ECA(Event-Condition-Action)의 형식으로 제약조건과 능동규칙을 표현되고 있다[9].

on *event*
if *condition*
then *action*

하지만, 본 논문에서는 표현의 단순화를 위해 다음과 같은 일차 논리의 형태로 표현한다.

PR) : condition → action

이것은 조건이 만족하면 행동이 실행됨을 의미한다. 여기서 행동은 위에서 가정했던 바와 같이, UPDATE, INSERT, DELETE만을 가진다.

4.2 확장된 클래스형 제시

본 장에서는 2장의 클래스형을 확장하여 제약조건 및 능동규칙을 병합하여 객체지향 데이터베이스에서 클래스(class)를 선언하려 한다.

```
class class_name {
        OID        : oid-type
        attribute (
                              :
        ),
        constraint (// 무결성제약조건에 관한 rule
                              :
        ),
        rule (                // 능동규칙에 관한 rule
                              :
        ),
        method (              // method
                              :
        )
}
```

이 클래스 구조는 기존의 클래스 구조에서 제약조건과 능동규칙을 포함하는 형태이다. 여기서, 제약조건부분은 2.1절에서 소개되었고, 능동규칙은 4.1절에서와 같이 일차 논리로 표현하여 구성하고 있다. 이 새로운 클래스형은 다음과 같은 장점이 기대된다: 1) 설계가 간단하다. 즉, 규칙들의 추가, 수정, 삭제 등의 작업을 간단하게 처리할 수 있다. 2) 기존 객체지향 개념의 장점을 그대로 적용할 수 있다. 3) 의미 파악이 용이하다. 4) 객체의 표현력이 증가한다.

4.3 예 제

예제 4.1 예제 1.2의 능동규칙을 위와 같은 확장된 클래스 형태로 표현하면 다음과 같다.

```
Class EMP {
   OID                :          oid-type,
   attribute (
         name         :          string(20),
         sex          :          { M , F },
         salary       :          int,
         years_of_expr:          int,
         tex_rate     :          float,
         num_depn     :          int ),
   constraint (
         IC1 : (years_of_expr<4) → (salary<50K)
         IC2 : (salary>=50K) → (tex_rate>0.15) ),
   rule (
         PR1 : (salary>=50K)&(num_depn>=3)
                              → (tex_rate:=0.20)
         PR2 : (years_of_expr>20) → DELETE(OID)
         PR3 : (salary>=80K) →
               INSERT('high_paid' object(name,salary)) )
}
```

```
class ENGINEER {
   OID          :          oid-type,
   inherit                 EMP;
   attribute (
         dept         :          string(20),
         job_time_day:           int ),
   constraint (
         IC1 : (dept="반도체") → (years_of_exp>=3)
         IC2 : (salary<40K) → ),
   rule (
         PR1 : (years_of_expr>=10)
                              → DELETE(OID)
         PR2 : (salary>=50K)&(years_of_exp<2)
                              → (job_time_day:=10) )
}
```

```
class SECRETARY {
   OID        :          oid-type,
   inherit            EMP;
   attribute (
         typing_speed:          int ),
   constraint (
         IC1 : (sex!="F") →
         IC2 : (typing_speed<300) →
         IC3 : (salary<30K) → ),
   rule (
         PR1 : (typing_speed< =350)→(salary<=45k) )
}
```

위에서 보여주는 바와 같이, 각 클래스에 해당하는 제약조건과 능동규칙이 기존의 속성(attribute)과 방법(method)과 함께 클래스에 캡슐화(encapsulation)되어 있음을 알 수 있다. □

4.4 능동규칙을 갱신 표현으로 전환하기

능동규칙 PR은 데이터베이스의 상태를 변화시킬 수 있다. 4.1에서 언급한 능동규칙에 대한 형식을 고려하자. (PR) : condition → action

여기서 행동(action)부는 수정, 추가, 삭제만이 가능하다고 앞서 가정하였다. 만약, 행동부가 갱신에 관한 것이라면 다음과 같은 형식이 된다.

(PR) : condition → Class.UPDATE(value)

이를 다시 SQL의 형식으로 표현하면 다음과 같다.

```
UPDATE        Class
SET           value
PRECOND EXIST     PR's condition
```

능동규칙은 규칙의 왼쪽, 오른쪽 모두가 조건으로 이루어진 제약조건과는 달리, 규칙의 왼쪽만 조건이고, 오른쪽은 연산자의 실행 부분이다. 따라서, 사후조건 부분은 능동규칙에서는 의미가 없어진다. 따라서, 'PRECOND'는 기존의 형식인 'WHERE'로 대체하여도 무방하게 된다.

예제 4.2 예제 4.1의 EMP객체에 대한 능동규칙중 PR1은 임금이 50K이상이고, 부양가족이 3명 이상이면, 세율을 20%로 한다는 내용이다. 이를 일차 논리로 표현하면 다음과 같다.

EMP.PR1 : (salary ⟩= 50K)&(num_depn ⟩= 3)→
(tex_rate : = 0.20)

이것을 SQL의 형식으로 바꾸면 다음과 같다.

```
UPDATE        EMP
SET           tex_rate : = 0.20
WHERE EXIST (SELECT       *
                FROM EMP
     WHERE (salary ⟩= 50K)AND (num_depn ⟩= 3))
```

능동규칙은 제약조건과는 달리 데이터베이스의 상태를 변화시킬 수 있기 때문에, 갱신의 전파를 유발시킨다. 이것은 다음 장에서 다루어 본다. □

5. 갱신의 전파

데이터베이스의 상태가 변하면, 능동데이터베이스의 규칙들은 사용자의 간섭없이 실행한다. 즉, 데이터베이스 상태 변화는 규칙을 트리거시키고, 더 나아가 규칙의 행동 부분을 실행한다. 간단히 예를 들어, 능동규칙 PR_k가 관계형 데이터베이스에서 갱신 U에 의해 트리거되었다고 가정하자. 그렇게 되면, 주어진 U는 다음과 같이 일련의 갱신으로 표현된다.

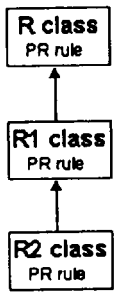
```
UPDATE    relation
SET       assignment in U
WHERE     the condition of U
UPDATE    relation
SET       assignment in PR_k
WHERE     the condition of PR_k
```

이를 객체지향 데이터베이스의 일반화에 대해 적용을 하자면, 2장의 제약조건에서와 같이 갱신 언어에 대한 확장이 요구된다. 그 방법은 다음과 같다.

5.1 적용 방식

아래와 같은 일반화 클래스에 간단한 사용자의 갱

신이 요구되었을 때 데이터 변화 전파에 의거하여 전체 데이터베이스의 일관성을 관리하기 위한 방법을 제시하려고 한다. 3장에서와 같이 일반화 계층에서 나타나는 데이터 변화의 3가지 경우에 대하여 알아보려 한다.



(그림 2) 능동규칙을 가진 일반화된 클래스
(Fig.2) Class generalization with Active Rules

5.1.1 수퍼클래스 R에 대해 갱신이 가해질 경우

```
UPDATE    R
SET       value
```

능동규칙에 의한 갱신은 수퍼클래스의 능동규칙들만을 가지고 구성한다.

```
UPDATE    R
SET       value
UPDATE    R
SET       value
WHERE     R-PR's Condition
```

이렇게 구성된 일련의 갱신 언어로 수퍼클래스의 갱신 작업을 시작한다. 수퍼클래스에서의 갱신 작업이 끝나면 바로 아래의 서브클래스, 즉, R1 클래스로 작업 영역을 옮긴다. 즉, (그림 2)에 대해 적용할 경우, "UPDATE R"문을 "UPDATE R1"문으로 바꾸어 실행하는 것이다. 여기서, 능동규칙에 대한 수정과 조건 부분인 "SET-WHERE" 부분은 계속 유지된다.

```
UPDATE    R1
SET       value
UPDATE    R1
SET       value
WHERE     R-PR's Condition
```

R1클래스에 대한 갱신 작업이 끝나면, 같은 방법으로 그 아래의 서브클래스, 즉, R2 클래스로 작업 영역을 옮긴다.

```
UPDATE    R2
SET       value
UPDATE    R2
SET       value
WHERE     R-PR's Condition
```

5.1.2 임의의 서브클래스(ex)(그림 2)의 R1 클래스)에 대해 갱신이 가해질 경우

```
UPDATE    R1
SET       value
```

능동규칙에 의한 갱신은 그 서브클래스의 수퍼클래스를 찾아 슈퍼클래스에서 차례로 서브클래스로 내려오면서 관련된 능동규칙 수만큼의 "UPDATE-SET-WHERE"가 모두 추가된다. 이 경우, 수퍼클래스에서 내려오면서 찾아지는 능동규칙의 순서대로 갱신 작업에 대해 SQL의 형태를 만든 후, 사용자의 질의 뒤에 추가시키는 형식으로 진행한다.

```
UPDATE    R1
SET       value
UPDATE    R1
SET       value
WHERE     R-PR's Condition
UPDATE    R1
SET       value
WHERE     R1-PR's Condition
```

해당 서브클래스의 갱신 작업이 끝나면, 각 능동규칙에 대한 "SET-WHERE"부분은 그대로 유지하면

서, 그 아래의 서브클래스로 작업영역을 옮긴다. 즉, (그림 2)에 대해 적용할 경우, R1 클래스에서 갱신 작업이 끝나면, "UPDATE R1"문을 "UPDATE R2"문으로 바꾸어 실행하는 형식으로 진행한다.

```
UPDATE   R2
SET      value
UPDATE   R2
SET      value
WHERE    R-PR's Condition
UPDATE   R2
SET      value
WHERE    R1-PR's Condition
```

5.1.3 모든 클래스 계층을 통합하였을 때의 갱신 작업(ex)SELECT R FROM R)

```
UPDATE    SELECT R FROM R
SET       value
```

이 경우도 제약조건 때와 같이 먼저 수퍼클래스의 능동규칙들로 갱신 언어를 만들어 내어 작업한 후, 아래 서브클래스로 내려갈 때마다 각 클래스가 가지고 있는 능동규칙들에 대한 갱신 언어를 만들어 내어 추가시키는 형식으로 진행한다. (그림 2)로 처리 방법을 설명하면 다음과 같다.

우선, R 클래스의 능동규칙을 다음과 같이 갱신 언어로 바꾸어 사용자 질의에 추가시켜 수퍼클래스에 대해 갱신 작업을 진행한다.

```
UPDATE   R
SET      value
UPDATE   R
SET      value
WHERE    R-PR's Condition
```

R 클래스에 대한 갱신 작업이 끝나면, 작업 영역을 수퍼클래스의 서브클래스인 R1 클래스로 옮기고, 다음과 같이, R1의 능동규칙을 갱신 언어로 바꾸어 R 클래스의 갱신 작업시의 갱신 언어에 추가시켜 갱신 작업을 진행한다.

```
UPDATE   R1
SET      value
UPDATE   R1
SET      value
WHERE    R-PR's Condition
UPDATE   R1
SET      value
WHERE    R1-PR's Condition
```

R1 클래스에 대한 갱신이 끝나면, 다음과 같이, 그 아래의 클래스인 R2 클래스로 작업 영역을 옮겨 R2 클래스의 능동규칙을 갱신 언어로 바꾸어 R1 클래스의 갱신 작업시의 갱신 언어에 추가시켜 갱신 작업을 진행한다.

```
UPDATE   R2
SET      value
UPDATE   R2
SET      value
WHERE    R-PR's Condition
UPDATE   R2
SET      value
WHERE    R1-PR's Condition (계속)
UPDATE   R2
SET      value
WHERE    R2-PR's Condition
```

결국, 이 작업은 슈퍼클래스에서 모든 서브클래스로 범위를 이동시키면서, 능동규칙에 대한 갱신 작업을 추가하는 작업을 반복한다.

5.2 예 제

예제 5.1 4.2절에서 제시된 클래스의 선언에 대하여 예제 3.2와 같이 사원들의 임금을 10% 인상한다는 질의가 주어졌다고 가정하자.

```
UPDATE    Class
SET       salary := salary * 1.1
```

임금 갱신에 대해 관련되는 능동규칙들은 다음과 같다.

· EMP : PR1, PR3
· ENGINEER : PR2
· SECRETARY : PR1

5.2.1 EMP클래스(수퍼클래스)에 갱신이 가해지는 경우

위의 갱신문이 EMP클래스에 요구되었을 때를 고려하자. 주어진 갱신문에 의한 데이터 변화와 그의 연속적인 전파에 대한 일관성을 관리하기 위하여, 다음과 같이 EMP클래스가 가지고 있는 능동규칙중, 임금 갱신과 관련 있는 PR1과 PR3에 대한 갱신 언어를 만들어 추가시킨다.

```
UPDATE    EMP
SET       salary := salary * 1.1

UPDATE    EMP
SET       tex_rate : = 0.20
WHERE EXIST (SELECT    *
             FROM      EMP
   WHERE (salary ) = 50K) AND (num_depn ) = 3))

INSERT       HIGH_PAID
VALUE        name, salary
WHERE  EXIST
  (SELECT    *
  FROM       EMP
  WHERE      salary > = 80K)
```

이와 같은 연속적인 갱신, 갱신, 삽입문을 결국 EMP클래스의 일관성을 관리하게 된다. 그러나, 이로 인한 데이터 변화가 EMP의 서브클래스에서 선언된 또 다른 능동규칙 (예, ENGINEER에서의 PR2)을 실행시킬 수도 있다. 예컨대, EMP클래스에 대한 갱신 작업이 끝나면, 다음과 같이 갱신 범위를 EMP클래스의 서브클래스인 ENGINEER클래스로 바꾸면서, 갱신 작업은 ENGINEER클래스에 대해 일어난다.

```
UPDATE       ENGINEER
SET          salary : = salary * 1.1

UPDATE       ENGINEER
SET          tex_rate : = 0.20
WHERE EXIST (SELECT    *
             FROM      ENGINEER
             WHERE (salary ) = 50K) AND
             (num_depn ) = 3) AND (years_of_exp ( 2))
INSERT       HIGH_PAID
VALUE        name, salary
```

위에서 살펴보듯이, 두 번째 UPDATE문은 조건절이 확장 (EMP의 PR1과 ENGINEER의 PR2의 병합)되었음을 알 수 있다. ENGINEER클래스에 대한 갱신 작업이 끝나면, ENGINEER의 서브클래스가 더 이상 없기 때문에, 다시 수퍼클래스인 EMP클래스의 다른 서브클래스인 SECRETARY객체로 범위를 옮겨서 같은 방법으로 갱신에 대한 일관성을 관리하게 된다.

5.2.2 ENGINEER클래스(서브클래스)에 갱신이 가해지는 경우

다음과 같이 사용자에 의한 데이터 변화가 일어날 경우를 고려하자.

```
UPDATE    ENGINEER
SET       salary : = salary * 1.1
```

ENGINEER클래스의 수퍼클래스는 EMP클래스이다. 우선은, EMP클래스에서 임금 갱신과 관련 있는 능동규칙인 PR1과 PR3에 대한 갱신 언어를 만들어 차례로 사용자의 갱신 질의에 추가시키고, 그 다음 ENGINEER클래스에서 임금 인상과 관련 있는 능동규칙, PR2를 갱신 언어로 바꾸어 추가시켜 갱신을 ENGINEER클래스에서부터 시작한다. 이 경우, ENGINEER클래스에 대해서 더 이상의 서브클래스는 없기 때문에, 작업은 ENGINEER클래스에 대해서만 발생한다.

```
UPDATE       ENGINEER
SET          salary : = salary * 1.1

UPDATE       ENGINEER
SET          tex_rate : = 0.20
WHERE EXIST (SELECT    *
```

```
                FROM          ENGINEER
                WHERE (salary)=50K) AND
                (num_depn)=3))

INSERT          HIGH_PAID
VALUE           name, salary
WHERE           EXIST
  (SELECT       *
  FROM          ENGINEER
  WHERE         salary>=80K)

UPDATE                ENGINEER
SET             job_time_day:=10
WHERE EXIST (SELECT       *
                FROM          ENGINEER
                WHERE (salary)=50K) AND
                (years_of_exp(2))
```

이와 같은 일련의 갱신문도 역시 간단한 사용자 질의문이 능동데이터베이스의 기능에 의해 데이터의 변화가 전파될 때 일관성을 관리하기 위하여 새로이 UPDATE와 INSERT가 되어지는 방법을 살펴보았다.

5.2.3 (그림 2)의 클래스 계층을 모두 통합하였을 때

```
UPDATE      SELECT EMP FROM EMP
SET         salary:=salary*1.1
```

제약조건으로 일관성을 검증하는 경우와 같이 갱신 작업은 수퍼클래스인 EMP클래스로부터 시작되고, 아래의 서브클래스들로 범위가 이동될 때마다 갱신 언어는 달라진다. 우선, 다음과 같이 EMP클래스에서 임금 갱신과 관련 있는 PR1과 PR3을 갱신 언어로 바꾸어 사용자의 갱신 언어에 추가시켜 갱신 작업을 진행한다.

```
UPDATE      EMP
SET         salary:=salary*1.1

UPDATE      EMP
SET         tex_rate:=0.20
WHERE EXIST (SELECT       *
                FROM EMP
                WHERE (salary)=50K) AND
                (num_depn)=3))

INSERT          HIGH_PAID
VALUE           name, salary
WHERE           EXIST
  (SELECT       *
  FROM          EMP
  WHERE         salary>=80K)
```

EMP클래스에 대한 갱신 작업이 끝나면, 작업 영역을 서브클래스인 ENGINEER클래스로 옮겨야 한다. 우선, EMP클래스에 대한 갱신 작업때 사용하던 갱신 언어에서 "UPDATE EMP"를 모두 "UPDATE ENGINEER"로 바꾸고, 거기에 ENGINEER클래스에서 임금 갱신과 관련된 능동규칙인 PR2를 갱신 언어로 바꾸어 추가시킨 뒤, ENGINEER클래스에 대한 갱신 작업을 시작한다.

```
UPDATE      ENGINEER
SET         salary:=salary*1.1

UPDATE      ENGINEER
SET         tex_rate:=0.20
WHERE EXIST (SELECT       *
                FROM          ENGINEER
                WHERE (salary)=50K) AND
                (num_depn)=3))

INSERT          HIGH_PAID
VALUE           name, salary
WHERE           EXIST
  (SELECT       *
  FROM          ENGINEER
  WHERE         salary>=80K)

UPDATE      ENGINEER
SET         job_time_day:=10
WHERE EXIST (SELECT       *
                FROM          ENGINEER
                WHERE (salary)=50K) AND
                (years_of_exp(2))
```

ENGINEER클래스에 대한 서브클래스가 더 이상 없기 때문에, ENGINEER클래스에 대한 갱신 작업이 끝나면 EMP클래스의 다른 서브클래스인 SECRETARY클래스로 작업 영역을 옮긴다. 이 때, 방법은 ENGINEER클래스에 대한 갱신 언어를 만들어 작업할 때와 동일하다.

이러한 일련의 갱신 표현들은 변화 효과의 전파에 대해 설명하고 있다. 임금 인상에 대한 질의가 들어오면, 나열된 순서대로 각 능동규칙들은 자기의 조건을 비교하고, 만약 만족한다면, 행동을 실행하여 데이터베이스의 일관성을 유지 할 수 있다.

6. 결 론

본 논문은 능동데이터베이스에서의 기존의 제약조건 관리에 대한 새로운 접근 방법[24]을 객체지향 데이터베이스의 특성인 일반화 계층에 대한 적용에 관한 것이다. 간단한 데이터베이스의 변화 (예 갱신문)에 대하여 제약조건과 능동규칙의 의미를 포함함으로써, 변화 전파가 유발하는 전체 데이터베이스의 비일관성을 방지할 수 있다. 본 논문의 내용을 요약하면 다음과 같다.

· 객체지향 데이터베이스에 대해서 새로운 클래스 구조를 제시하였다. 기존의 속성과 방법만 가지던 클래스구조에 제약조건과 능동규칙을 병합하여 표현력이 강해졌다.

· 객체지향 데이터베이스에 대해서 데이터베이스 갱신 해석(calculus)을 발전시켰다. 본 논문에서는 사용자가 지정한 데이터베이스 갱신에 제약조건과 능동규칙을 병합하여 기존의 객체지향 데이터베이스의 의미보다 나은 일련의 갱신으로 변환하였다.

· 갱신에 제약조건과 능동규칙을 병합하여 데이터베이스의 일관성을 효과적으로 유지할 수 있도록 하였다.

· 더 나아가, 데이터 변화에 사후조건으로 제한을 둠으로써, 능동성이 미치는 비일관성과 그로 인한 보수작업을 불필요하게 하였다.

· 객체지향 데이터베이스에서의 갱신의 전파에 대해 살펴보았다. 일반화된 클래스 구조에서는 이러한 전파가 더욱 더 많아진다.

본 논문에서 얻을 수 있는 이점은 다음과 같다.

· 본 논문에서 제안하는 일련의 갱신문은 사용자에게 데이터베이스의 능동성이 어떻게 이루어지는가를 알 수 있게 해 준다.

· 객체지향 데이터베이스에 능동적인 요소를 넣음으로써, 더 강력한 표현력을 가지게 한다.

본 논문에서 논하였던 방법들은 능동데이터베이스에 대해 설계자와 사용자들이 데이터베이스의 변화에 대한 일관성을 관리하고, 조절할 수 있도록 해준다. 특히, 객체지향 데이터베이스에 대해 이러한 능동 데이터베이스의 장점들을 적용하여 객체지향 데이터베이스의 표현력을 더욱 더 강력하게 하였으며, 제약조건과 능동규칙의 적용 방법에 대해 논하였다.

참 고 문 헌

[1] S. Abiteboul and R. Hull. "Update propagation in the IFO database model", In Proceedings of the International Conf. on Foundations of Data Organization, pages 243-251, Kyoto, 1985.

[2] Alexander Aiken, Jennifer Widom, and Jeseph M. Hellerstein. "Behavior of database production rules: Termination, confluence, and observable determinism", In Michael Stonebraker, editor, Proc. ACM SIGMOD Intl. Conf. on Management of Data, pages 59-68, San Deigo, 1992.

[3] F. Cacace, S. Ceri, S. Crespi-Reghizzi, L. Tanca, and R. Zicari. "Integrating object-oriented data modeling with a rule-based programming paradigm", In Hector Garcia-Molina and H. V. Jagadish, editors, Proc. ACM SIGMOD Intl. Conf. on Management of Data, pages 225-236, Atlantic City, NJ, 1990.

[4] Upen S. Chakravarthy, John Grant, and Jack Miller. "Logic-based approach to semantic query optimization", ACM Transactions on Database Systems, 15(2):163-207, June 1990.

[5] S. Ceri and J. Widom. "Deriving production rules for constraint maintenance", In Proc. Intl. Conf. on Very Large Data Bases, pages 650-661, Brisbane, Australia, 1990.

[6] S. Ceri and J. Widom. "Deriving production

rules for incremental view maintenance", In Proc. Intl. Conf. on Very Large Data Bases, pages 577-589, Barcelona, 1991.

[7] U. Dayal, B. Blaustein, A. Buchmann, U. Chakravarthy, R. Ledin, M. Hsu, D. McCarthy, A. Rosenthal, S. Sarin, M. J. Carey, M. Livny, and R. Jauhari. "The HiPAC project: Combining active databases and timing constraints", ACM SIGMOD Record, 17(1): 51-70, March 1988.

[8] Umeshwar Dayal, Meichun Hsu, and Rivka Ladin. "Organizing long running activities with triggers and transactions", In Hector Garcia-Molina and H. V. Jagadish, editors, Proc. ACM SIGMOD Intl. Conf. on Management of Data, pages 204-214, Atlantic City, NJ, 1990.

[9] Umeshwar Dayal, Eric Hanson, Jennifer Widom. "Ch 21. Active Database Systems", MODERN DATABASE SYSTEMS, Kim Won, ACM PRESS, pages 434-456.

[10] G. Gottlob, P. Paolini, and R. Zicari. "Properties and update semantics of consistent views", ACM Transactions on Database Systems, 13(4): 486-524, 1988.

[11] Matthew S. Hecht and Larry Kerschberg. "Update semantics for the functional data model", Technical Report Database Research Report, No. 4, Bell Laboratories, Holmdel, New Jersy, January 1981.

[12] M. Kramer, G. Lausem, and G. Saake. "Updates in a rule-based language for objects", In Proc. Intl. Conf. on Very Large Data Bases, pages 251-262, Vancouver, Canada, 1992.

[13] A. C. Kakas and P. Mancarella. "Database updates through abduction", In Proc. Intl. Conf. on Very Large Data Bases, pages 650-661, Australia, 1990.

[14] R. Kowalski. Logic for data description. In H. Gallaire and J. Minker, editors, "Logic and Data Bases", pages 77-103, New York, 1978. Plenum Press.

[15] Henry F. Korth and Abraham Silberschatz. "Database System Concepts", McGrawHill, Inc, New York. 1991.

[16] Sanjay Manchanda. "Declarative expression of deductive database updates", In Proceedings of the ACM Symposium on Principles of Database Systems, pages 93-100, Philadelphia, 1989.

[17] D. R. McCarthy and U. Dayal. "The architecture of an active database management system", In Proc. ACM SIGMOD Intl. Conf. on Management of Data, pages 215-224, Portland, Oregon, 1989.

[18] Guido Moerkotte and Peter C. Lockemann. "Reactive consistency control in deductive database", ACM Transactions on Databas Systems, 16: 670-702, 1991.

[19] M. Morgenstern. Constraint equations: "Declarative expression of constraints with automatic enforcement", In Proc. of VLDB, pages 111-125, 1984.

[20] M. Morgenstern. "The role of constraints in databases", In Larry Kerschberg, editor, Expert Database Systems, pages 351-368. Benjamin/Cummings, 1986.

[21] Xialolei Qian and Richard Waldinger. "A transaction logic for database specification", In Proc. ACM SIGMOD Intl. Conf. on Management of Data, pages 243-250, Chicago, 1988.

[22] A. Shepherd and L. Kerschberg. "Constraint management in expert database systems", In Larry Kerschberg, editor, Expert Database Systems, pages 309-368. Benjamin/Cummings, 1986.

[23] J. Wildom, R. J. Cochrane, and B. G. Lindsay. "Implementating set-oriented production rules as an extension to Starburst", In Proc. Intl. Conf. on Very Large Data Bases, pages 275-285, Barcelona, Spain, 1991.

[24] Jong P. Yoon and Larry Kerschberg. "Semantic Update Optimization in Active Databases", International Federation Information Proc. Data Bases Application Semantics(DS-6) 6th. Conference, Atlanta, GA, 1995.

김 형 민

성균관대학교 정보공학 석사 및 학사
현재 : LG-soft 근무

김 웅 모

성균관대학교 수학 학사
(미) Old Dominion University 전산학 석사
(미) Northwestern University 전산학 박사
현재 : 성균관대학교 정보공학과 부교수
관심분야 : 능동데이터베이스, 지리정보시스템

윤 종 필

연세대학교 전기공학 학사
(미) University of Florida 전자공학 석사
(미) George Mason University 전산학 박사
현재 : 숙명여자대학교 전산학과 조교수
관심분야 : 능동데이터베이스, 멀티미디어 데이터베이스, 분산데이터베이스, 데이터마이닝