# 분산환경에서 객체지향 능동 규칙 시스템 구현

고 굉 욱[†] · 유 상 봉[††] · 김 기 창[†††] · 차 상 균[††††]


## 요 약

본 논문에서는 이종 분산환경에서 효율적으로 공간 무결성 제약조건을 관리하기 위한 컴파일러 기반의 능동 규칙 시스템을 제시한다. 제시된 프로토타입 능동 규칙 시스템은 특별히 SDBC(Spatial DataBase Connectivity)라는 전체 미들웨어 시스템의 한 콤포넌트(component)로 개발되었다. 이러한 이유로 본 능동 규칙 시스템은 분산환경에서 특정 시스템에 독립적으로 이종 ODBMS들을 위해 제공되며, 공간 무결성 제약조건을 E-C-A(Event-Condition-Action) 형태의 능동 규칙으로 정의하고 관리하기 위해 이용된다. 이러한 능동 규칙 시스템을 사용함으로써 응용 프로그래머는 응용 객체들의 무결성 유지에 대한 과중한 부담으로부터 벗어날 수 있게 된다.

컴파일러 기반 방식에서는 데이터베이스 프로그램에 의해 발생된 이벤트에 적용 가능한 능동 규칙들이 전처리기에 의해 함수 형태로 직접 데이터베이스 프로그램에 삽입된다. 그 다음 데이터베이스 프로그램에 삽입된 능동 규칙 코드들은 응용 프로그램 소스 코드들과 함께 컴파일 된다. 이러한 연구 방식의 한 장점은 전처리 된 프로그램이 실행될 때, 모든 데이터베이스 상태 변화들을 모니터 하는데 수반되는 실행시간 오버헤드가 전혀 존재하지 않는다는 것이다. 또한, 본 능동 규칙 시스템은 변경된 규칙들을 관리하고, 그런 규칙들을 실행시간에 동적으로 해석하기 위한 기능을 제공한다.


# Implementation of Object-oriented Active Rule System in Distributed Environment

Koeng-wook Ko[†] · Sang-Bong Yoo[††] · Ki-Chang Kim[†††] · Sang Kyun Cha[††††]


## ABSTRACT

In this paper we present compiler-based active rule system to efficiently maintain spatial integrity constraints in a heterogeneous, distributed environment. Specially, the prototype active rule system presented has been developed as a component of a whole middleware system called SDBC(Spatial DataBase Connectivity). Due to this reason, our active rule system is provided for heterogeneous ODBMSs in a distributed environment and used to define spatial integrity constraints using the active rules in E-C-A(Event-Condition-Action) type. Using this active rule system, an application programmer can free himself from a heavier burden on the integrity maintenance of application objects.

In the compiler-based approach, active rules applicable to events raised by a database program are directly inserted into the program in a function type by the preprocessor, and then they are compiled with the application program source codes. One advantage of this approach is that there is no run-time overhead accompanied by monitoring all the database transitions when preprocessed program is executed. This active rule system also provides facilities to manage changed rules and dynamically interpret those rules at run-time.


※ 본 연구는 인하대학교의 1998년 교내연구비 지원에 의하여 수행되었음

† 준 회 원 : 인하대학교 대학원 자동화공학과
†† 정 회 원 : 인하대학교 자동화공학과 교수
††† 정 회 원 : 인하대학교 컴퓨터공학과 교수
†††† 정 회 원 : 서울대학교 전기공학부 교수

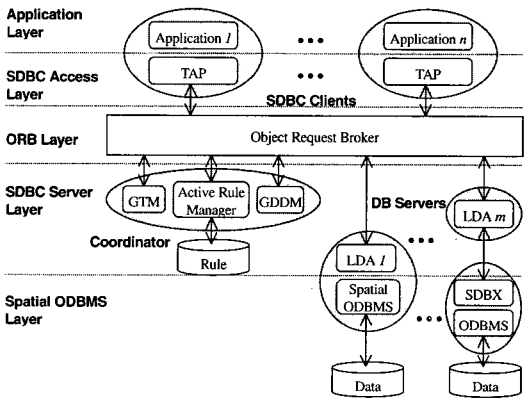
논문접수 : 1999년 1월 28일, 심사완료 : 1999년 10월 4일

## 1. 서 론

사용자들에게 데이터베이스 프로그래밍에 독립적인 지식관리 기능을 제공하려는 노력의 일환으로 데이터베이스 공동체는 무결성 제약조건 관리, 뷰 관리, 작업흐름 관리 등의 응용에 능동 데이터베이스 규칙을 활용하는 많은 연구 및 개발 과제들을 수행하였다[1]. 그러나 기존의 능동 데이터베이스 시스템들은 다른 문법과 의미론을 갖는 능동 규칙들로 설계 및 구현되었기 때문에, 분산환경에서 다수의 이종 데이터베이스 시스템들을 갖는 응용들(예를 들어, IDE [2], 데이터 웨어하우스 [3])을 구축하기가 어렵다.

본 논문의 목적은 이종 분산환경에서 효율적으로 공간 객체의 무결성을 확인하기 위한 능동 규칙 시스템을 제공하는 것이다. 이러한 목적을 달성하기 위해 본 능동 규칙 시스템은 CORBA [4]와 ODMG [5]의 표준들을 활용하여 구축된 ODBMS를 위한 미들웨어 시스템인 SDBC의 한 콤포넌트로서 제공된다. SDBC는 프로그래머에게 데이터베이스와 프로그래밍에 대한 단일의 객체지향적 관점을 제공함으로써 데이터베이스의 물리적 위치 및 ODBMS 프로세스 구조와 같은 특정 구현에 따른 개별 데이터베이스의 상세정보를 프로그래머가 알고 있어야 하는 부담을 줄여준다. 이러한 투명성에 대한 핵심 해결책은 OpenGIS [6]에 기초한 공간 데이터 타입들 및 연산자들로 ODMG 객체 모델 [5]을 확장하는 것이다. 부가적으로, SDBC는 다수의 이종 데이터베이스들을 액세스하는 SDBC 트랜잭션들의 일관성 보장을 위한 전역관리 기능들을 통합한다. SDBC에서 능동 규칙 시스템은 SDBC의 전역관리를 위한 한 구성요소로서 공간 객체 데이터의 무결성 정의 및 관리에 적용된다. SDBC의 전체 구조는 (그림 1)과 같다.

현재 상용 데이터베이스 시스템들은 고비용의 이유로 데이터 무결성 확인을 위해 단지 제한된 지원만을 하고 있다. 그러나, 공유 데이터의 무결성이 적절히 유지되지 않는 한 전체 시스템의 행위는 예측될 수 없으므로 무결성 관리는 특히 공조적 엔지니어링 환경들(예를 들어, GIS, CAD/CAM, CIM 등)에서 그 중요성이 강조되고 있다. 일반적으로 데이터 무결성 확인에 적용 가능한 서비스가 전혀 없을 때, 응용 프로그래머가 응용 프로그램에 제약조건 확인을 위한 코드를 삽입해야 한다. 그러나 무결성 제약 조건들의 수와 복잡성이 증가할수록 이런 방식은 응용 프로그래머들에게 제약조건에 대한 지식획득, 코딩, 그리고 코드의 최적화 측면에서 과중한 부담을 주게 된다. 이런 문제에 대한 적절한 해결책은 프로그래머를 데이터 무결성 확인의 책임으로부터 벗어나게 하고 별도의 시스템이 독립적으로 데이터 무결성 확인을 다루게 하는 것이다. 무결성 제약 조건들은 관련 영역에 관해 상세한 지식을 가진 전문가들에 의해 수집되어 규칙들의 집합으로 규칙 베이스에 저장될 수 있다. 그 다음, 무결성 제약조건 관리시스템은 응용 프로그램의 데이터베이스 액세스 활동을 감시하고 필요할 때 제약 조건들이 만족되는 지를 확인할 수 있다.




(그림 1) SDBC의 레이어 구조

무결성 제약조건은 E-C-A(Event-Condition-Action) 규칙으로 표현되고 독립적으로 유지될 수 있으므로, 새로운 규칙이 삽입될 때 규칙들 간의 일관성을 확인하는 것이나 규칙의 복잡한 컨디션 부분들에 대한 계산 시간을 최적화하는 것이 가능하다. 그러나 무결성 제약 조건들을 적절히 실행하기 위해, 무결성 제약조건 관리자는 응용 프로그램을 지속적으로 감시해야 한다. 데이터베이스 연산이 있을 때마다 무결성 제약조건 관리자는 해당되는 규칙이 규칙 베이스에 존재하는지를 확인하고, 있을 경우 그 규칙을 적용해야 한다. 이러한 작업은 실행시간에 발생하므로 데이터베이스 액세스를 상당히 지연시킬 수 있다. 본 연구에서는 이러한 실행시간 오버헤드를 줄이기 위해 전처리기를 이용하여 응용 프로그램 내에 미리 제약조건 확인 코드를 삽입함으로써 실행시간 동안 무결성 제약조건 관리

시스템의 간섭을 제거하는 방식을 사용한다. 이는 규칙과 응용 프로그래머의 독립성을 보장하고 더 빠른 실행을 가능하게 한다. 컴파일 방식의 한 단점은 컴파일 된 코드에 상응하는 규칙들이 규칙 베이스에서 변경될 때 역시 변경되어야 한다는 것이다. 본 논문은 이런 경우에 응용 프로그램을 다시 컴파일하는 대신 컴파일 된 규칙 코드의 변경된 규칙 코드에 대한 동적해석 방법을 제시한다(상세한 설명을 위해 4.3절 참조).

본 논문은 다음과 같이 구성된다. 2절에서는 관련 연구들을 언급하고, 3절에서는 능동 규칙 정의 언어와 의미론이 약간의 예제들과 함께 다루어진다. 4절에서는 능동 규칙 관리자에 관해 설명한다. 5절에서 규칙 코드 삽입을 위한 컴파일 절차를 상세히 설명하고, 또한 변경된 규칙의 실행시간 해석에 관해 설명한다. 마지막으로 6절에서 결론을 언급한다.

## 2. 관련 연구

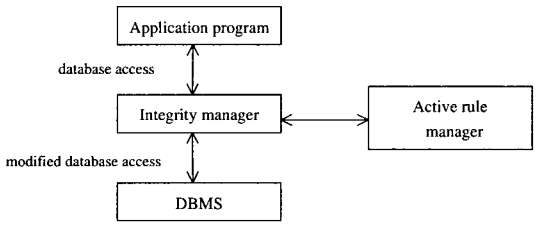
규칙 베이스에 규칙을 저장하고 이에 기반하여 무결성 제약조건을 확인하는 것은 학계와 산업계의 많은 연구 과제들에 의해 연구된 방식이다[8, 9, 10]. 대부분의 연구들이 주로 규칙 실행 알고리즘의 세 측면(즉, 컨디션 최적화, 규칙 언어 정의, 그리고 효율적인 이벤트 모니터링)에 집중되었다. 규칙 언어에 관한 연구는 [8, 12] 등에서 발표되었다. [8]은 능동 데이터베이스를 위해 유연한 실행 모드들을 갖는 규칙 언어를 서술한다. 이 언어는 단순 트리거링 이벤트 뿐만 아니라 복합 트리거링 이벤트의 표현 방법을 제공한다.

효율적인 이벤트 모니터링 기법들은 [8, 12, 13]에서 보고되었다. [12]는 신속한 이벤트 탐색을 지원하기 위해 모니터링 시스템을 직접 데이터베이스 시스템의 커널 속으로 통합하였다. [13]은 클래스 설계자가 데이터베이스 스키마를 설계할 때 규칙들을 정의하기 위해 사용할 수 있는 언어들을 제안하였다. 규칙들은 상응하는 클래스 정의들 내부에서 정의되고, 응용 프로그래머는 무결성 제약조건을 확인하거나 어떤 규칙 코드를 트리거하는 것에 관해 관여하지 않는다. 규칙 처리의 실제 구현(즉, 이벤트 탐색, 컨디션 평가, 그리고 액션 코드의 호출)은 컴파일러가 생성한 stub 코드에 의해 이루어진다. 그러나 규칙의 선언이 응용 프로그램 속으로 하드코딩 되므로, 규칙을 효율적으로 관리하기 힘들고 규칙의 변화는 곧 데이터베이스 스키마의 재설계를 의미한다. [8]은 규칙관리를 응용 프로그램으로부터 분리하였다. 외부 이벤트 탐색자가 이벤트 탐색을 위해 응용 프로그램을 감시하며, 규칙 관리자, 객체 관리자, 그리고 컨디션 평가자와의 상호작용에 의해 해당 규칙을 실행한다. [8]은 규칙처리 코드를 응용 프로그램으로부터 분리하였으므로 [13]에 비해 규칙 컨디션들의 최적화와 규칙관리를 더 효율적으로 할 수 있다. 그러나 이벤트 탐색자는 지속적으로 응용 프로그램을 감시해야 하고, 이벤트가 검출되면 해당 규칙은 실행시간에 잠재적으로 방대한 규칙 베이스에서 검색되어야 한다.

본 논문의 연구방식은 응용 프로그램에서 규칙과 관련된 코드를 생성하기 위해 컴파일러(또는 전처리기)를 사용한다는 점에서 [13]의 연구방식과 유사하다. 그러나 응용 프로그램을 규칙 베이스와 완전히 분리한다는 점에서 다르다. 본 연구방식에서 규칙은 응용 프로그램에 무관하게 정의되고 변경될 수 있다. 이러한 분리성은 시스템 개발자에게 더 효율적인 규칙관리 기법들을 허용하고, 응용 프로그래머에게는 규칙과 관련된 코딩으로부터 완전히 자유롭게 해준다. 이를 가능하게 하기 위해, 본 연구의 전처리기는 더 많은 작업을 수행한다. 전처리기는 지속 객체들에 관련된 모든 가능한 이벤트들을 탐색하고, 규칙 베이스에 있는 적용 가능한 규칙들을 능동 규칙 관리자로부터 전달 받아 적절한 곳에 규칙 확인 코드를 삽입한다. 또한 본 연구의 이벤트 모니터링 방식은 프로그램의 지속적인 모니터링이 필요 없다는 것과 실행시간 동안 이벤트와 관련된 규칙들을 (규칙의 변경이 전혀 없을 경우) 검색하지 않아도 된다는 점에서 [8]의 모니터링 방식과 구별된다. 본 연구의 전처리기는 응용 프로그램이 그 내부에 자체로 필요한 모든 규칙들을 포함하고 이벤트가 발생할 때마다 적당한 규칙 코드를 실행할 수 있도록 응용 프로그램을 수정한다. 그와 동시에 변경된 규칙의 실행시간 해석을 지원하기 위한 코드들도 응용 프로그램 속에 함께 삽입한다.

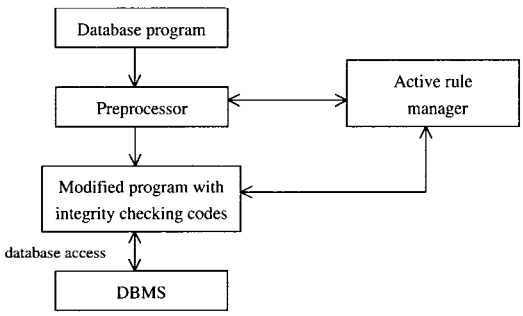
일반적으로 능동 데이터베이스 시스템을 위한 구조는 레이어 구조(layered architecture), 내장 구조(built-in architecture), 그리고 컴파일 구조(compiled architecture)와 같이 세가지 범주로 분류될 수 있다[1]. 층상 및 내장 구조에서 무결성 관리자는 (그림 2)와 같이 데이터베이스 시스템과 응용 프로그램 사이에 위치한다. 무결성 관리자는 응용 프로그램의 모든 데이터베이스 연

산들을 중간에서 가로채어 상응하는 규칙 코드들을 실행하며, 만일 제약 조건들이 만족되면 데이터베이스 시스템에 그 연산들을 전달한다. 무결성 관리자는 상응하는 규칙들을 찾기 위해 능동 규칙 관리자와 상호작용하며, 능동 규칙 관리자는 규칙들의 삽입, 변경 및 삭제에 관한 책임을 맡는다.




(그림 2) 실행시간 모니터링 모듈을 갖는 능동 데이터베이스 시스템의 구조

본 연구에서는 미들웨어 환경과 이종 ODBMS를 고려하므로 (그림 3)에서 보여지는 바와 같이 무결성 제약조건 관리자는 제거된다. 응용 프로그램은 적절한 제약 조건들의 확인에 필요한 코드들을 갖기 위해 전처리기에 의해 수정되고 데이터베이스 시스템에 직접 액세스할 수 있게 된다. 전처리 된 코드들은 포함된 규칙들을 확인하기 위해 실행시간에 능동 규칙 관리자와 상호작용한다. 만일 포함된 어떤 규칙이 변경되면, 실행 프로그램은 시효가 지난 규칙은 무시하고 능동 규칙 관리자로부터 전달된 변경된 규칙들을 동적으로 해석한다.




(그림 3) 컴파일 방식의 컴파일 및 실행 흐름

## 3. 능동 규칙

### 3.1 능동 규칙 정의 언어

능동 규칙은 이벤트(Event), 컨디션(Condition), 액션(Action)의 세가지 주요부분과 부가적인 변수선언 및 질의 부분으로 구성된다. 본 논문에서 능동 규칙은 하나 이상의 특정 클래스에 대해 정의되는 목표설정 규칙이며, 문법적으로 FOR 절로 정의된다. 본 능동 규칙 언어는 선언적(declarative) 언어와 절차적 언어의 특성을 모두 가진다. 선언적 언어 요소는 컨디션 구문과 객체 질의문을 위해 사용되고, 절차적 언어 요소는 주로 액션 구문을 위해 사용된다. 본 능동 규칙 언어의 문법 정의는 다음과 같다.

```
active_rule = rule_head  event_stmt  [declaration]  condition_stmt
              action_stmt END ;
rule_head = DEFINE rule_mode RULE rule_id FOR ( class_name
            {, class_name} ) ;
rule_mode = INSTANCE-ORIENTED
          | SET-ORIENTED
event_stmt = EVENT trigger_event ;
declaration = var_decl [ query ]
var_decl = VAR {var_name {, var_name} : [ SET OF ] class_name
           ;} ;
query = var_name := QUERY (temp_var from_clause where_clause) ;
from_clause = FROM class_name
where_clause = WHERE  logical_expression  {connector  logical_
               expression}
condition_stmt = CONDITION condition_stmt ;
action_stmt =ACTION action_stmt ;
```

능동 규칙에서 이벤트는 데이터베이스의 지속 객체들에 대해 상태변화를 야기하는 연산들(즉, INSERT, UPDATE, 그리고 DELETE에 해당하는 데이터베이스 연산들) 중의 하나로 정의된다. 능동 규칙의 컨디션은 규칙에 정의된 액션을 취하기 전에 먼저 평가되어야 할 조건들을 기술한다. 컨디션 술어(predicate)들은 AND 또는 OR에 의해 재귀적으로 연결될 수 있다. 규칙에서 액션은 내장 함수들 또는 사용자 정의 함수들과 같은 절차적 연산으로 정의된다. 그리고 규칙의 액션으로 데이터베이스 트랜잭션 연산인 ABORT가 정의될 수 있으며, 이 연산의 호출에 의해 현재의 트랜잭션에 의한 모든 데이터베이스 상태변화는 취소된다.

본 능동 규칙 정의 언어를 이용하여 정의될 수 있는 공간 무결성 제약조건을 위한 규칙들의 예는 다음과 같다.

**[예 1]** 이 예에서 다음의 규칙은 만일 사용중인 하수도관이 삭제될 경우 새로운 하수도관을 생성하여 삭제된 관을 대체할 것을 규정한다.

```
DEFINE INSTANCE-ORIENTED RULE R1 FOR(SewerPipe) ;
VAR sp, new_sp: SewerPipe ;
```

```
EVENT DELETE(sp) ;
CONDITION sp.usage == TRUE ;
ACTION replace(new_sp, sp) ;
END_RULE ;
```

**[예 2]** 다음의 규칙은 만일 하수도관의 타입이 Transport이거나 재질이 Steel일 때 관경이 350보다 작으면 두 경우 모두 관경을 350으로 변경할 것을 규정한다.

```
DEFINE INSTANCE-ORIENTED RULE R2 FOR(SewerPipe) ;
VAR sp: SewerPipe ;
EVENT INSERT(sp) ;
CONDITION (sp.type = Transport OR sp.material = Steel)
                    AND sp.diameter < 350 ;
ACTION update(sp.diameter, 350) ;
END_RULE ;
```

**[예 3]** 이 예에서는 하수도 관들의 전체수가 MAX_NUM을 초과하지 않도록 제한하는 규칙을 정의한다. 이 규칙은 만일 하수도 관들의 전체수가 MAX_NUM을 초과하면, 현재의 트랜잭션을 취소할 것을 정의한다.

```
DEFINE SET-ORIENTED RULE R3 FOR(SewerPipe) ;
VAR sp: SewerPipe ;
EVENT INSERT(sp) ;
CONDITION count(SewerPipe) > MAX_NUM ;
ACTION ABORT ;
END_RULE ;
```

**[예 4]** 다음의 규칙은 하수도 관의 타입이 Transport이면 그 재질이 Steel이어야 하는데 그렇지 않은 경우로 변경되면, 변경된 하수도 관을 삭제할 것을 정의한다.

```
DEFINE INSTANCE-ORIENTED RULE R4 FOR(SewerPipe) ;
VAR sp: SewerPipe ;
EVENT UPDATE(sp) ;
CONDITION sp.type = Transport AND sp.material != Steel ;
ACTION remove(sp) ;
END_RULE ;
```

### 3.2 능동 규칙의 의미론

능동 규칙은 instance-oriented 또는 set-oriented 규칙으로 정의된다. Instance-oriented 규칙은 단일 인스턴스에 대해 적용되고, set-oriented 규칙은 같은 클래스 또는 다른 클래스들에 속한 인스턴스들의 집합에 대해 적용된다. 이 둘의 구별은 사용자들이 규칙을 정의할 때 하게 되며, 최적의 코드 생성을 위해 전처리기는 이러한 구별을 고려한다.

본 논문에서는 데이터베이스의 무결성이 트랜잭션 초기에 유지되고 트랜잭션이 완료 되기 전에 보장되어져야 한다고 가정한다. 이는 데이터베이스의 무결성이 트랜잭션 동안 일시적으로 위배될 수 있음을 의미한다. 예를 들어, 설계 객체에 대한 한 쌍의 잘라내기 및 붙여넣기(cut-and-paste) 연산 도중에 객체의 소멸은 일시적으로 설계 규칙들을 위배할 수 있다. 본 연구에서는 무결성 규칙들의 실행을 트랜잭션의 완료지점까지 연기함으로써 무결성 규칙에 대한 시간적 위배를 허용한다.

규칙의 액션들은 다른 규칙들을 트리거 할 수 있다. 규칙의 액션 부분의 실제 실행은 실행시간에 규칙의 컨디션 부분의 계산결과에 의해 결정된다. 이벤트 부분이 활성 규칙(triggered rule)들의 액션 부분과 일치하는 어떤 규칙도 실행시간에 트리거 될 수 있다. 그러므로 전처리기는 응용 프로그램에 실제의 트리거링을 결정하기 위한 코드들을 삽입해야 한다. 이러한 이유로 규칙들 간의 트리거링 관계는 사이클을 형성할 수 없다. 그렇지 않으면, 컴파일 시간에 코드 생성의 무한 루프를 초래한다.

## 4. 능동 규칙 관리자

### 4.1 트리거링 그래프

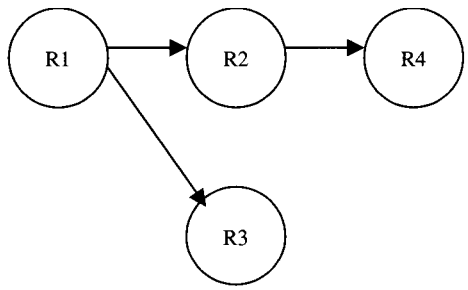
능동 규칙 언어에 의해 정의된 능동 규칙들은 규칙 베이스에 저장되고 관리된다. 사용자 데이터베이스 프로그램이 컴파일 될 때, 프로그램에 정의된 이벤트들과 관련된 규칙들이 규칙 베이스에 있는지 규칙 관리자에게 문의한다. 규칙 관리자는 이벤트 타입을 받으면 그 이벤트에 의해 트리거 되는 모든 후보 규칙들을 검색한다. 이러한 후보 규칙들은 컴파일러에게 전달되고 사용자 프로그램에 적절한 코드로 삽입된다. 일단 규칙들이 삽입되고 나면, 컨디션들의 검사 및 무결성 확인은 프로그램 실행시간에 실행된다. 능동 규칙들 간의 트리거링 관계를 표현하기 위해 트리거링 그래프를 구성하고 관리한다.

**[정의 1]** *트리거링 그래프*는 directed graph $G(V, E)$로 표현되며, $V$는 vertex들의 집합이고 $E$는 directed edge들의 집합이다. 규칙 베이스에서 각 규칙은 vertex로 표현된다. 만일 vertex

*v1*에 해당하는 규칙의 액션이 vertex *v2*에 해당하는 규칙의 이벤트로 고려될 수 있으면 *v1*에서 *v2*를 향한 edge *e1*이 존재한다.

트리거링 그래프에 관한 자세한 분석은 [7]에서 논의된다. 본 연구에서는 컴파일 시간에 사용자 프로그램 속으로 규칙들이 삽입되기 때문에, 트리거링 그래프에서 사이클이 허용되지 않는다. 그렇지 않으면, 사이클들을 따라 코드들이 무한히 생성되는 결과를 초래한다.

**[예 5]** 예제 1, 2, 3, 4에서 정의된 규칙들에 대한 트리거링 그래프를 도시하면 다음의 (그림 4)와 같다.




(그림 4) 예 1, 2, 3, 4에 정의된 규칙들에 대한 트리거링 그래프

규칙 베이스는 상용 ODBMS인 Objectivity/DB 상에서 구현되었다. 규칙이 주어지면 파싱 되어 기존의 트리거링 그래프와 병합된다. 그래프의 신속한 검색을 위해, 능동 규칙들은 Objectivity에서 제공하는 동적 해싱을 이용한 인덱스를 지원하는 Keyed Object 형태로 규칙 베이스에 저장된다. 규칙 베이스는 데이터베이스 프로그램 전처리기의 요청들에 응답한다. 전처리기는 이벤트 타입을 능동 규칙 관리자에게 전달하고, 능동 규칙 관리자는 활성 규칙들이 존재할 경우 되돌려준다. 이벤트에 해당하는 활성 규칙이 존재하지 않을 때 능동규칙 관리자는 NO_TRIG_RULE이란 메시지를 되돌려준다.

### 4.2 규칙의 변경관리

데이터베이스의 인스턴스들처럼 규칙 베이스의 규칙들은 변경될 수 있다. 새로운 규칙들이 생성될 수 있고 기존의 규칙들은 변경되거나 삭제될 수 있다. 어떤 규칙들이 변경될 때, 트리거링 그래프는 따라서 변경된다. 컴파일러 기반 방식에서 만일 규칙 베이스가 변경되면 프로그램은 시효가 지날 수 있다. 프로그램에 포함된 규칙들 중 단 하나의 규칙이 변경될 지라도 전체 프로그램은 그 효과를 반영하기 위해 다시 컴파일 되어야 한다. 이러한 오버헤드를 피하기 위해, 본 연구에서는 변경된 규칙들의 동적 해석 기능을 제공한다. 컴파일 된 프로그램은 능동 규칙 관리자와 상호 작용하고 변경된 규칙이 있을 경우 전달 받는다. 이러한 변경된 규칙들은 실행시간에 동적으로 해석된다.

동적 해석 기능을 지원하기 위해 능동 규칙 관리자는 타임스탬프 정보를 저장함으로써 각 규칙에 대한 최종 변경을 계속 관리한다. 프로그램의 실행 초기에 프로그램은 능동 규칙 관리자에게 프로그램에 포함된 규칙들의 리스트를 보낸다. 규칙 관리자는 전달 받은 규칙들의 타임스탬프를 확인하고 변경된 규칙들이 있으면 그 규칙들을 결과로 반환한다.

## 5. 전처리기에 의한 규칙 코드의 삽입

### 5.1 알고리즘 및 예제

무결성 제약 조건들은 지속 객체가 응용 프로그램에서 생성, 변경, 또는 삭제 될 때마다 확인되어야 한다. 전처리기는 어떤 객체가 지속 객체인지, 그리고 이 객체들에 대해 어떤 종류의 이벤트(즉, 삽입, 변경, 삭제)가 발생했는지를 기억하고 있어야 한다. 이를 위해 pclist(persistent class list), polist(persistent object list), celist(class-event list) 등 세가지 종류의 리스트가 관리된다. pclist는 프로그램에서 선언된 모든 지속 클래스들을 기억한다. 전처리기는 지속 클래스가 선언될 때마다 그 이름을 pclist에 삽입한다. polist는 프로그램에서 선언된 모든 지속 객체들을 기억한다. 전처리기는 지속 객체가 선언될 때마다 객체의 이름, 클래스 이름, 참조 형태를 pclist에 삽입한다. celist는 프로그램에서 발생한 모든 이벤트들을 기억한다. 전처리기는 지속 객체의 생성, 변경, 삭제에 해당하는 이벤트를 발견할 때마다 관련 클래스 이름과 이벤트 타입을 celist에 삽입한다.

또한 전처리기는 어떤 이벤트들에 대해 어떤 규칙들이 적용되는지를 응용 프로그램이 알 수 있게 해야 한다. 이러한 목적으로 전처리기는 규칙 관리자로부터 규칙 코드를 가져와서 특정 파일(rulefile.C)에 저장한

다. 각 규칙은 전처리기에 의해 부여된 이름을 갖는 함수의 형태로 이 파일에 저장되며, 함수의 이름은 해당하는 클래스 이름, 이벤트 타입 등과 함께 rule_func_array라는 배열에 저장된다. 또한 이 배열은 응용 프로그램이 올바른 규칙 코드 함수를 찾기 위해 이러한 배열을 사용할 수 있도록 전역 변수로서 rulefile.C에 기록된다.

규칙 코드를 삽입하는 최적위치는 이벤트가 발생한 곳이다. 그러나 각 이벤트 발생지점 바로 뒤에 규칙 코드를 삽입하는 것은 실질적이지 못하다. 객체의 삭제는 보통 단일 함수의 호출에서 완료되지만, 객체의 생성 및 변경은 일반적으로 이러한 이벤트의 완료지점을 정확히 찾기 위해 프로그램을 추적해 들어갈 필요가 있는 여러 명령들로 이루어진다. 더욱이 어떤 필드들은 여러 번 변경될 수 있으며, 이런 경우 단지 최종 변경만이 유효하게 된다. 만일 전처리기가 이러한 모든 경우들을 다 고려하여 정확한 이벤트 완료지점에 해당 규칙 코드들을 삽입한다면, 그러한 작업을 매우 어렵게 할 뿐만 아니라 많은 비용을 요구하게 된다. 본 논문에서는 이벤트의 정확한 완료지점을 추적하는 대신 모든 이벤트가 commit 구문 직전에 완료된다고 가정한다. 그러므로 전처리기는 프로그램에서 발생한 각 이벤트에 대한 규칙 코드를 commit 구문 직전에 삽입한다. 이는 전처리기가 모든 이벤트들을 관련 객체의 ID, 클래스 이름, 이벤트 타입, 그리고 해당 규칙 코드 표현으로 기억해야 한다는 것을 의미한다. 그러나 객체에 관한 정보는 단지 실행시간에만 획득될 수 있다. 전처리기는 단지 어떤 이벤트가 어떤 클래스에 대해 발생한지와 어떤 규칙 코드가 그런 이벤트에 대해 적용되어야 하는지 만을 알 수 있다. 객체 자체에 관한 정보의 수집은 실행시간 동안에 프로그램에 의해 수행된다. 이를 위해 전처리기는 이벤트가 발생할 때마다 관련 객체가 스택에 저장될 수 있도록 하기 위한 코드를 프로그램에 삽입한다. 전역 데이터 구조인 molist(modified object list)는 이러한 목적을 위해 사용된다. 이벤트가 발생하면, 전처리기는 응용 프로그램에 프로그램이 스스로 객체 ID, 클래스 이름, 그리고 이벤트 타입을 molist에 저장하기 위한 코드를 삽입한다.

데이터 구조 molist에 있는 변경된 객체들을 기억하는 것 외에, 응용 프로그램은 또한 실제로 발생한 이벤트들을 기억해야 한다. 이러한 목적으로 데이터 구조 active_celist가 사용된다. 이 데이터 구조는 이 절의 앞부분에서 설명한 전처리기에 의해 관리되는 데이터 구조인 celist와 차이가 있다. 전처리기는 모든 가능한 이벤트들을 찾아내고 상응하는 규칙 코드를 준비해야 한다. 그러나 응용 프로그램은 실행시간 동안에 실제로 발생한 이벤트들만 기억한다. 데이터 구조 active_celist에 있는 각 엔트리는 클래스 이름, 이벤트 타입, 그리고 카운터 필드를 갖는다. 어떤 특정 클래스에 대해 이벤트가 동일한 이벤트 타입으로 발생할 때마다 카운터 필드의 값은 증가한다. 이 필드는 나중에 이런 이벤트들에 대해 set-oriented 규칙이 적용될 필요가 있을 때 사용된다. 카운터 필드는 처리되어야 할 이벤트들이 얼마나 남아있는지를 보여준다. 마지막을 제외한 모든 이벤트들은 규칙 코드를 실행하는 대신 단지 카운터 값을 감소한다.

응용 프로그램이 commit 구문에 도달하면, 실제로 발생한 모든 이벤트들에 대한 규칙 코드를 검사해야 한다. 이벤트들은 active_celist에 보고되고 상응하는 규칙들은 rule_func_array에서 찾을 수 있다. 이벤트와 관련된 객체들은 molist에 기록된다. 응용 프로그램은 molist에서 객체들을 하나씩 꺼내고, active_celist와 rule_func_array의 도움으로 정확한 규칙 함수들을 찾아낸다. 그 다음 응용 프로그램은 이러한 규칙 함수들을 실행한다.

응용 프로그램은 규칙 코드를 실행하기 위해 active_celist, molist, rule_func_array 등을 사용하므로, 전처리기는 이러한 데이터 구조들의 정의와 지원 루틴들을 별도의 파일에 포함하여 나중에 응용 프로그램에 링크될 수 있도록 해야 한다. 또한 전처리기는 자신에 의해 채워질 rule_func_array를 제외하고, 응용 프로그램이 이러한 데이터 구조들을 채워넣고 모든 이벤트들에 대한 규칙 코드들을 검사하기 위해 그러한 데이터 구조들을 사용할 수 있도록 응용 프로그램을 수정해야 한다. 다음의 알고리즘은 전처리기가 응용 프로그램을 적절히 수정하기 위해 따라야만 하는 알고리즘이다.

**[알고리즘 1] 규칙 코드의 삽입**

입력 : 데이터베이스 응용 프로그램
출력 : 규칙 코드가 삽입된 입력 프로그램
방법 :
0. 전처리기는 입력 프로그램 파일을 파싱하는 동안 다음의 과정들을 반복한다.
1. 만일 지속 클래스가 선언되면, 그 클래스의 이름을 pclist에 삽입한다.
2. 만일 지속 객체가 선언되면, 그 객체의 이름을 polist에 삽입한다.
3. 만일 지속 객체에 대한 이벤트(생성, 변경, 또는 삭제)가 발견되면, 다음의 과정들을 수행한다.

3.1 그 지속 객체의 클래스 이름 및 이벤트 타입을 celist에 삽입한다.
3.2 molist에 관련 객체의 reference, 클래스 이름, 그리고 이벤트 타입을 저장한다.
3.3 실행시간에 실제로 발생한 이벤트를 active_celist에 보고할 수 있도록 다음과 같이 응용 프로그램을 수정한다.
3.3.1 만일 같은 종류의 이벤트(즉, 동일한 클래스 이름과 동일한 이벤트 타입을 갖는 이벤트)가 active_celist에 이미 존재하면, 단지 카운터만 증가시킨다.
3.3.2 그렇지 않으면, 이러한 이벤트에 대해 새로운 엔트리를 하나 생성하고, 카운터를 1로 설정한다.
4. 만일 commit 구문을 만나면, celist에 이벤트가 남아 있는 동안 다음의 과정들을 반복한다.
4.1 celist에서 이벤트를 꺼내고, 그 이벤트에 관련된 규칙들을 규칙 베이스로부터 모두 인출한다. 그 다음 인출된 각 규칙마다 다음의 절차들을 수행한다.
4.1.1 규칙을 C 함수로 변환하여 rulefile.C 라는 파일에 기록한다.
4.1.2 규칙에 관한 정보들로 rule_func_array를 구성한다.
4.2 트랜잭션이 commit 되기 전에, 응용 프로그램이 molist에 있는 변경된 객체들을 꺼내어 해당 규칙 코드를 실행할 수 있도록 응용 프로그램을 수정한다.

상기의 알고리즘에서 rule_func_array는 어떤 이벤트에 어떤 규칙이 적용되어야 하는지를 보여주기 위해 구성된다. 알고리즘 1을 위해 사용되는 주요 데이터 구조들은 다음의 (그림 5)와 같다.

```
list <pc> pclist;
struct pc { char *name; };
list<po> polist;
struct po {
    char *oname;        // object name
    char vtype;         // object reference type (V_HANDLE or V_ITERATOR)
    char *cname;        // class name this object belongs to
};
list<ce> celist;
struct ce {
    char *cname;        // class name
    char event;         // event type (E_CREATE, E_DELETE OR E_UPDATE)
};
list<mo> molist;
struct mo {
    T_OID oid;          // object id
    char *cname; char event; char vtype;
};
list<active_ce> active_celist;
strcut active_ce { char *cname; char event; int count; };
list<rf> rule_func_array;
struct rf {
    char *cname;
    char event;
    char mode;          // 0 if instance-oriented rule; 1 if set-oriented rule
    void (*rule_func) ();
    int lmt;            // last modified time
    char interp;        // 1 if this rule should be interpreted; 0 otherwise
    int rid;            // rule id
};
```

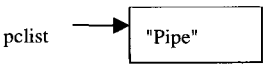
(그림 5) 전처리가 사용하는 데이터 구조들

다음의 (그림 6)은 알고리즘 1을 설명하기 위한 SDBC 응용 프로그램의 예를 보여주고 있다. 아래 그림에서 줄 번호는 단지 설명을 위해 부가되었다.

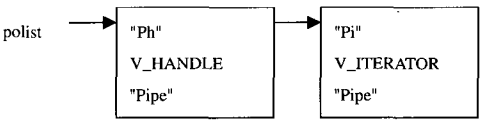
```
1  class Pipe: public s_Object{
2  public:
3    Pipe(Type pipe_type, int pipe_thickness);
4    static s_Set<s_Ref<Pipe>>> pipes; // a reference to class extent
5    void set_thickness(int val) {mark_modified(); thickness = val;}
6    Type type; int thickness;
7  };
8  .....
9  main() {
10   s_Database DBh;
11   s_Ref<Pipe> Ph;
12   s_Iterator<s_Ref<Pipe>> Pi;
13   s_Transaction trans;
14   .....
15   trans.begin(DBh);
16   .....
17   Ph = new(DBh,"Pipe") Pipe(Transport, 20);
18   pipes->insert_element(Ph);
19   Pi = pipes.create_iterator();
20   while (Pi.next(Ph)){
21     Ph->set_thickness(40);
22   }
23   trans.commit();
24 }
```

(그림 6) SDBC 응용 프로그램의 예

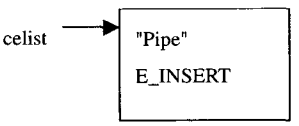
줄 1 : SDBC에서 지속 클래스는 클래스 s_Object의 하위 클래스로 선언된다. 그러므로, 클래스 Pipe는 pclist에 삽입된다.




줄 11, 12 : 지속 객체 Ph와 Pi가 선언되었다. 지속 객체들은 polist에 삽입된다.




줄 17 : 지속 객체가 생성되었다. 즉, INSERT 타입의 이벤트가 발생하였으므로 두 가지 절차가 수행된다. 먼저 이 이벤트가 celist에 기록된다.

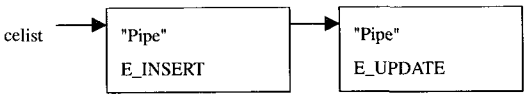



다음으로 전처리기는 응용 프로그램이 실행시간에 스스로 관련 객체의 객체 ID, 클래스 이름, 그리고 이벤트 타입을 molist에 삽입할 수 있도록 응용 프로그

램을 수정한다. 또한, 이러한 이벤트를 active_celist에 보고할 수 있도록 프로그램을 수정한다. 수정된 코드의 형태는 다음과 같다.

```
17 : Ph = new(DBh, Pipe) Pipe ;
     mo = build_mo(Ph, Pipe, E_INSERT, V_HANDLE) ;
     molist.insert(mo) ;
     active_ce = build_active_ce(Pipe, E_INSERT) ;
     report_active_ce(active_ce) ;
```

줄 21 : 지속 객체에 대한 UPDATE 타입의 이벤트(즉, mark_modified()를 동반한 클래스 Pipe의 멤버 함수 set_thickness()의 호출)가 발견되었다. 그러므로 이 이벤트는 먼저 celist에 삽입된다.




그 다음 응용 프로그램은 앞의 INSERT 타입 이벤트가 발생한 경우와 유사한 방식으로 수정된다.

```
20 : while (Pi.next(Ph)) {
21 :   Ph->set_thickness(40);
       mo = build_mo(Pi, Pipe, E_UPDATE, V_ITERATOR) ;
       molist.insert(mo) ;
       active_ce = build_active_ce(Pipe, E_UPDATE) ;
       report_active_ce(active_ce) ;
22: }
```

줄 23 : "commit" 구문이 발견되면, 우선 전처리기는 celist에 있는 이벤트들을 위해 필요한 모든 규칙 코드들을 가져와야 한다. 규칙 코드는 전처리기에 의해 부여된 이름을 갖는 함수로 "rulefile.C"라는 파일에 기록된다. 또한 각 규칙 함수에 관한 정보는 "rule_func_array"에 보고된다. 아래의 <표 1>은 본 예제를 위한 "rule_func_array"의 한 예를 보여준다.

본 예제를 위한 "rulefile.C"는 다음과 같은 규칙 함수를 포함한다.

```
void rule_pipe_insert1(ooHandle(Pipe) Ph) {
 if (Ph->thickness < GAS_MIN_THICKNESS)
 remove(Ph);
}
```

<표 1>에서 이벤트 ("Pipe", E_INSERT)에 대해, 세 개의 규칙들이 존재함을 볼 수 있다. 이 중 둘은 instance-*oriented*(필드 rmode에서 "instance"로 표시됨)이며, 이러한 규칙들은 단지 관련 객체에 대해서만 적용되는 것을 의미한다. 마지막 규칙은 set-oriented(필드 rmode에서 "set"으로 표시됨)이며, 이런 규칙은 관련 클래스에 속한 모든 객체들에 대해 적용되는 것을 의미한다. 전처리기는 파일 "rulefile.C"를 생성한 후, molist에 있는 각 객체를 꺼내어 상응하는 규칙 코드를 실행할 수 있도록 아래와 같이 응용 프로그램을 수정한다. 그러므로 원래의 "commit" 구문은 다음과 같은 프로그램 코드에 의해 모든 규칙 코드들이 실행된 후에야 비로소 실행하게 된다.

```
while (!molist.empty()) {
        mo = molist.pop();
        execute_rule_code(mo);
}

23 : trans.commit();
```

상기의 예제에서 함수 "execute_rule_code(mo)"는 "rule_func_array"를 이용하여 mo에 기록된 이벤트에 대한 규칙 함수들을 검색한다. 이렇게 검색된 모든 규칙 함수들은 하나씩 차례로 실행된다.

다음의 (그림 7)은 이제까지 (그림 6)의 입력 프로그램에 대해 부분적으로 설명되었던 전처리 과정들이 완료된 후에 새로 생성된 SDBC 프로그램의 예를 보여준다. 그림에서 진한 글씨체로 쓰여진 코드들이 전처리기에 의해 생성된 코드들을 나타낸다.

〈표 1〉 rule_func_array의 한 예

| cname | event | rmode | rule func | lmt | rid | . |
|---|---|---|---|---|---|---|
| Pipe | insert | instance | ptr to "rule pipe insert1()" | 9802021234 | 344 | . |
| Pipe | insert | instance | ptr to "rule pipe insert2()" | 9802121235 | 345 | . |
| Pipe | insert | set | ptr to "rule pipe insert3()" | 9802121236 | 346 | . |
| Pipe | update | instance | ptr to "rule pipe update1()" | 9803021237 | 456 | ... |
| ..... | ..... | ..... | ..... | ..... | . | ... |

```
1  class Pipe: public s_Object{
2  public:
3     Pipe(Type pipe_type, int pipe_thickness);
4     static s_Set<s_Ref<Pipe>>> pipes; // a reference to class extent
5     void set_thickness(int val) {mark_modified(); thickness = val;}
6     Type type; int thickness;
7  };
8  .....
9  main() {
10    s_Database DBh;
11    s_Ref<Pipe> Ph;
12    s_Iterator<s_Ref<Pipe>> Pi;
13    s_Transaction trans;
14    .....
15    trans.begin(DBh);
16    .....
17    Ph = new(DBh,"Pipe") Pipe(Transport, 20);
18    mo = build_mo(Ph, "Pipe", E_INSERT, V_HANDLE);
19    molist.insert(mo);
20    active_ce = build_active_ce("Pipe", E_INSERT);
21    report_active_ce(active_ce);
22    pipes->insert_element(Ph);
23    Pi = pipes.create_iterator();
24    while (Pi.next(Ph)) {
25       Ph->set_thickness(40);
26       mo = build_mo(Pi, "Pipe", E_UPDATE, V_ITERATOR);
27       molist.insert(mo);
28       active_ce = build_active_ce("Pipe", E_UPDATE);
29       report_active_ce(active_ce);
30    }
31    while (!molist.empty()) {
32       mo = molist.pop();
33       execute_rule_code(mo);
34    }
35    trans.commit();
36 }
```

(그림 7) 전처리된 SDBC 프로그램의 예

### 5.2 트리거링 규칙의 처리

한 규칙의 액션 실행은 또 다른 규칙들을 트리거 할 수 있다. 다음의 예는 3.2 절에서 이미 예시된 규칙으로 이러한 경우에 해당된다.

```
DEFINE INSTANCE-ORIENTED RULE R1 FOR(SewerPipe);
VAR sp, new_sp: SewerPipe;
EVENT DELETE(sp);
CONDITION sp.usage == TRUE;
ACTION replace(new_sp, sp);
END_RULE;
```

이 규칙은 실행될 때 새로운 상수도 관의 생성을 야기하고, 그 다음 3.2 절에 있는 규칙 R2와 같은 INSERT 타입의 다른 규칙에 대한 확인을 요구한다. 이런 종류의 규칙을 위해 응용 프로그램은 자동으로 R2와 같은 규칙을 확인하는 것이 보장되어야 한다. 만일 규칙 R2가 또 다른 규칙을 트리거 하면, 프로그램은 역시 그러한 규칙을 확인할 수 있어야 한다. 이러한 전방향 연쇄화(forward chaining)가 잠시동안 진행되고, 프로그램은 모든 관련된 규칙들을 확인할 수 있어야 한다. 이런 종류의 규칙을 다루기 위해 전처리기는 원래의 응용 프로그램 코드에서와 마찬가지로 유사한 방식으로 규칙 코드를 수정한다. 즉, 전처리기는 규칙 코드에 있는 모든 이벤트들을 탐색하여 celist에 그 이벤트들을 기록하고, 실행시간 동안에 molist와 active_celist를 구성할 수 있도록 규칙 코드를 수정한다(알고리즘 1의 단계 3 참조). 이렇게 하기 위해 알고리즘 1의 단계 4는 다음과 같이 수정된다.

```
if ("commit()" statement is found) {
     open "rulefile.C"
     call rulefetch()
     modify application program to include code for
     poping "molist" and
     executing rule code.
}
```

위에서 함수 "rulefetch()"는 다음과 같이 정의된다.

```
void rulefetch()
{
  while (celist is not empty) {
     take out one "ce"
     fetch all related rules for this "ce"
     for each of these rules {
       if it is a triggering rule {
          process it according to step 3 of algorithm
          1. during this process,
          all additional events occurring in this rule
          code will be added to "celist", and
          the rule code will be modified appropriately.
       }
       dump its code in "rulefile.C"
       report this rule function in "rule_func_array"
} } }
```

트리거링 규칙에 있는 모든 이벤트들은 celist의 끝에 추가되므로 그 이벤트들은 관련된 활성 규칙들을 가져온 후에 처리된다. 만일 이런 규칙들이 다른 트리거링 규칙들의 집합을 포함하면, 그 규칙들은 전처리기가 규칙 인출(fetching) 과정을 반복하게 하여 또 다시 이벤트들의 집합을 celist에 추가한다. 이런 식으로 실행시간 동안 요구될 수 있는 모든 규칙 코드들을 rulefile.C에 준비해 놓는다. 실제 규칙의 실행은 다음에서와 같이 규칙을 확인하는 과정 동안에 발생한다.

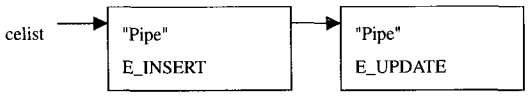
```
while(!molist.empty) {
   mo = molist.pop();
   execute_rule_code(mo);
}
```

위의 execute_rule_code(mo)에서, 만일 관련된 규칙이 트리거링 규칙이면, 그 규칙은 자동으로 관련 객체를 molist에 삽입하고 active_celist에 그 이벤트를 보

램을 수정한다. 또한, 이러한 이벤트를 active_celist에 보고할 수 있도록 프로그램을 수정한다. 수정된 코드의 형태는 다음과 같다.

```
17 : Ph = new(DBh, Pipe) Pipe ;
     mo = build_mo(Ph, Pipe, E_INSERT, V_HANDLE) ;
     molist.insert(mo) ;
     active_ce = build_active_ce(Pipe, E_INSERT) ;
     report_active_ce(active_ce) ;
```

줄 21 : 지속 객체에 대한 UPDATE 타입의 이벤트(즉, mark_modified()를 동반한 클래스 Pipe의 멤버 함수 set_thickness()의 호출)가 발견되었다. 그러므로 이 이벤트는 먼저 celist에 삽입된다.




그 다음 응용 프로그램은 앞의 INSERT 타입 이벤트가 발생한 경우와 유사한 방식으로 수정된다.

```
20 : while (Pi.next(Ph)) {
21 :    Ph->set_thickness(40);
        mo = build_mo(Pi, Pipe, E_UPDATE, V_ITERATOR) ;
        molist.insert(mo) ;
        active_ce = build_active_ce(Pipe, E_UPDATE) ;
        report_active_ce(active_ce) ;
22: }
```

줄 23 : "commit" 구문이 발견되면, 우선 전처리기는 celist에 있는 이벤트들을 위해 필요한 모든 규칙 코드들을 가져와야 한다. 규칙 코드는 전처리기에 의해 부여된 이름을 갖는 함수로 "rulefile.C"라는 파일에 기록된다. 또한 각 규칙 함수에 관한 정보는 "rule_func_array"에 보고된다. 아래의 <표 1>은 본 예제를 위한 "rule_func_array"의 한 예를 보여준다.

본 예제를 위한 "rulefile.C"는 다음과 같은 규칙 함수를 포함한다.

```
void rule_pipe_insert1(ooHandle(Pipe) Ph) {
 if (Ph->thickness < GAS_MIN_THICKNESS)
 remove(Ph);
}
```

<표 1>에서 이벤트 ("Pipe", E_INSERT)에 대해, 세 개의 규칙들이 존재함을 볼 수 있다. 이 중 둘은 instance-oriented(필드 rmode에서 "instance"로 표시됨)이며, 이러한 규칙들은 단지 관련 객체에 대해서만 적용되는 것을 의미한다. 마지막 규칙은 set-oriented(필드 rmode에서 "set"으로 표시됨)이며, 이런 규칙은 관련 클래스에 속한 모든 객체들에 대해 적용되는 것을 의미한다. 전처리기는 파일 "rulefile.C"를 생성한 후, molist에 있는 각 객체를 꺼내어 상응하는 규칙 코드를 실행할 수 있도록 아래와 같이 응용 프로그램을 수정한다. 그러므로 원래의 "commit" 구문은 다음과 같은 프로그램 코드에 의해 모든 규칙 코드들이 실행된 후에야 비로소 실행하게 된다.

```
while (!molist.empty()) {
        mo = molist.pop();
        execute_rule_code(mo);
}

23 : trans.commit();
```

상기의 예제에서 함수 "execute_rule_code(mo)"는 "rule_func_array"를 이용하여 mo에 기록된 이벤트에 대한 규칙 함수들을 검색한다. 이렇게 검색된 모든 규칙 함수들은 하나씩 차례로 실행된다.

다음의 (그림 7)은 이제까지 (그림 6)의 입력 프로그램에 대해 부분적으로 설명되었던 전처리 과정들이 완료된 후에 새로 생성된 SDBC 프로그램의 예를 보여준다. 그림에서 진한 글씨체로 쓰여진 코드들이 전처리기에 의해 생성된 코드들을 나타낸다.

<표 1> rule_func_array의 한 예

| cname | event | rmode | rule func | lmt | rid | . |
|---|---|---|---|---|---|---|
| Pipe | insert | instance | ptr to "rule pipe insert1()" | 9802021234 | 344 | . |
| Pipe | insert | instance | ptr to "rule pipe insert2()" | 9802121235 | 345 | . |
| Pipe | insert | set | ptr to "rule pipe insert3()" | 9802121236 | 346 | . |
| Pipe | update | instance | ptr to "rule pipe update1()" | 9803021237 | 456 | ... |
| ..... | ..... | ..... | ..... | ..... | . | ... |

```
1  class Pipe: public s_Object{
2  public:
3     Pipe(Type pipe_type, int pipe_thickness);
4     static s_Set<s_Ref<Pipe>>> pipes; // a reference to class extent
5     void set_thickness(int val) {mark_modified(); thickness = val;}
6     Type type; int thickness;
7  };
8  .....
9  main() {
10    s_Database DBh;
11    s_Ref<Pipe> Ph;
12    s_Iterator<s_Ref<Pipe>> Pi;
13    s_Transaction trans;
14    .....
15    trans.begin(DBh);
16    .....
17    Ph = new(DBh,"Pipe") Pipe(Transport, 20);
18    mo = build_mo(Ph, "Pipe", E_INSERT, V_HANDLE);
19    molist.insert(mo);
20    active_ce = build_active_ce("Pipe", E_INSERT);
21    report_active_ce(active_ce);
22    pipes->insert_element(Ph);
23    Pi = pipes.create_iterator();
24    while (Pi.next(Ph)) {
25       Ph->set_thickness(40);
26       mo = build_mo(Pi, "Pipe", E_UPDATE, V_ITERATOR);
27       molist.insert(mo);
28       active_ce = build_active_ce("Pipe", E_UPDATE);
29       report_active_ce(active_ce);
30    }
31    while (!molist.empty()) {
32       mo = molist.pop();
33       execute_rule_code(mo);
34    }
35    trans.commit();
36 }
```

(그림 7) 전처리된 SDBC 프로그램의 예

5.2 트리거링 규칙의 처리

한 규칙의 액션 실행은 또 다른 규칙들을 트리거 할 수 있다. 다음의 예는 3.2 절에서 이미 예시된 규칙으로 이러한 경우에 해당된다.

```
DEFINE INSTANCE-ORIENTED RULE R1 FOR(SewerPipe);
VAR sp, new_sp: SewerPipe;
EVENT DELETE(sp);
CONDITION sp.usage == TRUE;
ACTION replace(new_sp, sp);
END_RULE;
```

이 규칙은 실행될 때 새로운 상수도 관의 생성을 야기하고, 그 다음 3.2 절에 있는 규칙 R2와 같은 INSERT 타입의 다른 규칙에 대한 확인을 요구한다. 이런 종류의 규칙을 위해 응용 프로그램은 자동으로 R2와 같은 규칙을 확인하는 것이 보장되어야 한다. 만일 규칙 R2가 또 다른 규칙을 트리거 하면, 프로그램은 역시 그러한 규칙을 확인할 수 있어야 한다. 이러한 전방향 연쇄화(forward chaining)가 잠시동안 진행되고, 프로그램은 모든 관련된 규칙들을 확인할 수 있어야 한다. 이런 종류의 규칙을 다루기 위해 전처리기는 원래의 응용 프로그램 코드에서와 마찬가지로 유사한 방식으로 규칙 코드를 수정한다. 즉, 전처리기는 규칙 코드에 있는 모든 이벤트들을 탐색하여 celist에 그 이벤트들을 기록하고, 실행시간 동안에 molist와 active_celist를 구성할 수 있도록 규칙 코드를 수정한다(알고리즘 1의 단계 3 참조). 이렇게 하기 위해 알고리즘 1의 단계 4는 다음과 같이 수정된다.

```
if ("commit()" statement is found) {
     open "rulefile.C"
     call rulefetch()
     modify application program to include code for
     poping "molist" and
     executing rule code.
}
```

위에서 함수 "rulefetch()"는 다음과 같이 정의된다.

```
void rulefetch()
{
  while (celist is not empty) {
     take out one "ce"
     fetch all related rules for this "ce"
     for each of these rules {
       if it is a triggering rule {
          process it according to step 3 of algorithm
          1. during this process,
          all additional events occurring in this rule
          code will be added to "celist", and
          the rule code will be modified appropriately.
       }
       dump its code in "rulefile.C"
       report this rule function in "rule_func_array"
} } }
```

트리거링 규칙에 있는 모든 이벤트들은 celist의 끝에 추가되므로 그 이벤트들은 관련된 활성 규칙들을 가져온 후에 처리된다. 만일 이런 규칙들이 다른 트리거링 규칙들의 집합을 포함하면, 그 규칙들은 전처리기가 규칙 인출(fetching) 과정을 반복하게 하여 또 다시 이벤트들의 집합을 celist에 추가한다. 이런 식으로 실행시간 동안 요구될 수 있는 모든 규칙 코드들을 rulefile.C에 준비해 놓는다. 실제 규칙의 실행은 다음에서와 같이 규칙을 확인하는 과정 동안에 발생한다.

```
while(!molist.empty) {
   mo = molist.pop();
   execute_rule_code(mo);
}
```

위의 execute_rule_code(mo)에서, 만일 관련된 규칙이 트리거링 규칙이면, 그 규칙은 자동으로 관련 객체를 molist에 삽입하고 active_celist에 그 이벤트를 보

고한다. 그 다음 상응하는 mo가 molist로부터 꺼내지고 난 후에 활성 규칙이 실행된다.

### 5.3 규칙의 동적 처리

1절에서 지적한 바와 같이 컴파일러(또는 전처리기) 기반 방식의 심각한 단점은 규칙들의 부분 또는 전체가 변경되었을 때 응용 프로그램이 그 변경된 규칙을 적용하기 위해서는 다시 컴파일 되어져야 한다는 것이다. 만일 단지 규칙의 작은 부분만이 매우 빈번히 변경될 경우, 그 때마다 전체 프로그램을 다시 컴파일하는 것은 매우 많은 비용을 요구하게 된다. 그러한 비용 오버헤드를 해결하기 위해 본 논문에서는 이런 경우에 동적 규칙 해석기를 사용한다. 동적으로 규칙들을 처리하기 위해 응용 프로그램은 컴파일 된 규칙 코드들을 실행하기 전에 먼저 변경된 규칙들이 있는지 능동 규칙 관리자에게 문의한다. 문의결과 변경된 규칙들이 존재하면, 그 변경된 규칙들을 바이트 코드 형태로 전달 받아 해석기를 이용하여 동적으로 실행하게 된다. 이렇게 하기 위해 commit 구문 전의 규칙 확인 과정을 변경하여야 한다. 다음의 코드들은 변경된 규칙의 실행시간 해석을 위한 과정의 전체 흐름을 개략적으로 설명하는 코드들이며, 상세한 코드들은 생략되어 있다.

```
while(!molist.empty) {
  check_result = identify_rule_change(rule_info);
  if(check_result.change_message == UPDATED or INSERTED) {
    update_rfa(rule_func_array);
    execute_changed_rule_bytecode(check_result.
    changed_rule_bytecode);
  }
  else if(check_result.change_message == UNCHANGED) {
    mo = molist.pop();
    execute_compiled_rule_code(mo);
  }
  else if(check_result.change_message == DELETED)
    update_rfa(rule_func_array);
}
```

함수 identify_rule_change는 우선 5.1절에서 예로 든 것과 같은 rule_func_array에 있는 각 엔트리로부터 <cname, event, lmt, rid>과 같은 형태의 4-튜플(4-tuple) 리스트(상기의 프로그램에서 rule_info를 구성하는 데이터에 해당됨)를 추출하여 능동 규칙 관리자에게 전달한다. 능동 규칙 관리자는 그 응답으로 각 엔트리에 대해 <cname, event, rmode, lmt, rid, change_message, changed_rule_bytecode>과 같은 형태의 6-튜플(6-tuple) 리스트(상기의 프로그램에서 check_result를 구성하는 데이터에 해당됨)를 되돌려준다. rule_func_array에는 응용 프로그램에서 발생할 수 있는 모든 가능한 <class, event> 쌍들과 응용 프로그램에게 이미 알려진 상응하는 규칙들이 기록되어 있기 때문에, 능동 규칙 관리자는 어떤 이벤트에 해당하는 규칙이 변경되었는지에 관한 문의에 응답할 수 있다. 규칙 관리자에 의한 응답 결과인 6-튜플에서 change_message 필드는 UNCHANGED(변경되지 않았을 나타냄), UPDATED(변경되었음을 나타냄), DELETED (삭제되었음을 나타냄), INSERTED (새로 추가되었음을 나타냄) 중 하나를 나타낸다. 이중 UPDATE와 INSERTED의 경우에 changed_rule_ bytecode 필드는 변경되거나 새로 생성된 규칙 코드를 바이트 코드 형태로 포함한다.

변경된 규칙들에 대해 응용 프로그램은 변경된 규칙의 바이트 코드를 포함한 그 외의 변경된 정보를 전달받아 rule_func_array의 각 엔트리를 수정한다. 예를 들어, <표 1>에서 만약 rid가 344인 규칙이 lmt(last modified time) 이후로 변경되고 rid가 512인 규칙이 이벤트 (Pipe, INSERT)에 추가되었다면, 응용 프로그램은 변경된 규칙들의 바이트 코드를 해석하기 전에 rule_func_array를 갱신한다. 갱신된 rule_func_array는 <표 2>와 같이 표현된다. <표 2>에서 보여지는 바와 같이 rid가 344인 규칙에 관한 엔트리의 lmt가 변경되고, rid가 512인 규칙에 관한 엔트리가 응용 프로그램에 의해 rule_func_array에 추가된다.

〈표 2〉 응용 프로그램에 의해 변경된 rule__func__array의 예

| cname | event | rmode | rule func | lmt | rid | |
|---|---|---|---|---|---|---|
| ***Pipe*** | ***insert*** | ***instance*** | ***삭제됨.*** | ***9808073434*** | ***344*** | |
| Pipe | insert | instance | ptr to "rule pipe insert2()" | 9802121235 | 345 | |
| Pipe | insert | set | ptr to "rule pipe insert3()" | 9802121236 | 346 | |
| Pipe | update | instance | ptr to "rule pipe update1()" | 9803021237 | 456 | |
| ..... | ..... | ..... | ..... | ..... | . | |
| ***Pipe*** | ***insert*** | ***instance*** | ***공백임.*** | ***9901192513*** | ***512*** | |

규칙 바이트 코드의 동적 해석기는 자체 데이터 구조로 object_info와 interp_result를 관리한다. 해석기가 규칙 바이트 코드를 해석할 때 규칙의 적용 대상이 되는 객체 데이터를 액세스(즉, 검색 및 저장)할 수 있어야 한다. 이를 위해 object_info를 이용하며, 이것은 객체의 OID, 데이터베이스 reference, 그리고 객체 reference 정보를 관리한다. 한편, 또 다른 데이터 구조인 interp_result는 해석 결과를 관리하기 위한 것이며, 해석 결과 메시지 및 결과의 객체 데이터를 관리하기 위해 사용된다.

## 6. 결 론

본 논문에서는 공간 무결성 제약조건을 효율적으로 관리하기 위한 능동 규칙 시스템을 제시하였다. 제시된 능동 규칙 시스템의 프로토타입은 ODMG 인터페이스 및 ODBMS에 대한 공간 확장을 제공하는 더 큰 규모의 미들웨어 시스템의 한 콤포넌트로 구현되었다. 본 논문에서는 능동 규칙 시스템을 미들웨어 방식으로 구현함으로써 분산 환경에서 특정 시스템에 독립적인 방식으로 이종 ODBMS들을 위해 제공된다. 본 능동 규칙 시스템에서 능동 규칙은 E-C-A 형태로 표현되며, 공간 무결성 제약조건을 정의하고 확인하기 위해 이용된다.

다른 능동 규칙 시스템들의 구조와 비교하여, 본 논문에서 제시한 시스템은 데이터베이스 응용 프로그램을 컴파일 함으로써 모든 데이터베이스 상태 변화들을 실행시간에 모니터링 하는데 수반되는 오버헤드를 감소한다. ECA 규칙들에서 표현된 무결성 제약 조건들을 확인하기 위한 코드들은 공간 데이터베이스 프로그램에 직접 삽입된다. 또한 본 시스템은 변경된 규칙들에 대한 실행시간 해석을 제공한다. 실행시간 해석을 위해 변경된 규칙들의 stub 코드들은 능동 규칙 관리자로부터 전달된다. 능동 규칙들의 변경정보를 관리하기 위해 타임스탬프 정보가 트리거링 그래프에 저장된다. 능동 규칙들에서 약간의 변화가 있을 때, 사용자는 프로그램을 다시 컴파일 할지 또는 변경된 규칙들을 실행시간에 해석할 지에 관한 선택을 할 수 있다. 그러나, 규칙의 동적 처리 기능은 분산 환경에서 작동하므로 실행시간 오버헤드(overhead)를 수반하게 된다. 이러한 오버헤드 문제는 크게 두 가지 측면에서 고려될 수 있다. 하나는 규칙을 실행시간에 규칙 베이스로부터 인출하는데 드는 통신 오버헤드이고, 다른 하나는 규칙 코드를 동적으로 해석하는데 소요되는 시간 오버헤드이다. 본 연구에서는 이러한 오버헤드를 최소화 하기 위해 바이트 코드 해석 기법을 이용하고 있으며, 또한 통신 오버헤드를 최소화 하기 위해 규칙 코드의 캐싱(caching)에 관한 기법에 관한 연구를 진행하고 있다.

향후 최적화에 대해 남겨진 여지가 많이 존재한다. 우선, 서로 다른 능동 규칙들의 규칙 컨디션들은 종종 공통 부분들을 포함한다. 의존성(dependency) 그래프를 이용한 분석에 의해 주어진 규칙 컨디션들에 대한 공유 부분들과 최적의 산정 순서를 확인할 수 있다[14]. 다음으로, 목표 ODBMS로부터 최적화를 위한 가치 있는 정보들을 얻을 수 있으며, 그 중 인덱스 정보는 특히 공간 객체들에 대해 더 중요하다. 그 이유는 공간 인덱스들이 응용에 더 종속적이기 때문이다[15].

## 참 고 문 헌

[1] J. Widom and S. Ceri, Active Database Systems, Morgan Kaufmann Publishers, Inc., 1996.

[2] R. Kidwell and J. Richman, Preliminary IDE Framework Implementatin Concept, and Rationale Report for the DoD CALS IDE Project, ManTech Advanced Technology Systems, 1996.

[3] H. Garcia-Molina, W. Labio, and Jun Yang, Expiring Data in a Warehouse, Proceedings of the 24th International Conference on Very Large Data Bases, pp.500-511, New York, 1998.

[4] OMG, CORBA Services : Common Object Services Specification, 1996.

[5] R. Cattell and D. Barry ed., The Object Database Standard : ODMG 2.0, Morgan Kaufmann Publishers, Inc., 1997.

[6] Open GIS, OpenGIS Simple Features Specification for CORBA, Revision 0, Open GIS Consortium, Inc., 1997.

[7] E. Baralis, S. Ceri, and S. Paraboschi, Compile-Time and Runtime Analysis of Active Behavior, IEEE Trans. On Knowledge and Data Engi-

neering, Vol.10, No.3, pp.353-370, 1998.

[8] S. Chakravarthy, B. Blaustein, A.P. Buchmann, M. Carey, U. Dayal, D. Goldhirsch, M. Hsu, R. Jauhari, R. Ladin, M. Livny, D. McCarthy, R. McKee, and A. Rosenthal, "HiPAC : A research project in active, time-constrained database management," Technical Report XAIT-89-02, Xerox Advanced Information Technology, Carnbridge, Massachusetts, July 1989.

[9] S. Chakravarthy and K. Ono, "ECA Rule Support for Distributed Heterogeneous Environments," Proceedings of the Fourteenth International Conference on Data Engineering, pp.601, Orlando, Florida, February 1998.

[10] L.M. Haas, W. Chang, G.M. Lohman, J. McPherson, P.F. Wilms, G. Lapis, B. Lindsay, H. Pirahesh, M. Carey, and E. Shekita, "Starburst mid-flight : As the dust clears," IEEE Transactions on Knowledge and Data Engineering, 2(1) : 143-160, March 1990.

[12] C. Collet, T. Coupaye, and T.Svensen, "NAOS-efficient and modular reactive capabilities in an object-oriented database system," Proceedings of the Twentieth International Conference on Very Large Data Bases, pp.132-143, Santiago, Chile, September 1994.

[13] G. Weiss, J. Ros and A. Singhal, ANSWER : Network Monitoring Using Object-Oriented Rules, Proceedings of the 10th Innovative Applications of AI Conference (IAAI-98), July 1998.

[14] S. Yoo and S. Cha, Integrity maintenance in a heterogeneous engineering database environment, Data & Knowledge Engineering, Vol.21, Elsevier Science, pp.347-363, 1997.

[15] H. Samet, The Design and Analysis of Spatial Data Structures, Addison Wesley, 1990.

**고 굉 욱**

e-mail : kwko@islab.auto.inha.ac.kr

1993년 인하대학교 자동화공학과 학사

1996년 인하대학교 자동화공학과 석사

1996년~현재 인하대학교 자동화공학과 박사과정

관심분야 : 시스템통합, 객체지향 DB, Active DB, CALS/EC

**유 상 봉**

e-mail : syoo@inha.ac.kr

1982년 서울대학교 제어계측공학과 학사

1986년 Arizona 주립대학교 전기 및 컴퓨터공학 석사

1990년 Purdue 대학교 전기 및 컴퓨터공학 박사

1989년 미국 AT&T Bell 연구소 연구원

1990년 삼성전자 선임연구원

1992년~현재 인하대학교 자동화공학과 부교수

관심분야 : 객체 및 지식 데이터베이스, 시스템 통합, GIS, CALS/EC, PDM 등

**김 기 창**

e-mail : kchang@inha.ac.kr

1986년 California State Polytechnic University at Pomona, Computer Science 학사

1988년 University of California at Irvine, Computer Science 석사

1992년 University of California at Irvine, Computer Science 박사

1992년~1994년 IBM T.J. Watson 연구소 연구원

1994년~현재 인하대학교 컴퓨터공학과 부교수

관심분야 : 병렬 컴파일러, 웹 프락시 등


## 차 상 균

e-mail : chask@kdb.snu.ac.kr

1980년 서울대학교 전기공학 학사

1982년 서울대학교 제어계측공학 석사

1991년 Stanford대학교 컴퓨터 공학 박사

1982년~1983년 데이콤 연구원

1984년~1992년 Stanford 대학교 전산학과 연구 조교, Texas Instruments, IBM Palo Alto Scientific Center, HP 연구소 연구원

1992년~현재 서울대학교 전기공학부 부교수

관심분야 : 주메모리 DBMS, 통합 분산 공간 데이터베이스, DB 인덱싱 및 클러스터링, ITS.