지능정보연구 제17권 제4호 2011년 12월(pp.37~58)

전사적 응용시스템 테스트를 위한 DB이미지 생성에 관한 연구

권오승
서울시립대학교 대학원 경영학과
(koschoco@naver.com)

홍사능
서울시립대학교 대학원 경영학과
(snhong@uos.ac.kr)

데이터베이스를 사용하는 프로그램을 테스트하는 것은 일반 소프트웨어의 경우보다 훨씬 더 복잡하고 어렵다. 테스트 데이터에 더하여 데이터베이스 상태가 테스트의 절차와 결과에 결정적인 영향을 미치는 것이 주요 원인이다. 테스트에 적합한 데이터베이스 상태를 만들어주려면 많은 시간과 노력이 필요한 것은 물론이거니와 IT와 업무에 대한 상당한 지식이 있어야 한다. 이러한 어려움에도 불구하고 데이터베이스 응용 프로그램의 테스트에 대한 연구와 지원은 매우 부족하다.

이 논문은 테스트에 알맞은 데이터베이스 상태의 생성과 유지에 관한 연구 결과를 보고한다. 연구의 핵심은 프로그램에서 사용하는 SQL을 로그파일에서 추출하여 분석한 결과와 데이터베이스 스키마와 테이블, 로그, 전문 등의 다양한 원천(source)에서 수집한 정보를 결합하여 프로그램의 테스트에 적합한 사전, 사후 상태를 자동으로 만들어주는 테스트 지원도구의 개발이다. 연구에서 제시한 절차와 도구는 단위 테스트와 통합 테스트의 지원과 더불어 회귀 테스트의 수행에 따르는 어려움을 극복하는데 큰 도움이 될 것이다

실무적으로는 연구의 결과가 데이터베이스 상태의 생성과 유지에 소요되는 시간과 노력을 줄여 개발인력의 생산성을 제고하고, 다양한 케이스의 테스트와 회귀 테스트를 지원하여 대상 프로그램의 품질 향상에 기여할 것으로 기대한다. 학문적으로는 프로그램에서 사용하는 SQL의 패턴을 분석할 수 있는 상태 전이 도형과, 패턴의 표현 및 추론이 가능한 문법을 정의하여 전사적 응용 프로그램 테스트에 대한 폭 넓은 이해와 새로운 접근 방식을 가능하게 하였다.

논문접수일 : 2011년 12월 12일 게재확정일 : 2011년 12월 24일
투고유형 : 국문급행 교신저자 : 홍사능

1. 서 론

데이터베이스 응용 프로그램은 특성상 전사적으로 구성된 인프라를 필요로 하기 때문에 테스트에 필요한 환경도 매우 복잡하다(Bohner and Arnold, 1996). 또한 응용 프로그램과 데이터베이스의 상호작용을 검증하기 위한 데이터의 준비에 많은 노력과 시간이 필요하며, 테스트 결과에 상응하는 데이터베이스의 상태를 확인하는 것도 어려울 수밖에 없다. 이러한 복합적인 요인으로 인하여 오랜 기간에 걸쳐 소프트웨어 테스트에 관한 다양한 연구가 있었음에도 불구하고 데이터베이스 응용 프로그램의 테스트에 대한 연구와 지원은 아직 부족한 상태이다(Chays et al., 2002). 연구와 지원도구의 부족은 테스트의 상당부분이 수작업으로 이루질 수밖에 없도록 하고, 수작업에 의존한 테스트 전략은 불충분한 테스트로 이어져서 전사적 시스템의 품질에 확신을 갖기 어렵게 한다.

소프트웨어 품질은 절대적으로 테스트에 의존하지만, 단순히 테스트를 많이 하였다고 품질이 좋아지는 것은 아니다. 동일한 데이터를 사용한 테스트를 수백 번 실행하였다고 해서 소프트웨어의 품질이 높아진다고 할 수 없다. 소프트웨어가 취할 수 있는 모든 상태(state)의 전부 또는 가장 중요한 부분을 점검할 수 있도록 테스트케이스를 도출하고 이를 테스트해야만 소프트웨어의 품질이 높아진다. 그러나 테스트가 소프트웨어 품질의 신뢰성을 가질 수 있도록 다양한 테스트케이스를 도출하는 데에는 많은 시간과 노력(Aditya, 2008; James and Gregg, 2003)은 물론 고도의 IT 지식과 업무 지식이 필요하다.

소프트웨어는 근본적으로 복잡할 뿐만 아니라(Brooks, 1987), 특히 분산 소프트웨어의 경우에는 노드(CPU) 간의 상호작용에 시간요소(timing)가 개입되기 때문에 완벽한 테스트라는 개념 자체를 적용할 수 없다(Royce, 1998). 이에 더하여 데이터베이스 응용시스템의 경우에는 테스트가 수행되기 이전의 데이터베이스 상태를 구축해야 하고, 테스트가 수행된 이후에는 프로그램의 결과 값뿐만 아니라 데이터베이스의 상태까지도 확인해야 한다(Chays et al., 2002; 2004; 2005). 더구나 하나의 응용 프로그램에서 작게는 2~10개, 많게는 수십에서 백여 개에 이르는 SQL문을 수행한다는 점까지를 감안하면 전사적 응용시스템의 품질을 보장할 수 있는 충분한 테스트의 어려움과, 이에 대한 연구와 지원도구가 부족한 까닭을 이해할 수 있다.

이 논문은 전사적 응용시스템 테스트에 필요한 DB 사전/사후 이미지(이하 'DB이미지')를 자동으로 생성하는 방안에 관한 연구결과이다. DB 사전 이미지(이하 '사전 이미지')는 테스트가 실행되기 이전의 데이터베이스 상태로서; 응용 프로그램에서 사용하는 테이블들에 테스트에서 필요로 하는 레코드는 있어야 하고, 있어서는 안 되는 레코드는 제거되어야 한다. DB 사후 이미지(이하 '사후 이미지')는 테스트에서 수행한 데이터베이스 작업(SQL 실행)을 사전 이미지에 반영한 상태를 의미한다.

사전 이미지의 생성은 응용 프로그램이 실행 중에 남긴 로그에서 SQL을 추출하는 것으로 시작한다. 추출된 텍스트 형태의 SQL을 구문요소의 계층관계로 분해한 구조(parse tree)를 이용하여, 정제와 더불어 SQL 사이의 의존관계를 유지하는 SQL 순차집합(sequence)으로 분할한다. 이어서 개별 순차집합에 나타나는 SQL 사용 패턴에 따라 이미지 생성에 필요한 조작의 종류를 판별한 뒤에; 속성, WHERE 조건, 속성 값에 관한 정보를 수집하여 DB이미지 생성 SQL 문을 조립한다. 사전 이미지는 필요한 레코드의 '등록'이나 불필요한 레코드의 '삭제' 작업으로, 사후 이미지는 예상 결과와 이를 확인할 수 있는 '조회' 작업으로 만들어진다. DB이미지 생성 과정의 대부분은 자동화가 가능하고, 일부 작업에만 사람의 판단을 필요로 한다. 후자의 경우에도 부가적인 단순 수작업은 모두 자동화 대상이다(Carsten et al., 2008).

연구의 결과를 적용하면 1) 테스트케이스의 입력 값이 작성되면 DB이미지와 예상 값을 자동으로 생성할 수 있어서 테스트케이스 작성에 소요되는 시간과 노력을 획기적으로 줄일 수 있으며, 2) 업무에 포함된 SQL 순서에 따른 오류를 발견하고, 3) 반복적 테스트를 지원하기 위한 회귀 테스트를 수행할 수 있는 테스트 데이터와 데이터베이스 상태를 생성 및 유지해 줄 수 있다.

다음 장에서는 지원의 대상이 되는 전사적 시스템의 기본 구조와 기능을 설명하고, 데이터베이스 프로그램의 예제와 더불어 테스트 절차와 주요 이슈를 검토한다. 제 3장에서는 SQL의 추출, Pars-

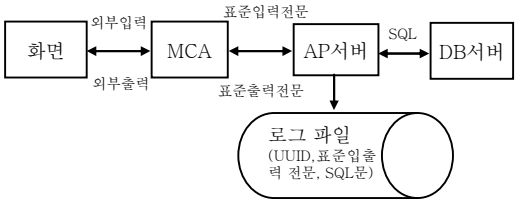
ing, SQL 의존관계 추론, 필요정보 수집, 그리고 이미지 생성 SQL 조립과 예제의 순서로 이미지 생성 과정을 기술하고, 제 4장에서는 관련 문헌을 검토하여 이 연구와의 관계를 설명한다. 이어서 제 5장에서는 기술한 내용의 구현과 적용 사례를 요약하고, 제 6장의 결론으로 논문을 마친다.

2. 문제의 정의

2.1 전사적 시스템 아키텍처

전사적 시스템은 다양한 방법으로 구성할 수 있지만, 이 연구는 <그림 1>과 같은 전문(message)으로 정보를 교환하는 아키텍처로 구성된 응용 시스템의 테스트 지원을 목표로 한다. 다층 구조(multi-layer)를 갖는 이 아키텍처는 그림에서 보는 바와 같이 화면, MCA, AP, DB의 계층으로 시스템의 기능을 전문화(specialize)한다. 화면은 사용자 인터페이스를 의미하며 여러 종류의 사용자 기기(client device)를 지원할 수 있다. MCA는 multi-channel architecture의 약어이며, 다양한 사용자 채널로부터 입/출력되는 다양한 형식의 데이터를 응용시스템에서 처리할 수 있도록 표준 전문 체계로 변환하여 준다. AP(application program)서버는 MCA를 통하여 사용자와 정보를 주고 받으면서 비즈니스 프로세스를 처리하고, DB서버는 AP의 비즈니스 수행에 필요한 데이터를 지원한다. 응용 프로그램은 사용자의 요청과 업무를 처리한 결과를 로그 파일에 기록하며, 로그에는 응용 프로그램에서 실행하는 SQL 문도 포함되어 있다.

이러한 아키텍처는 대규모 거래처리 시스템에서 많이 사용하는 구조로써, 우리나라 금융권에서 이루어지는 거래(transaction)는 모두 이 아키텍처로 구성된 시스템으로 처리되고 있다고 하여도 과언이 아니다.



<그림 1> 전사적 시스템 아키텍처

2.2 단순 프로그램 테스트

<그림 2>는 신용카드의 등급을 확인하는 클래스인 CardGrade를 정의한 프로그램이며, getCardGrade()는 문자열 타입으로 카드번호를 입력 받고, 처리 결과로 정수 타입인 등급을 반환하는 함수(method)이다. 이 프로그램은 외부 프로그램이나 데이터베이스와 인터페이스가 없기 때문에 단독 프로세서(CPU)에서 테스트가 가능하다. 테스트는 카드번호의 입력 값으로{50, 150, 250, 350}와 같은 4가지의 경우에 대해 실행하고 등급의 그 결과 값으로 각각{1, 2, 3, 4}를 확인한다.[1] 이와 같이

```
public class CardGrade {
    public static void main(String [] args) {
        int grade = getCardGrade(116);
    }
    public int getCardGrade(String 카드번호) {
        if (카드번호 >= 1 && 카드번호 < 100) {
            return 1;
        } else if (카드번호 >= 100 && 카드번호 < 200) {
            return 2;
         else if (카드번호 >= 200 && 카드번호 < 300) {
            return 3;
        } else {
            return 4;
        }
    }
}
```

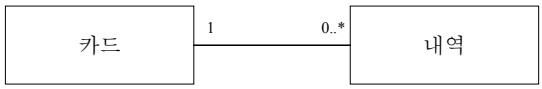
<그림 2> DB을 사용하지 않는 프로그램

1) 설명의 편의를 위하여 오류처리 루틴과 오류 및 한계값 테스트는 생략하였다.

DB를 사용하지 않는 프로그램의 경우에는 입력 값과 결과 값만 고려하면 되기 때문에 테스트 데이터를 쉽게 작성할 수 있을 뿐만 아니라 JUnit과 같은 도구를 이용하여 테스트의 실행과 결과 비교를 자동화할 수 있다(Beck and Gamma, 2011).

2.3 데이터베이스 응용 프로그램

데이터베이스 응용 프로그램은 프로그램과 데이터베이스의 상호작용을 포함하기 때문에 단순 프로그램에 비해서 훨씬 더 복잡한 구조로 작성될 수 밖에 없다. <그림 3>은 이 논문의 예제에서 사용하는 데이터베이스 구조이다.



<그림 3> 신용카드 데이터베이스

카드 실체(entity)는 신용카드를 의미하며, **내역** 실체는 신용카드의 사용내역이며, 양자는 일대다(1 : N) 관계를 갖는다. **카드**와 **내역** 실체의 속성은 <그림 4>의 테이블 생성 DDL(data definition language)에서 보는 바와 같이 예제의 설명에 필요한 최소한으로 제한하였다.

```
CREATE TABLE 카드 (
    카드번호          VARCHAR2(16) PK,
    효효년월          VARCHAR2(6)
    합계              INTEGER
)
CREATE TABLE 내역 (
    순번              NUMBER PK,
    카드번호          VARCHAR2(16) FK,
    사용일시          VARCHAR2(14),
    사용금액          NUMBER
)
```

<그림 4> 테이블 생성 DDL

<그림 5>는 신용카드의 사용내역을 기록하는 CardHistoryRecord 클래스 정의로서; card와 history의 두 개의 속성과 addService()와 addCard History()의 두 개의 함수를 포함하고 있음을 보여주고 있다. 속성 타입 카드_t와 내역_t는 사용자가 정의한 별도의 클래스이다.

```
public class CardHistoryRecord {
   카드_t card = new 카드_t();
   내역_t history = new 내역_t();
   public static void addHistory (카드_t card, 내역_t history);
   public boolean recordHistory (Connection con, UUID uuid
                          , 카드_t card, 내역_t history);
}
```

<그림 5> CardHistory 클래스 정의

```
public static void addHistory (카드_t card, 내역_t history) {
    String uuid = getUUID();
    Connection con = DriverManager.getConnection
                          (url,"","");

    card.set카드번호 ("1234567890123456") ;
    card.set유효년월 ("202009") ;
    card.set합계 (0) ;

    history.set카드번호 (카드.get카드번호( )) ;
    history.set사용일시 ("20111120182030") ;
    history.set사용금액 (6000) ;

    CardHistoryRecord ch = new CardHistoryRecord( ) ;
    boolean ret = ch.recordHistory (con, uuid, card, history) ;
    if(ret == true) {
        log (uuid, "[output] 거래 등록이 정상처리
             되었습니다.");
    } else {
        log (uuid, "[output]거래 등록이 실패 하였습니다.");
    }
    con.close ( );
}
```

<그림 6> addHistory 함수 정의

<그림 6>의 addHistory()는 카드(card)와 내역(history) 객체로 카드와 사용 정보에 대한 입력 값을 설정한 후에 recordHistory()를 호출하여 반

환된 거래처리 결과를 정상처리/실패로 구분하여 로그파일에 출력한다. getUUID()는 거래의 식별에 필요한 고유 거래식별자(unique identifier)를 생성하는 함수이며, log()는 거래에 관한 주요 사항을 input과 output으로 구분하여 로그파일에 기록하는 함수이다.

```
public boolean recordHistory (Connection con, String uuid)
{
    PreparedStatement stmt = null;
    String sql;

    log (uuid, "[input] 카드번호 ="+card.get카드번호( ) ;
    log (uuid, "[input] 유효년월 ="+card.get유효년월( ) ;
    log (uuid, "[input] 합계 =" +card.get합계());
    log (uuid, "[input] 카드번호 =" +history.get카드번호( ) ;
    log (uuid, "[input]사용일시 =" +history.get사용일시( ) ;
    log (uuid, "[input]사용금액 =" +history.get사용금액( ) ;
    try {
      sql = "INSERT 내역 VALUES (MAX(순번)+1,'"
            +history.get카드번호( )+"','"
            +history.get사용일시( )+"','"
            +history.get사용금액( )+"') ;
      stmt.executeUpdate(sql) ;
      log (uuid, sql) ;

      sql = "SELECT 합계"
            +"FROM 카드"
            +"WHERE 카드번호=?";
      pstmt = conn.prepareStatement(sql);
      pstmt.setString(1, card.get카드번호());
      ResultSet rs = pstmt.executeQuery();
      log (uuid, sql);

      int 금액 = rs.getInt("합계")+history.get사용금액( ) ;
      sql = "UPDATE 카드"
            +"SET 합계 ="+금액
            +"WHERE 카드번호 ='"
            +card.get카드번호( )+ "'" ;
      stmt.executeUpdate (sql) ;
      log (uuid, sql);
      return true;
    } catch (Exception e) {
      return false;
    } finally {
      stmt.close();
    }
}
```

<그림 7> recordHistory 함수 정의

<그림 7>의 recordHistory()는 데이터베이스 연결 식별자(con)와 거래식별자(uuid)를 인자로 받고, 거래의 처리결과를 정상처리와 실패로 구분한 Boolean 값으로 반환한다. recordHistory()의 처리 프로세스를 보면, 신용카드(card)와 사용내역(history) 정보를 로그 파일에 저장하고, 동일한 정보로 SQL 문을 작성하여 이를 실행한 후에 실행된 SQL 문을 로그 파일에 저장한다. 기술된 SQL 문은 내역 테이블에서 지불금액을 조회하고, 다시 기존 합계와 지불금액을 더해서 카드 테이블의 합계를 수정한다.

addHistory()의 실행이 종료되면 <그림 8>과 같은 로그가 기록된다. 로그 파일은 정상으로 처리된 경우와 오류가 발생한 두 경우를 예시로 보여준다. 각 줄의 처음에 나오는 큰 숫자는 거래의 고유 식별자이며(예 : 123456789001, 123456790001), 그 이후는 신용카드와 거래 정보에 관한 메시지이다. 메시지는 [input]과 [output]으로 입력과 출력을 구분하고 있으며, 변수의 값은 "변수명 = 값"의 형태로 표시된다. SQL 문은 recordHistory()에서 작성된 값 또는 JDBC에서 사용하는 '?'를 포함하는 형태로 저장한다.

2.4 수작업에 의한 테스트

만일 데이터베이스를 사용하지 않는 프로그램이라면 <그림 9>의 1과 3에 해당하는 부문의 작업만으로 프로그램을 테스트할 수 있으나, 데이터베이스 응용 프로그램의 경우에는 2와 4의 작업이 추가되어야 테스트를 마칠 수 있게 된다.

이러한 추가의 작업이 개발자의 생산성을 저하시키는 요인은 크게 세 가지로 나타난다. 첫 째는 <그림 1>의 전사적 아키텍처에서 살펴 본 바와 같이 응용 프로그램이 실행되는 환경이 매우 복잡

하여 테스트 환경을 구성하는데 많은 노력이 필요하다. 이와 연관하여 두 번째 문제로 테스트의 결과를 확인하는 절차도 단순 프로그램과 달리 복잡한 절차를 거쳐야 하는 경우가 많다. 세 번째는 프로그램의 품질 향상을 위하여 반복적인 테스트가 필요하나, 수작업의 특성 때문에 동일한 작업을 똑같이 반복하여야 한다. 프로그램에서 사용하는 테이블의 개수가 늘어나면 개발자에게 부가되는 작업의 양은 감당할 수 없을 정도로 급격하게 증가한다.

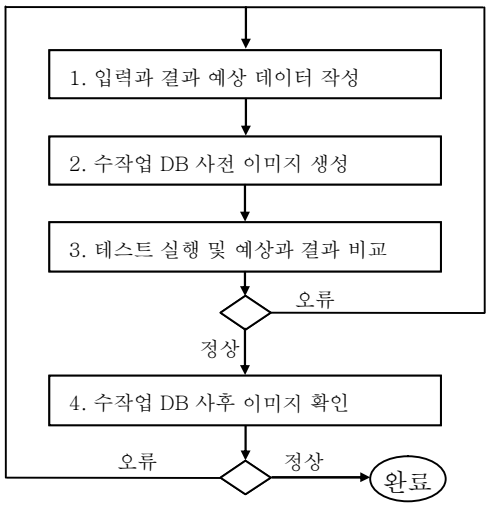
이 논문은 세 번째 문제의 해결 내지는 현격한 완화를 목표로 하며, 부가적으로 두 번째의 문제의 해소에도 크게 기여할 것으로 예상한다.

```
…
123456789001 [input]카드번호 = 1234567890123456
123456789001 [input]유효년월 = 202009
123456789001 [input]합계 = 0
123456789001 [input]카드번호 = 1234567890123456
123456789001 [input]사용일시 = 20111120182030
123456789001 [input]사용금액 = 6000
123456789001 INSERT 카드(카드번호, 합계)
             VALUES('1234567890123456', ?)
123456789001 INSERT 내역 VALUES(MAX(순번)+1,
             '1234567890123456', '20111120182030',
             60000)
123456789001 SELECT A.합계 FROM 카드 A WHERE
             A.카드번호 = ?
123456789001 UPDATE 카드 SET 합계 = 6000
             WHERE 카드번호 = '1234567890123456'
123456789001 [output]거래 등록이 정상처리
             되었습니다.
123456790001 [input]카드번호 = 1234567890654321
123456790001 [input]유효년월 = 202009
123456790001 [input]합계 = 0
123456789001 [input]카드번호 = 1234567890123456
123456789001 [input]사용일시 = 20111120182030
123456789001 [input]사용금액 = 6000
123456790001 INSERT 카드(카드번호, 합계)
             VALUES('1234567890654321', ?)
123456790001 [output] 거래 등록이 실패하였습니다.
…
```

<그림 8> 프로그램이 남긴 로그

2.5 테스트 도구를 이용한 테스트

<그림 10>에서 보는 바와 같이 첫 회의 테스트(그림의 1번 항목)는 수작업 테스트와 동일하다. 그러나 테스트가 한 번 실행된 이후에는 로그에 남아있는 정보를 이용하여 사전 이미지를 자동으로 생성하여 줌으로써, 개발자가 추가 작업에 대한 부담을 갖지 않고 테스트를 반복할 수 있다. 앞에서 언급한 것처럼, 프로그램에서 사용하는 테이블의 개수가 증가할수록 DB이미지의 자동 생성에 대해 개발자가 느끼는 업무의 경감 효과는 급격하게 커질 것이다. 또한 전사적 응용시스템에서 핵심 업무를 다루는 중요한 프로그램일수록 다루어야 할 테이블 개수가 많아지는데 비례해서, 개발자가 사전 이미지 생성에 부담을 느껴 테스트의 수행을 꺼리게 되는 모순된 현상의 해소도 가능해 진다. 이러한 점으로 미루어볼 때, 로그를 이용한 사전 이미지 생성의 자동화는 개발자의 생산성 향상과 더불어 프로그램과 시스템의 품질 향상에 크게 기여할 것으로 예상할 수 있다.

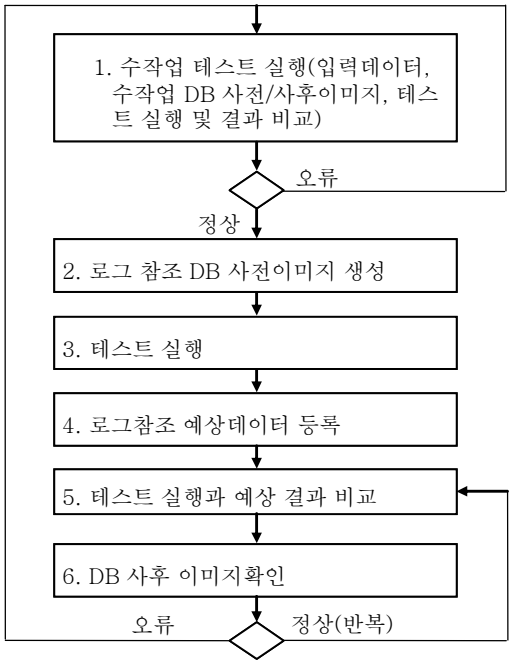


<그림 9> 수작업 테스트 절차

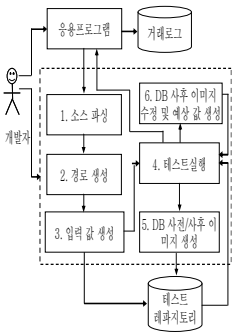
이 논문의 주제인 DB이미지 자동 생성은 테스트케이스 자동생성 연구의 일환으로 수행되었다. 이에 대한 연구는 현재 상당한 진전을 보이고 있으며, 전체적인 프레임워크는 <그림 11>과 같다. 다음은 그림에서 점선으로 표시된 단계별 처리 내용의 개략적인 소개이다.

1. 소스코드를 파싱(parsing)하여 얻어진 정보를 이용하여 테스트 데이터 생성에 필요한 정보(예 : 입력변수, 대입문, 조건문)를 추출하여 추론에 필요한 데이터베이스를 구축한다. 데이터베이스는 CFG(Control Flow Diagram)와 DFG (Data Flow Diagram)를 구조화된 데이터로 표현한다.
2. 생성된 데이터베이스를 이용하여 프로그램에서 독립적으로 실행할 수 있는 경로로 구성된 집합(independent paths)을 도출한다(McCabe, 1989).
3. 생성된 경로에 포함된 개별 경로를 실행할 수 있는 테스트 데이터를 생성한다. 이를 위해서는 1에서 생성된 데이터베이스에서 해당 경로 선택에 필요한 정보를 추출하여 선형 방정식을 작성하고, 이에 대한 해를 구해서 경로를 실행할 수 있는 입력 데이터를 생성한다(Gupta et al., 1998; Sen et al., 2005).
4. 생성된 입력 값을 이용하여 테스트를 실행한다.
5. 4의 과정에서 남겨진 로그 정보에서 SQL 문을 추출하고, 이 논문에서 제시하는 방안으로 사전 이미지를 생성한다.
6. 생성된 입력 값과 사전 이미지를 이용하여 테스트를 실행하여 결과를 확인하고, 이 결과를 테스트 저장소에 기록한다.
7. 회귀테스트를 위해서 4에서 6까지를 반복한다.

이 논문은 앞에서 기술한 것처럼 위의 5항(<그림 11>에서도 5번 항목)에 대한 연구 결과이다.



<그림 10> 테스트 도구를 이용한 절차

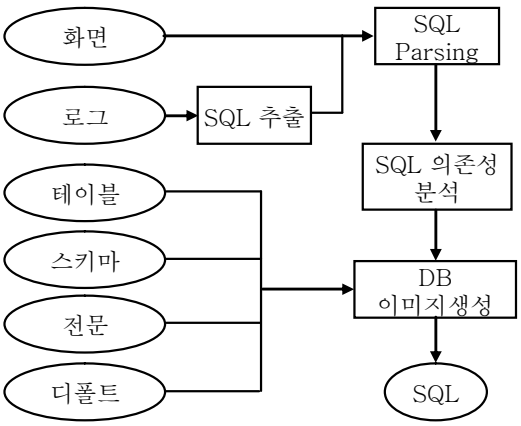


<그림 11> 테스트 데이터 자동생성 프레임워크

3. DB이미지 생성

데이터베이스 응용 프로그램을 테스트하려면 데이터베이스가 적합한 상태로 구축되어 있어야 하며, 테스트가 종료된 시점에는 프로그램에서 조작

한 내용이 반영된 상태로 변경되어야 한다. 이러한 조건을 만족시키는 DB이미지를 생성하려면 복잡한 절차를 거쳐야 한다. <그림 12>는 DB이미지를 자동으로 생성하는 절차를 개념적으로 보여주고 있다.



<그림 12> DB이미지 생성 개념도

DB이미지 생성의 첫 단계는 화면으로부터 SQL을 입력 받거나 이전 테스트 결과로 남은 로그파일의 기록에서 텍스트 SQL 목록인 OQS(Original Query Sequence)를 생성한다. 다음의 SQL Parsing 단계에서는 OQS에서 후속 작업을 지원하는 parse tree의 목록인 PQS(Parsed Query Sequence)를 도출한다. 세 번째 단계에서는 SQL들 사이의 의존관계를 분석하여 함께 고려해야 할 순차집합들을 분리해 내고, 각 순차집합에 필요한 작업의 종류를 결정한다. 마지막 단계에서는 DB이미지 생성에 여러 원천에서 수집한 정보를 이용하여 DB 이미지 생성 SQL을 조립한다. SQL 조립에는 로그 이외에도 생성 대상 테이블에 존재하는 레코드, 테이블 정의 스키마, 메시지 전문에 포함된 정보를 이용하며, 때에 따라서는 사전에 정의한 디폴트 값을 사용하기도 한다. 사전 이미지로는 UPDATE/INSERT 또는 DELETE SQL을 조립하고, 사후 이미지로는 데이터베이스의 사후 상태를 조회할 수 있는 SELECT SQL을 조립한다.

3.1 SQL 추출

최근에 구축되는 시스템들은 거래처리의 기록을 파일이나 데이터베이스에 저장한다. 테스트 결과도 일반적인 거래처리와 마찬가지로 로그에 기록을 남기게 할 수 있으며, DB이미지 생성에 필요한 SQL도 로그의 일부로 포함된다. 그 외에도 데이터베이스 언어로 작성된 stored procedure와 trigger, 타 응용 프로그램과 모듈, 그리고 연관된 다른 응용시스템에서 남기는 로그가 있다. 이 연구에서는 응용 프로그램과 모듈이 남기는 로그에 중점을 두고, DBMS에 내장되는 stored procedure 및 trigger와 Java stored procedure로 존재하는 SQL의 지원은 향후 연구 과제로 남긴다.

거래처리 기록은 동일 거래와 관련된 처리 내용의 추적을 위해서 거래식별자인 UUID(Universal Unique Identifier)를 사용한다. DB이미지 생성을 위한 SQL 추출에서도 동일한 UUID로 시작되는 로그 기록은 동일한 테스트(운용의 경우에는 동일한 거래)에 포함되는 SQL로 파악한다.

로그는 UUID로 시작된다. 한 건의 로그는 줄(line)의 수에 상관없이 다음 UUID가 나오기 전까지 읽어서 추출되는 SQL을 차례대로 OQS의 마지막 원소(member)로 추가한다. OQS는 동일한 UUID(동일한 테스트)로 기록된 SQL들을 원소로 하는 순차집합으로 정의한다. 현재 처리하고 있는 테스트의 UUID와 다른 UUID를 만날 때까지 이 절차를 계속하여 하나의 테스트에 해당하는 OQS를 생성한다. 로그는 프로그램이 실행되는 순서대로 기록되기 때문에 OQS는 SQL의 실행 순서를 유지한다.

개발자가 화면을 이용하여 DB이미지 생성에 필요한 SQL을 입력하는 경우에는 로그 파일의 처리 없이 직접 OQS를 생성한다. 최초 테스트의 경우에는 입력 화면을 이용하여 DB이미지를 생성할 수 있으며, 입력된 순서와 OQS의 순서는 동일하다.

3.2 SQL Parsing

Parsing 단계에서는 텍스트 SQL의 순차집합인 OQS을 차례로 읽어서 SQL 구문요소 계층구조(parse tree)의 순차집합인 PQS를 도출한다. Parsing과 더불어 각 SQL은 SELECT, INSERT, UPDATE, DELETE 중의 하나로 분류한다. 프로그램 언어의 컴파일과 마찬가지로 SQL의 Parsing에서도 구문의 정확한 사용 여부를 확인한다. 다음은 SQL 구문 확인의 몇 가지 예시이다:

1. SELECT, INSERT, UPDATE, DELETE의 표준 문법(syntax)을 점검한다. 예를 들어 조회 SQL의 경우SELECT, FROM, WHERE와 같은 구문과 DISTINCT, ALL과 같은 키워드의 정확한 사용여부를 확인한다.
2. subquery는 SELECT에 준하여 확인한다.
3. 대소문자, 줄 바꿈과 공란, comment, EXEC 등을 변환하거나 제거하여 표준 SQL로 바꿔준다.
4. alias, meta-character(*), 그리고 DBMS에서 지원하는 함수의 정확한 사용을 확인한다. alias의 경우에는 영문, 한글 및 한영 혼합에 대한 정확한 사용 여부를 확인한다.

정제된 PQS는 후속 작업을 위해서 필요한 다음의 특징을 갖는다 :

1. OQS에 포함된 SQL은 텍스트인데 비하여, PQS에 포함된 SQL은 구문요소의 계층구조로 표현된다. 이 구조는 사전 이미지 생성 작업에서 SQL 구성요소의 검색과 조작을 용이하게 한다.
2. OQS와 PQS의 모든 원소는 1:1 대응 관계를 가지며 동일한 순서를 유지한다. 즉, PQS에 포함된 SQL은 테스트 결과로 로그에 기록되거나 화면을 통해 입력된 순서를 유지하고 있다.
3. PQS에 포함된 SQL은 이에 대응하는 OQS에 포함된 SQL의 구문 오류가 제거된 상태이다.
4. PQS에 포함된 모든 SQL은 4가지 DB 조작 중의 하나로 분류되어 있다.

3.3 QDS(Query Dependency Set)

3.3.1 QDS의 정의

PQS를 단순한 집합이 아닌 순차집합(Charles, 2002)으로 유지하는 이유는, 실행되는 순서를 이용하여 데이터베이스 조작 의도를 추론할 수 있기 때문이다. 물론 순서만 가지고 조작 의도를 추론하는 것은 불가능하다. 하지만 복수의 SQL의 조작 대상 데이터가 동일하다면 SQL의 순서에 의해서 그 실행의도를 유추하는 것이 가능하다. 예를 들어, <SELECT, INSERT>의 조작 대상 데이터가 동일하다면, 그 순서에 의해서 SELECT는 INSERT로 입력하려는 데이터가 없음을 확인하려는 조회임을 알 수 있고, <SELECT, DELETE>의 경우의 SELECT는 삭제 대상 데이터가 있음을 확인하려는 조회라는 추론이 가능하다. 여기서 조작 대상 데이터가 같다는 것은 SQL에 포함된 WHERE 구문이 동일하다는 것을 의미한다. WHERE 구문이 없는 INSERT의 경우에는 VALUE 구문의 데이터 값들이 조건과 같은 역할을 한다. 이상의 논의를 바탕으로 다음과 같은 작업을 정의할 수 있다:

1. PQS에 포함된 SQL에서 WHERE 구문을 추출하여 서로 다른 조건만으로 구성된 CS(Condi-

tion Set)를 생성한다.

2. PQS를 CS로 등가 분할(equivalence partition)하고 그 결과로 나타나는 n개(CS의 원소의 수)의 순차집합을 생성한다. 생성되는 순서에 따라 첨자를 부여하여 $QDS_i(0 < i < n+1)$라고 부른다.

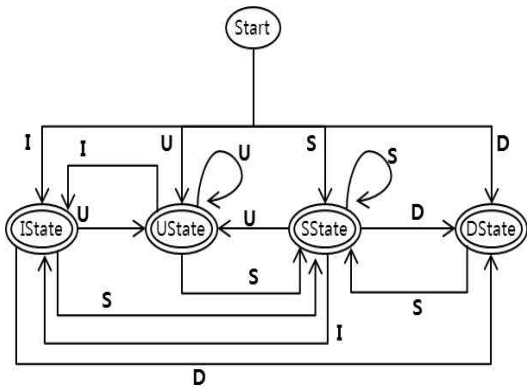
QDS는 Query Dependence Sequence를 의미하며, 동일한 QDS_i에 포함된 SQL들은 OQS와 PQS에서 가지고 있던 순서를 유지하기 때문에, 원소들 사이에 의존관계가 있음을 표현하고 있다. 그러나 서로 다른 QDS_i에 포함된 SQL들은 조작 대상 데이터가 달라 실행 순서가 테스트(프로그램 실행 결과)에 영향을 미치지 않는다.

QDS의 예로, PQS = $\{I_{C1}, I_{C2}, S_{C1}, U_{C1}\}$에서 I, S, U는 SQL의 조작유형인 INSERT, SELECT, UPDATE를 의미하고, C_i의 i값에 따라 조건(WHERE 구문)이 동일하거나 상이함을 나타낼 때, CS = $\{C_1, C_2\}$이고, PQS를 CS로 등가 분할 한 결과는 $QDS_1 = \{I_{C1}, S_{C1}, U_{C1}\}$와 $QDS_2 = \{I_{C2}\}$가 된다. QDS_2와 같이 WHERE 구문이 없는 INSERT가 한 개인 경우에는 주 키(primary key)가 조작 대상 데이터를 결정하는 조건이 된다.

3.3.2 QDS 패턴과 문법

QDS_i에 포함된 SQL을 순차적으로 실행하면 그에 따라 테스트 대상 프로그램의 상태가 변화해 간다. 예를 들어 SELECT 문이 실행되면 프로그램은 조회 상태에 있게 되며, 이어서 DELETE 문이 실행되면 다시 삭제 상태로 전환된다. 기술적으로는 어떤 조합의 SQL도 가능하지만, 실질적으로는 연속된 조작이 의미가 없거나 바람직하지 않을 수 있다. 예를 들어 동일한 데이터를 반복해서 입력하거나 삭제하는 일련의 조작은 데이터의 정합성 측면에서도 바람직하지 않을 뿐만 아니라 실무적인 상식에도 어긋난다.

QDS_i에 포함된 SQL의 실행은 추상적 기계(abstract machine)의 상태를 전이(state transition)시키는 것으로 볼 수 있다. 여기서 추상적 기계는 테스트 대상 프로그램 또는 조작 대상 데이터이다. <그림 13>에서 Start는 시작 상태이며, 입력, 수정, 조회, 삭제 상태를 나타내는 IState, UState, SState, DState는 기계가 거칠 수 있는 중간 상태이다. 크기가 다른 두 개의 타원으로 표시한 네 개의 상태는 프로그램의 상태가 진입할 수 있는 중간 상태임을 의미한다. 그림의 단순화를 위하여 종료 상태는 표시하지 않았다. 화살표는 상태의 전이(변환)를 표시하며, 화살표에 병기된 문자가 전이의 조건 또는 전이를 야기하는 이벤트를 표시한다. 즉, 화살표에 병기된 문자에 해당하는 조작이 시작되면 기계는 화살표가 시작되는 쪽의 상태에서 화살표가 가리키는 쪽의 상태로 전환하게 된다.



<그림 13 > QDS의 상태 전이 도형

<그림 13>에서 보는 바와 같이, QDS_i에 포함된 첫 번째 SQL이 시작되면 프로그램은 그 유형에 해당하는 화살표가 가리키는 상태로 진입한다. 시작과 조회 상태에서는 다음 조작에 제약이 없지만, 다른 상태에서는 다음에 진입할 수 있는 상태, 즉

다음에 실행할 수 있는 조작의 유형에 제약이 있다. 예를 들어서 삭제한 데이터를 이어서 다시 수정하는 조작은 발생하지 않는 업무 패턴이다. 그러나 자료를 임시로 등록(입력)하고 작업이 종료된 후에 삭제하는 작업은 가능하다. <그림 13>은 일반적인 업무환경에서 나타나는 QDS_i 패턴의 상태 전이 도형이다. 업무의 종류에 따라 <그림 13>과 다른 SQL 패턴을 지원하는 것도 가능하다.

<그림 13>의 상태 전이 도형으로 표현한 QDS 패턴은 다음과 같은 문법(context free grammar)으로도 정의할 수 있다:

Start → S *SState*|I *IState*|U *UState*|
D *DState*
SState → S *SState*|I *IState*|U *UState*|
D *DState* | *End*
IState → S *SState*|U *UState*|D *DState*|*End*
UState → S *SState*|I *IState*|U *UState*|*End*
DState → S *SState*|*End*
End → ε

문법에서 ε는 더 이상의 작업이 없거나 치명적인 오류가 발생하여 종료 상태로 진입해야 함을 의미하고, ***End***는 종료 상태로 진입하였음을 의미한다. 상태 전이 도형과 문법은 허용되지 않는 QDS 패턴의 사용을 원천적으로 막아준다. 또한 정형적인 문법은 오류의 방지에 추가해서 컴파일러 기법을 이용한 메시지 지원이 가능하다(Aho et al., 1986). 예를 들어, 중복된 삭제에 대한 오류 메시지를 보여주기 위해서 DState의 구문을 다음과 같이 보완할 수 있다:

DState → S *SState*|D *DError*|*End*
DError → {print "중복 삭제 입니다."} *End*

<그림 13>의 모형을 지원하는 도구에서는 <I, I>, <U, S>, <D, I>, <D, U>로 나타나는 조작 유형에 대해서도 앞에서 기술한 <D, D>와 유사하게 정의할 수 있다.

3.3.3 DB이미지 생성 작업 유형 추론

개별 QDS_i에 포함된 일련의 SQL은 동일한 조건(CS의 특정 원소인 WHERE 구문)이 지정하는 동일한 데이터를 조회, 입력, 수정 또는 삭제한다. 따라서 사전 이미지의 생성으로 데이터베이스 상태가 QDS_i에 포함된 첫 번째 SQL의 의도를 반영할 수 있도록 만들어주어야 한다. 예를 들어, QDS_1 = {I_{C1}, S_{C1}, U_{C1}}는 조건 C_1에 해당하는 데이터를 입력한 후에 결과를 조회하고 다시 수정하려는 일련의 SQL이 제대로 수행되는지의 여부를 테스트하고자 하므로, 이에 대한 사전 이미지 생성은 C_1에 해당하는 데이터를 삭제하는 조작이 필요하다. 이처럼 QDS_i가 두 개 이상의 SQL을 포함하고 있는 경우에는 유일한 조작을 추론하는 것이 가능하지만, SQL이 하나 밖에 없는 경우에는 그 의도를 자동으로 결정할 수 없는 것이 보통이다. 예를 들어, QDS_2 = {I_{C2}}의 경우에는 조건 C_2에 해당하는 데이터의 입력을 테스트 하거나, 동일한 데이터가 이미 존재하기 때문에 오류가 발생함을 테스트 할 수 있다. 따라서 전자의 경우에는 C_2에 해당하는 데이터를 삭제해야 하며, 후자의 경우에는 동일한 데이터를 입력해 주어야 한다.

지금까지 직관적으로 기술한 내용을 바탕으로 사전 이미지 생성을 위한 작업 유형을 다음과 같이 결정할 수 있다. QDS_i에서 첫 번째 SQL만 감안하면 아래의 규칙과 같이 사전 이미지 생성을 위한 삭제와 입력이 모두 가능하다.

SELECT → {INSERT, DELETE}

/* {정상, 0건 조회} */
INSERT → {DELETE, INSERT}
/* {정상, 중복 입력} */
UPDATE → {INSERT, DELETE}
/* {정상, 0건 수정} */
DELETE → {INSERT, DELETE}
/* {정상, 0건 삭제} */

결과적으로 QDS_i의 질의를 테스트하기에 적합한 사전 이미지를 생성하려면, QDS_i에 두 번째 SQL이 있는 경우와 그렇지 않은 경우를 모두 고려하여 사전 이미지 생성 유형을 파악해야 한다. 먼저 QDS_i에 두 개 이상의 SQL이 있는 경우에는, 첫째와 둘째 SQL의 패턴을 분석하여 사전 이미지 생성 유형을 결정한다:

1. 두 SQL의 패턴이 <S, S>, <S, U> <S, D>, <U, S>, <U, U>, 또는 <D, S>인 경우에는 '등록' 유형
2. <S, I>, <I, S>, <I, U>, <I, D> 또는 <U, I>인 경우에는 '삭제' 유형
3. <I, I>, <U, D>, <D, I>, <D, U> 또는 <D, D>의 경우는 앞 단계에서 이미 오류로 처리

'삭제' 유형은 조건에 해당하는 데이터의 삭제가 필요한 경우이며, '등록' 유형은 새로운 데이터의 입력 또는 기존 데이터의 수정이 필요한 경우를 의미한다.

QDS_i에 한 개의 SQL만 있으면 위와 같은 패턴에 의한 추론이 불가능하다. 이 경우에는 사전에 정해 놓은 유형을 적용하거나 도구의 사용자가 적절한 유형을 선택하는 방법이 있다. 사전에 정의된 방법은 자동화가 가능하다는 장점이 있고, 사용자의 선택은 테스트의 의도와 부합된 유형을 결정할 수 있다는 장점이 있다. 실제로 이미지 생성을 지원하는 도구를 구현할 때는 미리 DB이미지 생성 유형을 정해놓고 도구의 사용시에 정해진 유형을 변경할 수 있는 신축성을 부여하는 것으로 예상할 수 있다.

이상을 요약하면 다음과 같다:

1. QDS_i의 모든 i에 대해서 사전 이미지 생성을 위한 SQL 유형을 결정할 수 있다.
2. QDS_i에 대한 DB이미지 생성 SQL의 유형이 같더라도 i가 다르면 적용되는 데이터의 조건 C_i도 다르다.

PQS = {QDS_1, QDS_2}, QDS_1 = {I_{c1}, S_{c1}, U_{c1}}, QDS_2 = {I_{c2}}의 예에서, QDS_1를 위한 사전 이미지 생성은 삭제 유형이며, QDS_2는 삭제 또는 등록 유형이 된다. 후자의 경우에는 사전에 지정한 유형을 선택하거나 사용자가 유형을 결정한다. 사후 이미지 생성을 위한 작업은 모두 '조회' 유형이다.

3.4 DB이미지 생성 SQL 조립

일반적으로 테스트에 포함된 SQL, 특히 조회의 경우에는 그 구조가 매우 복잡하다. 그러나 사전 이미지는 테이블 단위로 생성해야 하기 때문에 해당 QDS_i의 첫 번째 SQL에서 생성 대상 테이블, 속성 및 WHERE 조건을 추출해야 한다. 이미지 생성을 위해서 필요한 테이블이나 조건은 PQS(QDS_i)가 유지하는 SQL 구문요소 계층구조에서 용이하게 검색하고 추출할 수 있다.

3.4.1 대상 테이블 목록의 결정

SQL은 한 개이지만 조인이나 subquery가 있는 경우에는 DB이미지를 생성해야 할 테이블은 여러 개가 될 수 있다. 이 경우 각 테이블은 별도의 생성 대상이다. 파악된 테이블 집합에서 제외 테이블

은 생성 대상에서 제거한다. 제외 테이블은 DB이미지 생성이 필요하지 않거나, 레코드를 변경하지 말아야 할 테이블을 의미한다. 예를 들어, 부서, 사용자, 영업일자와 같이 공통으로 사용하는 테이블과 데이터를 변경하지 말아야 할 코드 테이블이 제외 대상이다. 또한 통장번호와 같이 일련번호를 부여하기 위한 채번 테이블의 경우에도 DB이미지 생성을 위한 조작이 불필요하다. 제외해야 할 테이블 목록은 DB이미지 생성 이전에 결정되어 있는 것으로 가정한다.

QDS_i의 첫째 SQL에 참여하는 테이블 중에서 제외 테이블을 뺀 나머지가 DB이미지 생성 대상이 된다. 이 대상 목록에 포함된 각각의 테이블에 대해서 테스트를 위한 사전과 사후의 이미지를 생성해 주는 한 쌍의 SQL이 있어야 하기 때문에, 아래의 WHERE 구문의 조립부터 DB이미지 생성 SQL 조립까지에 기술된 내용이 대상 목록에 포함된 각각의 테이블에 적용된다.

3.4.2 WHERE 구문의 조립

테이블의 해당 레코드를 지정하는 WHERE 구문은 해당 QDS_i의 C_i에서 SQL을 생성하고자 하는 테이블에 속한 조건만 추출하여 구성한다. 조인 조건은 개별 테이블에 대한 조건으로 분해하여 해당 테이블의 조건으로 추가한다. 예를 들어, A.attr1 = B.att2라는 조인 조건은 테이블 A의 attr1과 테이블 B의 attr2에 아래의 속성 값 결정 방법에 의해 동일한 값을 부여하고, 예를 들어, 자료 유형에 따른 디폴트 값인 0을 부여하고, 테이블 A의 생성 SQL의 WHERE 구문에 attr1 = 0, 테이블 B의 SQL에는 attr2 = 0이라는 조건을 추가한다.

3.4.3 속성의 결정과 속성 값 부여

데이터베이스의 정합성을 유지하고 후속 SQL의 작업을 지원하기 위해서 테이블의 모든 속성을 생성 대상으로 한다. 속성 목록은 대상 데이터베이스 스키마에서 검색한다. 위의 WHERE 구문의 조립에서 사용한 모든 속성은 스키마에서 검색된 속성 목록에 포함되어 있어야 한다.

사전 이미지 생성 작업의 유형, 테이블, 속성 및 조건이 결정되어도 SQL을 작성할 수 없다. 아직 필요한 속성 값을 모르기 때문이다. 앞에서 부여한 WHERE 구문의 조인 조건을 만족시키는 이미지 생성 SQL 조립에 필요한 속성 값에는 다음과 같은 종류가 있다:

1. WHERE 구문의 '?'로 표시된 파라메터 값
2. UPDATE의 SET 구문에서 사용할 값
3. INSERT의 VALUES 구문에서 사용할 값
4. 조인 조건을 만족시키는 속성 값

값이 필요한 속성이 결정되면 다음의 목록에서 가장 적합한 방법을 선택하여 속성 값을 부여한다 :

1. 로그 파일에서 추출
2. 해당 테이블에서 조회
3. 해당 C_i인 WHERE 구문에서 조건으로 사용된 값
4. 프로그램 변수와 데이터베이스 속성 이름의 대응에 의한 입력/결과 예상 테스트 데이터
5. 해당 테이블 스키마에 정의된 데이터 유형과 제약조건을 이용한 동치 클래스 또는 경계 값 분석에 의한 속성 값
6. 미리 정의한 디폴트 값

나열한 방법들은 기존 테이블에 조회로 얻어지는 값, C_i의 WHERE 조건에서 사용된 값, 로그 파일에 기록된 속성 값, 테스트 데이터와 대응된 속성 값의 순서로 실제 업무환경을 반영하고 있으며, 속성 값을 부여하는 방법에 따라 생성되는 사전 이미지의 실효성도 달라질 수 있다.

3.4.4 DB이미지 생성 SQL의 조립

테이블, 속성과 속성 값 및 WHERE 조건이 결정되면 해당 QDS_i의 이미지 생성 유형에 따라 SQL을 작성한다. 등록 유형은 테스트가 실행될 때 데이터가 반드시 존재해야 하는 경우이다. 테이블에 해당 레코드가 이미 존재하고 있다면 기존 레코드를 UPDATE하고; 레코드가 없어서 UPDATE가 실패하는 경우에는 새로운 레코드를 INSERT를 할 수 있도록 한 쌍의 SQL을 조립한다. 따라서 레코드를 생성할 때, UPDATE를 실행하여 실행결과가 정상이면 등록 작업을 종료하고, 그 결과가 오류이면 INSERT를 실행한다. 삭제 유형은 앞에서 수집한 정보에 의해 DELETE를 조립한다. 사후 이미지는 모두 조회 작업이며, 사전 이미지와 쌍으로 만들어진다.

3.4.5. DB이미지 생성 예제

예를 들어, QDS_i에 <그림 14>와 같은 두 개의 테스트 대상 SQL이 있다고 가정하면,[2] <S, U>의 패턴으로 '등록' 유형의 작업이 필요함을 알 수 있다.

```
SELECT A.카드번호, A.유효년월, A.합계,
       B.순번, B.사용일시, B.사용금액
 FROM 카드 A, 내역 B
WHERE A.카드번호 = B.카드번호
   AND A.카드번호 = ?
   AND A.유효년월 = ?

UPDATE 카드
   SET 유효년월 = ?, 합계 = ?
WHERE 카드번호 = ?
   AND 유효년월 = ?
```

<그림 14> 등록 작업이 필요한 SQL

2) QDS는 parse tree 형태로 SQL을 저장하나, 설명의 편의를 위해서 텍스트로 표기하였다.

이어서 첫 SQL인 SELECT의 구문요소 계층구조를 참조하여 이미지 생성 대상 테이블이 **카드**와 **내역**이며, 데이터베이스 스키마를 검색하여 속성 목록이(카드번호, 유효년월, 합계, 순번, 카드번호, 사용일시, 사용금액) 임을 파악한다. 이어서 로그파일에서 카드번호는 '1234567890', 사용일시는 '20111120182030', 사용금액은 6000인 기록을 발견하고, 나머지 속성 값에 대해서는 카드 테이블에서 해당 레코드의 주 키인 카드번호, 이 레코드의 속성인 합계와 유효년월에 대해 각각 '1234567890', 3000, '202009'인 값을 조회를 통해 얻는다면, <그림 15>와 같은 한 쌍의 SQL을 자동으로 조립할 수 있다.

```
/* 카드 테이블 */
   UPDATE 카드 SET 유효년월 =' ',합계 = 3000
   WHERE 카드번호 = '1234567890';

   INSERT INTO 카드
   VALUES ('1234567890', '202009', 3000)

/* 내역 테이블 */
   UPDATE 내역
   SET 순번 = 100, 사용일시 = '20111120182030',
       사용금액 = 6000
   WHERE 카드번호 = '1234567890';

   INSERT INTO 내역
   VALUES (100, '1234567890', '20111120182030', 6000)
```

<그림 15> 등록 유형의 이미지 생성 SQL

테이블에서 유효년월과 합계의 값을 구할 수 없다면 (space)와 0과 같이 해당 속성의 자료유형에 해당하는 디폴트 값으로 지정할 수도 있다.

```
   SELECT 유효년월
   FROM 카드
   WHERE 카드번호 = ?

   INSERT INTO 카드 (카드번호, 유효년월, 합계)
   VALUES (?, ?, ?)
```

<그림 16> 삭제 작업이 필요한 SQL

<그림 16>과 같은 테스트 대상 SQL은 <S, I> 패턴에 의해서 '삭제' 유형의 이미지 생성 작업임을 파악하고 등록 유형과 유사한 절차를 밟아 <그림 17>의 DELETE 문을 조립한다.

```
DELETE FROM 카드
WHERE 카드번호 = "1234567890"
```

<그림 17> 삭제 유형의 이미지 생성 SQL

<그림 18>에 생성된 DB 사후 이미지는 카드 테이블과 내역 테이블의 조건에 맞는 레코드가 존재하는 경우이다. 상단의 '속성=값'의 목록은 과거 테스트 결과가 기록된 로그에서 추출한 속성 값의 목록이며, 하단은 사전 이미지와 유사한 방법으로 조립된 SQL이다. 테스트가 종료되면 상단의 속성 값 목록과 하단의 SQL 실행 결과를 비교하여 테스트의 성공 여부를 확인한다. 테스트케이스가 1건을 조회하는 경우라면 <그림 18>와 같이 하나의 레코드가 생성되지만 여러 건을 테스트하는 경우라면 지정된 개수만큼 레코드를 생성한다.

```
카드번호 = '1234567890'
유효년월 = '202009'
합계 = 9000

SELECT 카드번호, 유효년월, 합계
FROM 카드
WHERE 카드번호 = '1234567890'

순번 = 100
카드번호 = '1234567890'
사용일시 = '20111120182030'
사용금액 = 6000

SELECT 순번, 카드번호, 사용일시, 사용금액
FROM 내역
WHERE 순번 = 100
```

<그림 18> 레코드를 이용하여 생성된 사후 이미지

테이블의 제약조건으로 외래 키(foreign key)가 정의되어 있으면 사전 이미지 생성을 위해서 수행하는 입력과 삭제가 훨씬 복잡해 질 수 있으나 이 연구에서는 외래 키가 정의되지 않은 것으로 가정하였다. 권오승, 홍사능(2008)에서 제시한 외래 키 문제의 해결 방안은 이 논문에서 기술한 DB이미지 생성에도 적용할 수 있다.

4. 관련 연구

업무에 사용되는 응용시스템에서 데이터베이스는 핵심적인 역할을 담당하고 있음에도 불구하고, 환경의 복잡성과 데이터의 연관성 때문에 품질을 확신할 수 있을 만큼의 테스트를 수행하는 데에는 많은 어려움이 있다(Chan and Cheung, 1999a; Chan and Cheung, 1999b). 더욱이나 전통적인 소프트웨어 테스트 방법과 지원도구는 데이터베이스 응용 프로그램에는 적합하지 않아서(Florian et al., 2006), 수작업에 의존할 수밖에 없는 데이터베이스 테스트는 전체 개발과 유지보수에 소요되는 비용과 노력의 매우 큰 비중을 차지하고 있다(Davies et al., 2000; Carsten et al., 2008). 이러한 상황을 개선하고자 많은 연구와 투자가 이루어지고 있으며, 이 연구도 그러한 노력의 일환이다.

데이터베이스 테스트를 개선하는 방안으로, Barrasa et al.(2003)는 데이터베이스를 구성하는 실체, 관계, 속성의 매핑을 강조하였고, 여러 개의 SQL로 조합된 트랜잭션을 허용하고 입력과 DB 상태를 고려하는 연구(Yuetang et al., 2005)가 수행되었으며, 결함에 의거한 테스트 방법(Chan et al., 2005)이 제안되기도 하였다.

데이터베이스 레코드 생성에 관해서도 다양한 연구가 있다. Michael et al.(2007)은 프로그램의 실행 경로를 분석하여 입력 데이터와 데이터베이스

레코드를 생성하는 방법을 연구하였다. Chan and Cheung(1999a; 1999b)는 프로그램에서 사용된 SQL 문의 의미를 프로그램 언어로 해석한 후에 이에 대한 테스트케이스를 생성하는 연구를 수행하였으며, 데이터베이스의 스키마와 사용되는 SQL을 함께 분석하여 테스트 입력 데이터를 생성하는 연구도 있었다(Chays et al., 2002; 2004; 2005; Khalek et al., 2008). 지금까지의 데이터베이스 레코드 생성에 관한 연구와 이 논문의 가장 큰 차이점은 응용시스템 로그, 데이터베이스 스키마와 테이블 레코드 등의 다양한 자료 원천(information source)에서 필요한 정보를 지능적, 종합적으로 수집하고 활용하는 점이다.

실행 로그와 관련해서는 로그에서 CFG를 도출하여 프로그램의 실행 경로를 탐색하는 연구(James, 1995)와 동적 영향도 분석을 위한 연구(James and Gregg, 2003)가 있었다. 데이터베이스의 테스트 지원으로는 응용 프레임워크에서 기록한 로그에서 SQL문을 추출하여 테스트에 사용하는 방법에 관한 연구(권오승, 홍사능, 2009)가 있다.

업무환경에서 나타나는 실체 사이의 관계를 테이블의 제약조건인 외래 키로 구현하여 DBMS의 지원을 받을 수 있다(Date, 1981). Coral et al.(2001)은 응용 프로그램에서 사용하는 SQL 참조의 계층관계를 이용하여 잘못된 참조에서 발생하는 비정상적인 상태를 해소하는 방안을 연구 하였으며, Theo and Joachim(1995)은 접근 경로(access path tree)를 이용하여 참조 무결성(reference integrity)을 확보하는 방안에 대하여 연구하였다. 이처럼 외래 키의 사용은 데이터의 정합성을 유지하는데 도움이 되지만, 데이터베이스를 테스트하려면 참조되는 모든 레코드를 생성해야 되는 문제가 발생하여 수작업으로 테스트 레코드를 작성하는 경우에는 엄청난 부담으로 작용한다. 권오승, 홍사능(2008)은 테스트에 필요한 업무의 경감을 위해서 외래 키가 참조하는 레코드를 자동으로 생성하는 방안을 제시하였다.

로그 정보를 이용한 DB이미지 자동 생성은 개발 단계와 유지보수 기간의 회귀 테스트에 가장 큰 도움이 될 것으로 기대한다. 회귀 테스트의 중요성은 이미 잘 알려진 사실이다(Florian et al., 2006). 그러나 회귀테스트에는 많은 시간과 노력이 필요함은 물론이거니와 IT와 업무에 관한 깊은 지식이 요구되는 작업이기 때문에 도구의 지원이 절대적으로 필요한 분야이다(Sriraman et al., 2005). 이에 대한 연구는 영향도 분석과 테스트 데이터의 자동 생성으로 요약된다. 분석을 통한 테스트 대상의 최적화 필요성은 많은 연구에서 지적되고 있으며(Paul, 2002; Aditya, 2008; Alessandro et al., 2003; Bohner, 1996), James and Gregg(2003)은 실행 경로를 종합적으로 판단하여 영향도를 분석하는 방안에 대하여 연구하였다. 최근에는 기존의 정적인 영향도 분석 방식은 비용과 노력에 비해 정확도가 떨어진다는 지적과 함께 동적인 영향도 분석에 대한 연구가 수행되고 있다(James and Gregg, 2003; Taweesup, 2005). 테스트 데이터의 자동 생성으로는 사용자의 테스트 실행을 모사하여 자동으로 실행토록 하는 scripting에 관한 연구가 가장 많았으나(Alessandro and Kennedy, 2005; Gerard, 2003; John et al., 2000), GUI 테스트 자동화 도구에 대한 연구(Kanglin and Mengqi , 2005)와, 테스트케이스 작성 및 결과관리 상용 툴에 대한 제안(Mercury, 2009)도 있었다.

이 연구에서는 SQL에 1) 파라메터(?)로 지정된 속성 값을 추론하는 방법으로는 Barrasa et al.(2003)의 데이터베이스와 Ontology의 대응에 관한 연구결과를, 2) 기타 속성 값을 부여하는 방법으로는 Khalek et al.(2008)의 스키마 제약조건을 활용하

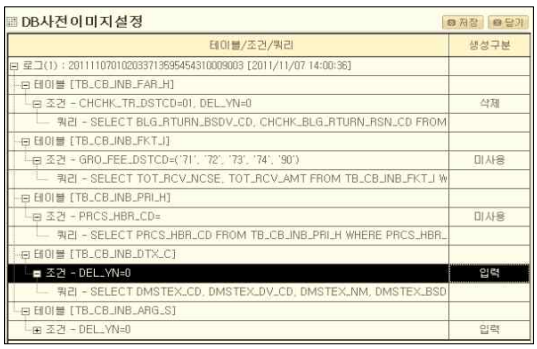
는 방법에 관한 연구 결과를, 3) 디폴트 값의 부여 방법으로는 Barrasa et al.(2003)과 Cohen et al.(1997)이 제시한 동치 클래스와 경계 값 분석에 관한 제안을 원용하였다.

5. DB이미지 생성 구현 및 적용 사례

이 연구의 결과는 상용 테스트 도구인 cb*Tester™의 DB이미지 생성 서브 시스템으로 구현하였다. 시스템은 연구 내용에서 기술한 작업을 수행하는 로그 파서, SQL 파서, 매퍼, 생성기 모듈로 구성되어 있다. SQL 파서로는 WITH, MERGE, UNICODE 구문을 JavaCC Grammar에 반영한 JSQLParser(Ultimoamore, 2011)를 사용하고, 나머지 세 모듈은 자체로 설계 및 구현하였다.

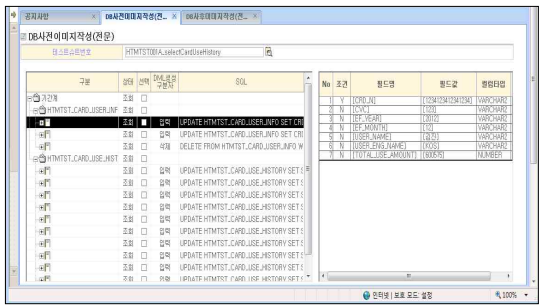
DB이미지 생성과 관련하여 사용자(테스트 담당자)에게 제공하는 주요 기능은 다음과 같다:

1. 화면을 통한 SQL 입력 및 실행
2. 프로그램 SQL 조회 및 편집
3. DB 사전 이미지의 설정과 조회
4. 부여된 속성 값의 확인과 불일치 처리
5. DB 사전/사후 이미지 생성 SQL 조회 및 편집



<그림 19> DB 사전 이미지 설정 화면

<그림 19>은 DB 사전 이미지 설정 화면이며, <그림 20>은 DB 사전 이미지 생성 SQL을 조회하거나 편집 할 수 있는 화면이다. 사후 이미지 작성 화면도 <그림 20>과 유사하다.



<그림 20> DB 사전 이미지 작성 화면

cb*Tester™는 국내에서 개발되었으며, 금융권에서는 잘 알려진 테스트 도구이다. cb*Tester™를 사용하는 한 사이트에 DB이미지 생성 서브 시스템을 시험적으로 적용하였다.

설치된 플랫폼은 HP Compaq dx7300 Microtower, CPU는 Intel® CoreTM 2 CPU 6320 1.86 GHz이고 메모리는 2GB였으며 운영체제로는 Service Pack3를 적용한 Windows XP를 사용하였다.

아직은 도구 적용의 초기 단계라 시스템적인 튜닝과 사용자 교육을 필요로 하지만, 시험적으로 1,309개의 프로그램 테스트에 적용한 결과를 요약하면 다음과 같다:

1. 하나의 프로그램에 포함된 SQL은 최대 111개, 평균(average) 7.8개, 표준편차 19개이었으며 중위수(median)와 최빈수(mode)는 모두 3개이었다
2. 42개의 프로그램은 100개 이상의 SQL을 포함하고 있었다.
3. 실행 시간은 13분에서 0.2초까지 분포하였다. 10분 이상이 2건, 4분 이상 3건, 3분대, 2분대, 1분대는 9건, 7건, 11건이었다. 1163건은 10초 이내

에 이미지를 생성하였으며, 그 중에서 827건은 2초 이내의 시간이 경과하였다.

4. 프로그램에 포함된 SQL의 수와 경과시간의 관계는 미미하였다. 예를 들면, 이미지 생성 시간이 10초 이상 걸린 146개의 프로그램 중에서 SQL의 수가 50개 이상인 프로그램은 2개에 불과하였다.
5. 총 1309건의 테스트에서 오류가 34건 발생했으며, 오류 건은 대부분이 10초 이상의 시간을 경과하였다.

이상의 내용으로부터 통계적으로 검증된 결론을 도출하는 것은 불가능하지만 직관적인 분석을 통하여 도구의 실효성을 가늠할 수는 있다. 일반적으로 수작업으로 1개의 테이블의 사전 이미지를 생성할 때 소요되는 시간은 1) 테이블 조회 SQL 문 입력 시간(20초), 2) 조회 시간(3초), 3) 레코드 선택시간(5초) 그리고 4) 저장 시간(3초)으로 가정하면 총 31초가 소요되는 것으로 산정할 수 있다. 이 시간은 업무 사이에 공백과 작업 결과의 확인이 없는 것을 가정하였기 때문에, 실제로는 최소 1~3분 정도의 소요 시간이 예상된다. 이러한 추정치를 자동으로 생성하는데 소요되는 시간과 비교하면 수 배에서 수 십 배의 차이를 예상할 수 있어 연구에서 제시한 방법을 적용하면 DB이미지 생성에 요구되는 시간을 획기적으로 단축할 수 있을 것으로 기대된다.

6. 결론

이 논문은 로그와 데이터베이스 스키마 등의 여러 곳에 분산되어 존재하는 지식과 정보를 효과적으로 수집하고, 지능적으로 활용하여 테스트에 필요한 DB이미지를 자동으로 생성해 주는 방안에 대한 연구이다. 연구의 결과는 1) 데이터베이스 테스트에 필요한 수작업의 부담을 덜어줌과 동시에 수작업에 따르는 오류의 가능성을 원천적으로 제거하고, 2) 테스트의 예상 결과 데이터도 자동으로 생성할 수 있으며, 3) 동일한 테스트케이스를 반복적으로 수행하여 반복 개발과 유지보수의 회귀 테스트를 용이하게 한다. 또한 4) 프로그램에서 사용하는 SQL을 분석하여 업무 패턴과 부합하지 않는 잘못된 데이터베이스 조작 패턴을 발견하고 이에 대한 정보를 제공할 수 있다.

다음과 같은 추가의 연구로 논문에서 제시한 방안의 완성도를 높일 수 있다. 1) 로그를 이용하여 DB 사전 이미지를 생성하는 시점이 이미 거래를 수행한 후이므로 거래 전 데이터로 사전 이미지를 만들 수 있는 연구가 필요하다. 2) 로그를 남기는 방식의 개선이다. UUID를 이용하여 추적 가능하도록 남기는 방식 외에, 원시 코드에 부가적인 코드를 삽입하여 로그를 남기는 것이다. 3) PL/SQL인 stored procedure와 trigger을 처리에 대한 연구가 필요하다. 4) 기업에서 중요한 거래를 처리하는 프로그램은 수십에서 백 개가 훨씬 넘는 수의 테이블을 이용하는 경우가 많아 이미지 생성 작업의 성능에 대한 고려가 필요하다.

참고문헌

권오승, 홍사능, “Automatic Maintenance of Referential Integrity for Unit Test Data Management”, *경영정보학회 추계학술대회*(2008), 609~616.

권오승, 홍사능, “로그 기반 효과적 반복 테스트”, *경영정보학회 추계학술대회*(2009), 685~690.

Aditya, P. M., *Foundations of software Testing*, Dorling Kindersley(India) Pvt. Ltd, 2008.

Aho, A. V., R. Sethi, and J. D. Ullman, *Compilers : Principles, Techniques, and Tools*, Addison Wesley, 1987.

Alessandro, O. and K. Bryan, "Selective Capture and Replay of Program Executions", *Proceedings of the third international workshop on dynamic analysis*, St. Louis, Missouri (2005), 1~7.

Alessandro, O., A. Taweesup, and J. H. Mary, "Leveraging Field Data for Impact Analysys and Regression Tesing", *ACM SIGSOFT Software Engineering Notes*, Vol.28, No.5 (2003), 128~137.

Barrasa, J., O. Corcho, and A. Gomez-Perez, "Fund Finder : A case study of database-to-ontology mapping", *Workshop on Semantic Integration*(2003), 9~15.

Beck, K. and E. Gamma, "Junit Cookbook", http://junit.sourceforge.net/doc/cookbook/cookbook.htm, 2011.

Bohner, S. and R. Arnold, "Software Change Impact Analysis", *IEEE Computer Society Press*, Los Alamitos, CA, USA, 1996.

Brooks, F. P., "No Silver Bullet Essence and Accidents of Software Engineering", *IEEE Computer Society*, Vol.20, No.4(1987), 10~19.

Carsten, B., D. Kossmann, and L. Eric, "Towards Automatic Test Database Generation", *Data Engineering Bulletin*, 2008.

Chan, M. Y. and S. C. Cheung, "Applying white box testing to database applications", *Technical Report*, HKUST-CS99-01, Dept. of Computer Science, Hong Kong Univ., 1999a.

Chan, M. Y. and S. C. Cheung. "Testing Database Applications with SQL Semantics", *Proceedings of 2nd International Symposium on Coorperative Database Systems for Advanced Applications*(1999b), 363~374.

Chan, W. K., S. C. Cheung, and T. H. Tse, "Fault-Based Testing of Database Application Programs with Conceptual Data Model", *Fifth International Conference on Quality Software*, (2005), 187~196.

Charles, C. P., *Set Theory*, KyungMoon Publishers, 2002.

Chays, D., Y. Deng, P. G. Frankl, S. Dan, F. I. Vokolos, and E. J. Weyuker, "AGENDA : A test generator for relational database applications", *Technical Report*, TR-CIS-2002-04, Dept. of Computer Science, Polytechnic Univ, 2002.

Chays, D., Y. Deng, P. G. Frankl, S. Dan, F. I. Vokolos, and E. J. Weyuker, "An AGENDA for testing relational database applications", *Journal of Software Testing, Verification, and Reliability*, Vol.14(2004), 17~44.

Chays, D., "Test Data Generation for Relational Database Applications", *Ph.D. diss., Dept. of Computer and Information Science, Polytechnic Univ.*, 2005.

Cohen, D. M., S. R. Dalal, and M. L. Fredman, and G. C. Patton, "The AETG System : An Approach to Testing Based on Combinatorial Design", *IEEE Transactions on Software Engineering*, Vol.23(1997), 437~444.

Coral, C., M. Piattini, and M. Genero, "Empirical validation of referential integiry metrics", *Information and Software Technology*, Vol. 43(2001), 949~957.

Date, C. J., "REFERENTIAL INTEGIRITY", *The VLDB Journal*, Vol.7(1981), 2~12.

Davies, R. A., R. J. Beynon, and B. F. Jones, "Automating the testing of databases", *Proceedings of First International Workshop on Automated Program Analysis, Testing and Verification,* 2000.

Florian, H., D. Kossmann, and L. Eric, "A frame-

work for efficient regression tests on database applications", *The VLDB Journal*, Vol.16(2006), 145~164.

Gerard, M., "Agile Regression Testing Using Record and Playback", *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, Anaheim, ca, USA*, (2003), 353~360.

Gupta, N., A. M. Mathur, and M. L. Soffa, "Automated Test Data Generation Using An Iterative Relaxation Method", *ACM SIGSOFT Software Engineering Notes*, Vol.23, No.6 (1998).

James, R. L., "Whole Program Paths", *ACM SIGPLAN Notices*, Vol.34, No.5(1995), 259~269.

James, L. and R. Gragg, "Whole Program Path-Based dynamic impact analysis", *Proceedings of the 25th International Conference on Software Engineering, Portland, Oregon, IEEE*, (2003), 308~318.

John, S., C. Pravir, and F. Bob, A. Podgurski, "jRapture : A Capture/Replay Tool for Observation-Based Testing", *ACM SIGSOFT Software Engineering Notes*, Vol.25, No.5 (2000), 158~167.

Kanglin, L. and W. Mengqi, *EFFECTIVE GUI TEST AUTOMATION : Development an Automated GUI Testing Tool*, SYBEX, 2005.

Khalek, S. A., B. Elkarablieh, Y. O. Laleye, and S. Khurshid, "Query-aware Test Generation Using a Relational Constraint Solver", *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*(2008), 238~247.

McCabe, T. J. and C. W. Butler, "Design complexity measurement and testing", *Communications of the ACM*, Vol.32, No.12(1989).

Mercury Interactive Software, "TestDirector test management and QuickTest test automation software", http://www-svca.mercuryinteractive.com/products, 2009.

Michael, E., M. Rupak, and S. Koushik, "Dynamic Test Input Generation for Database Applications", *Proceedings of the 2007 international symposium on software testing and analysis*, (2007), 151~162.

Paul, C. J., *SOFTWARE TESTING : A Craftsman's Approach Second Edition*, Grand Valley State University : CRC PRESS, 2002.

Royce W., *Software Project Management : A Unified Framework*, Addison-Wesley Professional, 1998.

Sen, K., D. Marinov, and G. Agha, "CUTE:a concolic unit testing engine for C", *ACM SIGSOFT Software Engineering Notes,* Vol. 30, No.5(2005).

Sriraman, T., G. Rajiv, and Z. Xiangyu, "Extended Whole Program Paths", *Parallel Architectures and Compilation Techniques, 14th Internaltional Conferenece*, IEEE(2005), 17~26.

Taweesup, A., O. Alessandro, and J. H. Mary, "Efficient and Precise Dynamic Impact Analysis Using Execute-After Sequences", *Software Engineering, Proceedings, 27th International Confernece*, IEEE(2005), 432~441.

Theo, H. and R. Joachim, "Access path support for referential integrity in SQL2", *The VL DB Journal*, Vol.5(1995), 196~214.

Ultimoamore, "Download", http://sourceforge.net/projects/jsqlparserfiles, 2011.

Yuetang, D., F. Phyllis, and D. Chays, "Testing Database Transaction with AGENDA", *Proceedings of the 27th international conference on Software engineering*(2005), 238~252.

Abstract

Automatic Generation of DB Images for Testing Enterprise Systems

Ohseung Kwon* · Saneung Hong*

In general, testing DB applications is much more difficult than testing other types of software. The fact that the DB states as much as the input data influence and determine the procedures and results of program testing is one of the decisive reasons for the difficulties. In order to create and maintain proper DB states for testing, it not only takes a lot of time and efforts, but also requires extensive IT expertise and business knowledge. Despite the difficulties, there are not enough research and tools for the needed help.

This article reports the result of research on automatic creation and maintenance of DB states for testing DB applications. As its core, this investigation develops an automation tool which collects relevant information from a variety of sources such as log, schema, tables and messages, combines collected information intelligently, and creates pre- and post-Images of database tables proper for application tests. The proposed procedures and tool are expected to be greatly helpful for overcoming inefficiencies and difficulties in not just unit and integration tests but including regression tests.

Practically, the tool and procedures proposed in this research allows developers to improve their productivity by reducing time and effort required for creating and maintaining appropriate DB sates, and enhances the quality of DB applications since they are conducive to a wider variety of test cases and support regression tests. Academically, this research deepens our understanding and introduces new approach to testing enterprise systems by analyzing patterns of SQL usages and defining a grammar to express and process the patterns.

Key Words : Test Automation, Test Case, Database Application System, Unit Testing, Regression Testing, Database Pre/Post Image, Query Dependency Sequence

* School of Business Administration, University of Seoul

저 자 소 개

권오승

1987년 동국대학교와 2007년 동대학교 산업대학원에서 각각 공학학사와 공학석사 학위를 취득하였다. (주) 한독과 삼성SDS에서 근무하였으며 현재는 컴포넌트베이시스(주)에 재직 중이면서 서울시립대학교 경영학과에서 MIS 전공으로 박사과정을 이수 중이다. 수 십 개의 SI 및 컴포넌트 기반 개발(CBD) 프로젝트를 수행하였고, 상용 black-box 테스트 도구를 개발하였다. 관심분야는 응용 프레임워크, 테스트케이스 자동생성, 테스트 프로세스 자동화(TMS)이다.

홍사능

서울시립대학교 농업경영학과를 졸업하고 University of Texas at Austin에서 경영학 박사 학위를 받았다. 대학을 졸업한 후에 한국산업은행에서 근무하였으며 유학 후에는 한국통신기술 신재무팀장, 금융감독원 정보관리국장을 역임한 경력이 있으며, 현재는 서울시립대학교 경영학부 교수로 재직하고 있다. Decision Support Systems 등에 논문을 게재하였으며 주요 관심분야는 정보시스템 개발, 정보기술 관리, 프로젝트 관리이다.