「정보시스템연구」 제19권 제4호
한국정보시스템학회
2010년 12월, pp. 25~49

`

임베디드 SQL 기반 정보시스템의 개발 및 관리 방법에 대한 연구

송 용 욱*

〈목 차〉

Ⅰ. 서 론

객체형 데이터베이스(OODB), 객체-관계형 데이터베이스(ORDB) 등의 새로운 개념의 등장에도 불구하고 관계형 데이터베이스(RDB)는 아직도 기업정보시스템의 근간이 되고 있다. 1990년대 중반 이후 전사 차원에서의 최적화된 업무 프로세스 구축 및 이를 통한 종합적 정보화 달성을 목표로 도입되기 시작하여 이제는 거의 대부분의 기업들이 도입한 전사적 자원관리 (ERP: Enterprise Resource Planning) 시스템들의 경우 관계형 데이터베이스를 기반으로 하고 있으며, 이러한 ERP 시스템의 구축을 통해 기업 데이터베이스는 집중화되고 거대화 되었다. 여기서 한 걸음 더 나아가 기업들은 이제 이렇게 ERP 구축을 통해 구현된 관계형 데이터베이스 위에 또 다른 응용 프로그램들도 구축하고 있다.

현재 기업의 정보기술 활용 단계는 전사 차원의 종합 정보화단계를 넘어 지식화 및 지능화의 단계로 들어서고 있다. 지식화 및 지능화를 가능하게 해주는 중요한 정보기술 중의 하나가 데이터마이닝(data mining)이며 기업의 데이터마이닝 응용 시스템들은 그동안 축적된 고객, 매출 등에 관한 자료 또는 정보를 이용하여 데이터마이닝을 수행하고 있다. 데이터마이닝의 대표적인 응용 분야로는 카드 도용 방지 (fraud

* 연세대학교 원주캠퍼스 경영학부 교수, yusong@yonsei.ac.kr

detection), 위험 관리 (risk management), 고객 불만 예방 (claim prevention), 고객 유치 및 유지 (customer acquisition and retention), 고객 세분화 및 프로파일링 (customer segmentation and profiling), 마케팅 효과분석 (campaign effect analysis), 목표 마케팅 (target marketing), 교차 판매 (cross-selling/up selling) 등을 들 수 있는데(이재규 등, 2002; 이재규 등, 2006), 이들 응용 시스템들이 기업의 고객 및 매출 정보를 이용한다. 그런데, 앞에서 이야기한 바와 같이 이러한 고객, 매출 등에 관한 정보가 관계형 데이터베이스에 저장되어 있으므로, 요즘 활발하게 개발되고 있는 기업의 데이터마이닝 응용 시스템들은 관계형 데이터베이스 기반 위에 구축되고 있다.

관계형 데이터베이스 기반 응용 프로그램 구축 도구 또는 기술로 많이 쓰이고 있는 것으로는 ODBC, JDBC 등과 같은 데이터베이스 접근 표준 인터페이스와, Pro*C와 같은 임베디드 SQL 프로그래밍 기술 등이 있다. 그런데, ODBC는 마이크로소프트 사의 플랫폼에 종속적이며 상대적으로 속도가 늦고, JDBC는 아직은 생소한[1] 자바(Java) 프로그래밍 언어 기반에서만 사용할 수 있는 데 비하여, C/C++은 다양한 플랫폼에서 상대적으로 빠른 속도의 응용 시스템을 구축할 수 있기 때문에 지금까지의 거의 대부분의 기업 정보시스템들은 Pro*C 임베디드 SQL 프로그래밍 환경 하에서 개발되고 있다. 그런데, 기업정보시스템의 경우 많은 수의 테이블, 그리고 더 많은 수의 필드를 갖는 데이터베이스를 바탕으로 하고 있기 때문에, 특히 임베디드 SQL 프로그램 개발 시 개발자가 이러한 모든 필드들의 이름, 자료형 및 크기 등을 일일이 고려하여 프로그램에 반영하는 것이 매우 피곤하고 지루한 작업이 된다. 나아가, 이렇게 개발한 프로그램을 지식의 변화 또는 데이터베이스 설계 변경 등의 이유로 수정하고자 할 때, 다시금 수많은 필드들의 이름, 자료형 및 크기 등을 일일이 고려하여 프로그램을 수정해야 하며, 이것은 어렵고도 오류가 발생하기 쉬운 작업이 된다.

따라서, 본 연구에서는 관계형 데이터베이스 기반 임베디드 SQL 프로그램의 개발 및 유지보수 관리를 용이하게 하기 위한 방법론을 개발하고 구현하고자 한다. 이를 위해 본 논문은 다음과 같이 구성되어 있다. 먼저, 2절에서는 연구의 이론적 배경으로서 ODBC, JDBC, Pro*C 등의 데이터베이스 접근 기술들을 살펴본 후, 3절에서는 Pro*C에 초점을 맞추어 Pro*C 기반 응용 프로그램 관리상의 문제점을 분석 정리하여 우리의 문제를 해결할 수 있는 실마리를 찾아본다. 이어서, 4절에서는 Pro*C 프로그램 개발 및 유지보수 관리 방법론을 도출하고, 이 방법론을 구현한 시스템 도구에 대해 알아본 후, 5절에서 이에 대해 토의하도록 한다. 그리고, 마지막으로 6절에서 결론을 맺도록 한다.

Ⅱ. 이론적 배경

2.1 ODBC

개방형 데이터베이스 연결(ODBC, Open Database Connectivity)은 데이터베이스 관리시

1) 자바 프로그래밍 언어는 현재 웹 환경에서는 JSP 등의 기반 언어로서 대중적으로 사용되고 있다. 그러나, 기업의 기간계 시스템들은 대부분 C/C++ 언어를 이용하여 개발되어 있다.

스템(DBMS)에 접근하기 위한 표준 인터페이스이다. ODBC는 프로그래밍 언어, 데이터베이스 시스템, 운영체제 등과 독립적이며, 따라서 ODBC를 이용하는 응용 프로그램은 플랫폼에 상관없이 데이터베이스의 데이터에 접근할 수 있다. ODBC는 이기종 컴퓨터의 관계형 및 비관계형 DBMS에 접근하기 위한 표준 응용 프로그램 인터페이스(API, Application Programming Interface)로서 SQL Access Group에 의해 1992년에 개발되었다. ODBC 등장 이전에 한 종류 이상의 데이터베이스에 접근할 수 있는 응용 프로그램을 개발하기 위해서는 각각의 데이터베이스와의 인터페이스를 별도로 구현하여야 하였다. ODBC를 이용할 경우 응용 프로그램 개발자는 ODBC라는 한 개의 인터페이스만 알면 되며, 개별 DBMS에 대한 접근방법이 변하더라도 ODBC 드라이버만 업데이트하면 되고 응용 프로그램 자체는 바꿀 필요가 없다. ODBC 드라이버는 프린터나 각종 컴퓨터 장치의 드라이버와 비슷하게 응용 프로그램이 사용할 표준화된 요청(call) 명령어 집합을 제공하며, 이러한 명령어 집합들은 응용 프로그램의 데이터베이스 접근 명령을 각 DBMS의 명령어로 전환하는 역할을 한다.

ODBC는 SQL Access Group, X/Open, ISO/IEC 등의 CLI (Call Level Interface) 표준을 바탕으로 하고 있다. 1992년에 마이크로소프트는 SQL Access Group의 CLI를 이용하여 ODBC 1.0을 개발, 발표하였다. ODBC 2.0과 3.0을 거치면서 X/Open, ISO/IEC의 표준을 포함시켰으며, 2009년 Windows 7용 ODBC 3.8을 발표하였다. (ISO/IEC, 2008; Easysoft; Microsoft) 마이크로소프트 비쥬얼 C++ 환경에서 ODBC 프로그래밍 단계는 다음과 같다.

(1) 환경변수 생성 (명령어 SQLAllocEnv)
(2) 데이터베이스 연결변수 생성
(명령어 SQLAllocConnect)
(3) 데이터베이스 연결 (명령어 SQLConnect)
(4) SQL 문장 변수 생성
(명령어 SQLAllocStmt)
(5) SQL 문장 실행 (명령어 SQLExecDirect)
(6) SQL 문장 변수 삭제
(명령어 SQLFreeStmt)
(7) 데이터베이스 연결 끊기
(명령어 SQLDisconnect)
(8) 데이터베이스 연결변수 삭제
(명령어 SQLFreeConnect)
(9) 환경변수 삭제 (명령어 SQLFreeEnv)

위 프로그래밍 단계 중 핵심은 (5) SQL 문장 실행 부분으로, 명령어 SQLExecDirect의 인수로서 문자열 형태의 SQL 명령문을 제공하게 된다. 예를 들어 <표 1>의 프로그램 예제를 보자. 명령문 “SQLExecDirect(gSrv.hstmt, SqlStatement, SQL_NTS);”에서 두 번째 인수인 SqlStatement는 문자열 형태의 SQL 명령문으로서 그 내용은 “sprintf(SqlStatement, "update authors set address = '%s_%d' where au_id = '%s'", "Address", i, "UserId");” 명령에 의해 만들어진다. 즉, SQL 명령문을 문자열로 표현한 후 명령어 SQLExec Direct의 두 번째 인수로 넘겨주면 된다. 예제 프로그램은 다음과 같은 5개의 SQL 명령문을 연속적으로 수행하고 있다.

update authors set address = 'Address_0' where

```
au_id = 'UserId'
update authors set address = 'Address_1' where
au_id = 'UserId'
update authors set address = 'Address_2' where
au_id = 'UserId'
update authors set address = 'Address_3' where
au_id = 'UserId'
update authors set address = 'Address_4' where
au_id = 'UserId'
```

위와 같이 데이터베이스 접근 명령어로 표준 데이터베이스 언어인 SQL을 사용함으로써 ODBC는 프로그래밍 언어, DBMS, 운영체제 등과 독립성을 유지할 수 있다.

2.2 JDBC

자바 데이터베이스 연결(JDBC, Java Database

<표 1> ODBC 프로그램 예제

```
void main(int argc, char **argv) {
    HENV gHenv = SQL_NULL_HENV;
    DBCONN gSrv;
    ITransactionDispenser *pTransactionDispenser;
    ITransaction *pTransaction;
    char SqlStatement[2014];
    DtcGetTransactionManager(NULL, NULL, IID_ITransactionDispenser, 0, 0, 0,
                                (void **) & pTransactionDispenser);
    SQLAllocEnv(&gHenv);
    SQLAllocConnect(gHenv, &(gSrv.hdbc));
    SQLConnect(gSrv.hdbc, "DSN", SQL_NTS, "username", SQL_NTS,
                "password", SQL_NTS);
    for (int i = 0; i < 5; i++) {
        pTransactionDispenser->BeginTransaction(NULL,
            ISOLATIONLEVEL_ISOLATED, ISOFLAG_RETAIN_DONTCARE,
            NULL, &pTransaction);
        SQLSetConnectOption(gSrv.hdbc, SQL_COPT_SS_ENLIST_IN_DTC,
                                (UDWORD) pTransaction);
        sprintf(SqlStatement, "update authors set address = '%s_%d' where "
                "au_id = '%s'", "Address", i, "UserId");
        SQLAllocStmt(gSrv.hdbc, &(gSrv.hstmt));
        SQLExecDirect(gSrv.hstmt, SqlStatement, SQL_NTS);
        SQLFreeStmt(gSrv.hstmt, SQL_DROP);
        gSrv.hstmt = SQL_NULL_HSTMT;
        pTransaction->Commit(0, 0, 0);
        pTransaction->Release();
    }
    pTransactionDispenser->Release();
    SQLDisconnect(gSrv.hdbc);
    SQLFreeConnect(gSrv.hdbc);
    gSrv.hdbc = SQL_NULL_HDBC;
    SQLFreeEnv(gHenv);
}
```

Connectivity)은 자바 프로그래밍 언어 환경에서 데이터베이스 관리시스템(DBMS)에 접근하기 위한 표준 인터페이스이다. ODBC와 마찬가지로 JDBC는 데이터베이스 시스템, 운영체제 등과 독립적이며, 따라서 JDBC를 이용하는 자바 응용 프로그램은 플랫폼에 상관없이 데이터베이스의 데이터에 접근할 수 있다. 단, JDBC는 자바 프로그래밍 언어만을 지원한다. JDBC를 이용할 경우 자바 응용 프로그램 개발자는 JDBC라는 한 개의 인터페이스만 알면 되며, 개별 DBMS에 대한 접근방법이 변하더라도 JDBC 드라이버만 업데이트하면 되고 응용 프로그램 자체는 바꿀 필요가 없다. JDBC 드라이버는 프린터나 각종 컴퓨터 장치의 드라이버와 비슷하게 응용 프로그램이 사용할 표준화된 요청(call) 명령어 집합을 제공하며, 이러한 명령어 집합들은 응용 프로그램의 데이터베이스 접근 명령을 각 DBMS의 명령어로 전환하는 역할을 한다.

JDBC는 1997년 2월 19일 선 마이크로시스템스(Sun Microsystems, 현 오라클)에 의해 자바 2 플랫폼 표준판 버전 1.1(Java 2 Platform, Standard Edition, Version 1.1), 즉 J2SE에 포함되어 발표되었다. 이때 JDBC-to-ODBC 브릿지와 함께 발표되어 ODBC에 의해 접근 가능한 모든 데이터 원천(data source), 예를 들어 MS SQL Server DB, MS Access DB, MS Excel 스프레드시트 파일 등에도 연결될 수 있었다. JDBC 3.0부터는 Java Community Process(JSP)에 의거하여 개발 중이며, JDBC 3.0은 JSP 표준인 JSR54, JDBC 4.0은 동 표준 JSR 221로 발표되었다. JDBC 4.0은 현재 Java SE 6에 포함되어 있다. (Sun Microsystems, Inc., 2006; Oracle)

JDBC 프로그래밍 단계는 다음과 같다.

(1) JDBC 드라이버 클래스 적재
(명령어 Class.forName)
(2) 데이터베이스 연결
(명령어 DriverManager.getConnection)
(3) SQL 문장 변수 생성
(명령어 Connection.createStatement)
(4) SQL 문장 실행 (명령어 Statement.execute)
(5) SQL 문장 변수 삭제 (명령어 Statement.close)
(6) 데이터베이스 연결 끊기
(명령어 Connectin.close)

위 프로그래밍 단계 중 핵심은 (4) SQL 문장 실행 부분으로, 명령어 Statement.execute의 인수로서 문자열 형태의 SQL 명령문을 제공하게 된다. 예를 들어 <표 2>의 프로그램 예제를 보자. 명령문 “stmt.execute(SqlStatement);”에서 인수 SqlStatement는 문자열 형태의 SQL 명령문으로서 그 내용은 “SqlStatement = String.format("update authors set address = + '%s_%d' where au_id = '%s_%d'", "Address", i, "UserId", i);” 명령에 의해 만들어진다. 즉, SQL 명령문을 문자열로 표현한 후 명령어 Statement.execute의 인수로 넘겨주면 된다. 예제 프로그램은 다음과 같은 5개의 SQL 명령문을 연속적으로 수행하고 있다.

```
update authors set address = 'Address_0' where
au_id = 'UserId_0'
update authors set address = 'Address_1' where
au_id = 'UserId_1'
update authors set address = 'Address_2' where
au_id = 'UserId_2'
```

update authors set address = 'Address_3' where au_id = 'UserId_3'
update authors set address = 'Address_4' where au_id = 'UserId_4'

위와 같이 데이터베이스 접근 명령어로 표준 데이터베이스 언어인 SQL을 사용함으로써 JDBC도 ODBC처럼 DBMS, 운영체제 등과 독립성을 유지할 수 있다.

2.3 Pro*C/C++

Pro*C(Pro*C/C++)는 오라클(Oracle) DBMS 및 사이베이스(Sybase) SQL 서버 DBMS에 의해 사용되는 임베디드(embedded) SQL 프로그래밍 언어이다. 임베디드 SQL 프로그램은 보통의 프로그램 소스 코드 내에 임베디드 SQL 문장을 포함한 프로그램을 말하며, 임베디드 SQL 문장은 프로그램 소스 코드 내에 포함된 데이터베이스 접근 SQL 문장을 말한다. 임베디드 SQL 문장은 SQL 전처리기에 의해 해석된 후 해당 라이브러리를 부르는 소스 프로그램 문장으로 대체된다. 이 과정을 통해 만들어진 새로운 소스 코드는 컴파일러에 의해 컴파일된다. Pro*C는 C 언어 또는 C++ 언어(C/C++ 언어)를 주 프로그래밍 언어(host language)로 사용한다. C/C++ 소스 프로그램을 컴파일할 때 소스 프로그램에 임베드(embed)된 SQL 문장들이 Pro*C 전처리기(precompiler)에 의해 해석된 후 각각 해당 SQL 라이브러리 함수를 부르는 문장으로 대체된다. Pro*C 전처리기는 표준 C/C++ 프로그램 코드를 생성하며, 이 프로그램은 보통의 C/C++ 컴파일러에 의해 수행 가능한 프로그램(executable)으로 컴파일된다. (Oracle, 2005) 다음 절에서는 Pro*C 프로그램의 기본 구문들을 설명하면서, Pro*C 임베디드 SQL 프로그램의 개발 및 관리상의 문제점을

<표 2> JDBC 프로그램 예제

```
public static void main(String[] args) {
    try {
        int i;
        String SqlStatement;
        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
        Connection conn = DriverManager.getConnection("jdbc:odbc:ydsn", "",
                          "");
        Statement stmt = conn.createStatement();
        for (i = 0; i < 5; i++) {
            SqlStatement = String.format("update authors set address = "
              + "'%s_%d' where au_id = '%s_%d'", "Address", i, "UserId", i);
            stmt.execute(SqlStatement);
        }
        stmt.close();
        conn.close();
    } catch (Exception e) {
    }
}
```

<표 3> Pro*C SELECT 프로그램 예제

```
{
    int a;
    EXEC SQL SELECT salary INTO :a
             FROM Employee
             WHERE SSN=876543210;
    printf("The salary is %d\n", a);
}
```

도출해 보도록 하겠다.

(1) SQL 문장

Pro*C에서 모든 임베디드 SQL 문장은 키워드 “EXEC SQL”로 시작하여 세미콜론(‘;’)으로 끝난다. SQL 문장은 C/C++ 수행문이 올 수 있는 C/C++ 프로그램 내 어디에든 위치할 수 있다. 예를 들어, <표 3>은 SQL SELECT 명령을 임베드한 프로그램의 일부이다.

(2) 호스트 변수

호스트 변수(host variable)는 응용 프로그램(host program)과 데이터베이스 간의 연결을 위한 핵심 도구이다. 호스트 변수는 C/C++ 변수가 선언될 수 있는 어떤 곳에든 선언될 수 있다. 호스트 변수를 선언할 때는 다음과 같이 declare section 선언을 해 주어야 한다.

```
EXEC SQL BEGIN DECLARE SECTION;
int x;
char *y;
int z;
EXEC SQL END DECLARE SECTION;
```

임베디드 SQL 문장에서 호스트 변수는 콜론(“:”)을 앞에 붙여서 표시하여야 한다. 물론, 임베디드 SQL 문장 내가 아닌 보통의 C/C++ 문장에서는 콜론을 붙이지 말아야 한다. 예를 들어, 다음은 위에서 선언한 호스트 변수를 이용하여 SQL INSERT 명령을 임베드한 프로그램의 일부이다.

<표 4> Pro*C INSERT 프로그램 예제

```
void f_insert() {
    EXEC SQL BEGIN DECLARE SECTION;
        int x;
        char *y;
        int z;
    EXEC SQL END DECLARE SECTION;
    x = 20100301;
    y = "Smith";
    z = 3;
    EXEC SQL INSERT INTO emp(empno, ename, deptno)
        VALUES(:x, :y, :z);
}
```

```
x = 20100301;
y = "Smith";
z = 3;
EXEC SQL INSERT INTO emp(empno, ename, deptno)
        VALUES(:x, :y, :z);
```

위 예제를 완성된 임베디드 SQL 프로그램으로 표현한 것이 <표 4>에 나타나 있다. 이 예제에서는 레코드 (20100301, "Smith", 3)을 테이블 emp의 새 레코드로 추가하고 있다. 단, 이 예제 프로그램에 데이터베이스와 연결하거나 연결을 끊는 문장은 포함되지 않았음에 유의하기 바란다.

Ⅲ. Pro*C 프로그램 관리상의 문제점

본 연구의 목적은 관계형 데이터베이스 기반 임베디드 SQL 프로그램의 개발 및 유지보수 관리를 용이하게 하기 위한 방법론을 개발하고 구현하는 것이다. 객체형 데이터베이스(OODB), 객체-관계형 데이터베이스(ORDB) 등의 새로운 개념의 등장에도 불구하고 관계형 데이터베이스(RDB)는 아직도 기업정보시스템의 근간이 되고 있다. 이에 따라, 대부분의 기업정보시스템들도 관계형 데이터베이스 기반의 임베디드 SQL 프로그램으로 개발되었고, 또 아직도 개발되고 있다. 또한, ODBC의 경우 마이크로소프트 사 플랫폼에 종속되어 있다는 점과 함께 속도가 상대적으로 느린 점, 그리고, JDBC의 경우 아직도 많은 기업의 정보시스템이 자바 언어보다는 C/C++ 언어 기반으로 되어 있다는 점 등의 이유 때문에 ODBC, JDBC 등에 비해 Pro*C가 기업정보시스템 개발에 더 많이 사용되고 있다. 따라서, 본 연구는 임베디드 SQL 프로그램의 개발 및 유지보수의 시간과 오류 가능성을 줄임으로서 개발 및 유지보수 비용을 줄이고 담당자의 업무 부담을 경감하는 것을 목표로 하며, 특히 임베디드 SQL 프로그래밍 언어 중 특히 Pro*C에 초점을 맞추도록 하겠다.

본 절에서는 먼저 Pro*C로 기업정보시스템을 개발 및 유지보수 하고자 할 때 그 소요 시간과 오류 가능성 측면에서 문제가 되는 부분을 알아보고자 한다. 이것은 크게 기업정보시스템 데이터베이스의 복잡성 문제, 그리고 이와 연관되어 Pro*C 자체에서 발생하는 문제로 나누어 볼 수 있다. 먼저 기업정보시스템에서 흔히 문제가 되는 것은 수많은 테이블과 필드 관리상의 복잡성이다. 현실 속 기업 데이터베이스의 경우 수십 개의 테이블과 각 테이블 당 수십~수백 개의 필드들로 이루어져 있다. 이것은 기업정보시스템의 특성상 불가피한 상황이지만, 이것이 Pro*C의 내재적인 구문 특성과 연결될 때 프로그램 개발 및 유지보수 관리상에 문제를 불러일으킨다. 따라서, 본 절에서는 Pro*C의 구문(Oracle, 2005)을 좀 더 자세히 살펴보면서, 이러한 프로그램 개발 및 유지보수 관리의 어려움이 발생하는 원인을 정리해 보도록 하겠다.

3.1 Pro*C 구문 분석

(가) 호스트 변수

기업정보시스템이 다루어야 하는 수많은 필드를 Pro*C에서 호스트 변수를 이용하여 처리하고자 할 때 작성하여야 하는 프로그램 라인 수는

필드의 개수보다 3~4배 정도가 된다. 예를 들어 <표 4>의 예제 프로그램을 분석, 정리한 <표 5>를 살펴보자. <표 5>는 SQL INSERT 명령을 수행하는 과정을 4 부분으로 나누어 표시하고 있다. 즉, (1) 호스트 변수 선언, (2) 호스트 변수 값 설정, (3) 필드 이름 선언, (4) 필드 값 설정 등 4 부분이 그것이다. 이 4 부분들은 각각 필드 개수만큼의 프로그램 라인으로 이루어져 있다. 따라서, 예를 들어 100개의 필드로 이루어진 데이터베이스 테이블에 대해 INSERT 명령을 프로그래밍 할 경우 적어도 400 라인 이상의 프로그램을 작성하여야 한다는 말이 된다. 보통은 호스트 변수들을 C/C++ 언어의 구조체(structure)를 이용하여 하나로 묶어서 처리하기 때문에 (4) 필드 값 설정 부분은 1 라인으로 처리된다. 그렇지만 이 경우에도 필드 수보다 적어도 3배는 많은 프로그램 라인을 작성하여야 한다는 것을 알 수 있다. 이러한 현상은 INSERT 외에 UPDATE, SELECT 등 다른 SQL의 명령어들을 작성할 때도 비슷하게 나타난다. 필드의 개수보다 3~4배의 프로그램 라인을 작성해야 한다는 사실은 프로그램 개발 시 많은 코딩 시간을 요구할 뿐만 아니라, 유지보수 시 중복 작업을 요구한다. 예를 들어 필드가 추가 또는 삭제되었을 경우 소스 코드에서 3~4 군데를 동시에 수정해야 하는 중복(duplication)의 문제가 발생한다. 이러한 중복의 문제는 동일한 테이블을 프로그램의 여러 군데에서 서로 다른 명령어로 접근(INSERT, UPDATE, SELECT 등)할 때 또 다시 발생한다.

(나) 문자열

C/C++ 기반 응용프로그램에서 필드를 처리할 때 또한 흔히 문제가 되는 것이 문자형 필드의 크기 관리이다. 각종 필드의 이름과 그 크기를

<표 5> Pro*C INSERT 프로그램 예제 분석

void f_insert() {	
EXEC SQL BEGIN DECLARE SECTION; int x; char *y; int z; EXEC SQL END DECLARE SECTION;	(1) 호스트 변수 선언

x = 20100301;
y = "Smith";
z = 3;
	(2) 호스트 변수 값 설정

EXEC SQL INSERT INTO emp(empno, ename, deptno)	(3) 필드 이름 선언
VALUES(:x, :y, :z);	(4) 필드 값 설정
}	

일일이 맞추는 작업은 개발 시뿐만 아니라, 필드 설계의 변경에 따른 정보시스템 유지보수 시에도 담당자들에 과중한 업무 부담을 주며 동시에 높은 에러 가능성을 내포한다. 개발 및 유지보수 담당자들은 프로그램 소스 코드의 곳곳에 위치한 필드 이름 및 그 크기들을 일일이 작성 또는 수정하여야 하고 또한 엄밀한 확인 작업을 거쳐야 한다.

Pro*C에서 문자열을 처리하기 위해 제공하는 자료형이 VARCHAR이다. VARCHAR는 Pro*C 전처리기가 인식하여 처리하는 의사자료형(psuedo-type)으로서, 문자열의 뒷부분을 빈 칸(‘ ’)으로 채운 가변 길이의 문자열을 표현해준다. 예를 들어, Pro*C 전처리기는 다음 선언을 아래와 같은 C/C++ 구조체 선언으로 바꾸어준다.

```
VARCHAR username[21];
→
struct
{
    unsigned short          len;
    unsigned char arr[21];
} username;
```

문자열 자료에 대해 SQL SELECT 명령을 처리하는 Pro*C 프로그램 예제가 <표 6>에 나타나 있다. 이 프로그램은 사용자명(username)이 "ysmith"인 고객의 주소(address)를 출력하기 위한 것이다. 여기서 사용자명 및 주소 필드가 VARCHAR 형 문자열로 처리되고 있다. 먼저 선언문 “VARCHAR h_username[20 + 1]”를 살펴보자. 사용자명을 위한 호스트 변수 h_username이 선언되고 있으며, 그 크기는 21이다. 원래 데이터베이스에서 username 필드의 크기는 20으로 선언되어 있지만, C/C++에서 문자열을 처리할 경우 문자열을 뒤에 빈 문자(null character, ‘\0’)를 덧붙이기 때문에 이를 위한 공간을 하나 추가해 주어야 하므로 호스트 변수 h_username의 크기는 21로 선언되어야 한다.

다음으로 2 개의 프로그램 라인 “strcpy (h_username.arr, "ysmith");”와 “h_username.len = strlen("ysmith");”를 살펴보자. 이것은 VARCHAR 형 호스트 변수 h_username이 C/C++ 측면에서는 len과 arr이라는 어트리뷰트(attribute)를 갖는 구조체이기 때문에, 문자열 복사를 위하여 2 라인으로 나타났다. 마지막으로 “printf("The address is %.*s\n", h_address.len, h_address.arr);”을 살펴보자. h_address 역시 구조체이므로 문자열 값은 h_address.arr에 나타나있다. 그런데, h_address.arr은 보통의 C/C++ 문자열처럼 빈 문자(‘\0’)로 끝나는(NULL-terminated) 형태가 아니라 빈 칸(‘ ’)이 문자열 뒤에 채워진 형태이므로, h_address.len을 이용하여 문자열의 크기를 printf 함수에 알려주어야 한다. 만약 문자열의 크기를 알려주지 않는다면, printf 함수는 단순하게 빈 문자(‘\0’)를 만날 때까지 문자들을 출력하려고 할 것이고, 이것은 심각한 수행 오류(run-time error)를 발생시킬 것이다.

이와 같은 문자열 처리 방식은 프로그래밍 측면에서 몇 가지 문제를 일으킨다. 첫째로 문자열이, 빈 문자(‘\0’)로 끝나는(NULL-terminated) 보통의 C/C++ 문자열과는 다른 형태이므로, 문자열 복사 또는 출력 시 문자열의 길이를 구조체의 len 어트리뷰트를 이용하여 일일이 처리해 주어야 하며, 이것은 개발자에게 별도의 처리업무를 부과하고 나아가 프로그래밍 시간을 증가시

<표 6> Pro*C 문자열 처리 프로그램 예제

```
{
    EXEC SQL BEGIN DECLARE SECTION;
    VARCHAR h_username[20 + 1];
    VARCHAR h_address[50 + 1];
    EXEC SQL END DECLARE SECTION;
    strcpy(h_username.arr, "ysmith");
    h_username.len = strlen("ysmith");
    EXEC SQL SELECT address INTO :h_address
                FROM Customer
                WHERE username=:h_username;
    printf("The address is %.*s\n", h_address.len, h_address.arr);
}
```

킨다. 특히, 문자열 필드의 개수가 많을 때는 개발자로 하여금 각 필드들의 크기를 일일이 확인하여 이에 맞게 프로그래밍 해주어야 하는 부담을 갖게 한다. 둘째로 VARCHAR 선언 시 필드의 크기(정확하게는 "크기 + 1")를 써주어야 하는데, 나중에 데이터베이스 설계 변경에 따라 필드의 크기가 바뀔 경우 담당자가 해당 호스트 변수들의 선언을 다 찾아서 그 크기를 일일이 바꾸어 주어야 하는 유지보수 상의 문제를 발생시킨다.

(다) 지시 변수

지시 변수(indicator variable)는 필드 값의 NULL 여부(missing 여부)를 표시해주기 위해 호스트 변수에 선택적으로 추가되는 변수이다. 지시 변수는 2 바이트 정수(short int)로 선언되며, Pro*C SQL 문장에서 호스트 변수 직후에 콜론(':')에 붙어서 나타난다. <표 7>에 지시 변수를 이용한 SQL SELECT 명령의 예제가 나타나 있다. 예제에서는 address 필드에 대한 지시 변수 i_address가 short int 형으로 선언되었으며, SELECT 문장 내에서 ":h_address:i_address"와 같이 호스트 변수 바로 뒤에 붙어 나타난다. 또한 i_address의 값이 0보다 작으면 address 필드의 값이 "없음"(NULL, missing)을 표시하고 0보다 크거나 같으면 "있음"을 표시하므로, 이에 맞추어 출력문(printf)이 if 문을 이용하여 작성되고 있다. 지시 변수의 값에 따른 호스트 변수의 구체적인 상태에 대해서는 Oracle Pro*C 설명서(Oracle, 2005)를 참조하기 바란다.

지시 변수는 필드, 즉 호스트 변수의 상태를 알기 위해 반드시 필요하기는 하지만, 호스트 변수와 유사하게 개발 및 유지보수 상의 문제점을 야기한다. 지시 변수는 (1) 선언되어야 하고, (2) 호스트 변수 뒤에 나타나야 하고, (3) 지시 변수의 값에 따라 호스트 변수 값이 처리되어야 한다. 따라서 지시 변수 처리를 위해 필드 개수의 3배 이상에 해당하는 프로그램 라인이 작성되어야 하고, 또 필드의 추가 또는 삭제 시 해당 부분을 전부 찾아 고쳐주어야 한다. 지시 변수는 호스트 변수와 함께 나타나므로 호스트 변수와 동시에 생각하면, 호스트 변수와 지시 변수의 처리를 위해서 필드 개수의 6~7배의 프로그램 라인을 작성해야 한다는 이야기가 되고, 이것은 또한

<표 7> Pro*C 지시 변수 이용 프로그램 예제

```
{
    EXEC SQL BEGIN DECLARE SECTION;
    VARCHAR h_username[20 + 1];
    VARCHAR h_address[50 + 1];
    EXEC SQL END DECLARE SECTION;
    short int i_address;
    strcpy(h_username.arr, "ysmith");
    h_username.len = strlen("ysmith");
    EXEC SQL SELECT address INTO :h_address:i_address
             FROM Customer
             WHERE username=:h_username;
    if (i_address < 0)
        printf("The address is 'missing'");
    else
        printf("The address is %.*s\n", h_address.len, h_address.arr);
}
```

6~7 번의 중복의 문제도 발생시킨다.

3.2 Pro*C 관리상의 문제 분석

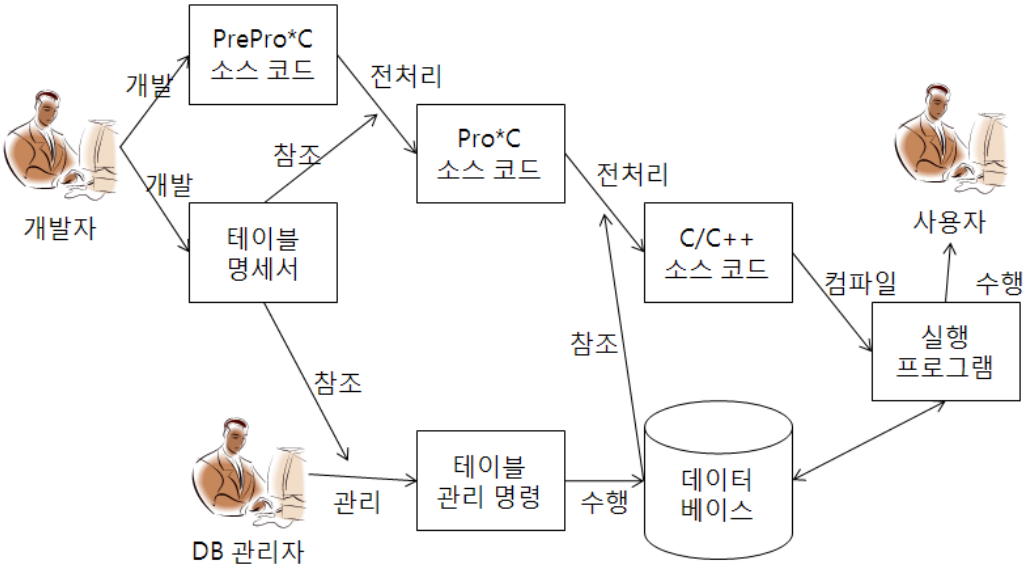
지금까지 Pro*C의 호스트 변수, 문자열, 지시 변수 등 구문을 살펴보면서 프로그램 개발 및 유지보수 관리상의 문제점들을 살펴보았다. 이들 문제점들은 크게 "반복"과 "중복"의 문제로 정리될 수 있다. 여기서 반복이란 유사한 프로그램 라인이 Pro*C 프로그램 내에서 계속해서 나타나는 것을 말하며, 이러한 반복에는 각 테이블의 필드 이름들이 Pro*C 프로그램 내에서 나열되는 것과 문자열 및 지시 변수에 대한 유사한 처리가 필드별로 동일하게 반복되는 것 등이 포함된다. 필드 이름은 서로 다르므로 필드 이름의 나열이 동일한 이름의 반복은 아니지만, 특정 집합(테이블)의 요소(필드 이름)들을 나열한다는 측면에서 반복이라고 볼 수 있다. 예를 들어, <표 7>에서 Customer 테이블의 username, address 필드는 변수로서 나열되어 선언되고 있으며, 이것은 Customer 테이블에 속한 필드 이름들을 나열, 선언한다는 측면에서 선언의 반복이라고 볼 수 있다. 그리고, i_address 지시 변수에 의해 address 필드 값을 출력하는 if 문장은 다른 필드(만약 있다면)의 출력에서도 거의 동일한 형태로 나타나게 되며, 이것도 동일한 if 문의 반복이라고 볼 수 있다. 반복은 필드의 개수가 늘어날수록 개발자에게 자료형, 자료 크기 등을 맞추면서 유사한 작업을 반복하여야 한다는 점에서 어려움을 가중시킨다.

한편, 중복이란 동일한 문장이 프로그램 내의 여러 곳에 여러 번 나타나는 것을 말한다. 예를 들어, <표 7>에서 address 필드와 관련하여 h_address 호스트 변수 선언, i_address 지시 변수 선언, SELECT 명령어 내에서 SELECT 뒤 address 필드 이름 선언, INTO 뒤 h_address 및 i_address 변수의 사용, 출력문에서 h_address 및 i_address 변수의 사용 등 총 7 부분에 address라는 필드 이름이 중복되어 있다. 만약 데이터베이스 설계의 변경에 의해 address라는 이름이 다

른 이름으로 변경될 경우 위 7 부분에 나타난 address들을 전부 고쳐주어야 하며, 한 군데라도 빠뜨리게 되면 이것은 오류로 연결된다. 이와 같이 중복은 필드의 변경 시 이에 맞추어 전부 고쳐주어야 한다는 점에서 유지보수 시 문제점, 즉 동일 사안에 대한 여러 번의 작업과 오류 가능성을 야기한다.

Ⅳ. Pro*C 프로그램 관리 방법론 및 구현

본 연구의 핵심은 앞 절에서 파악한 Pro*C 프로그램의 반복과 중복의 문제를 제거함으로써 임베디드 SQL 프로그램의 개발 및 유지보수 관리를 용이하게 하기 위한 방법론을 개발하는 데 있다. 이를 위한 기본 아이디어는 개발 프로그램에서 사용하는 데이터베이스 테이블과 그 필드들의 정보를 한 곳에 모아서 선언하고, 프로그램의 나머지 부분에서는 그것을 참조만 하도록 하는 것이다. 예를 들어, 호스트 변수 선언 시 반복의 문제를 해결하기 위해서 프로그램 소스 코드의 해당 부분에는 호스트 변수를 선언하라는 참조 지시만 해 놓는다. 그런 후에 별도의 소프트웨어(전처리기)가 테이블 및 필드 정보를 바탕으로 참조 지시문을 호스트 변수의 선언으로 대체하여 Pro*C 소스 코드를 생성한다. 즉, 전처리기인 Pro*C 이전에 수행되는 소프트웨어로서 새로운 전처리기를 만드는 것이다. 본 연구에서는 이 전처리기를 PrePro*C라고 부르도록 하겠다. 다시 말하면, 반복이나 중복의 문제를 가져올 Pro*C SQL 명령문 대신에 참조 지시문, 즉 PrePro*C 명령문을 포함한 소스 코드를 작성하면, PrePro*C 전처리기가 이것을 Pro*C 소스 코드로 바꾸어 주고, 이것을 다시 Pro*C 전처리기가 보통의 C/C++ 소스 코드로 바꾸어 주면, 이것을 컴파일하여 최종 수행 프로그램을 만들게 되는



<그림 1> PrePro*C 전처리기를 이용한 소스 코드 관리 구조

것이다. 이 과정을 그림으로 표시하면 <그림 1>과 같다.

위 그림에 나타난 PrePro*C 전처리기를 이용한 프로그래밍 과정을 단계별로 구체적으로 살펴보자.

(1) 관계형 데이터베이스 기반 응용 프로그램 개발자는 DB 설계서 등을 바탕으로 “테이블 명세서”를 작성한다.
(2) 또한, 개발자는 Pro*C SQL 명령어 대신에 참조 지시문만을 포함한 PrePro*C 소스 코드를 개발한다.
(3) PrePro*C 전처리기는 테이블 명세서를 참조하여 PrePro*C 소스 코드를 Pro*C 소스 코드로 변환한다. 생성된 소스 코드는 기존의 Pro*C 임베디드 프로그램이 되며, 따라서, 이 후의 절차도 기존 Pro*C 프로그램 처리 절차와 동일하다.
(4) Pro*C 전처리기는 데이터베이스를 참조하여 Pro*C 소스 코드를 C/C++ 소스 코드로 변환한다.
(5) C/C++ 컴파일러는 C/C++ 소스 코드를 실행 프로그램으로 컴파일한다.
(6) 사용자는 실행 프로그램을 실행하여 원하는 작업을 수행한다.

위 과정과 별도로 DB 관리자는 DB 설계서 또는 테이블 명세서 등을 이용하여 데이터베이스에 테이블을 생성하고 관리한다. 지금부터는 PrePro*C 전처리기를 위한 테이블 명세서의 구조, PrePro*C 명령어 구문 등을 설명하도록 하겠다.

4.1 테이블 명세서

PrePro*C 테이블 명세서는 다음과 같은 구조를 갖는다.

테이블이름$_1$
필드이름$_{11}$ [TAB] DB자료형 [TAB] 자료크기 [소수점크기] [TAB] C/C++자료형
필드이름$_{12}$ [TAB] DB자료형 [TAB] 자료크기 [소수점크기] [TAB] C/C++자료형
...
필드이름$_{1n1}$ [TAB] DB자료형 [TAB] 자료크기 [소수점크기] [TAB] C/C++자료형
[ENTER]
테이블이름$_2$
필드이름$_{21}$ [TAB] DB자료형 [TAB] 자료크기 [소수점크기] [TAB] C/C++자료형
필드이름$_{22}$ [TAB] DB자료형 [TAB] 자료크기 [소수점크기] [TAB] C/C++자료형
...
필드이름$_{2n2}$ [TAB] DB자료형 [TAB] 자료크기 [소수점크기] [TAB] C/C++자료형
...

테이블 명세서는 응용 프로그램에서 사용할 복수개의 테이블 각각에 대해서 명세하며, 그것은 테이블 이름으로 시작한다.

```
LG
LGSTRING  VARCHAR2  10    VARCHAR
LGINT     NUMBER    20    int
LGDOUBLE  NUMBER    30 2  double
```

예를 들어 위 테이블 명세는 LG라는 테이블에 대해 명세하고 있다. 테이블 이름 다음에는 필드에 대한 명세들이 따라온다. 위 예제에서는 LG 테이블의 LGSTRING, LGINT, LGDOUBLE

<표 8> 테이블 명세서 예제

LG			
LGSTRING	VARCHAR2	10	VARCHAR
LGINT	NUMBER	20	int
LGDOUBLE	NUMBER	30 2	double
LGUSHORT	NUMBER	40	unsigned short
LGLDOUBLE	NUMBER	50 0	long double
LGDATE	NUMBER	70	DATE
LGTIME	NUMBER	60	TIME
LGTINTERVAL	NUMBER	80	TINTERVAL
LGLONGLONG		10000	VARCHAR
LGLONGRAW	LONG RAW	100000	LONG VARRAW

등 세 개의 필드에 대해 명세하고 있다. 각 필드 이름 뒤에는 그 필드의 데이터베이스에서의 자료형, 필드의 크기, 그리고 Pro*C 자료형이 표시된다. 위 예에서 LGSTRING 필드의 데이터베이스 자료형은 VARCHAR2이고, 크기는 10 바이트, Pro*C 자료형은 VARCHAR이다. 실수 자료의 경우에는 필드의 크기가 두 개의 정수로 표시되며, 두 번째 정수는 소수점 이하 자리수를 표시한다. 위 예에서 LGDOUBLE 필드의 값은 데이터베이스에서 총 30 바이트로 표현되며, 소수점 이하 자리수는 2이다. 문자형 자료의 경우 필드의 크기는 데이터베이스 및 Pro*C 프로그램에서 모두 필요하지만, 수치형 자료의 경우에는 필드의 크기가 데이터베이스에서만 필요함에 유의하기 바란다. 지금까지 설명한 테이블 명세서 구문에 따라 작성한 테이블 명세서 예제가 <표 8>에 나타나 있다. 이 예제에서는 LG라는 1 개 테이블의 10 개 필드들을 명세하고 있다.

4.2 PrePro*C 명령문

PrePro*C 명령문은 다음과 같은 구조를 갖는다.

#PPC명령어 테이블이름 [선택사항$_1$] ... [선택사항$_n$]

PrePro*C 명령문(PPC 명령문)은 PrePro*C 소스 코드의 일부로 포함되며, 명령문은 #으로 시작한다. # 뒤에는 PPC 명령어가 나타나고, 그 뒤에는 테이블 이름, 그리고 다시 그 뒤에는 PPC 명령어에 따른 선택사항들이 나타난다. 예를 들어 다음 PPC 명령문을 살펴보자.

#PPCHVAR LG -1 extern LVR_%s %s LGSTRING LGUSHORT LGLDOUBLE

위 명령어 PPCHVAR는 호스트 변수를 선언하기 위한 명령이다. 이어서 LG는 테이블 이름이며, -1은 명령문 뒷부분에 나타난 LGSTRING, LGUSHORT, LGLDOUBLE 등을 제외하고 호스트 변수를 선언하라는 뜻이다. 이어서 extern은 Pro*C, 즉 C/C++ 변수 선언 시 앞에 extern 한정사를 붙이라는 뜻이다. LVR_%는 LONG RAW 형인 LGLONGRAW 필드의 구조체 형의 이름에 대한 형식인데, 여기서는 자세한 설명을 생략하기로 하겠다. 그리고, %s는 호스트 변수 이름 작성 형식인데, 이것은 필드 이름을 그대로 호스트 변수 이름으로 사용하라는 뜻이 된다. 단,

현재 구현된 PrePro*C 전처리기에서는 대문자를 전부 소문자로 변경한다. 위 PPC 명령문을 PrePro*C가 전처리하면 다음과 같은 Pro*C 소스 코드가 생성된다.

```
extern int                  lgint;
extern double               lgdouble;
extern long int             lgdate;
extern long int             lgtime;
extern long int             lgtinterval;
extern VARCHAR              lglong[10001];
extern LVR_LGLONGRAW        lglongraw;
```

위 예의 lglong 변수의 경우 데이터베이스 자료 크기가 10,000인데 비하여, C/C++ 자료 크기는 10,001임에 유의하기 바란다. 이것은 앞 절에서 설명한 바와 같이 C/C++에서 문자열의 끝을 빈 문자('\0')로 표시하기 때문에, 빈 문자를 넣기 위한 추가적인 1 바이트 공간을 고려한 결과이다. <표 8>의 테이블 명세서 예제를 바탕으로 예제 PrePro*C 프로그램을 전처리한 결과가 <표 9>에 나타나 있다. <표 9>의 "sample.ppc" 파일은 호스트 변수 선언(PPCCVAR 명령), 지시 변수 선언(PPCIVAR 명령), SELECT 문(PPCSELECT 명령), 출력문(PPCHPRINT 명령)으로 이루어진 응용 프로그램의 PrePro*C 소스 코드를 표시하고 있다. PrePro*C 전처리기는 이것을 변환하여 "sample1.pc"의 Pro*C 소스 코드를 생성해 준다. 한편, <표 10>에는 현재 구현된 대표적인 PrePro*C 명령어가 요약되어 있다.

<표 9> PrePro*C 전처리 예제

<sample1.ppc> 파일 (소스)

```
void select()
{
  // 호스트 변수
   EXEC SQL BEGIN DECLARE SECTION;
#PPCCVAR LG -1 0 LVR_%s %s
   EXEC SQL END DECLARE SECTION;
  // 지시 변수
#PPCIVAR LG -1 0 i_%s
  // select 처리
#PPCSELECT LG -1 %s i_%s
            WHERE     LGAAAA = 20;
  // 출력
#PPCHPRINT LG -1 %s i_%s buf <P> </P>
}
```

<sample1.pc> 파일 (생성)

```
void select()
{
  // 호스트 변수
   EXEC SQL BEGIN DECLARE SECTION;
  char                  lgstring[11];
  int         lgint;
  double      lgdouble;
  unsigned short int    lgushort;
  long double           lgldouble;
```

```
long int   lgdate;
long int   lgtime;
long int   lgtinterval;
char                lglong[10001];
LVR_LGLONGRAW  lglongraw;
EXEC SQL END DECLARE SECTION;
// 지시 변수
short int  i_lgstring;
short int  i_lgint;
short int  i_lgdouble;
short int  i_lgushort;
short int  i_lgldouble;
short int  i_lgdate;
short int  i_lgtime;
short int  i_lgtinterval;
short int  i_lglong;
short int  i_lglongraw;
// select 처리
SELECT
                    LGSTRING,
                    LGINT,
                    LGDOUBLE,
                    LGUSHORT,
                    LGLDOUBLE,
                    LGDATE,
                    LGTIME,
                    LGTINTERVAL,
                    LGLONG,
                    LGLONGRAW
          INTO      :lgstring :i_lgstring,
                    :lgint :i_lgint,
                    :lgdouble :i_lgdouble,
                    :lgushort :i_lgushort,
                    :lgldouble :i_lgldouble,
                    :lgdate :i_lgdate,
                    :lgtime :i_lgtime,
                    :lgtinterval :i_lgtinterval,
                    :lglong :i_lglong,
                    :lglongraw :i_lglongraw
          FROM      MBTNLG
          WHERE     LGAAAA = 20;
// 출력
if (i_lgstring == -1)
   printf(" <P>%10s</P>\n", "");
else
   printf(" <P>%-10.*s</P>\n", ((int)lgstring.len < 10 ? (int)lgstring.len : 10), (char *)lgstring.arr);
if (i_lgint == -1)
   printf(" <P>%20s</P>\n", "");
else
   printf(" <P>%20d</P>\n", lgint);
if (i_lgdouble == -1)
   printf(" <P>%30s</P>\n", "");
else
   printf(" <P>%30f</P>\n", lgdouble);
if (i_lgushort == -1)
   printf(" <P>%40s</P>\n", "");
else
   printf(" <P>%40u</P>\n", (unsigned int)lgushort);
```

```
  if (i_lgldouble == -1)
    printf(" <P>%50s</P>\n", "");
  else
    printf(" <P>%50lf</P>\n", lgldouble);
  if (i_lgdate == -1)
    printf(" <P>%70s</P>\n", "");
  else
    printf(" <P>%-70s</P>\n", ICECNum2Date(lgdate, buf));
  if (i_lgtime == -1)
    printf(" <P>%60s</P>\n", "");
  else
    printf(" <P>%-60s</P>\n", ICECNum2Time(lgtime, buf));
  if (i_lgtinterval == -1)
    printf(" <P>%80s</P>\n", "");
  else
    printf(" <P>%-80s</P>\n", ICECNum2TInterval(lgtinterval, buf));
  if (i_lglong == -1)
    printf(" <P>%11s</P>\n", "");
  else
    printf(" <P>%-11.*s</P>\n", ((int)lglong.len < 11 ? (int)lglong.len : 11), (char *)lglong.arr);
}
```

<표 10> PrePro*C 명령어 목록

명령어	설명
PPCCRTTABLE	테이블을 생성하는 Pro*C 명령문을 생성한다.
PPCDROPTABLE	테이블을 삭제하는 Pro*C 명령문을 생성한다.
PPCINSERT	레코드를 생성(insert)하는 Pro*C 명령문을 생성한다.
PPCUPDATE	레코드를 수정(update)하는 Pro*C 명령문을 생성한다.
PPCDELETE	레코드를 삭제(delete)하는 Pro*C 명령문을 생성한다.
PPCSELECT	레코드를 검색(select)하는 Pro*C 명령문을 생성한다.
PPCFETCH	검색(select) 결과가 복수 개일 경우 한 레코드씩 가져오는(fetch) Pro*C 명령문을 생성한다.
PPCCVAR	호스트 변수들을 선언하는 C/C++ 문장들을 생성한다.
PPCIVAR	지시 변수들을 선언하는 C/C++ 문장들을 생성한다.
PPCSET	호스트 변수의 각 값들을 할당(set)하는 C/C++ 문장들을 생성한다.
PPCHPRINT	호스트 변수의 값들을 HTML 문장으로 출력한다.

4.3 PrePro*C 구현

PrePro*C는 현재 명령 프롬프트(command-line prompt) 기반으로 구현되어 있다. 이것은 명령 프롬프트 기반의 Pro*C 전처리기 및 보통의 C/C++ 컴파일러와 동일한 환경을 갖도록 고안된 것이다. 예를 들어, UNIX 환경에서 PrePro*C 및 Pro*C 전처리기 그리고 C/C++ 컴파일러가 명령 프롬프트 기반으로 제공되면 UNIX의 make 명령 및 <표 11>과 같은 makefile을 이용하여 모든 전처리 및 컴파일 과정을 다음과 같은 단 한 번의 명령으로 처리할 수 있다. 결과적으로 makefile은 응용 프로그램의 소스 코드, 테이블 명세서, 수행 프로그램 등에 대한 정보를 하나의 프로젝트 형태로 묶어 일관성 있게 관리할 수 있도록 해 준다.

```
make -f makefile sample
```

<표 11>에서 프롬프트 명령어 ppc는 PrePro*C 전처리기 수행 프로그램이며, sample1.spc는 테이블 명세서 파일이다. ppc 명령어의 두 번째 인수는 PrePro*C 소스 코드 파일 이름이다. 예를 들어, 이 파일의 이름이 “sample.ppc”라면 “ppc sample1.spc sample.ppc” 명령문은 sample.ppc를 PrePro*C로 전처리하여 “sample.pc” Pro*C 소스 코드를 생성해 준다. Pro*C 전처리기 수행 프로그램(명령어)인 proc는 “proc sample.pc $(OPT_ORA) $(DEF_PROC)” 명령문에 의해 “sample.pc”를 전처리하여 “sample.cpp” C++ 소스 코드를 생성해 주며, 다시 C++ 컴파일러인 gcc가 “gcc -o sample.o -c sample.cpp” 컴파일 명령과 “gcc -o sample sample.o -lclntst9 -lld -lpthreads -ldl -lodm” 링크 명령에 의해 이것을 sample이라는 수행 프로그램으로 만들어준다.

현재 PrePro*C 전처리기는 Pro*C 전처리 및 SQL 명령어 전처리 기능을 갖도록 UNIX 명령 프롬프트 환경 하에서 개발되어 있으며, 앞으로 ODBC, JDBC 등도 지원할 수 있도록 확장 중에 있다. 이러한 확장은 PrePro*C의 “테이블 명세서를 참조한 전처리”라는 기본적인 구조의 변경을 요구하지는 않으며, 단지 PrePro*C 명령어의 추가만으로 가능하고, 또한 새로운 데이터베이스 인터페이스의 표준이 개발되어도 무리 없이 지원 가능할 것으로 예상된다.

<표 11> makefile 예제

```
.SUFFIXES:          .ppc .pc .cpp
.ppc.pc:
        ppc sample1.spc $*
.pc.cpp:
        proc $* $(OPT_ORA) $(DEF_PROC)
.cpp.o:
        gcc -o $@ -c $<
sample: sample.o
        gcc -o $@ sample.o -lclntst9 -lld -lpthreads -ldl -lodm
```

Ⅴ. 토의

앞 절에서는 Pro*C 프로그램의 반복과 중복 문제를 해결하기 위한 방법론으로 PrePro*C라는 전처리기를 이용한 데이터베이스 기반 응용 프로그램 개발 구조를 제안하고 구현하였다. PrePro*C 전처리기는 테이블 명세서의 내용을 참조하여 Pro*C 프로그램에서 필요한 호스트 변수의 선언, 지시 변수의 선언, SQL 문에서의 필드의 표시 등 반복적 나열이 필요한 것을 대신 생성해 줌으로써 개발자의 반복 작업을 줄여준다. 또한, 필드의 수정, 삭제 등 변경 발생 시 테이블 명세서만 수정한 후 PrePro*C 전처리기에 의해 Pro*C 소스 코드를 재생성함으로써 필드의 변경에 따라 프로그램 소스 코드의 여러 부분에서 수행해야 할 수정 작업들을 하지 않아도 되게 함으로써 중복의 문제도 해결하였다.

예를 들어, <표 9>의 예제를 살펴보면, 단지 4 개의 PrePro*C 명령문 만으로 90 여 라인에 이르는 Pro*C 소스 코드를 생성함으로써, 개발 시 지루한 반복 업무와 유지 보수 시 오류를 범하기 쉬운 중복 업무를 줄이고 있다. 만약 예제 테이블 LG의 필드가 100 개로 증가한다고 하며, PrePro*C 소스 코드 프로그래머 입장에서는 소스 코드에 추가하거나 변경할 내용은 하나도 없다. 단지 테이블 명세서에 100개의 새로운 필드를 명세해 준 후 Pre*ProC 전처리기를 수행하기만 하면 900 여 라인에 이르는 새로운 Pro*C 소스 코드를 얻게 된다. <표 9>의 예제는 단지 한 개의 SQL SELECT 문장으로 이루어진 프로그램만 살펴본 것이고, 실제 기업정보시스템은 다수의 여러 종류의 SQL 문장으로 이루어지기 때문에 Pre*ProC 전처리기를 이용한 반복과 중복의 제거는 데이터베이스 기반 응용 프로그램의 개발 및 유지보수 시간 및 노력을 현저히 줄여줄 것임에 틀림없다.

PrePro*C 전처리기의 개념은 데이터베이스 관리에도 적용할 수 있다. 데이터베이스 관리자(administrator)는 Pro*C 프로그래밍 전에 데이터베이스에 해당 테이블을 생성해 주어야 한다. 이것은 보통 데이터베이스 관리 프로그램(예를 들어, Oracle sqlplus, QuestSoft TOAD 등)을 이용하여 SQL 명령문을 작성, 수행함으로써 이루어진다. 이때도, 예를 들어 테이블을 생성하기 위하여 CREATE TABLE 명령을 작성하려면, 모든 필드의 이름과 그 크기를 정확히 작성해야 한다. 이것은 앞에서 이야기한 반복의 문제를 다시 야기한다. 또한, 관리자의 사소한 실수로 특정 필드를 빠뜨리거나, 크기를 잘못 표시하거나, 또는 필드 이름을 잘못 썼을 경우 이것은 응용 프로그램과의 불일치성을 가져와 프로그램 오류로 이어지게 된다. 이 문제 역시 PrePro*C 전처리기를 SQL 명령문 생성에 사용하도록 함으로써 해결할 수 있다. 예를 들어 다음과 같은 SQL 전처리 명령을 이용하여 아래의 SQL 명령문을 생성할 수 있다.

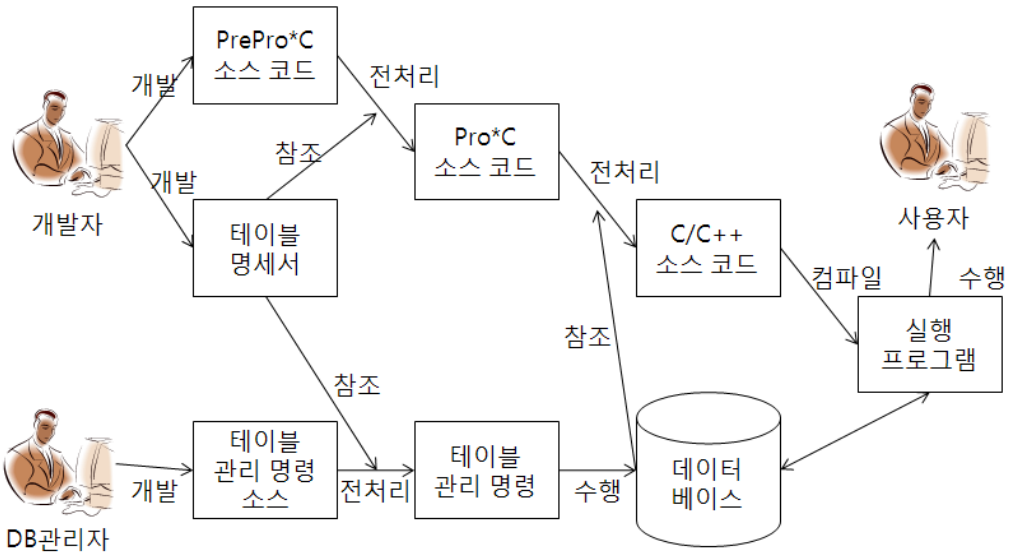
```
#PSQLCRTTABLE LG 0
→
CREATE TABLE LG
        (LGSTRING       VARCHAR2(10),
         LGINT          NUMBER(20),
         LGDOUBLE       NUMBER(30, 2),
         LGUSHORT       NUMBER(40),
         LGLDOUBLE      NUMBER(50),
         LGDATE         NUMBER(70),
         LGTIME         NUMBER(60),
```

```
LGTINTERVAL     NUMBER(80),
LGLONG          LONG,
LGLONGRAW       LONG RAW);
```

위 예에서 PSQLCRTTABLE 명령어는 SQL의 CREATE TABLE 명령문을 생성해주는 전처리 명령어이다. 따라서, 데이터베이스 관리자도 데이터베이스 관리 명령어를 일일이 작성하기 보다는, 테이블 명세서와 PrePro*C 전처리기를 이용하여 SQL 명령문을 생성함으로써, 데이터베이스 관리 시 지루한 반복 작업과 불일치에 의한 오류 가능성을 배제할 수 있다. PrePro*C 전처리기를 데이터베이스 관리에도 사용하는 데이터베이스 기반 응용 프로그램의 개발 구조가 <그림 2>에 나타나 있다. 이와 같이 PrePro*C 전처리기를 활용하면 데이터베이스 기반 응용 프로그램의 개발, 유지보수 및 관련 데이터베이스의 관리상의 반복, 중복 작업을 없애고, 프로그램 간의 일치성(consistency)을 확보할 수 있다.

Ⅵ. 결 론

본 연구에서는 관계형 데이터베이스 기반 임베디드 SQL 프로그램의 개발 및 유지보수 관리를 용이하게 하기 위한 방법론을 개발하고 구현하였다. 객체형 데이터베이스(OODB), 객체-관계형 데이터베이스(ORDB) 등의 새로운 개념의 등장에도 불구하고 관계형 데이터베이스(RDB)는 아직도 기업정보시스템의 근간이 되고 있으며, 대부분의 기업정보시스템들이 관계형 데이터베이스 기반의 임베디드 SQL 프로그램으로 개발되고 있다. 본 연구에서는 임베디드 SQL 프로그래밍 언어 중 가장 대표적인 Pro*C에 초점을 맞추어, Pro*C 기반 프로그램의 개발 및 유지보수 관리 방법론을 개발, 구현하고자 하였다. 이를 위해 먼저 Pro*C의 SQL 문장과 호스트 변수 구문을 살펴보고, 이로부터 Pro*C에서 호스트 변수, 지시 변수, 문자열의 처리 등이 개발 및 유지보수 담당자에게 많은 부담을 주는 것으



<그림 2> PrePro*C 전처리기를 이용한 소스 코드 및 데이터베이스 관리 구조

로 파악하였다. 특히, 다수 필드 이름의 나열, 문자열 변수의 크기 처리, 프로그램 곳곳에 나타나는 동일한 필드 이름의 관리 등이 그 부담의 주요 요인으로 파악되었고, 본 연구에서는 이 부담 요인들을 일반화하여 “반복”과 “중복”이라는 용어로 개념화하였다. 데이터베이스 테이블의 수십, 수백 개의 필드들은 유사한 형태로 반복적으로 나열되며, SQL 문장, 호스트 변수, 지시변수 등에 동일한 이름이 중복적으로 나타나므로, 이것들이 개발자의 개발 시간 중 많은 부분을 차지하게 되고 또 나중에 유지보수를 어렵게 한다.

반복과 중복의 문제를 해결하기 위한 방법론으로 본 연구에서는 Pro*C 소스 코드를 생성해 주는 전처리기를 제안하고, 그것을 PrePro*C라고 이름 붙였다. PrePro*C 명령문을 포함하여 작성된 PrePro*C 소스 코드는 PrePro*C 전처리기에 의하여 Pro*C 소스 코드로 변환된다. 이 Pro*C 소스 코드는 기존의 Pro*C 임베디드 SQL 프로그램 처리 순서에 따라 C/C++ 소스 코드로 변환 된 후, 다시 C/C++ 컴파일러에 의하여 수행 프로그램으로 컴파일 및 링크된다.

나아가, 본 연구에서는 데이터베이스 관리자의 반복과 중복의 문제도 해결하기 위한 방법론으로 PrePro*C에 의해 데이터베이스 관리 SQL 명령문을 생성하는 방법론도 제안하였다. 데이터베이스 관리 PrePro*C 명령문을 작성하면 PrePro*C 전처리기가 이것을 데이터베이스 관리 SQL 명령문으로 변환, 생성한다. 이렇게 생성된 SQL 명령문을 데이터베이스 관리 프로그램에서 수행하여 데이터베이스를 관리함으로써 이 데이터베이스를 이용하는 응용 프로그램과의 일치성을 보장할 수 있었다. 또한, PrePro*C 전처리기를 Pro*C 전처리기 및 C/C++ 컴파일러와 동일하게 명령 프롬프트 상에서 작동하게 함으로써 관계형 데이터베이스 기반 응용 프로그램 개발 시 소스 코드, 테이블 명세서, 수행 프로그램 등에 대한 정보를 하나의 프로젝트(makefile)로 묶어 일관성 있게 관리할 수 있도록 하였다.

현재 PrePro*C 전처리기는 Pro*C 소스 코드 전처리 및 SQL 명령어 전처리 기능을 갖도록 UNIX 명령 프롬프트 환경 하에서 개발되어 있으며, ODBC, JDBC 등 다른 데이터베이스 인터페이스 환경에서도 사용할 수 있도록 기능을 확장 중이다. 본 연구에서 제시, 구현한 방법론 및 PrePro*C 전처리기를 사용할 경우 관계형 데이터베이스 기반 기업정보시스템의 효율적인 개발 및 효과적인 유지보수가 가능할 것으로 기대된다.

참고문헌

김동호, 김진석, 류근호, "e-로지스틱스에서 효율적인 차량관제를 위한 질의 처리기 구현," 한국지리정보학회지, 제 7 권, 제 3 호, 2004, pp. 35-47.

김신희, 류명춘, 박정량, "데이타베이스 공유 환경에서 분산 동시성 제어를 위한 캐쉬 일관성 기법," 정보시스템연구, 제 7 권, 제 2 호, 1998, pp. 259-279.

김은주, 용환승, 이상원, "데이터 웨어하우스 성능 관리를 위한 DBMax의 확장," 정보처리학회논문지D, 제 10-D 권, 제 3 호, 2003.6, pp. 407-416.

김천식, 김경원, 이지훈, 장복선, 손기락, "XQL-SQL 질의 변환을 통한 XQL 질

의 처리 시스템의 설계 및 구현," 정보처리학회논문지D, 제 9-D 권, 제 5 호, 2002.10, pp. 789-800.

심송용, 강희모, 이윤환, "R 언어를 통한 데이터베이스 접근," 한국통계학회논문집, 제 15 권, 제 1 호, 2008, pp. 51-64.

유재건, "관계형 데이터베이스 설계를 위한 개체-관계 모델링 시스템 개발," 대한산업공학회 추계학술대회 논문집, 2003, pp. 64-68.

이재규, 권순범, 김우주, 김민용, 송용욱, 최형림, 전자상거래 원론, 제3판, 법영사, 2002.

이재규, 최형림, 김현수 편저, 인터넷 환경의 지식시스템, 법영사, 2006.

이종민, 강현철, "LOB 캐쉬를 위한 SQL CLI의 확장," 정보처리학회논문지, 제 8-D 권, 제 1 호, 2001.2, pp. 1-9.

이중화, 박유현, 김경석, "멀티미디어 데이터베이스 : 멀티미디어 데이터를 지원하기 위한 SQL 확장," 멀티미디어학회 논문지, 제 2 권, 제 2 호, 1999.6, pp. 109-119.

정석찬, 안태우, 신준기, "데이터 통합 방식의 중소기업용 미들웨어 개발," Entrue Journal of Information Technology, 제 6 권, 제 2 호, 2007, pp.139-149.

정윤수, 이춘열, 김남규, "토픽맵의 다중역할 토픽 보존을 위한 관계형 데이터베이스 구조," 정보시스템연구, 제 18 권, 제 3 호, 2009, pp. 327-349.

정채영, 최규원, 김영옥, 김영균, 강현석, 배종민, "관계형 데이터베이스에서 XML 뷰 기반의 질의 처리 모델," 정보처리학회논문지D, 제 10-D 권, 제 2 호, 2003.4, pp. 221-232.

조동일, 류성열, "SQL 기반 퍼시스턴스 프레임워크," 정보처리학회논문지D, 제 15-D 권, 제 4 호, 2008.8, pp. 549-556.

주진웅, 김학수, 황진호, 손진현, "관계형 데이터 스트림에서 고급 키워드 검색을 위한 질의 최적화," 정보처리학회논문지D, 제 16-D 권, 제 6 호, 2009.12, pp. 859-870.

최현종, 황성욱, 김태영, "XML 웹서비스와 JDBC를 이용한 분산 메타데이터 검색 시스템의 설계 및 구현," 컴퓨터교육학회논문지, 제 7 권, 제 2 호, 2004.3, pp. 25-34.

Easysoft. "ODBC Versions", http://www.easysoft.com/developer/interfaces/odbc/.

Hoffer, Jeffrey A., Mary B. Prescott, and Heikki Topi, Modern Database Management (9th Ed.), Prentice Hall, 2009.

ISO/IEC 9075-3, Information technology, Database languages, SQL, Part 3: Call-Level Interface (SQL/CLI), 2008.

John W. Satzinger, Robert B. Jackson, and Stephen D. Burd, Systems Analysis and Design in a Changing World, 4/E, Course Technology, 2006.

Microsoft, "What's New in ODBC 3.8", http://msdn.microsoft.com/en-us/library/ee388580%28VS.85%29.aspx.

Oracle, JavaTM Platform, Standard Edition 6, API Specification.

Oracle, Pro*C/C++ Programmer's Guide, 10g

Release 2 (10.2), Part Number B14407-01, June 2005.

Sun Microsystems, Inc., JSR 221: JDBC™ 4.0 API Specification, December 11, 2006.

송용욱(yusong@yonsei.ac.kr)

현재 연세대학교 원주캠퍼스 경영학부 부교수로 재직 중이다. 서울대학교 국제경제학과를 졸업하고, 한국과학기술원(KAIST) 경영과학과 및 산업경영학과에서 석사 및 박사학위를 취득하였다. Management Science, Annals of Operations Research, Expert Systems with Applications 등에 논문을 게재한 바 있다. 주요 관심분야는 전자상거래, 정보시스템 개발, 경영분야 문제에의 전문가시스템 응용, 전문가시스템 및 수리계획법과 전자상거래의 통합 등이다.

<Abstract>

A Study on the Development and Maintenance of Embedded SQL based Information Systems

Yong Uk Song*

As companies introduced ERP (Enterprise Resource Planning) systems since the middle of 1990s, the databases of the companies has become centralized and gigantic. The companies are now developing data-mining based applications on those centralized and gigantic databases for knowledge management. Almost of them are using Pro*C/C++, a embedded SQL programming language, and it's because the Pro*C/C++ is independent of platforms and also fast. However, they suffer from difficulties in development and maintenance due to the characteristics of corporate databases which have intrinsically large number of tables and fields. The purpose of this research is to design and implement a methodology which makes it easier to develop and maintain embedded SQL applications based on relational databases. Firstly, this article analyzes the syntax of Pro*C/C++ and addresses the concept of repetition and duplication which causes the difficulties in development and maintenance of corporate information systems. Then, this article suggests a management architecture of source codes and databases in which a preprocessor generates Pro*C/C++ source codes by referring a DB table specification, which would solve the problem of repetition and duplication. Moreover, this article also suggests another architecture of DB administration in which the preprocessor generates DB administration commands by referring the same table specification, which would solve the problem of repetition and duplication again. The preprocessor, named PrePro*C, has been developed under the UNIX command-line prompt environment to preprocess Pro*C/C++ source codes and SQL administration commands, and is under update to be used in another DB interface environment like ODBC and JDBC, too.

Keywords: Administration, Embedded SQL, Preprocessor, Relational Database, SQL

* Yonsei University Wonju Campus, yusong@yonsei.ac.kr

* 이 논문은 2010년 8월 25일 접수하여 1차 수정을 거쳐 2010년 9월 15일 게재 확정되었습니다.