제20회 한국정보처리학회 추계학술발표대회 논문집 제10권 제2호 (2003. 11)

컴포넌트 개발 도구를 위한 CICS/COBOL 및 SQL 생성 규칙 개발

이신혜*, 배철성*, 정민수*
*경남대학교 전자계산학과
e-mail : shi-ny@hanmail.net

Development Of The CICS/COBOL And SQL For a Component Development Tool

Shin-hye Lee*, Cheol-Seong Bae*, Min-Soo Jung*
*Dept. of Computer Science, Kyung-Nam University

요 약

기존에 운용되고 있는 시스템은 새로운 컴퓨팅 기술 및 환경에 적응하기 위하여 현대화 되어야 한다. 최근 소프트웨어 개발 환경은 컴포넌트 지원 도구 개발을 중심으로 발달 되어지고 있다.

COBOL2EJB 는 소프트웨어 컴포넌트로서 기존 시스템을 변화시킬 연계 도구이다. 이 COBOL2EJB 는 코볼 문법을 중심으로 하고 있기 때문에 코볼에 내장되어 있는 CICS 와 SQL 을 지원하기엔 한계점이 있다. 이러한 한계점을 지적하고 해결하기 위한 방안으로서 JJTREE 를 소개한다. 그리고 COBOL2EJB 연계 도구의 사용성을 높이고자 IBM 코볼 소스에 내장된 CICS 와 SQL 을 처리할 수 있도록 JavaCC 를 이용하여 문법 생성규칙을 확장하였다

1. 서론

최근 사용자들의 폭발적인 소프트웨어 개발 요구를 충족시키기 위해서는 새로운 개발 기법들이 등장하고 있다. 이런 소프트웨어 개발 기법중 주목할 할 만한 것은 소프트웨어 재사용 기술과 컴포넌트 기술이다.[2]. 객체지향 프로그래밍 언어들은 이러한 기술을 충족시키기에 충분한 대안이다. 따라서 최근에 개발되는 대부분의 소프웨어는 비주얼 C 계통의 언어 또는 Java 기반의 언어들을 사용하고 있다. 그러나 이러한 현재의 현상에도 불구하고 기존 시스템의 중요성으로 인하여 쉽게 컴포넌트 기반 시스템으로 변환하지 못하는 경우가 있다.

기존에 운용되고 있는 시스템은 새로운 컴퓨팅 기술 및 환경에 적응하기 위하여 현대화 (Modernization)되어야한다[1].

본 논문에서는 COBOL 프로그램을 EJB 로 변환하기 위해 EJB 래퍼 컴포넌트 지원 도구 개발을 소개하고 현재 연구의 한계점을 지적하고 극복을 위한 방안을 제시하고 구현하고자 한다. 본 논문의 구성은 다음과 같다. 2 장에서는 관련 연구로서 기존 시스템을 현대화 하는 방법과 COBOL2EJB 시스템의 구성과 코볼 코드 분석기에 대해 고찰 하며, 연구수행 결과의 문제점과 대안을 제시한다. 3 장에서는 JJTREE 에 대해 소개하고 4 장에서는 JJTREE 를 이용하여 코볼에 내장된 CICS 와 SQL 을 지원하기 위한 문법 규칙을 정의, 구현한다. 그리고 5 장에서는 기존 연구와 본 연구에서 구현된 결과와의 차이점을 설명하며 결론을 맺고 향후 연구 방향을 제시하고자 한다.

2. 관련연구

기존 시스템의 현대화

기존 시스템을 현대화 하는 방법중 가장 주목 받고 있는 것 중의 하나는 래핑(Wrapping)이다.

래핑은 기존 시스템은 그대로 두고 새로운 시스템 환경에서 사용 가능하도록 중간에 미들웨어의 일종인 래퍼(Wrapper)를 둠으로써 신뢰성과 안정성이 높이는 방법이다. 그러나 시스템의 유연성과 확장성 측면을 고려할 때 많은 문제점을 갖고 있다.

본 논문에서는 IBM 메인 프레임에서 운용되고 있는 CICS 코볼 시스템을 EJB 래퍼 컴포넌트로 연계하는 일련의 프로세스를 지원하는 연계 도구(COBOL2EJB)를 효율적으로 활용하기 위한 CICS 와 SQL 문법 생성 규칙을 정의하고 있다.

COBOL2EJB 시스템의 구성

COBOL2EJB(그림 1)는 현재 IBM 메인 프레임에서 운용되고 있는 CICS 코볼 시스템을 EJB 래퍼 컴포넌트로 연계하는 일련의 프로세스를 지원하는 연계 도구이다. 이 시스템은 COBOL 코드 분석기, 시각화 정보 생성기, EJB 래퍼 컴포넌트 생성기, 컴포넌트 시험기로 구성되어 있다 [1].

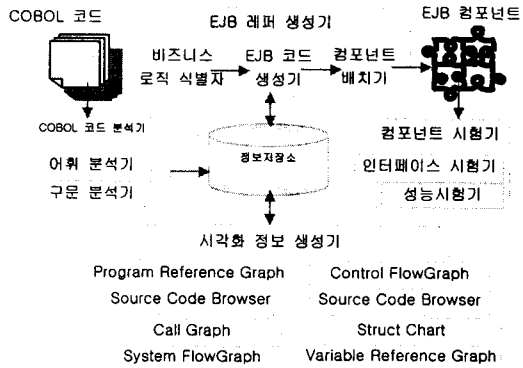

그림 1. COBOL2EJB 시스템 구성도

COBOL 코드 분석기는 코볼소스를 입력으로 받아 어휘 분석과 구문 분석 과정을 거쳐 파서 트리, 관계 테이블, AST 그리고 심볼 테이블 을 생성하고 이를 정보 저장소에 저장한다. 코드 분석기를 통해 분석된 각 정보들은 시각화 정보 생성기 및 EJB 래퍼 생성기 등에서 사용된다.

EJB 레퍼 생성기는 COBOL 코드 분석기를 통해 생성된 각 정보를 이용하여 EJB 컴포넌트화 대상이 될 비즈니스 로직을 식별하는 비즈니스 로직 식별기, 식별된 비즈니스 로직을 EJB 컴포넌트로 생성하는 EJB 코드 생성기와 생성된 EJB 컴포넌트를 웹 어플리케이션 서버에 배치하는 컴포넌트 배치기로 구성되어 있다.

시각화 정보 생성기는 COBOL 코드 분석기에서 분석한 정보들을 이용하여 이들 자료들을 그래픽을 통하여 시각적으로 제공해 줌으로써 자료 분석에 도움을 제공한다. 시각화 정보 생성기를 통해 제공되는 주요 자료로는 시스템 레벨의 시스템 흐름 그래프, 프로그램 참조 그래프가 있으며 프로그램 레벨에서는 호출 그래프, 구조도, 제어 흐름 그래프, 변수 참조 그래프 등이 있으며 그 외 소스 코드 브라우저, 맵화일 뷰어 등이 있다.

기존 연구의 문제점

COBOL2EJB 시스템을 사용하려면 코드 분석 정보를 이용하여 다양한 정보를 생성하므로 먼저 코볼 소스 코드를 분석하는 과정을 거치게 된다. 따라서 적용할 코볼 시스템에 대한 충분한 문법적 정의가 필요하다.

지금까지의 연구는 단순한 IBM 코볼만을 분석하는 수준에 있다. 그러나 대부분의 코볼 소스 파일들은 순수 코볼 소스보다 CICS 와 SQL 소스들이 더 많이 포함되어 있다. 그러므로 이들 언어들에 대한 정확한 문법 정의와 코볼과의 관계에 대한 이해가 필요하다.

따라서 본 논문에서는 CICS 와 SQL 에 적용된 문법을 정의하고 코볼과 연계하기 위해 코볼과 이들 사이의 관계를 분석하고 이를 토대로 새로운 코볼 문법을 정의 하고자 한다.

3. JJTREE

JJTREE 소개

JJTree 는 JavaCC 용 전처리기로서 JavaCC 소스상의 여러 위치에서 구문 분석 트리를 만드는 메소드를 삽입한다. JJTree 의 출력은 구문 분석기를 생성하기 위한 JavaCC 를 통한 실행이다.

기본적으로 JJTree 는 언어상의 각각의 비단말(non-terminal)에 대응하는 구문 분석 트리 노드를 만드는 코드를 생성한다. JJTree 의 기본 동작을 변경하여 일부 비단말들에 대해서는 트리 노드를 생성하지 않도록 하거나, 문법 생성 규칙의 확장 부분에 대한 노드를 생성하도록 할 수도 있다.

JJTree 는 모든 구문 분석 트리 노드가 구현하여야 하는 노드 인터페이스를 정의한다. 노드 인터페이스는 노드의 부모 노드 설정, 자식 노드의 추가 및 검색과 같은 동작을 위한 메소드를 제공한다.

JJTree 는 단일 모드 및 다중 모드의 두가지 모드로 동작한다. 단일 모드인 경우, 각각의 구문 분석 트리 노드는 SimpleNode 형의 명확한 타입이며, 다중 모드에서의 구문 분석 트리 노드의 타입은 노드의 이름으로부터 만들어진다. node 클래스에 대한 구현을 하지 않으면, JJTree 는 사용자를 위하여 SimpleNode 에 기반하여 예제 구현을 생성한다. 사용자는 이를 적절하게 수정하여 노드 클래스를 변경할 수 있다.

BNF 생성 규칙

BNF 생성 규칙은 JavaCC 에서 문법을 정의하기 위하여 사용하는 표준 생성 규칙이다. 각각의 BNF 생성 규칙은 비단말을 지정하는 좌변을 가지고 있다. BNF 생성규칙은 좌변의 비단말에 대한 BNF 확장을 우변에 정의한다. 비단말은 자바에서 선언되는 메소드와 같이 정확하게 쓰여져야 한다. 각각의 비단말은 생성되는 구문 분석기에서 메소드로 변환되기 때문에, 자바의 메소드와 동일한 스타일의 비단말작성은 이러한 변환 관계를 명확하게 한다. 비단말의 이름이 메소드의 이름이 되며, 비단말에 대해 선언된 파라메터 및 반환값은 구문 분석 트리의 아래 위로 값을 전달하는 수단이 된다. BNF 생성 규칙의 우변에 있는 비단말은 메소드 호출 형태로 변환되므로, 구문 분석 트리의 아래 위로 값을 전달하는 것은 메소드의 호출과 리턴과 동일한 방식으로 수행된다.

확장은 확장 유닛들의 시퀀스로 이루어진다.
확장 유닛은 지역적인 LOOKAHEAD SPECIFICATION (LOCAL_LOOKAHEAD), JAVA_BLOCK, 하나 이상의 확장선택들이 "(" 와 ")" 괄호로 묶인 집합, 하나 이상의 확장 선택들이 "["와 "]" 괄호로 묶인 집합, 정규 표현 또는 비단말인 경우와 같이, 6 가지 종류가 있다.

```
expansion_unit  ::=   local_lookahead
                  |   java_block
                  |   "(" expansion_choices ")" [ "+" | "*" | "?" ]
                  |   "[" expansion_choices "]"
                  |   [java_assignment_lhs                    "="]
                  |  regular_expression
```

```
| [java_assignment_lhs "="] java_identifier
  "(" java_expression_list ")"
```

확장 유닛이 지역적인 LOOKAHEAD SPECIFICATION(LOCAL_LOOKAHEAD)인 경우, 생성되는 구문 분석기에게 선택 지점에서 선택을 위한 방법을 지시한다.

확장이 "{" 와 "}"로 둘러싸인 자바 선언부와 코드인 경우, 파서 메소드라고도 한다. 파서 메소드는 비단말을 구문 분석하는 메소드의 적절한 위치에 생성되며, 구문 분석과정에서 이 지점을 성공적으로 통과할때마다 수행된다. JavaCC 가 자바 블럭 처리시에는 해당 부분에 대한 구문이나 의미를 검사하지 않는다. 따라서 JavaCC 가 처리한 파서 메소드 부분의 에러가 자바 컴파일러에 의해 발견될 수도 있다.

확장 유닛이 하나 이상의 확장 선택들이 "(" 와 ")" 괄호로 묶인 집합인 경우, 확장 유닛의 일반적인 파서는 중첩된 확정 선택들 중의 임의의 일반적 파서이다. 하나 이상의 확장 선택들이 "(" 와 ")" 괄호로 묶인 집합은 "+", "*"또는 "?"과 같은 메타문자를 접미사로 사용하여 추가적인 의미를 정의할 수 있다.

■"+": 하나 이상의 확장 선택들이 "(" 와 ")" 괄호로 묶인 집합의 일반적인 파서의 1 회 이상의 반복과 일치하는 경우를 나타낸다.

■"*": 하나 이상의 확장 선택들이 "(" 와 ")" 괄호로 묶인 집합의 일반적인 파서의 0 회 이상의 반복과 일치하는 경우를 나타낸다.

■"?": 하나 이상의 확장 선택들이 "(" 와 ")" 괄호로 묶인 집합의 일반적인 파서의 0 회(empty) 또는 1 회 반복과 일치하는 경우를 나타낸다. 확장 선택을 "[...]"내에 기술한 것과 동등한 의미이다.

확장 유닛이 하나 이상의 확장 선택들이 "[" 와 "]" 괄호로 묶인 집합인 경우는 하나 이상의 확장 유닛들이 "("와 ")" 괄호로 묶인 집합에 메타문자 "?"이 접미사로 붙은 것과 동일하다. 즉 하나 이상의 확장 유닛들이 "["와 "]" 괄호로 묶인 집합의 일반적 파서의 0 회 또는 1 회 반복과 일치하는 경우를 나타낸다.

마지막으로 확장 유닛이 비단말인 경우, 비단말의 이름을 메소드의 이름으로 가지는 메소드를 호출하는 형태이다. 비단말의 성공적인 파서가 되면 해당 메소드가 주어진 파라메터에 대해 수행된 후 비단말이 void 가 아니라면 메소드의 리턴값이 리턴된다.

BNF 생성 규칙을 이용한 코볼 문법 적용 예제

이 절에서는 코볼의 명령어 가운데 ADD 명령어가 어떻게 JavaCC 문법으로 정의되는지 간단히 설명한다. 먼저 ADD 명령문의 일반적인 형식은 다음과 같다.

```
ADD { identifier - 1 | literal -1 } { identifier -2 | literal -2 } ... To identifier - m[ROUNDED]
[.identifier - n[ROUNDED]] ... [:ONSIZE ERROR imperative -statement]
```

위 수식에서 알 수 있듯이 "ADD"와 "TO" 사이에는 identifier 또는 literal 이 여러개 올수 있다. 그리고 "TO" 절에는 identifier 들이 여러개 올 수 있고 각각은 반올림을 하기위해 "ROUNDED"라는 키워드가 선택적으로 올 수 있다. 마지막으로 ADD 명령어수행도중 오류가 발생하면 처리할 수 있는 "ON SIZE ERROR"절이 선택적으로 올 수 있다. 이것을 BNF 로 표기하면 다음과 같다.

```
add_statement ::= "ADD" { ( identifier | literal ) }+ "TO"
  { identifier [ "ROUND" ] }+ [ [ "ON" ] "SIZE"
"ERROR" statement_list ] [ "NOT" [ "ON" ] "SIZE"
"ERROR" statement_list ] [ "END-ADD" ]
```

여기서 굵은 글씨체는 키워드 들이다. 위의 BNF 식을 JavaCC 문법으로 표기하면 다음과 같다.

```
void add_statement() : {}
{
  "ADD" ( identifier() | literal() )+
   "TO"  ( identifier ["ROUNDED"] )+
   [ [ "ON" ]  "SIZE" "ERROR" statementList() ]
   [ LOOKAHEAD(2) "NOT" [ "ON" ] "SIZE"
  "ERROR" statementList() ] [ LOOKAHEAD(2)
  "END_ADD" ]
  }
```

4. CICS와 SQL 생성 규칙 정의 및 구현

일반적으로 기존 코볼을 사용하는 시스템에서는 단순한 코볼만을 사용하지는 않는다. IBM 의 경우 화면 출력등을 제어하기 위해 멥(Map)파일과 CICS 를 제공하고 있다. 그리고 데이터베이스를 엑세스하기 위해 SQL 을 지원한다. 따라서 이러한 SQL 명령어를 코볼에서 사용하기 위해서는 별도의 문법 정의가 요구된다.

맵파일을 위한 생성 규칙 정의

IBM 의 맵파일은 코볼의 DATA DIVISION 의 WORKING-STORAGE SECTION 에 삽입된다. 맵파일의 주요 기능은 화면 디스플레이를 제어하는 것이다. 맵파일의 코볼에서의 사용은 각각의 자료들이 화면에서 어느 위치에 출력될 것인지 글자체는 어떤 것으로 할 것인지를 자세히 설명한다.

```
void workingStorageSectionClause() : {}
{
<TOK_WORKING_STORAGE><TOK_SECTION>
<TOK_PERIOD>   (  dataItemDescriptionEntry()
|<TOK_PRINT><TOK_NOGEN>              (cicsBmsMacro())+
<TOK_END>
  )*
}
```

맵파일은 "PRINT-NOGEN" 이라는 키워드로 시작하여 "END"라는 키워드로 끝나며 이 블록 속에 맵정보들이 들어 있다. 따라서 맵 파일을 코볼에서 적용하려면 문법생성 규칙에서 앞에서와 같이 수정 하여야 한다.

기존의 dataItem 들만 기술하도록 되어있는 부분을

맵파일을 사용할 수 있도록 확장하였다.

cicsBmsMacro()는 CICS 의 맵파일을 위한 명령어들에 대한 문법 생성규칙을 기술하고 있다.

CICS 명령어 처리를 위한 생성 규칙 정의

CICS 명령어들은 PROCEDURE DIVISION 에 코볼 명령어의 또다른 하나의 명령어 형태로 기술된다. 이들 명령어는 맵파일의 구성요소에 값을 저장, 삭제 및 초기화하는 등의 기능을 갖고 있다. 코볼에서는 CICS 명령어들과 코볼 명령어들을 구분하기 위해 CICS 명령어들은 내장형으로 사용하고 있다.

CICS 명령어들은 코볼 PROCEDURE DIVISION 에 명령어의 하나로 사용되므로 코볼 명령어와 구분하기 위해 "EXEC"와 "END-EXEC" 키워드 사이에 기술한다. CICS 명령어들을 기존의 코볼에 삽입하여 사용하려면 다음과 같이 문법을 수정하여 CICS 명령어들을 삽입하면 된다.

```
void statement() : {}
{
  ( acceptStatement()   | addStatement()
  . . . .
  | stopStatement()   | cicsStatement()
  )
}
. . . .
void cicsStatement() : {}
{
<TOK_EXEC> <TOK_CICS>
cicsCommand()
  <TOK_END_EXEC>
}
```

SQL 명령어 처리를 위한 생성 규칙 정의

SQL 명령어는 관계형 데이터베이스를 엑세스하고 처리하는 명령어들이다. 이들 명령어 가운데 사용자들이 주로 사용하는 명령어들은 데이터베이스의 테이블 데이터를 질의하는 명령어들이다. 그러므로 이들 명령어들은 명령어 자체만으로는 애플리케이션에 적합하지 않다. 따라서 이들 명령어들은 어플리케이션 프로그램과 결합될때 유용하다. 이 절에서는 SQL 명령어들이 COBOL 에 내장되어 사용되는 예와 이것을 처리하기 위한 문법 생성규칙을 정의한다.

SQL 문을 지원하기 위해서는 문법 생성 규칙을 다음과 같이 수정해야 한다.

```
void workingStorageSectionClause() : {}
{
<TOK_WORKING_STORAGE><TOK_SECTION>
<TOK_PERIOD>
( dataItemDescriptionEntry()
| "EXEC SQL" sqlStatement() "END-EXEC"
<TOK_PERIOD>
)*
}
void linkageStorageSectionClause() : {}
{
<TOK_LINKAGE><TOK_SECTION><TOK_PERIOD>
( dataItemDescriptionEntry()
| "EXEC SQL" sqlStatement() "END-EXEC"
<TOK_PERIOD> )*
}
void statement() : {}
{
( acceptStatement()
. . . .
| sqlStatement() )
}
void sqlStatement() : {}
{
  <TOK_EXEC> <TOK_SQL>
sqlCommand()
  <TOK_END_EXEC>
}
```

5. 결론 및 향후 연구 과제

본 논문에서는 프로그램 시각화 정보 생성기에 적용하기 위해 기존 코볼을 JavaCC 를 이용하여 문법생성규칙을 재구성하였다. 이전의 연구에서는 단순히 순수 코볼만 적용이 가능하였다. 그러나 일반적으로 여러 기업체 및 은행권등에서는 출력 화면처리 하기 위해 CICS 및 SQL 을 코볼 소스에 내장하고 있다. 따라서 이들 처리를 위한 지원이 없이는 COBOL2EJB 연계 도구는 그 실효성이 떨어진다.

따라서 본 논문에서는 COBOL2EJB 연계 도구의 효율적인 사용성을 위해서 IBM 코볼 소스에 내장된 CICS 와 SQL 을 처리할 수 있도록 JavaCC 를 이용하여 문법 생성규칙을 확장하였다. 이러한 문법 생성 규칙의 확장은 구 시스템의 현대화에 좀더 많은 발전을 가질수 있도록 할 것이다.

본 연구는 기존 시스템으로 코볼이 가장 많이 사용되고 있어 코볼을 중심으로 연구가 진행되었다. 그러나 우리나라의 경우 기존 시스템으로 PL/I 으로 구현된 사례도 많다. 따라서 이러한 언어의 현대화에도 관심을 가지고 기능확장 측면에서 향후 과제로 고려할 수 있다.

참고문헌

[1] 정효택, 김동관 "COBOL 시스템을 위한 EJB 래퍼 컴포넌트 지원 도구 개발," 제 18 회 한국정보처리학회 추계학술발표대회 논문집 제 9 권 제 2 호, 2002.

[2] "Creating Components from Legacy Applications", *CBDi Forum Journal*, Dec. 1998.

[3] H.Huang, "Business rule extraction from legacy code", *Proc. 20Th Computer software and Applications Conference*, pp.922-pp.926, 1996.

[4] H. M. Sneed, "Extracting business logic from existing COBOL programs as a basis for redevelopment", *Proc. 9th, Program Comprehension(IWPC 2001)*, 167-175. 2001.

[5] H. M. Sneed, "A Case Study in Software Wrapping", *Proc. IEEE Software Maintenance*, pp86-92, 1998.