COBOL 레거시 시스템의 재사용을 위한 효율적인 트리의 설계 및 구현

전하용, 최원호, 정민수
경남대학교 컴퓨터공학과
e-mail : hayongj@hotmail.com

Design and Implementation of Efficient Tree for Reuse of COBOL Legacy System

Ha-Yong Jeon, Won-Ho Choi, Min-Soo Jung*
Dept of Computer Engineering, Kyungnam University

요 약

컴퓨팅 환경의 급격한 변화에 따라 기존의 레거시 시스템을 객체지향이나 웹과 같은 새로운 환경에서도 운용할 수 있도록 시스템을 현대화하고자 하는 연구가 활발히 진행되고 있다. 특히 소프트웨어 재사용 기술과 함께 컴포넌트 기술이 개발되고 보급됨에 따라 레거시 시스템을 컴포넌트로 변환하거나 연계하는 방법들이 연구되고 있다. 본 논문은 IBM 메인 프레임에서 운용되고 있는 CICS 및 SQL 코볼 시스템을 EJB 래퍼 컴포넌트로 연계하는 일련의 프로세스를 지원하는 연계 도구에 대한 내용으로서 코볼 소스코드를 어휘분석과 구문분석을 통하여 새로운 형태의 AST 트리를 구성한다.

1. 서론

기존에 운영되고 있는 레거시 시스템(Legacy System)을 객체지향이나 웹과 같은 새로운 컴퓨팅 기술 및 환경에 적용시켜 재사용하기 위해서는 객체화라는 진화단계가 필요하다. 시스템의 진화는 데이터베이스의 한 필드를 갱신하는 것부터 시스템 전체를 재구현하는 것까지를 이르는 매우 광범위하게 사용된다. 레거시 시스템을 현대화하기 위해서는 레거시 시스템의 비즈니스 로직 일부를 컴포넌트화하는 것이 일반적이다. 이 방법으로는 레거시 전이, 컴포넌트 인터페이스로의 래핑, 레거시 재구조화, 레거시 향상, 컴포넌트 인터페이스로 통합 등이 있다. 본 논문에서는 이러한 방법들에서 공통적으로 필요한 작업인 주요 모듈의 비즈니스 로직(Business Logic)을 추출하여 컴포넌트화하기 위한 방법을 제안하고, 이를 CICS(Customer Information Control System) 및 SQL COBOL 프로그램에 적용하여 EJB(Enterprise Java Beans) Beans 생성에 필요한 COBOL 소스코드 파일을 객체지향의 기법에 맞게 알맞은 AST트리로 구성하는 방법을 제안한다.

2. 관련 연구

2.1 레거시 시스템의 현대화 방법

레거시 시스템의 현대화 방법으로는 크게 재개발, 래핑(Wrapping), 변환 등으로 나눠 볼 수 있다. 재개발 방법은 기존 시스템에 대한 이용이 가장 적으며 개발비용과 기간이 길다. 또한 개발 후 기존 시스템을 대체할 경우 신뢰성과 안전성에 대한 보장이 약하다. 이에 반해 래핑 방법은 기존 시스템은 그대로 두고 새로운 시스템 환경에서 사용 가능하도록 중간에 래퍼(Wrapper)를 둠으로써 신뢰성과 안정성이 좋지만, 시스템의 유연성과 확장성에서 많은 단점이 있다. 변환 방법은 레거시 시스템을 새로운 시스템으로 1:1로 매핑(Mapping)해 주는 방법으로 가장 이상적이지만 시스템들간의 환경, 소프트웨어 아키텍처, 프로그램의 특성이 다르기 때문에 구현이 힘들다.

2.2 COBOL2EJB 시스템의 이해

본 논문은 현재 사용되는 CICS 및 SQL COBOL 프로그램과 같은 레거시 시스템에 새로운 환경으로 전환하는 방법론 및 이를 지원하는 도구의 개발을 목표로 작성되고 있는 것이 COBOL2EJB 시스템이다. 기존의 COBOL 시스템을 웹 등의 새로운 컴퓨팅 환경으로 현대화하기 위한 방법으로는 재사용 가능한 비즈니스 로직을 컴포넌트로 변환하거나 래핑하는 방법이 주로 사용되고 있다.

2.2.1 COBOL2EJB 시스템 구성

본 논문에서 소프트웨어 재사용을 위해 연구 개발되고 있는 COBOL2EJB 시스템의 구성도는 아래의 그림1과 같다. COBOL2EJB 시스템은 COBOL 코드 분석기, EJB 래퍼 생성기, 시각화 정보 생성기 및 컴포넌트 생성기로 구성되어 있다.

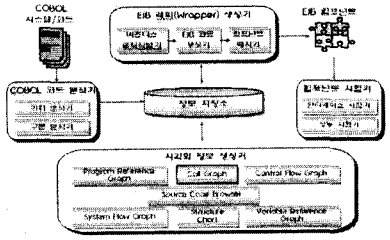

그림 1. COBOL2EJB 시스템의 구성도

COBOL 코드 분석기는 IBM CICS 및 SQL COBOL 소스 코드를 어휘 분석과 구문 분석을 통하

여 AST 및 각종 테이블을 생성함으로써 시각화 정보 생성기 및 EJB 래퍼 생성기에서 필요한 정보를 추출하는 기능을 담당한다. 시각화 정보 생성기는 코드 분석기를 통하여 얻은 정보를 이용하여 시스템의 구성을 시각적으로 보여줌으로써 시스템에 대한 이해를 증진시킨다. EJB 래퍼 생성기는 소스 코드를 분석한 정보를 이용하여 재사용 가능한 비즈니스 로직을 식별하는 비즈니스 로직 식별기, 식별된 로직을 EJB 코드로 생성하는 EJB 코드 생성기 및 생성된 컴포넌트를 웹 어플리케이션 서버에 배치하는 컴포넌트 배치기로 구성되어 있다. 그리고 컴포넌트 시험기는 서버에서 운용되는 EJB 컴포넌트에 대한 인터페이스 테스트와 성능 테스트를 수행하는 모듈이다.

3. COBOL 코드분석기를 위한 자료구조 설계

본 장에서는 레거시 시스템의 재사용을 위한 COBOL2EJB 시스템에서 CICS 및 SQL COBOL 프로그램을 자바빈즈 컴포넌트로 변환하여 사용하기 위해서 기본적인 프로그램의 어휘분석 및 구문분석 뿐만 아니라 비즈니스 로직과 컴포넌트를 추출하기 위한 각종 심볼 테이블 등의 모든 정보들을 생성한다. 아래의 그림 2는 COBOL 코드분석기의 구성도이다.

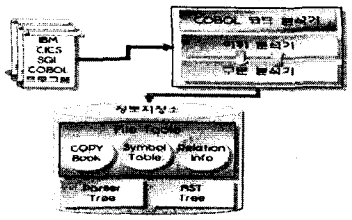

그림 2. COBOL 코드 분석기

코드분석기는 먼저 전처리기를 통해서 COBOL 프로그램에 관련된 코드들을 어휘분석 한다. 이 과정에서는 COBOL 프로그램에 사용되는 자료정의에 관련된 COPY구문에 따라 해당되는 파일들을 COBOL프로그램내에 첨부시킨다. COBOL 프로그램을 어휘분석기와 구문분석기 거쳐 시각화 정보기와 EJB 래퍼 생성기에서 사용할 형태로 정보로 객체로 변환하여 정보 저장소에 저장한다. 코드분석기를 거쳐서 나오는 정보는 카피북(Copy Book) 정보와 파서 트리(Parser Tree), 심볼 테이블(Symbol Table)과 관계정보(Relation Information)를 저장하는 파일 테이블(File Table)을 작성하고, 끝으로 시각화 정보 생성기 내의 호출 그래프, 제어 흐름 그래프, 구조도, 소스코드 브라우저에서 사용될 AST 트리를 작성한다.

3.1 심볼 테이블

코드 분석기에서는 COBOL 프로그램을 읽어들여 어휘분석을 한다. 이때 생성되는 각 토큰들을 이용해서 시각화 정보기에서 사용할 수 있는 심볼들을 생성한다. COBOL 프로그램에서 사용되는 형태대로 심볼 객체를 생성하여 심볼 테이블을 구축한다. 생성된 심볼 테이블은 파일 테이블에 저장되고, 이 파일 테이블이 COBOL 프로그램에 관한 모든 정보를 가지게 되고 정보 저장소에 저장된다. 심볼 테이블의 구성은 일반적인 COBOL 프로그램에서 사용되는 형태에 따라서 상수와 변수에 대한 정보를 기록하는 각각의 객체들이 생성되고, COBOL 프로그램의 구조에 따라 프로그램과 패러그래프 심볼객체가 생성된다. 어휘분석 과정에서 생성되는 심볼들은 기본적으로 토큰 이름과 COBOL 프로그램 상의 라인번호에 관련된 정보들을 갖는다. 다음의 그림 3은 코드 분석기에서 사용되는 심볼들의 계층구조를 나타낸다.

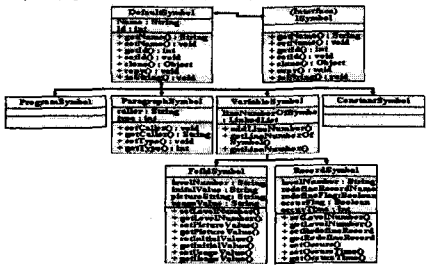

그림 3. 심볼들간의 계층구조

3.2 릴레이션 정보

COBOL 코드분석기를 거쳐서 나오는 여러 가지 정보들 중에서 심볼 테이블과 릴레이션 정보는 레거시 시스템을 컴포넌트로 재사용하기 위해서 반드시 필요한 정보들이다. 릴레이션 정보는 패러그래프 내의 문장들간의 흐름 정보를 보여주고, 패러그래프 내부에서 사용된 여러 가지 변수들과 상수들의 사용여부를 관계를 파악할 때 이용이 된다. 그리고 릴레이션 정보를 구성하는 기본 자료는 이미 생성된 심볼들을 기반으로 작성이 되고, 패러그래프들간의 관계와 패러그래프와 변수들간의 관계, 변수들간의 관계를 파악하기 위해서 만들어진다. 릴레이션 정보는 COBOL 시각화정보기에서 COBOL 프로그램내에서 사용된 핵심 변수 식별과, 프로그램 패턴 식별에서 사용된다. 이를 기반으로 해서 비즈니스 로직을 추출하여 자바빈즈를 이용한 컴포넌트로 재사용된다.

3.3 파일 테이블

COBOL 코드분석기에서 생성되는 정보는 크게 심볼테이블과 릴레이션 정보들이다. 이 정보는 COBOL 프로그램 하나당 각각 한개씩 생성이 되고 파일 테이블이라는 클래스파일에 저장된다. 그리고 이 파일 테이블이 실제의 정보 저장소에 저장되는 내용이다. 이 정보를 기반으로 해서 COBOL 레거시 시스템의 재사용을 위한 시각화 정보를 만든다.

3.4 파서 트리와 AST 트리

COBOL 코드분석기에서 사용되는 파서(Parser)는 기존의 일반적인 프로그램에서 이용되는 어휘분석기(Lex)나 구문분석기(Yacc)이 아니라 JavaCC라고 하는 어휘분석기를 이용한다. JavaCC를 사용하는 이유는 객체지향의 개념이 없는 COBOL 프로그램을 컴포넌트라고 하는 객체지향적 언어로 재사용하기 위해서이다. 그리고 COBOL 프로그램 하나를 어휘분석한 후에는 심볼 테이블과 릴레이션 정보, 카피 북에 관한 정보를 저장하는 파일 테이블 객체가 만들어지고, 또한 어휘분석에 결과인 파서 트리가 작성이 된다. 코드분석기를 통해서 생성한 여러 가지 정보들을 이용해서 시각화 정보기가 비즈니스 로직을 추출하다가 필요에 따라서 해당 패러그래프간의 관계나 핵심 변수

의 사용여부, 프로그램 내부에서 자주 사용되는 COBOL 프로그램의 명령어 등을 쉽게 찾고, 프로그램 전체의 구조를 쉽게 파악하기 위해서 구문분석 과정을 통해 AST트리를 생성한다. 이러한 AST트리는 소스코드 브라우저와 함께 연동을 해서 필요한 부분만을 추출할 수 있도록 만들어진다.

4. COBOL2EJB를 위한 AST 트리 재설계

COBOL 코드분석기는 'JJTree'와 'JavaCC'를 이용해서 어휘분석과 구문분석을 한다. 이 과정을 통해서 심볼 테이블과 릴레이션 정보, 파일 테이블을 구성하고 파서 트리와 AST트리를 작성한다. 그러나, 위의 도구들을 이용해서 AST 트리를 작성하게 되면 트리를 구성하는 노드와 노드들 간의 추적관계에 문제가 발생한다. 최상위 노드에서부터 하향식 방법으로 시각화 정보기에서 요구하는 노드를 검색할 수 있지만 상향식 방법으로는 필요한 노드를 찾는 것이 불가능하다. 이러한 문제점을 해결하기 위해서 AST 트리를 재설계하였다.

4.1 AST 트리의 노드 구성

기존에 사용되는 AST 트리의 각 노드들은 'JavaCC'에서 만들어진 노드들을 그대로 이용해서 트리를 구성하였다. 따라서 각 노드들이 가질 수 있는 관계정보는 부모노드에 대한 참조 값과 그 자체에 연결되는 자식노드들에 대한 배열의 자료구조 갖게된다. 다음의 그림 4은 AST 트리의 노드를 재설계하기 전의 노드의 자료구조 형태이다.

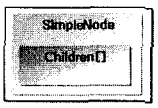

그림 4. 이전 AST 노드의 구조

이러한 형태의 노드들은 부모노드에 대한 포인터만을 갖고 있고, 자식노드에 접근하는 방법은 배열의 인덱스 값을 이용해서 접근할 수 있다. 그리고 형제들 간의 이동 경로는 존재하지 않으므로 AST 트리에서 특정한 노드를 찾았을 때는 항상 노드의 시작부분부터 순차적으로 검색해 내려와야 한다. 본 논문에서는 이러한 문제를 해결하고 더 빠른 시간에 필요한 노드들을 찾을 수 있도록 AST트리의 노드들을 재설계한다. 재설계된 노드들을 구성하기 위해서는 각 노드에 부모 노드, 왼쪽 형제노드, 오른쪽 형제노드, 자식 노드에 관한 포인터를 따로 추가하여 노드를 구성하고, 실제 COBOL 프로그램 상의 라인번호를 입력하여 소스 코드와의 연관성을 더욱 좋게 만들 수 가 있다. 다음 그림 5가 새로 설계한 AST트리 구조이다.

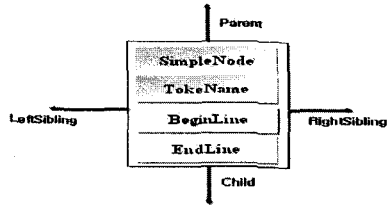

그림 5. 새로 설계한 AST 노드의 구조

4.2 AST 트리의 노드 관계

기존의 노드들 간의 문제를 해결한 방법이 다음에 나오는 그림 6이다. 각 노드에는 부모노드에 대한 참조 값과 노드의 양쪽의 형제노드에 대한 참조 값, 그리고 자식 노드에 대한 참조 값을 연결한다. 이렇게 구성함으로서 한 노드에서 가져야할 모든 자식노드에 대한 메모리를 줄일 수 있게 되고 또한 다른 노드들로 쉽게 이동을 할 수 있게 된다. 그리고 하향식 검색방법과 상향식 검색방법도 가능하게 되었다. 따라서 원하는 노드에 대한 검색이 더 쉽고, 빨리 찾을 수 있고, 프로그램 내부의 핵심이 되는 노드들만 따로 추출하여 사용할 수 있게 되었다.

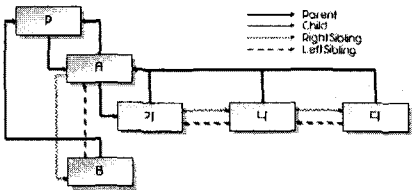

그림 6. 노드들 간의 연결관계

5. 구현 및 테스트

COBOL2EJB 시스템에서 사용되는 COBOL 코드분석기는 COBOL 프로그램을 어휘분석하기 위해 "IBM COBOL V1.03"을 기반으로 하여 설계되었다.

5.1 파서트리와 AST 트리의 노드 비교

```
if(node != null) {
    if(buildNodeCheck(node) == true) {
        buildNode = new BuildNode(node, node.toString(), node.getBeginLine(), node.getEndLine());
        parent = buildNode;
        ancestor = buildNode;
        if(node.children != null) {
            for(int i=0; i<node.children.length; i++) {
                SimpleNode son = (SimpleNode)node.children[i];
                child = buildTree(son, parent);
                if((child != null) && (childFlag == false)) {
                    parent.setChildNode(child);
                    child.setParentNode(parent);
                    childFlag = true;
                    leftSibling = child;
                    leftFlag = true;
                } else if((child != null) && (childFlag == true)) {
                    child.setParentNode(parent);
                    if(leftFlag == true) {
                        child.setLeftSiblingNode(leftSibling);
                        leftSibling.setRightSiblingNode(child);
                        leftSibling = child;
                    } else {
                        leftSibling = child;
                        leftFlag = true;
                    }
                }
            }
        } else {
            return buildNode;
        }
    }
```

그림 7. AST 트리의 노드 구성 코드

본 개발결과의 가장 핵심 사항으로 COBOL 프로그램을 코드분석기로 분석한 결과인 파서 트리와 AST 트리가 만들어진다. 이 두가지 트리는 코드분석기에서 만든 심볼 테이블과 릴레이션 정보, 카피 북에 관련된 모든 정보들과 별도로 동작하도록 구성이 되어 있다. 그리고, 각 정보들을 원할 때마다 따로 추출할 수 있도록 파서 트리와 AST 트리를 따로 작성한다. COBOL 코드분석기는 두가지 트리를 구성을 하는데 파서 트리인 경우는 코드분석을 하기 위해서 사용한 "JavaCC" 도구로 간단하게 생성 할 수 있다. 하지만 생성된 트리는 소스 코드 브라우저와 시스템 흐름 그

표 1. 파서트리와 AST 트리 비교

구분	파서트리	AST트리
노드수	906	421 (46% 정도 감소)
결과	paragraph paragraphName userDefinedWord sentence statementList statement addStatement addToGivingStatement literal numeric identifier qualifiedDataName dataName alphabeticUserDefinedWord	[paragraph : ADD-RTN], (P : procedureDivision, C : addStatement, LS : paragraph, RS : paragraph)(begin : 48, end: 49] [addStatement : ADD], (P : paragraph, C : addToGivingStatement, LS : null, RS : addStatement)(begin : 49, end: 49] [addToGivingStatement : TO], (P : addStatement, C : literal, LS : null, RS : null)(begin : 49, end: 49] [literal : 1], (P : addToGivingStatement, C : null, LS : null, RS : identifier)(begin : 49, end: 49] [identifier : I], (P : addToGivingStatement, C : dataName, LS : literal, RS : null)(begin : 49, end: 49] [dataName : I], (P : identifier, C : null, LS : null, RS : null)(begin : 49, end: 49]

래프, 프로그램 호출 그래프 및 패러그래프 흐름 그래프에서 이용 할 수 없다. 또한 특정 심볼을 찾고자할 때 트리의 시작부분부터 순차적으로 검색을 해야한다. 따라서, 이러한 문제점을 고려하여 새로운 AST 트리를 설계하고 적용시켰다. 그 결과 트리를 구성하는 노드의 수에서 큰 변화를 볼 수 있었고, 또한 검색 시간도 감소되었다. 그리고, COBOL 생성규칙의 원하는 부분만을 선택하여 AST 트리로 구성을 할 수 있게되어, COBOL2EJB 시스템을 사용하는 이용자의 목적에 맞게 원하는 형태로 AST 트리를 구성할 수 있게 되었다. 그림8은 COBOL 코드분석기에서 사용되는 AST 트리를 구성하는 부분을 구현한 것이다.

작성된 AST 트리를 파서 트리와 간단한 코볼 예제 프로그램을 비교하면 표1의 실행결과와 같은 트리 노드의 구조를 쉽게 파악할 수 있다. 그리고, 각 노드들이 갖는 정보들을 쉽게 비교할 수 있고 사용자가 원하는 형태의 노드들만을 출력할 수 있다. 파서 트리와 AST 트리간의 비교를 위해서 간단한 COBOL 프로그램인 "ex00pgm"을 수행시킨 결과가 표1과 같다. 이 프로그램은 120라인으로 구성이 되고 대부분의 COBOL 명령어들이 사용되었다.

5.2 각 노드들 간의 추적방법 비교

파서 트리를 이용해서 각 노드들을 추적할 수 있는 방법은 단 한가지이다. 트리의 최상위 시작 노드를 찾은 다음, 아래로 하나씩 내려오면서 모든 노드들을 비교해한다. 이런 방법으로 특정 노드를 찾은 후 이 노드와 연관된 노드를 찾기 위해서는 다시 최상위 노드로부터 하나씩 순차적으로 검색을 해야 된다. 이러한 방법으로 필요한 노드를 찾는 것은 시간적으로 엄청난 손실을 볼수 밖에 없다. 그러나 재설계된 AST 트리를 이용하면 각 노드들마다 연결된 부모, 왼쪽 형제, 오른쪽 형제, 자식 노드에 대한 참조 값이 존재하기 때문에 쉽게 노드들간의 이동할 수 있다. 그리고 최상위의 노드부터 차례로 검색해 와야 한다는 비중을 줄일 수 있다.

5.3 트리 구성의 시간 비교

앞의 표1에서 나온 결과에서 AST 트리는 파서 트리가 생성하는 노드들을 거의 50%까지 줄여서 생성할 수 있다. 따라서 각 트리를 작성하는데 걸리는 시간도 파서 트리를 구성하는데 걸리는 절반의 시간만으로 가능하다.

6. 결 론

COBOL2EJB 시스템은 COBOL 레거시 시스템을 재사용하여 객체지향의 컴포넌트 방식으로 재구성하는 시스템이다. 개발비를 새로 투자하여 새로운 시스템을 개발사는 것이 아니라 기존에 사용하였던 시스템을 COBOL2EJB라는 방식으로 전환하여 효율적으로 재사용하여 개발비뿐만 아니라 시간의 절약을 가져왔다. 이 시스템에서 가장 중요한 부분이 바로 COBOL 코드분석기이다. COBOL 프로그램을 COBOL2EJB 시스템에서 사용할 수 있는 형태로 가공해주는 것이 바로 코드분석기 내부의 구문분석기가 하는 역할이다. 그리고 현재에 사용되는 어휘분석과 구문분석 방법을 COBOL2EJB 시스템에 맞게 다시 변형하여 개발한 것이 본 논문의 주된 내용이다. 본 논문의 구문분석기는 기존에 파서 트리가 갖는 문제점들을 쉽게 해결하였고 또한 효율성을 높였다.

향후 과제로는 시스템의 비즈니스 로직을 확장하거나 시스템의 구조를 변경하는 등 시스템을 재구조화 할 수 있는 기능을 추가하거나, 대상 시스템을 CICS COBOL에 한정하지 않고 다양한 COBOL 시스템으로 확장하거나 웹 프로그램을 컴포넌트로 변환하는 도구와의 통합을 고려할 수 있다.

참고문헌

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

[2] D. C. C. Poo, Events in use cases as a basis for identifying and specifying classes and business rules, Proc. Technology of Object-Oriented Languages, 1999, 204-213.

[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.

[6] H.M.Sneed, "Extracting business rules from source code", Proc.4th Program Comprehension, pp.240-pp.247, 1996

[7] J. K. Joiner, Data-Centered Program Understanding, Proc. Software Maintenance, 1994, 272-281.

[8] J.Q. Ning, "Recovering reusable components from legacy systems by program segmentation", Proc. Reverse Engineering, pp.64-pp.72, 1993.

[9] K. Kawwabe, "Variable Classification Technique for Software Maintenance and Application to The Year 2000 Problem," Proc. 6th Software Engineering Conf.(ASPEC'99), pp.500-506. 1999.