제20회 한국정보처리학회 추계학술발표대회 논문집 제10권 제2호 (2003. 11)

레거시 시스템을 위한 EJB 레퍼 컴포넌트 지원 코볼 코드 분석기 설계 및 구현

주상현*, 임동기*, 정민수*
*경남대학교 컴퓨터공학과
e-mail : jshstart@naver.com

Design and Implementation of EJB Wrapper Component Supporting Cobol Code Analyzer for Legacy System.

Sang-Hyun Joo*, Dong-Ki Im*, Min-Soo Jung*
*Dept. of Computer Engineering, Kyungnam University

요 약

최근에 기존 레거시 시스템의 새로운 컴퓨팅 환경으로의 전환을 위해, 재개발, 변환, 래핑 등의 방법을 사용하고 있으며, 이러한 경향은 레거시 시스템의 현대화를 촉진시키고 있다. 레거시 시스템을 컴포넌트화 함으로써 기존의 시스템을 재사용하고 개발기간의 단축 및 시스템의 유지·보수가 용이하도록 한다. 본 논문에서는 레거시 시스템에서 사용되는 시스템의 핵심 변수와 패턴 식별에 사용되는 정보 저장소를 생성하여 레거시 시스템을 컴포넌트 기반 시스템으로 변환하는데 용이하도록 하였다. 정보 저장소의 보다 효율적인 구성을 위해서 코볼 코드의 릴레이션간의 릴레이션 중복을 방지하는 최적화된 코볼 코드 분석기를 설계하고 구현함으로써 보다 효율적인 정보 저장소를 구성할 수 있도록 하였다.

1.서론

전 세계적으로 소프트웨어를 개발하는 방식이 소프트웨어를 통째로 개발하던 방식에서 컴포넌트 기반 개발 방식으로 빠르게 변화하고 있다. 이는 하드웨어 제품의 부품별 개발방식과 같이 필요한 소프트웨어 컴포넌트만을 모아서 접목시키는 방식으로 인력과 시간을 절약해 개발비용을 감소 시킨다. 그러나 기존의 레거시 시스템을 컴포턴트 기반으로 전환할 경우 기존의 시스템의 안정성을 헤치지 않는 측면에서 접근해서 현대화된 컴포넌트 기술 개발로 진행해 가야 한다. 이 과정에 레거시 시스템의 핵심 변수와 패턴 식별이 요구되는데 이 역할을 하는 것이 바로 코드 분석기이다. 본 논문에서는 기존의 코볼 언어용 코드 분석기 보다 더 향상된 성능과 효율성을 위한 방법들을 제시한다.

2. 관련연구

2.1 레거시 시스템

정보기술에서 레거시 프로그램과 데이터는 프로그래밍 언어, 플랫폼 그리고 기술 등이, 과거로부터 물려 내려온 것들을 의미한다. 일반적으로 레거시 시스템들은 그것들을 개발했던 플랫폼에서만 운영될 수 있었다. 현재, 많은 회사들이 자신들의 레거시 프로그램들을 개방형이나, 표준 프로그래밍 인터페이스를 따르는 새로운 프로그래밍 언어와 운영체계에 맞게 변환하고 있다. 대체로 새로운 개발환경은 레거시 시스템과 데이터를 계속 지원 해야할 필요가 있다.

2.2 COBOL2EJB

COBOL2EJB(COBOL To Enterprise Java Beans)

연계 도구는 현재 IBM 메인 프레임에서 운용되고 있는 CICS(Customer Information Control System) 코볼 시스템을 EJB(Enterprise Java Beans) 래퍼 컴포넌트로 연계하는 일련의 프로세스를 지원하는 연계 도구로서, 코볼 코드 분석기, 시각화 정보 생성기, EJB 래퍼 컴포넌트 생성기, 컴포넌트 시험기로 구성되어 있다

COBOL2EJB 연계 도구의 동작 순서는, 먼저 코볼 코드로 작성된 시스템을 입력받아 코드를 분석하고, 다음으로 시각화 정보 생성기에서는 여러 그래프를 활용하여 시스템에 대한 이해를 높이는데 제공되는 그래프를 생성한다. 이 그래프들은 시스템 흐름 그래프, 프로그램 참조 그래프, 호출그래프, 제어 흐름 그래프, 변수 참조 그래프와 소스 코드 브라우저, 맵화일 뷰어가 있다. 그리고 EJB 래퍼 컴포넌트를 생성, 실행하기 위하여 컴포넌트화 할 대상 비즈니스 로직을 식별하고, EJB 코드를 생성한 다음 이를 웹 어플리케이션 서버에 배치한다. 생성된 컴포넌트는 컴포넌트 시험기를 통하여 인터페이스와 성능을 테스트한다.

3. 코볼 코드 분석기 설계

코볼 코드 분석기에서는 소스코드, 맵화일, 카피북 등 여러 요소들을 입력으로 받아 어휘 분석, 구문 분석 과정을 거쳐 AST(Abstract Syntax Tree), 심볼 테이블 등을 생성하고 정보 저장소에 저장한다. 저장된 는 정보는 그림-1 의 코볼코드 분석기 구성도와 같이 저장된다. 이렇게 생성된 정보는 시각화 정보 생성기 및 EJB 래퍼 컴포넌트 생성기 등 에서 사용하게 된다.

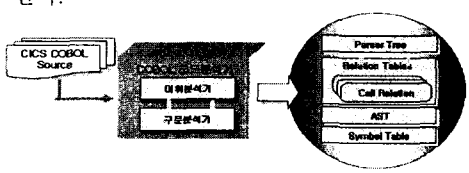

그림-1. COBOL 코드 분석기 구성도

코드 분석기가 이러한 정보를 얻기 위해서는 파서를 생성해야 하는데 전체과정이 그림-2 에서 나타나고 있다.

3.1. 파서 개발 과정

우리는 파서를 생성하기 위해서 토큰과 생성규칙을 가진" .jjt " 파일을 입력으로 받아 자바 컴파일러 컴파일러 툴을 사용했다.

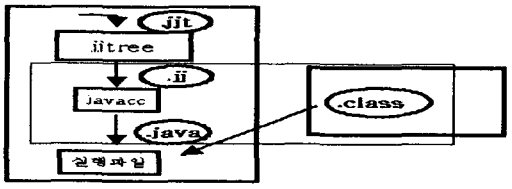

그림-2. 파서 개발 과정

3.2 javacc 입력파일의 구성요소

코볼 코드 분석기를 생성하기 위해서는 파서를 먼저 만들어야 한다. 이 파서를 생성하기 위한 입력파일의 구성요소에는 옵션부, 파서 클래스 부분, 어휘 정의부, 구문 정의부로 구성되어 있다.

옵션부는 파서를 생성하면서 필요한 여러가지 옵션을 설정하는 부분이며, 파서 클래스 부분에는 파서가 이용하는 프로그램 코드가 들어간다. 어휘 정의부는 파싱에 참여할 토큰을 설정하는 부분이며, 마지막으로 구문 정의부에서는 코볼 BNF(Backus-Naur form)를 보고 해당 넌터미널을 각각 함수로 구현한 부분이 포함된다.

3.3 javacc 구문 정의부 생성 방법

코볼의 context-free syntax 를 가지고 터미널과 넌터미널에 대한 처리를 실시한다.

터미널은 어휘 정의부에서 설정하고 넌터미널은 다시 새로운 함수를 생성해서 다른 넌터미널이나 터미널을 가지도록 정의한다.

```
cobol-source-program=( "IDENTIFICATION" | "ID" )
"DIVISION" "." program-id-cobol-source-program
[ identification-division-content ]
[ "ENVIRONMENT" "DIVISION" "." environment-division-
content ]
[ "DATA" "DIVISION" "." data-division-content ]
[ procedure-division ][ { nested-cobol-source-
program}*"END" "PROGRAM" program-name "." ]
```

표-1. 코볼 context-free syntax

터미널은 해당 토큰으로 매칭이 되고 넌터미널은 계속해서 다른 함수로 정의가 된다.

표-1 에 있는 BNF 를 토대로 표-2 와 같은 생성규칙을 작성하게 된다.

```
void cobolSourceProgram() : {
IdentificationDivision() programId()
[ identificationDivisionContent() ]
[ environmentDivision()
environmentDivisionContent() ]
[ dataDivision() dataDivisionContent() ]
[ procedureDivision() ] [ <TOK_END>
<TOK_PROGRAM>
  programName() <TOK_PERIOD> ]
}
```

표-2. 생성규칙 작성예

3.4 코볼 분석기 전체 자료구조도

시각정보화기에서는 핵심변수와 패턴을 찾기 위해서 각종 정보가 필요하게 된다. 이런 정보를 처리하기 위해 FileTable 이라는 자료구조를 두어서 코볼 프로그램마다 생성되는 FileInfo 객체를 저장하게 된다. 이 FileTable 객체 하나만 관리하게 되면 시각정보화기와 콤포넌트 생성기에서 필요로 하는 모든 정보를 손

쉽게 관리할 수 있게 된다.

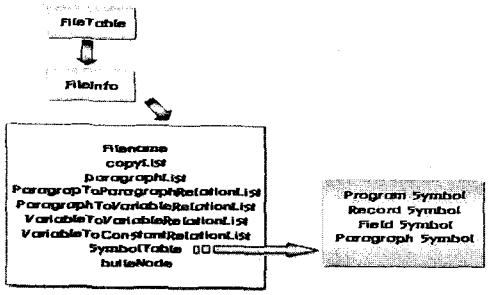

그림-3. 코볼 분석기 전체 자료구조도

3.5 Token, Node 의 자료구조

코볼 코드 분석기에서 해당 정보들을 저장하기 위해서 파서를 실행하게 되면 파싱이 된 토큰들은 자바 컴파일러 컴파일러에 의해서 토큰 자료구조에 토큰에 대한 정보들이 저장되게 된다.

시각화 정보화기에서는 이 토큰자료구조의 정보를 바탕으로 생성한 여러 객체를 사용하게 된다.

노드는 시각화 정보기에서 토큰을 가진 객체인 노드간의 트레이스를 위해서 사용하는 자료구조를 말한다.

3.6 릴레이션 생성

릴레이션 생성 부분은 노드간의 관계를 맺어 주는 정보를 처리한다. 릴레이션은 총 4 가지를 가지며 각각의 릴레이션 마다 정해진 자료구조를 가진다.

ParagraphToVariable,VariableToVariable, VariableToConstant 릴레이션은 RELATION_NONE, RELATION_USE, RELATION_DEFINE 과 같은 공통된 자료구조를 가지며, ParagraphToParagraph 만 RELATION_NOTE, RELATION_SEQUENCE, RELATION_PERFORM, *RELATION_GOTO* 과 같은 조금 다른 릴레이션을 가진다.

3.7 노드 생성

파서를 통해서 사용될 자료들을 모두 생성한 후 이 자료를 가진 노드들을 트레이스하기 쉽도록 연결한다.

4.코볼 분석기의 성능향상 방안

4.1 코볼 COPY 문 처리

코볼에서 COPY 문을 만나게 되면 해당 파일을 원래 존재하던 코볼 소스에 복사하게 되어 고유의 코볼 라인번호 정보를 활용할 수 없게 된다. 그래서 COPY 문을 파싱하면서 해당 라인 만큼을 읽어 저장해 놓았다가 실제 코볼 코드를 파싱할 때에 COPY 문 만큼의 라인 수를 빼어서 고유의 코볼 라인번호를 유지 할 수 있게 설계했다.

4.2 중복 릴레이션 방지

노드 별로 릴레이션이 생성되면서 중복된 릴레이션 객체로 인해 많은 메모리 공간과 시각화 처리 시간을 가지게 된다. 이러한 문제를 해결하기 위해서 릴레이션을 생성할 때 존재 여부를 따져서 존재할 경우 다시 생성하지 않도록 함으로써 메모리와 시각화 처리 시간을 대폭 줄였다.

표-3. 릴레이션 생성 비교표

	소스라인	P To P	P To V	V To V	V To C
1	120	0/0	72/102	83/145	63/63
2	749	12/12	89/390	130/283	173/195
3	795	14/14	95/406	101/297	177/199
4	391	9/9	42/143	37/99	53/71
5	558	15/15	62/250	31/177	111/127
6	113	0/0	2/15	2/11	12/25
7	213	4/5	23/66	45/63	15/15
8	393	10/10	70/154	61/89	91/133
합계	3332	64/65	455/1526	490/1164	695/828
비율	416.5 라인	8/8.1 98.8%	56.9/ 190.8 29.8%	61.3/ 145.5 42.1%	86.9/ 103.5 84%

표-3 에서 릴레이션의 중복 여부에 따라 생성된 릴레이션 객체의 개수를 나타내었다.

평균 416 라인의 코볼 소스 코드를 가지고 중복을 허용 했을때와 중복 허용을 방지했을 경우 패러그래프와 패러그래프간의 릴레이션에서는 거의 차이가 없었다. 그러나 패러그래프와 변수들간의 릴레이션 객체의 개수는 중복 허용을 방지한 경우 29.8%의 비율로 릴레이션 객체가 줄어 든 것을 볼 수 있다. 변수와 변수간의 릴레이션 객체를 살펴보면 약 42%로 역시 릴레이션 객체가 줄었고 마지막으로 변수와 상수와의 릴레이션 객체도 84%의 비율로 객체의 개수가 줄어든 것을 볼 수 있다.

이렇게 릴레이션의 중복 허용을 방지함으로써 시각화 정보기에서 핵심변수나 패턴을 뽑아 낼 수 있는 보다 효율적인 그래프를 생성할 수 있게 된다. 중복 허용을 방지하고자 릴레이션 객체를 비교함으로써 속도의 손해가 발생하나 이것은 보다 정확한 그래프를 그리고 필요한 정보를 찾아내는 것에 비해 조그만 손실이라고 생각된다.

4.3 자바 가상 기계 한계 처리

코볼 언어에 대한 풀셋의 파서 분석기를 생성함에 있어서 자바가상기계의 한계로 인한 파서 분석기의 특별한 설정이 요구된다. 코볼 언어의 풀셋의 파서를 생성함에 있어서 앞으로 추가될 언어의 파서 능력을 예상할 경우 발생하는 엄청난 양의 토큰과 생성규칙으로 인해 그림-4 와 같이 많은 양의 초기화 작업이 요구된다. final 로 설정된 초기화 작업은 자바가상기

계의 clint 이라는 곳에서 실시되는데 이곳에는 자바 가상기계의 스펙상에 2 바이트를 설정하도록 되어있다.

```
final private int[] jj_la1_0 = {0x0,0x0,0x0,0x0,
,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,
x0,0x8080000,0x0,0x8000000,0x0,0x0,0x0,0x8080000,
final private int[] jj_la1_1 = {0x0,0x0,0x0,0x0,
0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,
final private int[] jj_la1_2 = {0x0,0x20000000,
0,0x0,0x200000,0x200000,0x200000,0x0,0x0,0x0,0x0,
final private int[] jj_la1_3 = {0x0,0x0,0x0,0x0,
,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,
```

그림-4. 가상기계 처리 문제

그래서 이 부분이 계속 늘어나게 되면 표-4 와 같은 에러 발생과 함께 파서 분석기가 실행을 멈추게 된다.

표-4. 발생 에러 메시지

```
Exception in thread "main"
java.lang.ClassFormatError
: Cobol (Code of a method longer than 65535 bytes)
at java.lang.ClassLoader.defineClass0 (Native Method)
at java.lang.ClassLoader.defineClass  (Unknown Source)
at java.security.SecureClassLoader.defineClass (Unknown
  Source)
at java.net.URLClassLoader.defineClass   (Unknown Source)
```

즉, 위에서 언급한 자바가상기계가 정해놓은 클래스 파일의 크기인 65535bytes 를 넘어서게 된다.

그래서 코볼의 COPY 문에서 추가될 CICS 의 풀셋의 추가도 그림-5 와 같은 초기화 설정부분을 변경하지 않으면 실행할 수 없게 된다.

```
public void init1() {
      jj_la1_0 = new int[]{0x0,0x0,0x0,0x0,
x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,
080000,0x0,0x8000000,0x0,0x0,0x0,0x8080000,0x0,
      jj_la1_1 = new int[]{0x0,0x0,0x0,0x0,0x0,
0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,
  }

public void init2() {
      jj_la1_2 = new int[]{0x0,0x20000000,0x0,
0x200000,0x200000,0x200000,0x0,0x0,0x0,0x0,0x0,0x0,
      jj_la1_3 = new int[]{0x0,0x0,0x0,0x0,0x0,
x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,
  }
```

그림-5. 가상기계 문제 해결 모습

이것은 자바가상기계의 스펙상에 정의되어진 크기이다. 클래스가 가질 수 있는 메소드의 크기가 스펙상에 2 바이트(2 의 16 승)으로 정해져 있기 때문에 65535bytes 이상은 자바가상기계의 검정과정에서 검사되어져서 프로그램이 65535bytes 보다 더 많은 크기의 기억장소를 사용하려고 할 경우 실행되기 전에 검증과정에서 클래스를 거부하게 되어 실행되지 않게 된다. 해결 방안은 토큰과 생성규칙을 줄이거나 초기화 부분에 해당하는 코드를 객체로 만들어서 초기화 부분인 clint 의 양을 줄여 주어야 한다. 이렇게 함으로써 계속적인 파서 분석기의 확장이 가능하게 될 것이다.

5. 결론 및 향후 연구 방향

COBOL2EJB 코볼 코드 분석기, 시각화 정보 생성기, EJB 래퍼 컴포넌트 생성기, 컴포넌트 시험기로 구성되어 있으며, 이 구성 요소를 통해서 EJB 컴포넌트를 생성하여 기존의 레거시 코볼 시스템과 연동해서 기존 시스템의 변경없이 새로운 환경에서 쉽게 운용하고 새로운 기능을 컴포넌트로 확장이 가능하다.

향후 연구과제로는 보다 효율적인 코볼 분석기의 연구와 함께 핵심변수와 패턴을 보다 정확히 찾아내고 이것을 EJB 에서 사용할 수 있도록 하는 것이 필요하다. 덧붙여 객체의 중복을 방지한 보다 정확하고 빠르게 실행되는 분석기의 설계가 요구된다.

참고문헌

[1] A. Perkins, Business rules=meta-data, Proc.34th, Technology of Object-Oriented Languages(TOOLS 34'), 2000, 285-294.

[2] COBOL II grammar Version 1.0.3

[3] "Creating Components from Legacy Application s", CBDi Forum Journal, Dec. 1998

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

[5] H.Huang, "Business rule extraction from legacy code", Proc. 20Th Computer software and Applications Conference, 1996

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

[7] H.M.Sneed, "Extracting business rules from source code", Proc.4th Program Comprehension, 1996

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

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

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

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