자바프로그램 분석을 위한 바이트코드 시뮬레이터

김 도 우[†] · 정 민 수[††]

요 약

자바와 같이 객체지향 언어로 작성된 프로그램은 프로그램의 실행과 관련된 정보들과 제어의 흐름이 숨겨져 있기 때문에 분석하기가 쉽지 않다. 그러나, 자바의 경우는 컴파일 과정을 통해 생성된 클래스 파일에 프로그램의 수행과 관련된 정보가 포함되어 있다.

자바 가상 기계는 클래스 파일에 포함된 바이트코드를 실행시킨다. 따라서 바이트코드가 실행되는 과정을 살펴보면 자바 소스 프로그램에 대해 보다 명확한 분석과 쉬운 이해가 가능해 진다.

본 논문에서는 자바 프로그램의 이해와 분석을 위한 효율적이고 강력한 바이트코드의 실행을 위한 시각화 도구를 설계 구현했다. 이것은 프로그램의 구조와 객체들 사이의 제어의 흐름을 이해하는데 도움을 준다.

Bytecode Simulator for Analyzing Java Programs

Do-Woo Kim[†] · Min-Soo Jung[††]

ABSTRACT

It is not easy to analyze object-oriented programs, including those in Java, because the control flows of the programs is not visible to the users. The users, however, can utilize class files to trace the process of execution, since a lot of information related on control flow are stored in the control flows.

A Java virtual machine can then execute the bytecodes included in classfiles It means that understanding the execution process of the bytecodes leads users to comprehend and analyze source programs in Java.

We design and implement a visual tool for bytecode execution that is an efficient and powerful tool to understand and analyze source programs in Java. It can aid users to thoroughly grasp not only the structure of a program but also the flow of controls among objects.

1. 서 론

자바는 기존의 객체지향 언어를 네트워크 기반의 인터넷 환경에서 사용하기 알맞게 개발된 언어이다. 자바 언어는 특정한 하드웨어나 운영체제에 영향을 받지 않고 동작할 수 있는 높은 이식성을 가지고 있다. 이것은 자바 언어로 작성된 프로그램의 경우 생성되는 목적코드가 특정한 프로세서에 맞춰져 있지 않고 자바 가상 기계(Java virtual machine)라는 가상의 명령어 집합과 실행 환경에 맞춰져 있기 때문이다.

객체지향 언어로 작성된 프로그램은 프로그램의 수행 과정과 관련되어 있는 많은 정보가 숨겨져 있다. 그 이유는 여러 개의 객체들로 구성된 전체 프로그램 중에서 일부 객체만을 사용자가 실제 작성하고 대부분의 객체는 시스템 혹은 타인이 작성한 객체를 재사용하기 때문이다. 이러한 이유로 객체지향 프로그램을 분석하기가 쉽지 않다. 자바 프로그램의 경우도 마찬가지다. 그러나 자바의 경우는 컴파일 과정을 통해 생성된 클래스 파일에 프로그램의 수행과 관련된 정보가

※ 본 연구는 한국학술진흥재단의 97년도 우수연구소과제 연구비의 지원에 의해 수행되었음.
† 준 회 원 : 경남대학교 대학원 컴퓨터공학과
†† 종신회원 : 경남대학교 정보통신공학부 교수
논문접수 : 1999년 5월 26일, 심사완료 : 2000년 6월 29일

숨겨져 있다[1-3].

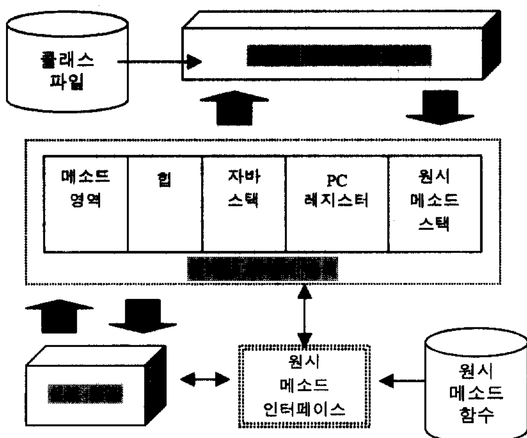
자바 소스 프로그램은 자바 컴파일러에 의해 바이트코드로 번역되고, 자바 가상 기계는 단순히 규정된 바이트 스트림인 바이트코드라 불리는 특별한 명령어 집합을 실행시키는데, 각각의 바이트코드는 자바 가상 기계에 대해 무엇을 해야만 하는지에 대해 정확하고 정교한 세부 규정을 갖고 있다. 바이트코드에는 사용자가 정의한 객체뿐만 아니라 미리 정의되어 숨겨져 있는 객체에 대한 정보가 모두 저장되어 있다. 따라서 바이트코드가 실행되는 과정을 살펴보면 자바 소스 프로그램에 대해 보다 명확하고 쉬운 이해가 가능해 진다. 바이트코드가 실행되는 과정을 가장 효율적으로 보여 줄 수 있는 것은 자바 가상 기계의 각 구성 요소들을 시각화하여 그 동작 과정을 보여주는 것이다[4].

본 논문에서는 자바 바이트코드 실행 시각화 도구의 개발을 통해 자바 프로그램을 분석하고, 명확한 이해를 하는데 도움을 주고자 한다. 뿐만 아니라, 자바 가상 기계의 동작 과정 분석을 통한 컴파일러의 최적화 혹은 프로그램의 최적화에 활용이 가능하도록 하고, 사용자가 작성한 클래스 파일뿐 아니라 시스템 혹은 타인이 제공한 클래스 파일내의 바이트코드도 시각적으로 시뮬레이션하면서 실행시킬 수 있어 다른 사람이 작성한 프로그램도 손쉽게 이해할 수 있게 하고자 한다[6, 7].

2. 관련 연구

2.1 자바 가상 기계의 구조

자바 가상 기계의 구조는 (그림 1)과 같이 클래스 로더 서브시스템(class loader subsystem), 실행 엔진(execution engine)과 런타임 데이터 영역(runtime data areas)으로 구성되어 있다[3, 4].



(그림 1) 자바 가상 기계의 내부 구조

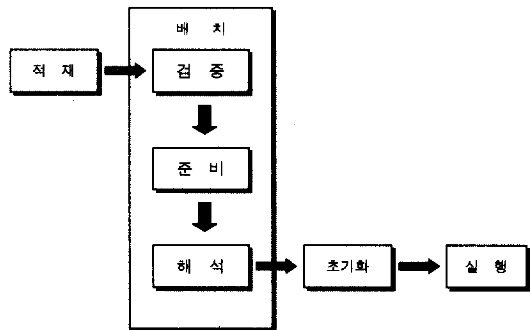
클래스 로더 서브시스템은 클래스나 인터페이스와 같은 형(types)들을 적재하고 검색하는 역할을 수행한다.

자바 가상 기계가 프로그램을 실행할 때 적재된 클래스 파일로부터 추출된 바이트코드를 포함한 프로그램이 인스턴스화한 객체, 메소드들의 매개변수, 리턴값, 지역 변수, 계산의 결과 값과 같은 많은 정보를 저장할 메모리가 필요하다. 이러한 정보들이 저장되는 부분이 런타임 데이터 영역이다. 이 부분은 보통 메소드 영역, 힙, 자바 스택, PC레지스터, 원시 메소드 스택으로 구성된다.

실행 엔진은 적재된 클래스의 메소드들에 포함된 명령어들을 실행하는 역할을 수행한다. 명령어들이 실행될 때 자바 가상기계는 현재 클래스의 콘스탄트 풀, 현재 프레임의 지역 변수, 현재 프레임의 오퍼랜드 스택의 최상위에 놓인 값들을 사용한다.

2.2 자바 가상 기계의 실행

자바 가상 기계는 (그림 2)와 같이 적재(loading), 배치(linking), 초기화(initialization), 실행(execution)의 과정을 통해 클래스 파일을 수행한다[2-4, 7].



(그림 2) 자바 가상 기계의 실행 과정 블럭도

2.2.1 적 재

적재는 세 단계의 과정을 통해서 이루어지는데, 첫 번째 단계는 그 형을 표현하는 이진 데이터의 스트림을 생성한다. 두 번째 단계는 메소드 영역의 내부 데이터 구조로 이진 데이터 스트림을 파싱한다. 세 번째 단계는 그 형을 표현하는 클래스 ***java.lang.Class***의 인스턴스를 생성한다. 적재의 마지막 단계에서 생성된

java.lang.Class 인스턴스는 자바 응용 프로그램과 내부 데이타 구조 사이의 인터페이스 역할을 한다. 내부 데이터 구조에 저장된 형에 관한 정보를 접근하기 위해 프로그램은 그 형에 관한 ***java.lang.Class*** 인스턴스의 메소드를 호출한다.

2.2.2 배 치

배치는 검증(verification), 준비(preparation), 해석(resolution)의 세 단계로 이루어진다. 배치 단계의 첫 번째 단계인 검증은 적재된 데이터의 형이 자바 언어의 의미를 따르고 있는지 혹은 자바 가상 기계의 내부 형식을 만족하는지를 검사한다. 배치 단계의 전 단계인 적재 과정을 통해 클래스 파일의 구성 요소들이 정해진 위치에 정해진 길이를 만족하는지를 검사하지만 적재된 데이터의 형에 대한 실질적인 검사가 검증 단계에서 이루어진다. 두 번째 단계인 준비는 클래스 변수들에 대한 메모리를 할당하고, 변수의 형에 대해 미리 정해진 초기값으로 클래스 변수들을 초기화한다. 하지만 클래스 변수들은 다음 단계인 초기화 단계까지는 변수의 정해진 값으로 초기화되지는 않는다. 마지막 단계인 해석은 클래스, 인터페이스, 필드, 메소드에 대한 이름(symbolic) 참조를 직접(dynamic) 참조로 변환한다.

2.2.3 초기화

초기화는 클래스 변수들에 대해 프로그래머가 미리 정의한 초기값을 할당한다. 자바 클래스 파일에서 클래스 초기화 메소드를 "***<clinit>***"이라 하는데, 자바 응용 프로그램의 내부 메소드들이 호출하는 것이 아니라 자바 가상 기계에 의해서 호출된다. 클래스의 초기화는 부모 클래스에 대해 먼저 이루어지고, 해당 클래스는 그 다음에 이루어진다.

2.2.4 실 행

자바 가상 기계의 구현에 있어서 가장 핵심적인 부분으로 자바를 직접 실행할 수 있는 자바 칩을 이용하여 직접 바이트코드나 원시 메소드들을 실행하는 하드웨어적 기법과 바이트코드 하나 하나를 읽어 해석하여 직접 실행하는 해석기 방식, 실행할 바이트코드를 읽어 들인 뒤, 이 바이트코드를 고유코드로 변환하여 실행하는 JIT(just-in-time) 컴파일 방식, 바이트코드를 직접 해당 기계 코드로 컴파일하여 직접 실행 가능한 이진코드를 생성하는 정적 컴파일 방식 등의 소프트웨어적인 기법이 있다[9].

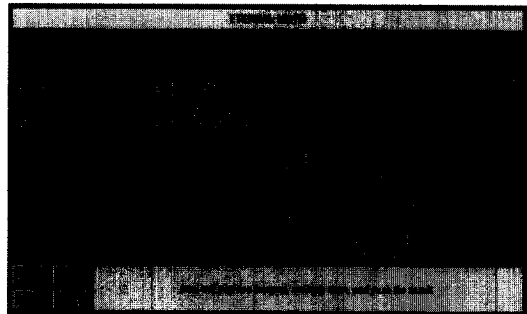
2.3 바이트코드 명령어 집합

바이트코드 명령어는 수행 동작을 구별하기 위한 한 바이트 크기의 연산코드(opcode)부분과 수행에 필요한 값을 정의하는 피연산자(operand)부분으로 구성된다. 대부분의 자바 가상 기계는 각 명령어들이 필요로 하는 피연산자를 스택으로부터 가져오고 또한 결과 값을 스택에 보존한다[3-5]. 다음은 자바 가상 기계 명령어를 기능별로 분류한 것이다.

- 로드(load) 및 저장 명령어들
- 산술 연산 명령어들
- 자료형 전환 연산 명령어들
- 제어 이동 명령어들
- 메소드 호출 명령어들
- 스택 관련 연산 명령어들

2.4 Artima Software사의 EteralMath

EternalMath는 자바 가상 기계가 소수의 바이트코드 명령어를 시뮬레이션하는 과정을 애플릿 뷰어나 웹 브라우저를 통해 보여주는 자바 애플릿이다. 이 시뮬레이터는 가상 기계 코드 입력을 파일이 아닌 프로그램내의 초기화문으로 처리하고 있으며, 특정 클래스의 한 메소드내의 가상 기계 코드가 실행되는 과정만을 보여준다. 버튼에 대한 마우스의 이벤트를 통해서 메소드의 실행 순서, 오퍼랜드 스택 및 지역 변수들의 변화를 보여준다[4, 22].



(그림 3) EternalMath의 바이트코드 시뮬레이션 화면

3. 자바 바이트코드 시뮬레이터의 설계

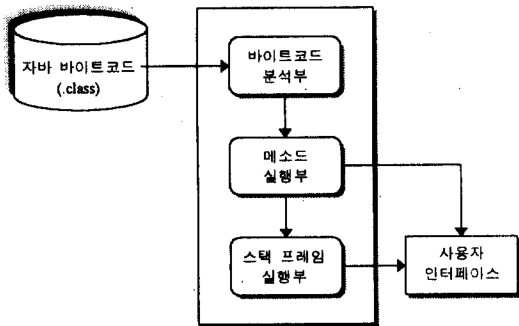
자바 소스 프로그램은 자바 컴파일러를 통하여 자바 바이트코드 즉 클래스 파일들로 생성된다. 자바 바이트코드 실행의 시각화 도구는 컴파일을 통해 생성된

클래스 파일을 입력으로 받아 각 클래스 내의 메소드들을 분석해서 그 분석 결과를 활용하여 메소드가 실제 실행되는 과정을 시각화한 자바 가상 기계의 구성 요소들을 통하여 보여주는 것이다.

자바 가상 기계는 해당 클래스 파일을 적재한 후 이를 가상 기계의 실행 상태로 배치하는 과정, 그리고 초기화 및 실행의 순서로 이루어지는데, 자바 바이트코드 실행의 시각화 도구는 적재에서 초기화 과정은 자바 가상 기계의 동작을 통해 완료한 상태로 가정하고, 마지막 단계인 실행을 비주얼하게 시뮬레이션한다[13-16].

3.1 자바 바이트코드 시뮬레이터의 구성

본 논문에서 자바 바이트코드 시뮬레이터는 (그림 4)에서 보는 바와 같이 크게 바이트코드 분석부, 메소드 실행부, 스택 프레임 실행부로 구성되어진다[19, 20].



(그림 4) 자바 바이트코드 시뮬레이터의 구성

3.1.1 자바 바이트코드 분석부

자바 가상 기계의 동작 과정을 시뮬레이션하기 위해서는 클래스 파일의 분석이 선행되어야 한다. 자바 컴파일러를 통해 생성된 클래스 파일을 구성하는 내용들, 이 내용들간의 상호 연관 관계 및 자바 가상 기계 동작 시에 어떻게 유기적으로 연동되는지 이해할 필요가 있다[13, 19].

자바 바이트코드 분석부는 입력된 클래스 파일을 분석하여 클래스 파일내부에 저장되어 있는 콘스탄트 풀 정보, 클래스 이름, 클래스 내부의 메소드들, 지역 변수 정보, 각 메소드를 구성하는 실제 바이트코드, 각각의 메소드를 구성하는 바이트코드에 해당하는 실제 명령어, 각각의 메소드가 필요로하는 스택의 크기 및 지역 변수부의 크기등 시뮬레이션에 필요한 정보를 추출하여 내부적으로 정의된 자료 구조인 ***methodPoolInfo*** 클래스에 저장한 후, 이 자료 구조를 다음 단계인 메소드 실행부의 입력으로 전달한다.

3.1.2 메소드 실행부

자바 바이트코드 분석부를 통해 methodPoolInfo 클래스에 저장된 정보 즉, 현재 클래스 이름, 현재 메소드 이름, 메소드의 실제 코드, 메소드 실제 코드에 해당하는 가상 기계의 명령어들, 실행될 다음 위치를 나타내는 프로그램 카운터를 우선 자바 가상 기계의 구성 요소들로 구성된 사용자 인터페이스를 통해 출력한다. 그리고 프로그램 카운터의 위치에 해당하는 가상 기계 명령어를 해석하고 명령어 수행으로 인해 발생하는 오퍼랜드 스택과 지역 변수부에 관한 정보를 스택 프레임 실행부로 전달한다.

3.1.3 스택 프레임 실행부

스택 프레임 실행부는 메소드 내의 실제 코드의 실행으로 인해 발생하는 스택 연산의 결과 반영으로 초래되는 오퍼랜드 스택의 변화 및 지역 변수의 변화를 사용자 인터페이스 구성 요소인 오퍼랜드 스택 영역, 지역 변수부를 통해 보여주는 역할을 한다.

3.2 제한 사항

자바 바이트코드 실행의 시각화 도구는 자바 가상 기계의 실행 과정 중 적재, 배치, 초기화가 가상 기계에 이루어졌다는 가정하에 실행 부분만을 클래스 파일의 분석을 통해 분석된 내용들을 기초로 시뮬레이션하기 때문에 문제점을 내포하고 있다.

자바 가상 기계 명령어 중에서 ldc, ldc_w, ldc2_w, anewarray, checkcast, getfield, getstatic, instanceof, invokeinterface, invokespecial, invokestatic, invokevirtual, multianewarray, new, putfield, putstatic과 같은 명령어들은 해석 과정을 거쳐서 실행되는데, 이 과정을 수행할 때 가상 기계의 실행 과정에서 적재 단계를 거치면서 클래스 내의 이진 데이터를 런타임 데이터 영역에 저장하고, 배치 단계를 거치면서 각 명령어 연산에 필요한 피연산자 값을 가리키는 런타임 데이터 영역의 포인터를 구하게 된다. 이러한 과정들의 생략으로 인해 연산에 적합한 피연산자를 적용하지 못한다. 따라서 자바 가상 기계의 구현이 먼저 선행되어야 하고, 실제 가상 기계의 동작 과정에 시각화해 줄 수 있는 구성 요소들을 첨가함으로써 위 문제점을 해결할 수 있다.

4. 자바 바이트코드 시뮬레이터

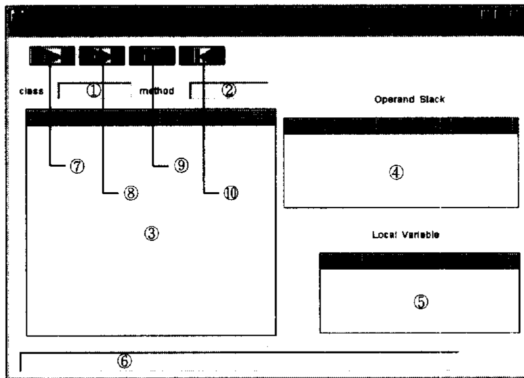
4.1 개발 및 실행 환경

자바 가상 기계의 구성 요소들을 시각화하여 바이트코드가 수행되는 모습을 보여주는 자바 바이트코드 시뮬레이터는 윈도우 95/NT 환경하에서 선사가 제공하는 자바 개발 도구(Java Developer Kit ; JDK) 1.1.6버전을 사용하여 개발하였으며, 전체적인 사용자 인터페이스에 대한 화면 구성 및 배치를 위해 시멘텍(Symantec)사의 비주얼 카페 프로1.0을 사용하였고, 디버깅을 위해 선사가 제공한 JDB를 사용하였다[8, 12].

자바 바이트코드 시뮬레이터는 유닉스나 윈도우 95와 같이 자바 가상 기계가 지원되는 모든 시스템에서 작동 된다. 하지만 그래픽을 표현하는 자바 응용 프로그램이기 때문에 GUI(Graphics User Interface)가 지원되는 유닉스 환경하의 X 윈도우나 윈도우 95/NT와 같은 시스템 환경을 요구한다.

4.2 자바 바이트코드 실행의 시각화 도구의 인터페이스

실행 단계를 가시화하여 보여주는 자바 바이트코드 실행의 시각화 도구는 클래스 이름, 메소드 이름, 메소드 영역, 오퍼랜드 스택 영역, 지역 변수부, 설명부를 시각화 구성 요소로 가진다. 자바 바이트코드 시뮬레이터의 인터페이스는 (그림 5)와 같다.



(그림 5) 자바 바이트코드 실행의 시각화 도구의 인터페이스

(그림 5)의 ①에서 ⑩으로 표시된 각 구성 요소에 대한 기능은 (그림 6)과 같다.

4.3 실험 결과

자바 바이트코드 시뮬레이션을 위해 (그림 7)과 같은 자바 예제 프로그램을 작성하였다. 예제 프로그램은 클래스 내에 산술 연산을 포함한 메소드, 클래스내의 다른 메소드를 호출하는 메소드등을 포함시켜서 가상 기계가 동작되는 과정을 잘 보여 줄 수 있도록 작성하였다.

① 클래스 이름
현재 실행되고 있는 클래스 이름을 출력한다.
② 메소드 이름
현재 실행되고 있는 메소드 이름을 출력한다.
③ 메소드 영역
현재 실행되는 메소드의 실제 바이트코드, 그 바이트코드에 해당하는 명령어, 명령어 연산 수행에 필요한 인자와 프로그램 카운터의 변화를 출력한다.
④ 오퍼랜드 스택 영역
메소드 내의 바이트코드에 해당하는 명령어를 수행하면서 발생되는 스택 내부 값의 변화 과정을 보여준다.
⑤ 지역 변수부
현재 메소드의 실행으로 인해 발생하는 지역 변수들에 대한 값의 변화를 보여준다.
⑥ 설명부
바이트코드에 해당하는 명령어가 어떤 연산 수행을 행하게 되는지를 보조 설명으로 출력한다.
⑦ 런(run) 버튼
메소드 내의 명령어를 연속적으로 수행시키는 버튼
⑧ 스텝(step) 버튼
메소드 내의 명령어를 하나씩 수행시키는 버튼
⑨ 스톱(stop) 버튼
연속적인 수행을 중지시키는 버튼
⑩ 리셋(reset) 버튼
처음 상태로 초기화하는 버튼

(그림 6) 각 구성 요소의 기능

```
class foobar {
    public static void main(String[] args) {
            foobar foo = new foobar();
                int i = 3;
                i *= 2;
                i = i + 4;
                int j = foo.invokeM();
    }

    public int invokeM() {
            int k = 1;
            foobar bar = new foobar();
            int m = bar.invokeS();
            k = m + 5;
            return k;
    }

    public static int invokeS() {
            int s = 3;
            s = s * 3;
            return s;
    }
}
```

(그림 7) 시뮬레이션을 위한 자바 예제 프로그램

위 자바 예제 프로그램 foobar.java는 자바 컴파일러를 통해 foobar.class파일로 생성되고, 이 클래스 파일을 분석해 메소드 관련 정보를 추출한 결과가 (그림 8)과 같다.

```
Method void main()

 0 bb  new #1 <Class foobar>  //객체 생성
 3 59  dup          //스택의top의 내용을 복사하여 스택에 다시 저장
 4 b7  invokespecial #3 <Method foobar()> //메소드 활성화
 7 4c  astore_1     //스택의 top의 내용을 지역 변수에 저장
 8 06  iconst_3     //int형 값을 스택에 저장
 9 3d  istore_2     //스택의 top의 내용을 지역 변수에 저장
10 1c  iload_2      //지역변수의 값을 스택에 저장
11 05  iconst_2     //int형 값을 스택에 저장
12 68  imul         //스택의 top의 내용을 팝하여 곱셈 연산
                      수행후 스택에 저장
13 3d  istore_2     //스택의 top의 내용을 지역 변수에 저장
14 1c  iload_2      //지역변수의 값을 스택에 저장
15 07  iconst_4     //int형 값을 스택에 저장
16 60  iadd         //스택의 top의 내용을 팝하여 덧셈 연산
                      수행후 스택에 저장
17 3d  istore_2     //스택의 top의 내용을 지역 변수에 저장
18 2b  aload_1      //지역변수의 값을 스택에 저장
19 b6  invokevirtual #5 <Method int invokeM()> //메소드 호출
22 57  pop          //스택의 top내용을 팝함
23 b1  return
```

(a) main() 메소드의 바이트코드와 그 의미

```
Method int invokeM()

 0 04  iconst_1   //int형 값을 스택에 저장
 1 3c  istore_1   //스택의 top의 내용을 지역 변수에 저장
 2 bb  new #1 <Class foobar>  //객체 생성
 5 59  dup        //스택의top의 내용을 복사하여 스택에 다시 저장
 6 b7  invokespecial #3 <Method foobar()> //메소드 활성화
 9 4d  astore_2   //스택의 top의 내용을 지역 변수에 저장
10 b8  invokestatic #6 <Method int invokeS()>  //메소드 호출
13 3e  istore_3   //스택의 top의 내용을 지역 변수에 저장
14 1d  iload_3    //지역변수의 값을 스택에 저장
15 08  iconst_5   //int형 값을 스택에 저장
16 60  iadd       //스택의 top의 내용을 팝하여 덧셈 연산
                    수행 후 스택에 저장
17 3c  istore_1   //스택의 top의 내용을 지역 변수에 저장
18 1b  iload_1    //지역변수의 값을 스택에 저장
19 ac  ireturn
```

(b) main()메소드에 의해 호출된 invokeM()메소드의 바이트코드와 그 의미

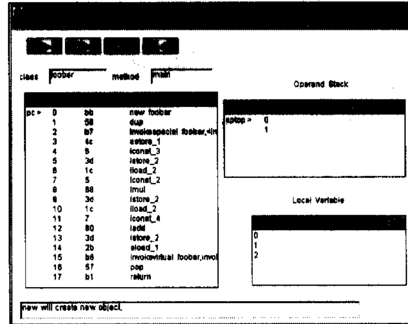
```
Method int invokeS()

0 06  iconst_3   //int형 값을 스택에 저장
1 3b  istore_0   //스택의 top의 내용을 지역 변수에 저장
2 1a  iload_0    //지역변수의 값을 스택에 저장
3 06  iconst_3   //int형 값을 스택에 저장
4 68  imul       //스택의 top의 내용을 팝하여 곱셈 연산 수행 후
                   스택에 저장
5 3b  istore_0   //지역변수의 값을 스택에 저장
6 1a  iload_0    //지역변수의 값을 스택에
```

(c) invokeM() 메소드에 의해 호출된 invokeS()메소드의 바이트코드와 그 의미

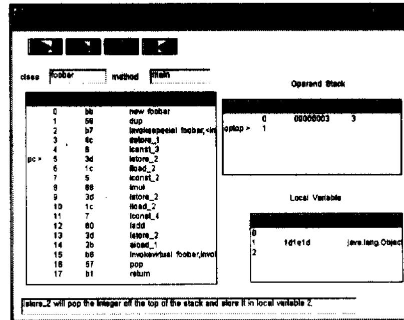
(그림 8) 클래스 파일 분석을 통해 생성된 메소드

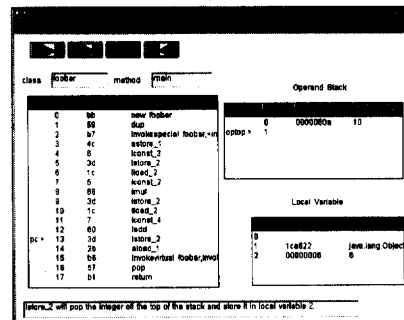
정보자바 바이트코드 실행의 시각화 도구는 시뮬레이션하고자 하는 클래스 파일을 파일 읽기를 통해 선택한다. (그림 9)는 foobar.class 파일을 선택하여 시뮬레이션하기 위한 초기화면이다.



(그림 9) foobar.class 파일을 시뮬레이션하기 위한 초기화면

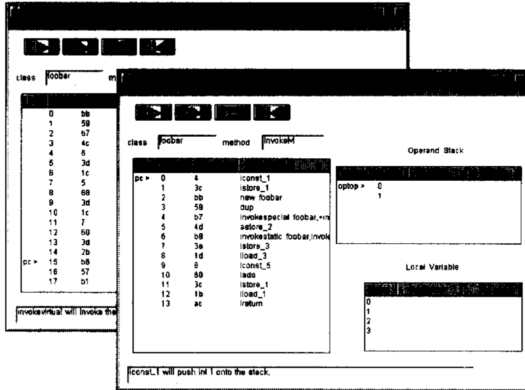
위 그림에서 스텝 버튼이나 런 버튼을 누르면 가상 기계가 동작하는 것과 같이 메소드의 실행을 시뮬레이션한다. (그림 10)은 위 그림에서 오프셋 5에서 13까지를 수행하면서 오퍼랜드 스택과 지역 변수들의 변화를 보여준다.





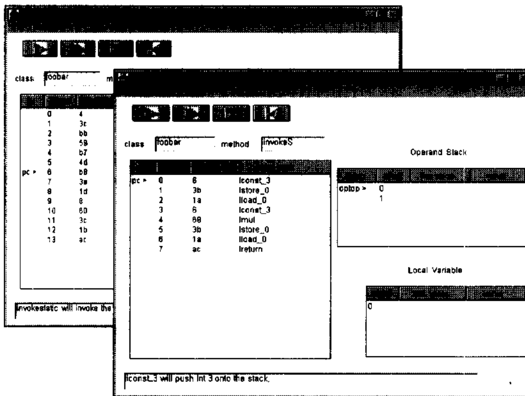
(그림 10) 산술 연산의 시뮬레이션 과정

메소드 실행을 시뮬레이션하는 과정에서 다른 메소드를 호출하면 (그림 11)과 같이 하나의 프레임이 더 생성되고 호출된 메소드의 내용이 출력된다. 호출된 메소드는 호출한 메소드의 실행 시뮬레이션과 같은 방법으로 실행이 이루어진다.



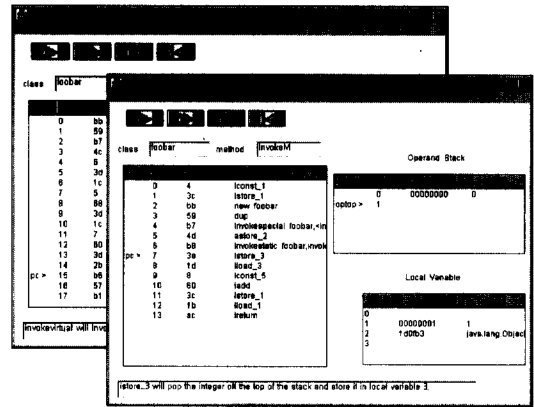
(그림 11) main() 메소드에 의한 invokeM() 메소드의 호출

호출된 메소드의 실행을 시뮬레이션하는 과정에서 또 다른 메소드를 호출하면 또 하나의 프레임이 생성되고 호출된 메소드의 내용이 프레임을 통해 생성된다.

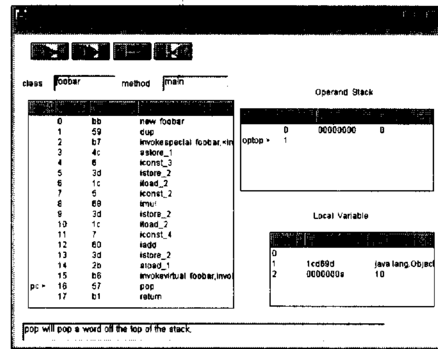


(그림 12) 호출된 invokeM() 메소드에 의한 invokeS() 메소드의 호출

호출된 메소드가 메소드의 실행을 완료하고 호출한 메소드로 제어 및 값을 리턴한다. 호출한 메소드는 프로그램 카운터 값을 이용 계속적인 실행을 할 수 있다. (그림 13)은 호출된 메소드가 실행을 완료한 후의 모습을 보여주고 있다.



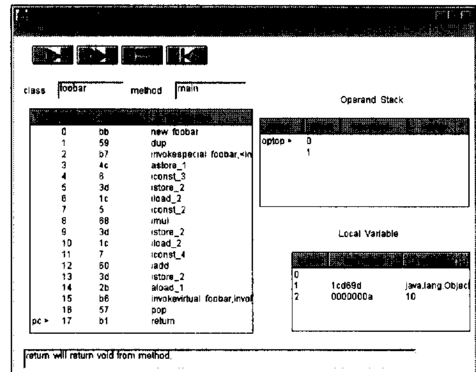
(a) invokeS()가 실행 완료 상태

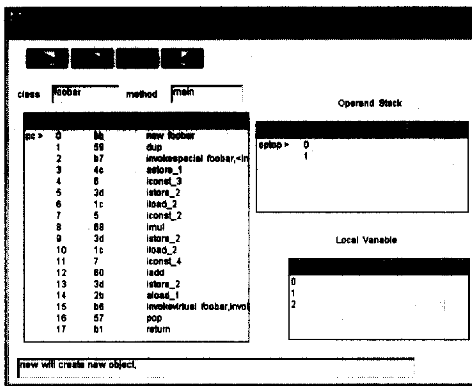


(b) 호출된 invokeM() 메소드의 실행 완료 상태

(그림 13) 호출된 메소드들의 실행 완료 상태

주 메소드의 실행이 완료되면 메소드의 실행을 반복할 수 있도록 (그림 14)와 같이 초기 상태로 복귀한다.





(그림 14) main() 메소드의 완료된 상태

4.4 구현 내역 및 비교 분석

본 논문에서 구현 자바 바이트코드 실행의 시각화 도구의 주요 소스 파일의 이름, 구동 함수의 이름, 소스 라인 수는 다음과 같다.

<표 1> 자바 바이트코드 실행의 시각화 도구의 구현 내역

소스 파일의 이름	소스라인 수
jvms.java	939
ClassInfo.java	1495
Method.java	2283
MethodArea.java	142
StackFrame.java	231
OperandStack.java	123
LocalVars.java	136
bytecode.java	267
합 계	5616

자바 바이트코드 실행의 시각화 도구와 유사한 Artima Software사의 EternalMath와 비교 분석한 내용은 <표 2>와 같다.

<표 2> 관련 도구와의 비교 분석

	본 논문	EternalMath
입력 형태	클래스 파일	초기화문
클래스 파일 분석 기능	지원	지원 못함
메소드 호출	지원	지원 못함
바이트코드 뷰어 기능	지원	지원
지원 언어	자바	자바

본 논문에서 구현한 자바 바이트코드 실행의 시각화 도구는 메소드 관련 정보가 초기화문 형태로 주어진 Eteralmath와는 달리 클래스 파일을 입력으로 받아 분석 과정을 거치면서 시뮬레이션에 필요한 메소드 관련 정보를 추출하고, 이 정보를 시각화된 인터페이스를 통해 보여주고, 메소드 호출이 발생하면 하나의 프레임을 더 생성해 호출된 메소드의 정보 출력 및 시뮬레이션이 가능하다.

5. 결론 및 연구 결과의 활용

본 논문에서 구현한 자바 바이트코드 실행의 시각화 도구는 객체 지향 언어인 자바 프로그램을 실행하는 자바 가상 기계의 수행 과정을 시각적으로 표현함으로써 프로그램의 구조 및 객체들간의 제어의 흐름을 이해하는데 도움을 줄 수 있고, 사용자가 작성한 클래스 파일뿐 아니라 시스템 혹은 타인이 제공한 클래스 파일내의 바이트코드를 시각적으로 시뮬레이션하면서 실행시킬 수 있어 다른 사람이 작성한 바이트코드도 쉽게 이해할 수 있다.

자바로 작성된 프로그램을 체계적으로 분석하게 하여 자바 프로그래밍 개발 환경으로 활용할 수 있고, 자바 가상 기계의 작동 분석 및 자바 프로그램과 가상 기계 코드의 연관성 분석을 통한 컴파일러의 최적화 혹은 프로그램의 최적화에 활용이 가능하다. 뿐만 아니라, 메소드의 실행 과정을 시뮬레이션함으로써 디버깅 도구로도 사용이 가능하다.

클래스 파일의 분석을 통해 분석된 내용들을 기초로 시뮬레이션하기 보다는 자바 가상 기계의 구현을 통해 클래스 파일의 실행 과정을 시뮬레이션할 수 있도록 연구해 나갈 것이다.

참 고 문 헌

[1] G. Gosling, B. Joy and G.Steele, *The Java Language Specification*, Addison-Welsley, 1997.

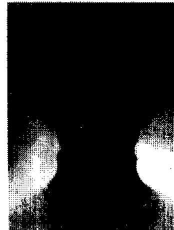
[2] J. Meyer and T. Downing, *Java Virtual Machine*, O'REILLY, 1997.

[3] F. Yellin and T. Lindholm, *The Java Virtual Machine Specification*, Addison-Wesley, 1996.

[4] B. Venners, *Inside Java Virtual Machine*, McGraw-Hill, 1997.

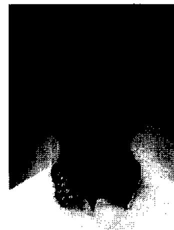
[5] A. V. AHO and R. SETHI, "*How hard is compiler code generation?*" SIAM J. Computing 1, pp.22-29,

1997.
[6] A. Taivalsaari, "*Implementation a Java Virtual Machine in the java programming Language*," SUN Lab., pp.134-162, 1997.
[7] F. yellin and T. Lindholm, "*Java Runtime Internals*," Java One Sun's Worldwide Java Developer Conference, pp.47-61, 1997.
[8] J. R. Jackson, *Java by Example 2/E*, Prentice hall, 1997.
[9] G. cornell and C. S. Horstmann, *Core Java, 2/E*, SunSoft Press, 1997.
[10] 양병선, 문수묵, "향상된 JIT 컴파일러 기법을 이용한 Java 가상 머신의 설계", 한국 정보과학회 제24회 추계발표대회, pp.313-316, 1997.
[11] 정민수, 옥재호, "교환기 프로그래밍 언어에 대한 제어 흐름 표시기의 설계 및 구현", 한국 정보과학회 제24회 추계발표대회, pp.273-276, 1997.
[12] 류동항, 정민수 "자바 바이트코드 분석기의 설계 및 구현", 한국 정보과학회 제25회 춘계발표대회, pp. 77-79, 1998.
[13] 정민수, 이종동, "자바 메소드 호출 관계 표시기의 설계 및 구현", 한국 정보과학회 제25회 춘계발표대회, pp.74-76, 1998.
[14] 이완석, 김홍근, "자바 보안 모델", 한국 정보과학회지, 제16권 제4호, 1998.
[15] 김도우, 정민수, 류동항, 진민, "자바 가상 기계 시뮬레이터의 설계 및 구현", 한국 정보과학회 제25회 추계발표대회, pp.422-424, 1998.
[16] http://java.sun.com/, Sun Microsystems, Java Home Page
[17] http://www.suntest.com/javacc, Sun Microsystems, Java Compiler Compiler-The Java Parser
[18] http://www.artima.com/insidejvm, Interactive Illustrations
[19] http://www.comla.ox.ac.uk/archive/redo/precc.html, Peter Breuer, Jonathan Bowen PRECC-APREttier, Compiler-Compiler



김 도 우

e-mail : dwkim@mail.com.kyungnam.ac.kr
1997년 경남대학교 전산통계학과 (학사)
1999년 경남대학교 컴퓨터공학과 (석사)
1999년~현재 경남대학교 컴퓨터공학과 박사과정 재학 중
2000년~현재 (주)디지털홈네트 대표이사
관심분야 : 자바 기술, 홈네트워킹, 네트워크보안



정 민 수

e-mail : msjung@eros.kyungnam.ac.kr
1986년 서울대학교 컴퓨터공학과 (학사)
1988년 한국과학기술원 전산학과 (석사)
1994년 한국과학기술원 전산학과 (박사)
1990년~현재 경남대학교 정보통신공학부 부교수
2000년~현재 (주)디지탈홈네트 기술이사
관심분야 : 자바 기술, 객체지향기술, 컴파일러, 홈네트워킹