자바 가상 머신에서 클래스 로딩에 관한 연구

A Study on Class Loading in Java Virtual Machine

김기태
인하대학교 전자계산공학과
이갑래
김천과학대 컴퓨터 정보계열
유원희
인하대학교 컴퓨터공학부

Ki-Tae Kim (g2011493@inhavision.inha.ac.kr)
Dept. of Computer Science, Inha University
Kab-Lae Lee (kllee@kcs.ac.kr)
Division of Computer, Kimchoen Science College
Weon-Hee Yoo (whyoo@inha.ac.kr)
School of Computer Science, Inha University

중심어 : 자바, 클래스 로딩, 정규화

Keyword : Java, Class Loading, Formalization

요 약

자바의 동적인 클래스 로딩은 자바 플랫폼에서 실행시간에 소프트웨어 컴포넌트를 동적으로 로딩하기 위한 강력한 메커니즘이다. 다른 시스템에서도 동적 로딩과 링킹을 제공하지만 지연 로딩, 타입안전 링크, 사용자 정의 로딩정책, 다중 이름 공간 등은 자바가 가진 중요한 특징들 이다.

클래스 로딩에서 핵심은 타입 안전을 보장하는 것이다. 타입 안전은 자바 보안에서 아주 중요한 부분을 차지한다. 하지만 자바 가상 머신에서 타입 안전에 대한 메커니즘은 매우 복잡하고, 접근이 명확하지 않아서 지금 까지 많은 버그가 발생하여 타입 안전에 문제가 되었다.

본 논문은 자바 가상 머신에서 동적인 클래스 로더의 동작을 분석하고, 연산적 의미론(operational semantics)으로 추상화하고 현재 로드되어진 클래스와 추가된 제한 등을 이용하여 이전에 제시되었던 타입 안전에 대한 문제를 분석한다.

Abstract

Dynamic class loading and class linking of Java is a powerful mechanism. Many other system also support some form of dynamic loading and linking, but lazy loading, type-safe linkage, user-definable class loading policy, and multiple namespaces are important features of Java.

The core of class loading is assured of type safety. The security of Java greatly depends on type safety. In JVM, type safety mechanism is very difficult and access of accuracy is not clear, so type safety problems were raised.

In paper, we analysis simple Java code and present a diagram graph and an operational semantics for dynamic class loading and type safety.

I. 서론

자바의 동적인 클래스 로딩은 자바 플랫폼에서 실행 시간에 소프트웨어 컴포넌트를 동적으로 로딩하기 위한 강력한 메커니즘이다[1]. 다른 시스템에서도 동적 로딩과 링킹을 제공하지만 지연 로딩, 타입 안전 링크, 사용자 정의 클래스 로딩 정책, 다중 이름 공간 등은 자바가 가진 중요한 특징이다[2],[3].

자바 가상 머신의 초기 버전에서 클래스 로더는 실행 시간 객체이기 때문에 클래스에 대한 참조는 단지 이름에 의해서만 이루어졌고, 그것을 로딩하는 클래스 로더는 포함되지 않았다. 따라서 클래스 로딩시 명확한 클래스의 이름을 요구하였다[4],[5].

자바 가상 머신 관점에서 동적 클래스 로딩을 살펴보면 우선 Saraswat는 동적 클래스 로더를 이용하여 타입 속이기를 보였다[6]. 그는 실행 시간에 타입 안전에 대한 부분을 체크하였다. 그의 클래스 로더와 타입 안전에 관한 비형식적인 논의는 자바 동적 클래스 로더의 형식화에 시작이 되

* 이 논문은 2002학년도 인하대학교의 지원에 의하여 연구되었음(INHA-22769).
접수번호 : #030607-001
접수일자 : 2003년 6월 7일, 심사완료일 : 2003년 6월 17일

었다. Liang과 Bracha에 의해 JDK1.2에서의 클래스 로딩 메커니즘이 바뀌었다[3]. 그들은 로드된 클래스 캐쉬와 로딩 제한을 제안하였다. Tozawa와 Hagiya는 Liang과 Bracha의 로딩 제한 시스템을 표현하였다[7]. 그들은 제한 스키마와 바이트 코드 검증 사이의 관계를 표현하였다. Qian[11]은 동적 클래스 로딩을 상태 변화 시스템으로 표현하는 또 다른 형식화를 제공하였다[9]. Qian은 정적 타입 정보, 현재 로드된 클래스, 그리고 현재 위치한 제한을 가진 동적 의미론과 관련 있는 타입 안전 증명을 표현하였다.

클래스 로딩에서 핵심은 타입 안전에 대한 확신이다. 또한 타입 안전은 자바 보안에서 아주 중요한 부분이다. 하지만 자바 가상 머신에서 타입 안전에 대한 메커니즘은 매우 복잡하고, 정확성에 대한 접근이 명확하지 않아서 지금까지 많은 버그가 발생되었고 또한 타입 안전에 문제가 되어왔다[6,7]. 따라서 본 논문에서는 기존의 타입 안전 문제를 분석하고 이를 바탕으로 자바의 동적인 클래스 로딩을 연산적 의미론(operational semantics)을 통해서 분석한다.

II. 관련 연구

1. 클래스 로더

자바에서 클래스 로더의 역할은 매우 복잡하다. 로딩되는 단위는 클래스가 된다. 클래스는 머신에 대해 독립적이고, 표준적이고, 이진형태로 표현된다. 새로운 클래스의 로딩뿐만 아니라 이름 공간을 조절해서 새롭게 로드된 애플릿이나 애플리케이션을 빠르게 접근할 수 있게 한다. 동적인 로더는 애플리케이션의 요구에 따라 프로그램 되어지고 조절 가능해 진다. 모든 클래스들은 한 번에 하나의 클래스만을 로드하고 필요할 때마다 새로운 클래스를 로드해 나간다. 즉 프로그래머는 모든 프로그램이 동적으로 로드 되므로, 별다른 구별 없이 프로그램 내의 모든 부분들에 대해서 동일하게 접근할 수 있게된다. 자바 가상 머신에서 클래스 파일을 동적으로 로드할 수 있는 이유는 java.lang.ClassLoader라는 클래스를 이용하기 때문이다.

자바 가상 머신은 로딩 되어진 클래스에 대해서 어떤 클래스 로더가 사용되었는지에 대한 정보를 가지고 있다. 그래서 로딩된 클래스가 다른 클래스 파일을 요구할 때 가상 머신은 같은 클래스 로더로 클래스를 로딩하게 된다. 서로 다른 클래스 로더는 각각 다른 이름 공간(name space)를 가진다. 따라서 같은 클래스 파일 이름을 가져도 문제가 발생하지 않는 것이다. 자바에서 사용 가능한 로더는 크게 두 가지로 나눌 수 있다. 우선 시스템 클래스 로더(system class loader)는 기본적으로 프로그램을 실행한 컴퓨터의 파일 시스템에서 클래스 파일을 찾아 로딩하는 역할을 한다. 두 번째는 클래스 로더 사용자 정의 로더(user defined loader)인데 시스템 클래스 로더처럼 동작할 수 있지만 다른 형태, 다른 장소에서 클래스 파일을 찾아 로딩할 수 있다.

2. 클래스 로더 동작

클래스의 로딩 과정을 살펴보면 크게 로딩(loading)과 링킹(linking)으로 나눌 수 있다[8],[9],[10],[11]. 로딩은 클래스의 이름을 사용하여 클래스 파일 형태의 바이트들을 찾고, 이를 자바 가상 머신에게 알리는 작업을 수행한다. 링킹은 클래스가 기본적인 형태를 갖추고 있는가와 가상 머신의 보안과 관련된 제약 조건을 거스르지 않음을 보장할 수 있도록 클래스를 검증하는 단계를 수행하고, 그리고 정적 초기화를 수행하는 메소드를 호출한다.

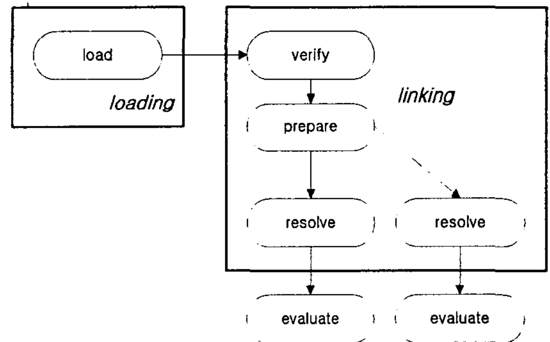

그림 1. 자바 가상 머신의 동작

그림 1은 가상 머신의 동작을 크게 로딩과 링킹으로 나누어 도식화 한 것이다. 그림 1에서 링킹과정은 세분화되어 표현되어진다. 로딩된 클래스는 검증 과정과 준비 단계, 그리고 결정 단계를 진행한 후 평가되어 진다.

III. 클래스 로딩 과정

자바의 클래스 로딩에서 핵심은 타입 안전에 대한 확신을 주는 것이다. 동적인 클래스 로딩의 동작과 타입 안전에 대한 확신을 높이기 위해 본 논문에서는 간단한 코드를 이용하여 동적인 클래스 로딩을 분석하고 도식화하고 형식적

인 방법으로 접근한다.

```
public class Test {
   public static void main(String args[]){
      CallCheck cc = new CallCheck();
      cc.call();
   }
}

public class CallCheck{
   public void call(){
      OtherCheck oc = new OtherCheck();
      Check c = new Check();
      oc.otherCall(c);
   }
}

public class FirstCheck {
   public void fcheck(Check c) {
      Object o = c.i;
   }
}

public class Check {
   Object i;                    C:/Test/test3
}                               loader1에서 동작
```

```
1
public class OtherCheck {
   public void otherCall (Check c) {
      (new FirstCheck()).fcheck(c);
      int f = c.i;
   }
}

public class Check {
   int i;
}                               C:/Test/test3/temp
                                loader2에서 동작
```

그림 2. 동적인 클래스 로딩 예제

그림 2는 동적인 클래스 로딩을 표현한 간단한 예제이다. 클래스 ClassLoader는 $loader_1$과 $loader_2$ 라는 두 가지 다른 객체를 가지며, Test, CallCheck, FirstCheck, OtherCheck, $Check_1$, 그리고 $Check_2$ 등 6개의 클래스를 가진다고 가정한다. 클래스 Test, CallCheck, FirstCheck, $Check_1$는 $loader_1$을 정의 로더로 가지며, 클래스 OtherCheck, $Check_1$는 $loader_2$를 정의 로더로 가진다. 여기서 $Check_1$과 $Check_2$는 같은 이름을 가진 클래스이다. OtherCheck와 Check 클래스를 위해 초기화 로더처럼 $loader_1$을 사용하여 클래스 OtherCheck와 $Check_1$을 생산하고, 반면에 $loader_2$는 초기화 로더처럼 사용되어 FirstCheck와 Check 클래스를 이용하여 클래스 FirstCheck와 $Check_2$ 클래스를 생성한다. 그림 2에서 $loader_1$은 클래스 OtherCheck를 로딩하는 것을 $loader_2$에게 위임한다. 그리고 $loader_2$는 클래스 FirstCheck 클래스의 로딩을 $loader_1$에게 위임한다.

그림 2의 public void otherCall (Check c)에서 자바 가상 머신은 $Check_2$가 로드되기 전까지는 Check1에 있는 i의 타입이 $Check_2$의 형식 인자 c와 매치 되는지 확인하지 않는다. 그래서 otherCall (Check c) 메소드가 수행될 때 예외가 발생하게 된다. 만약 이 메소드에 c.i가 존재하지 않는다면 자바 가상 머신은 예외를 발생시키지 않을 것이다. 타입 안전을 증명하는 보통의 접근 방법은 실행되는 동안 할당된 변수에 항상 같은 타입의 값이 있는가를 보이는 것이다. JDK1.1 버전에서는 otherCall(Check c) 메소드에서 c.i 필드에 대한 접근이 수행될 때, 클래스 $Check_1$과 $Check_2$의 동등성을 체크하지 못했다. 초기 버전의 자바 가상 머신에서는 같은 이름의 메소드가 어떤 변수에 접근할 때, 클래스의 동일성에 대한 체크를 하지 않았다. 그래서 이러한 필드의 접근은 예상할 수 없었고 잘못된 타입이 동작하기도 하였다.

그림 3은 그림 2의 동작을 그림으로 도식화한 것이다. 그림 3에서 $loader_2$의 OtherCheck 클래스에서 c.i를 평가할 때 NoSuchFieldError가 발생한다. 그 이유는 $loader_2$에 의해 로딩된 클래스 OtherCheck에서 c.i에 대한 평가가 이루어 질 때, 객체 c의 멤버필드 i에 대한 접근을 시도하는데, 이때 접근되어야 할 멤버필드 i의 타입은 정수형이기를 기대하는데 실제로 평가된 것은 $loader_1$에 의해 로딩된 Object 타입의 i가 나타나기 때문에 NoSuchFieldError가 발생하게 된다. 하지만 초기버전의 가상 머신은 이러한 오류를 체크하지 못했다.

그림 2에 대해 컴파일 시간에 Test코드의 동작을 보면 그림 5와 같다. 그림 4에서 보는 것과 같이 컴파일 시간에

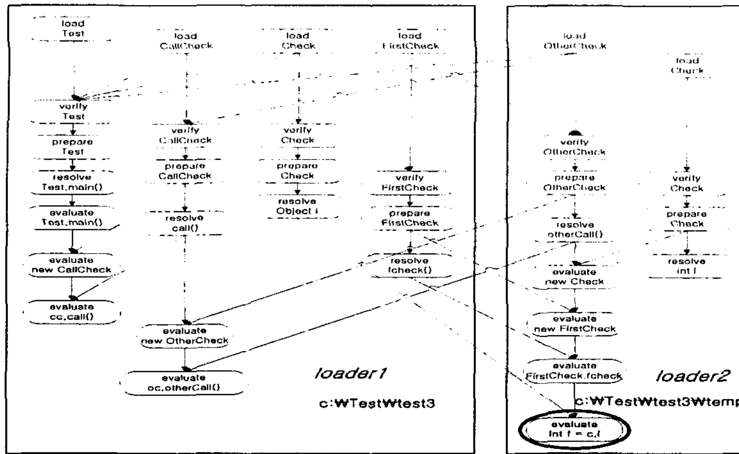

그림 3. 클래스의 동작 분석

```
[Loaded Test]
[Loaded CallCheck]
[Loaded OtherCheck]
[Loaded Check]
[Loaded FirstCheck]
[Loaded java.lang.NoSuchFieldError from C:\j2sdk1.4\jre\lib\rt.jar]
java.lang.NoSuchFieldError: i
[Loaded java.lang.StackTraceElement from C:\j2sdk1.4\jre\lib\rt.jar]
        at OtherCheck.otherCall(OtherCheck.java:12)
        at CallCheck.call(CallCheck.java:13)
        at Test.main(Test.java:12)
```

그림 4. 실행시간에 *Test* 코드의 동작

```
[parsing started Test.java]
[parsing completed 120ms]
[loading C:\j2sdk1.4\jre\lib\rt.jar(java/lang/Object.class)]
[loading C:\j2sdk1.4\jre\lib\rt.jar(java/lang/String.class)]
[checking Test]
[loading .\CallCheck.java]
[parsing started .\CallCheck.java]
[parsing completed 10ms]
[wrote Test.class]
[checking CallCheck]
[loading c:\TEST\test3\temp\OtherCheck.class]
[loading .\Check.class]
```

그림 5. 컴파일 시간에 *Test* 코드의 동작

는 모든 클래스들이 오류 없이 로딩되어진다. 하지만 그림 4에서 보는 것과 같이 실행시간에 NoSuchFieldError가 발생하는 것을 볼 수 있다.

그 이유는 같은 이름을 가진 클래스가 다른 로더에 의해 로딩되었을 때, 초기 버전의 가상 머신은 단지 이름만으로 클래스를 구별하였기 때문에 그와 같은 문제가 발생하였지만 최근 버전에서는 로드된 클래스 캐쉬(loaded class cache), 로딩 제한(loading constraints), 서브타입 제한(subtype constraints) 등을 제시하여 이러한 문제를 해결한다.

본 논문에서 실행되는 자바 가상 머신의 상태는 로드된 클래스 캐쉬, 로딩의 집합, 서브타입 제한, 객체를 저장하기 위한 힙으로 구성된 전역 상태로 구성된다. 상태는 프레임에 저장된 단일 스레드의 스택 수행을 위한 컴포넌트를 포함한다. 각 프레임은 클래스, 클래스의 메소드, 메소드의 프로그램 카운터, 그리고 지역 메모리의 상태를 가진다.

IV. 클래스 로딩 과정

그림 2의 과정을 정규화 표현으로 나타내기 내기 위해서는 몇 가지 정의가 필요하다. 우선 클래스와 클래스 로더에 대해 정의하면 표 1과 같다.

표 1은 cls_nm, sup_nm, cls_mem, refs, cls_f 등 클래스와 관련된 정보를 표현한 것이다. cls_nm은 클래스 파일로부터 클래스 이름을 가져오는 것이고 sup_nm은 상위 클래스 파일의 이름을 가져오는 것이다. 예를 들면 sup_nm (cls_f(Check))라 하면 클래스 파일인 Check의 상위 클래스를

표 1. 클래스 정의

```
cls_nm : ClassFile → Class
sup_nm : ClassFile → Class
cls_mem : ClassFile → Package(Member)
refs : ClassFile → Package(Class)
cls_f : ClassFile
∀m ∈ member(clsf). ClassMeber(m) = cls_nm(cls_f)
sys_cls_ld ∈ LoadedClass
cls_ld ∈ LoadedClass, cls_f ∈ ClassFile ⇒
(cls_f, cls_ld) ∈ LoadedClass
※ → : partial function
```

찾는데 그림 2에서 Check 클래스의 상위 클래스는 Object 클래스이기 때문에 sup_nm(cls_f(Check)) = java.lang.Object 라고 나타낸다. 그리고 sup_nm : ClassFile ⇸ Class 에서 ⇸ 은 부분 함수(partial function)를 표현하는데 이유는 Object클래스에서는 그 상위 클래스가 없기 때문에 →이 아닌 ⇸ 인 부분함수로 표현한다.

sys_cls_ld는 시스템 클래스 로더를 표현하는데 classpath에 있는 디렉토리 리스트로부터 클래스를 로딩 할 수 있다. 논문에서는 /test3와 /test3/temp는 같은 클래스 패스에 있다.

cls_ld는 클래스 로더를 표현하고, cls_f은 클래스 파일을 표현한다. c.cls_f은 로드된 클래스 c의 class file을 표현하는 것이다. c.cls_ld는 로드된 클래스 c의 클래스로더를 표현한다. 그리고 LoadedClass는 로드된 클래스의 집합을 의미한다.

표 2. 서브클래스 관계

```
S : set S of loaded classes
subR ⊆ LoadedClass × LoadedClass
c, c' ∈ S
sup_nm(cls_f(c)) = cls_nm(cls_f(c'))
c.cls_ld = c'.cls_ld
S ⇌ c subR c'          (1)

S ⇌ c subR* c'         (2)

c, c', c'' ∈ S
S ⇌ c subR c''
S ⇌ c'' subR* c'
S ⇌ c subR* c'         (3)
```

서브클래스는 표 2과 같이 표현할 수 있다. 표 2에서 S는 로드된 클래스들의 집합을 표현한다. S ⇌ c subR c'는 서브 클래스의 관계를 표현한다. c와 c'이 로드된 클래스들의 집합에 속하고 c.cls_f의 상위 클래스의 이름이 c'.cls_f과 같고 두 클래스 파일을 로딩한 로더가 동일하다면 클래스 c는 c'의 서브 클래스라고 할 수 있다. 또한 서브 클래스의 관계인 subR*는 subR에 대해 반사와 전사의 관계를 가진다.

표 3. loading 동작

```
W : LoadedClass × Class ⇸ LoadedClass
W(cls_ld, cls) = c ⇒ cls_nm(c, cls_f) = cls
W(cc.cls_ld, java.lang.Object) = cls_nm
S ⇌ load(cc, java.lang.Object) ⇨ S ∪ cls_nm        (4)

W(cc.cls_ld, cls) = cls_nm
sup_nm(cls_f(cls_nm)) = sup_cls_nm
S ⇌ load(cc, sup_cls_nm) ⇨ S'
S ⇌ load(cc, cls) ⇨ S' ∪ cls_nm                    (5)
```

자바 가상 머신의 동작 중 로딩은 표 3과 같이 표현할 수 있다.

W는 클래스 로더의 동작을 나타내는 함수이다. 이 함수에 의해 주어진 클래스 로더와 클래스 이름은 로드된 클래스를 반환한다. cls 는 클래스를 cc는 현재 클래스를 나타내고 sup_cls_nm은 수퍼클래스의 이름을 나타낸다. S ⇌ op(cc, cls) ⇨ S' 는 연산자 op가 수행되면 상태 S에서 상태 S'로 변화되는 것을 나타낸다.

표 4. linking 동작

```
S ⇌ load(cc, java.lang.Object) ⇨ S'
W(cc.cls_ld, java.lang.Object) = cls_nm
S' ⇌ verify(cc, cls_nm) ⇨ S''
S ⇌ link(cc, java.lang.Object) ⇨ S''    (6)

S ⇌ load(cc, cls) ⇨ S'
c_n = W(cc.cls_ld, cls)
super(cls_f(cls_nm)) = sup_cls_nm
S' ⇌ link(cc, sup_cls_nm) ⇨ S''
S'' ⇌ verify(cc, cls_nm) ⇨ S'''
S ⇌ link(cc, cls) ⇨ S'''                (7)
```

다음으로 링킹 동작을 표현하면 표 4과 같다. 링킹 동작에서는 클래스를 load(cc, cls_nm)에 의해 로딩하고, verify(cc, cls_nm)에 의해 클래스가 기본적인 형태를 제대로 갖추고 있는지, 가상 머신의 보안과 관련된 제약 조건을 거스르지 않음을 보장할 수 있도록 클래스를 검증하는 단계를 수행한다.

지금까지 정의한 내용들을 이용하여 그림 3을 분석해보면 다음과 같다. loader1에 의해 로드된 클래스들은 Test, CallCheck, FirstCheck, Check1 클래스이고 $loader_2$에 의해서 로드된 클래스는 OtherCheck와 $Check_2$ 클래스이다. 오류가 발생한 OtherCheck클래스를 중심으로 살펴보면 OtherCheck 클래스는 $loader_2$에 의해 로드되었고, OtherCheck의 현재 로더 역시 $loader_2$이다. 현재 로더를 cc.loader = $loader_2$로 나타낼 수 있다. OtherCheck에서의 동작을 보면 다음과 같이 나타낼 수 있다.

$$S \rightleftharpoons load(cc,\ OtherCheck) \Rightarrow S'$$
$$W(cc.cls_ld,\ OtherCheck) = (cls_f(OtherCheck),\ loader_2)$$
$$sup_nm(cls_f(OtherCheck)) = java.lang.Object$$
$$S' \rightleftharpoons link(cc,\ java.lang.Object) \Rightarrow S'$$
$$S' \rightleftharpoons verify(cc,\ (cls_f(OtherCheck),\ loader_2) \Rightarrow S''$$
$$S \rightleftharpoons link(cc,\ OtherCheck) \Rightarrow S''$$

여기서 S'와 S"은

$$S' = S \cup \{(cls_f(OtherCheck),\ loader_2)\}$$
$$S'' = S' \cup \{(cls_f(FirstCheck),\ loader_1),\ (cls_f(Check),\ loader_2)\}$$

로 표현할 수 있다. 그리고 현재 클래스인 OtherCheck 클래스의 참조를 보면 다음과 같다.

$$refs(cls_f(OtherCheck)) = \{FirstCheck,\ Check\}$$

OtherCheck 클래스는 FirstCheck와 Check 클래스를 참조한다. 이 클래스들의 로딩은 다음과 같이 표현된다.

$$S' \rightleftharpoons load^{*}(cc,\ \{FirstCheck,\ Check\}) \Rightarrow$$
$$S' \cup \{(cls_f(FirstCheck),\ loader_1),\ (cls_f(Check),\ loader_2)\}$$

그리고 OtherCheck 클래스에서 오류가 발생한 필드 i에 대한 접근은 getfield 명령어를 통해서 이루어진다. getfield(cc, (FirstCheck, i))은 다음과 같이 표현되어진다.

$$S'' \rightleftharpoons getfield(cc,\ (FirstCheck, i)) \Rightarrow S'' \cup \{(cls_f(Check),\ loader_1)\}$$

그리고

$$S'' \rightleftharpoons load(cc,\ OtherCheck) \Rightarrow S''$$
$$S'' \rightleftharpoons link(cc, OtherCheck) \triangleright S'' \cup \{\ (cls_f(Check),\ loader_1)\}$$
$$\{Check\} = refs(cls_f(FirstCheck))$$
$$S'' \rightleftharpoons load((cls_f(FirstCheck, loader_1), Check) \Rightarrow S'' \cup \{(cls_f(Check),\ loader_1)\}$$
$$S'' \rightleftharpoons verify((cls_f(FirstCheck, loader_1), Check) \Rightarrow S'' \cup \{(cls_f(Check), loader_1)\}$$

이 때 로드된 클래스들의 상태는

$$S \cup \{(cls_f(OtherCheck), loader_2), (cls_f(FirstCheck), loader_1), (cls_f(Check),\ loader_2), (cls_f(Check),\ loader_1)\}$$

이고

$$cls_nm((cls_f(Check), loader_2)) \neq cls_nm((cls_f(Check), loader_1))$$

인 관계가 성립하기 때문에 이때 찾고자하는 i에 해당하는 필드를 찾지 못하게 된다. 초기 버전에서는 이 부분에서 로더에 대해서는 고려하지 않고 단지 이름이 동일한가만 확인하였기 때문에 타입에 관련된 문제가 발생하였던 것이다. 이러한 문제는 클래스 이름을 표기하면서 함께 로더를 표시하여 주기 때문에 이름만 확인하여 클래스를 로딩하던 문제를 해결할 수 있고 정확한 클래스 로딩을 수행할 수 있다.

V. 결론

본 논문은 자바 가상 머신의 클래스 로더에 대한 동작을 분석하고 연산적 의미론으로 추상화하여 자바 보안에서 제시되었던 타입 안전에 대한 문제를 분석하였다. 그리고 자바 가상 머신에서 클래스가 로드된 상태를 정의하고, 로드와 링킹 동작을 정의하였다. 상태의 전위는 조건이 만족되어 질 때 이루어지고, 이러한 조건은 실제로 실행되는 자바 가상 머신에 의해 실행시간 체크에 상응하고 실패하면 자

바 가상 머신에서 예외가 발생하게 되었다. 본 논문에서는 간단한 예제 코드를 이용하여 타입 안전에 대한 문제를 다양한 방법으로 분석하였다. 그리고 다중 클래스 로딩을 사용하고 지연 로딩 접근 방식으로 클래스 로딩을 접근하였다. 또한 클래스 로딩 동작을 다루기 위해 몇 가지 함수와 명령도 정의하였다. 하지만 동시성과 예외에 대해서는 고려하지 않아 향후과제에서는 이 문제에 대해 고려할 것이다.

참 고 문 헌

[1] Jensen, T., Metayer, D. L., and Thorn, T. "Security and dynamic class loading in java: A formalization.," In Computer Language, pp. 25. IEEE Comput. Soc. Press, Los Alamitos, Calif., 1998.

[2] Bracha, "A Critique of Security and Dynamic Loading in Java : Foramlization.," Sun Java SoftWare, 1999.

[3] Liang, S. and Bracha, G., "Dynamic class loading in the Java virtual machine.," In Object-Oriented Programming, Systems, Languages and Applications (OOPSLA) (Vancouver, Canada, Oct.) SIGPLAN Notices 33, 10., pp. 36-44. 1998.

[4] Fritzinger,j. S. and Mueller, M., "Java Security," Sun Micro systems Inc, Mountain View, Calif. 1996.

[5] Gosling, J. Joy, B. Steele, G. and Bracha, G. "The Java Language Specification (second ed.).," Addision Wesley, Reading, Mass. 2000.

[6] Saraswat, V., "Java is not type-safe," Tech. Rep. (Aug.), AT&T Research, Florham Park, New Jersey. 1997.

[7] Tozawa, A, and Hagiya, M. "Careful analysis of type spoofing.," In C. H. Cap, Ed., JIT'99 Java Informations-Tage, Informatik aktuell, Springer-Verlag. pp. 290-296. 1999.

[8] Drossopoulou, S. "Towards an abstract model of Java dynamic linking and verification.," In ACM SIGPLAN Workshop on Type in Compilation(TIC)(Montreal, Canada, Sept.), Computer Science Department, Carnegie Mellon University. pp. Paper 19.

[9] Drossopoulou, S., Wragg, D. and Eisenbach, S., "Is the Java type system sound?," Theory and Practice of Object Systems 5, 1, pp. 3-24., 1999.

[10] Drossopoulou, S. and Eisenbach, S., "Manifestations of Java Dynamic Linking"

[11] Qian, Z., Goldberg, A., and Coglio, A., 200C. "A formal specification of JavaTM class loading.," In Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)(Minneapolis, Minnesota, Oct), ACM, New York. pp. 325-336. 2000.

김 기 태(Ki-Tae Kim) 정회원

1999년 2월 : 상지대학교 전산학과 (이학사)
2001년 2월 : 인하대학교 전자계산 공학과 (공학석사)
2001년 3월 ~ 현재 : 인하대학교 전자계산공학과(박사과정)

<관심분야> : 정보 보안, 컴파일러, 프로그래밍 언어

이 갑 래(Kab-Lae Lee) 정회원

1987년 2월 : 인하대학교 전산학과 (이학사)
1989년 2월 : 인하대학교 전자계산 공학과 (공학석사)
1997년 3월: 인하대학교 전자계산 공학과(박사수료)
1993년 9월 ~ 현재 : 김천과학대학 교수

<관심분야> : 콘텐츠 보호, 웹 프로그래밍, 정보 보안

유 원 희(Weon-Hee You) 정회원

1975년 2월 : 서울대학교 응용수학과 (이학사)
1978년 2월 : 서울대학교 대학원 계산학(이학석사)
1985년 2월 : 서울 대학교 대학원 계산학 전공(이학박사)
1979년 ~ 현재 : 인하대학교 컴퓨터 공학부 교수

<관심분야> : 컴파일러, 프로그래밍 언어, 실시간 시스템, 병렬시스템