디지털산업정보학회 논문지 제7권 제2호-2011년 6월

자바 바이트코드의 검증을 위한 프레임워크 설계

김 제 민* · 박 준 석** · 유 원 희***

A Design of Verification Framework for Java Bytecode

Kim, Je Min · Park, Joon Seok · Yoo, Weon Hee

〈Abstract〉

Java bytecode verification is a critical process to guarantee the safety of transmitted Java applet on the web or contemporary embedded devices. We propose a design of framework which enables to analyze and verify java bytecode. The designed framework translates from a java bytecode into the intermediate representation which can specify a properties of program without using an operand stack. Using the framework is able to produce automatically error specifications that could be occurred in a program and express specifications annotated in intermediate representation by a user. Furthermore we design a verification condition generator which converts from an intermediate representation to a verification condition, a verification engine which verifies verification conditions from verification condition generator, and a result reporter which displays results of verification.

Key Words : Verification, Java Bytecode, Proof Carrying Code,

Ⅰ. 서론

자바 언어로 작성한 소프트웨어는 재사용성과 플랫폼 독립적인 장점으로 인해 일반 응용프로그램 뿐만 아니라 웹 환경과 임베디드 시스템 환경에서도 활용되고 있다. 특히 항공, 우주, 운송, 산업 분야처럼 프로그램의 안전성과 보안성이 중요한 분야에서도 사용되고 있다. 따라서 자바 프로그램의 안전성과 보안성을 검증할 수 있어야 한다. 안전성과 보안성이 검증되지 않은 소프트웨어를 실행함으로써 발생할 수 있는 심각한 오류 및 위험성을 제거하기 위해서는 소프트웨어를 실행하기 전에 검증하는 것이 필요하다.

그런데 배포되며 다운로드되어 실행되는 소프트웨어는 소스코드를 포함하지 않는 경우가 대부분이다. 자바로 작성된 소프트웨어는 이런 경우 바이트코드 형태를 하고 있다. 그러므로 자바로 작성된 소프트웨어의 안전성을 검증하기 위해서는 바이트코드에 대한 정적검증이 필요하다.

정적 검증은 검증 범위와 자동화 정도에 따라 타입검사, (완전한) 프로그램 검증, 확장된 정적 검증(Extended Static Checking)으로 나눌 수 있다[1]. 타입검사는 자동화되어 있는 대신 다룰 수 있는 검증 범위가 제한되어 있다. 이에 반해 완전한 프로그램 검증은 이론적으로 완

* 인하대학교 컴퓨터정보공학과 박사과정
** 인하대학교 컴퓨터정보공학부 조교수
*** 인하대학교 컴퓨터정보공학부 교수

전한 검증이 가능하지만 자동화가 어렵다. 이에 대한 균형을 맞춘 방법이 확장된 정적 검증으로서 적절한 수준의 자동화와 검증 범위를 제공한다.

확장된 정적 검증은 주로 다음과 같은 부분으로 나누어져 수행된다. 검증하고자 하는 대상이되는 소스 코드와 검증하고자 하는 성질을 나타내는 명세가 함께 검증기에 입력으로 들어간다. 검증기는 소스코드와 명세를 가지고 검증조건(Verification Condition, 이하 검증조건 또는 VC)을 생성해내고 생성된 검증조건을 정리증명기(Theorem Prover)를 통해 증명한다. 증명 결과를 통해 프로그램이 명세한 성질을 만족하는지 분석할 수 있다.

기존의 확장된 정적 검증은 주로 고급 언어를 대상으로 수행되었다. 고급 언어를 대상으로 하는 확장된 정적 검증에 비해 바이트코드와 같은 저급 언어를 대상으로 하는 검증은 루프나 분기와 같은 구조가 명확하지 않으며 명령어가 길어지는 등의 문제가 발생한다. 본 논문에서는 자바 바이트코드가 저급 언어이기 때문에 발생하는 이러한 문제점을 해결하며, 자바 바이트코드에 대한 확장된 정적 검증을 통해 안전성을 검증할 수 있는 프레임워크를 설계한다.

본 논문의 구성은 다음과 같다. 2장에서는 검증 프레임워크를 설계하는데 있어서 관련이 되는 기존의 연구에 대해 설명한다. 3장에서는 본 연구의 동기가 된 바이트코드 프로그램의 예제를 보인다. 4장에서는 검증 프레임워크의 각 부분에 대해 설명한다. 4.1절에서는 바이트코드로부터 중간표현을 생성하는 중간표현 생성기에 대해 설명한다. 4.2절에서는 오류 분류 및 사용자 명세에 대해 설명한다. 4.3절에서는 명세와 프로그램 코드 부분이 합쳐진 중간표현을 가지고 검증조건을 생성하는 검증조건 생성기에 대해 설명한다. 4.4절에서는 검증조건 생성기에서 생성하는 검증조건을 가지고 SMT 해결기(Satisfiability Modulo Theories solver, 이하 SMT 해결기)를 이용해 검증을 수행하는 검증조건 검증기에 대해 설명한다. 마지막으로 5장에서는 결론과 향후의 연구 방향에 대해 설명한다.

Ⅱ. 관련연구

2.1 분석 및 검증도구

Julia[2]는 바이트코드 프로그램의 최적화 및 검증을 위해 요약해석(Abstract Interpretation)을 적용할 수 있도록 하는 포괄적(generic) 정적 분석 도구이다. 이를 위해 Julia는 요약 도메인을 쉽게 작성할 수 있는 방법을 제공하고 고정점 계산을 할 수 있는 고정점 계산 엔진을 제공한다.

프로그램 검증은 Proof Carrying Code(PCC)[3] 이후 꾸준히 주목받고 있다. 그 중 ESC/Java[1]는 Proof Carrying Code(PCC)[3] 패러다임을 이용해 자바 소프트웨어의 신뢰성과 보안성을 검증하는 도구이다. ESC/Java는 Java Modeling Language(JML)[4] 명세가 표기된 자바 소스코드로부터 증명을 생성하고 생성된 증명을 정리 증명기(Theorem Prover)를 통해 증명한다.

Java Applet Correctness Kit(JACK)[5]은 JML 명세로 검증하고자 하는 성질이 표기된 자바와 자바 카드 프로그램에 대해 검증할 수 있는 도구이다. 자바소스코드는 바이트코드로 변환되고 JML 표기는 BML 표기로 변환될 수 있다. 마찬가지로 JACK은 검증조건을 생성하고 생성된 검증조건을 정리 증명기를 통해 확인한다.

spec#[6]은 C#언어에 명세를 할 수 있도록 C#언어를 확장한 형태이다. Spec#으로 작성한 프로그램은 변환과정을 거쳐 중간 형태인 BoogiePL[7]로 바뀌게 된다. 변환된 BoogiePL은 Z3[8]와 같은 SMT 해결기에 의해 검증된다.

KeY[9]는 자바 프로그램에 대해 JML과 UML을 이용해 명세하고 자바를 위한 일차 동적 논리식(first-order dynamic logic)을 생성하여 KeY 증명기를 통해 검증을 수행하는 시스템이다. Key 증명기는 기호실행(symbolic execution), 산술 단순화(arithmetic simplication) 등을 이용한다.

Krakatoa[10]는 자바 소스 코드에 대한 검증을 위한 도구로 자바 소스코드와 JML을 이용해 검증 조건을 생

성하고 생성된 검증 조건을 가지고 여러 검증기를 통해 검증을 수행한다.

Julia의 경우 정적 분석을 손쉽게 수행할 수 있는 기능을 제공한다. 하지만 사용자가 검증하고자 하는 프로그램의 성질이나 오류를 판단하기 위해서는 사용자가 정적 검증 기술을 알아야 하며 요약 도메인 등을 제공해야 하는 단점이 있다.

ESC/Java, JACK, KeY, Krakatoa 같은 경우, 고급언어인 자바 소스코드에 대한 검증을 수행한다. 이러한 도구들의 경우 Juila와 같이 사용자가 프로그램 분석의 세부사항을 파악할 필요없이 검증하고자 하는 성질을 명세함으로써 프로그램의 성질이나 오류를 검증할 수 있다. spec#의 경우 검증 대상이 고급언어인 C#이라는 차이점 외에는 ESC/Java, JACK 등과 유사하다.

그런데 현재까지 고급 언어에 대한 검증 프레임 워크는 존재하였지만 저급 언어인 자바 바이트코드에 대한 검증 프레임워크는 존재하지 않았다. 고급언어에 대한 검증 프레임워크를 저급 언어를 대상으로 적용하기에는 저급언어는 루프나 분기와 같은 구조가 명확하지 않으며 명령어가 길어지는 등의 문제가 있다. 따라서 저급언어인 자바 바이트코드를 대상으로 검증을 수행하는 프레임워크가 필요하다. 따라서 본 논문에서는 자바 바이트코드 검증을 위한 프레임워크를 제안한다.

2.2 증명도구

생성된 검증 조건을 증명기를 통해 증명해야 한다. 증명기는 증명 보조기나 의사 결정함수(Decision Procedure)가 있다. 생성된 검증 조건을 사용자와의 상호 작용을 통해서 증명해나가는 방법이 증명 보조기를 이용하는 방법이다. 증명 보조기에는 Coq[11], PVS[12], HOL[13] 등이 있다. 의사 결정함수를 이용하는 방법은 검증 조건을 의사 결정함수의 입력에 맞게 변형하여 넣으면 자동으로 증명을 수행하는 방법이다. 의사 결정함수에는 SAT 해결기, SMT 해결기 등이 있다. SAT 해결기는 명제 논리식으로 나타난 검증 조건을 입력으로하여 만족 가능성을 증명하는 증명기이다. SAT 해결기를 통해 자동화된 검증을 수행할 수 있지만 프로그램을 검증하기 위해서는 프로그램에 존재하는 다양한 자료 구조들을 명제 논리식으로 모델링해야 한다. 이에 반해 SMT 해결기는 배열, 비트벡터, 레코드 등의 자료 구조를 다룰 수 있다. 따라서 SMT 해결기는 프로그램에 대한 검증을 위해 증명기로 적합하다.

Ⅲ. 사례(Motivating Example)

그림 2는 본 논문에서 설명을 위해 사용하는 바이트코드 형태의 예제 프로그램이다.

```
public class Example{
        int x;
        public void A(int y){
                if(y>0)
                        x+=y;
        }
}
```

<그림 1> 자바소스코드 형태의 예제 프로그램

```
public Example();
  Code:
    0:  aload_0
    1:  invokespecial       #10; //Method java/lang/Object."<init>":()V
    4:  return

public void A(int);
  Code:
    0:  iload_1
    1:  ifle    14
    4:  aload_0
    5:  dup
    6:  getfield            #18; //Field x:I
    9:  iload_1
    10: iadd
    11: putfield            #18; //Field x:I
    14: return
}
```

<그림 2> 바이트코드 형태의 예제 프로그램

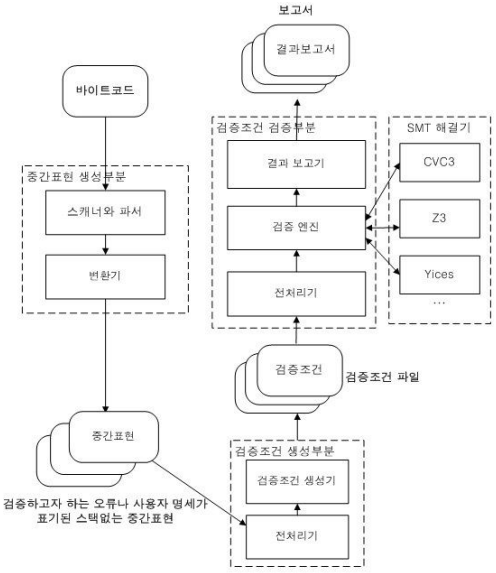
그림 1의 자바 소스코드를 그림 2와 같은 바이트코드로 컴파일하게 되면 컴파일된 바이트코드는 스택기반 코드이기 때문에 검증을 위한 기존의 고급언어를 대상으로 하는 분석 기법을 그대로 적용하기 힘들다. 또한 검증조건을 표기하기 위한 구조가 없다는 단점이 있다.

Ⅳ. 검증 프레임워크 구조 제안

그림 3과 같이 검증 프레임워크는 크게 중간표현식 생성기(IR Generator), 검증조건 생성기(VC Generator), 검증조건 검증기(VC Verifier)로 구성된다. 중간표현식 생성기는 검증대상이 되는 바이트코드를 표현이 명확하며 분석이 용이한 형태로 변환한다. 검증조건 생성기는 중간표현식 생성기에 의해서 생성된 중간표현식과 사용자가 검증하고자 하는 성질이나 이미 정의된 프로그램 오류를 명세한 명세파일을 함께 이용해 검증 조건을 생성한다. 검증조건 검증기는 검증조건 생성기에 의해 생성된 검증 조건을 기존에 개발되어있는 여러 SMT 해결기를 통해 검증하고 검증 결과를 보고한다.

4.1 중간표현식 생성기

검증 대상이 되는 프로그램은 자바 바이트코드 형태로 되어있다. 그런데 순수한 형태의 자바 바이트코드를 대상으로 검증 조건을 생성하는 것에는 몇 가지 단점이 있다. 자바 바이트코드는 스택 기반 코드이다. 따라서 바이트코드의 명령어는 다른 중간 형태보다 명령어가 길고, 표현이 불명확하며, 코드의 단편화가 존재한다. 또한 코드의 재사용이 용이하지 않기 때문에 불필요한 적재와 저장 명령어가 존재한다[14]. 또한 자바 소스코드로부터 컴파일된 바이트코드에는 소스코드가 가지고 있던 정보가 손실될 수 있다. 예를 들어, 자바 소스코드에서는 세분화되어 있던 타입 정보가 손실되거나 변형된다. 이러한 분석 및 변환 측면 뿐 아니라 바이트코드 자체에는



<그림 3> 검증 프레임워크의 구조

검증하고자 하는 성질을 나타낼 수 있는 방법이 없으며 스택기반 코드이므로 검증하고자 하는 성질을 나타내기 어렵다. 이러한 이유들로 인해 바이트코드를 피연산자 스택을 사용하지 않으며 검증하고자 하는 성질을 기술할 수 있는 코드로 변환하여 검증 조건 생성이 용이하도록 해야 한다. 이를 위해 중간 표현 생성기를 설계한다.

```
variable int x
predicate P(int y) := (y>0)
logicalfunction A :( int ) → unit
computationfunction A :( int y) → unit{
read ( y ) write ( x )
block 0 : 0 : ifd  y>0  1
assert P(y)
block 1 : 1 : check
          2 : check
          3 : affectField this.x : Example.int :=  y+x
block 4 : 4 : return
from 0 to 1 when y>0
from 0 to 4 when ¬ y>0
from 1 to 4
returnblock 4 }
```

<그림 4> 예제 프로그램의 중간표현

따라서 그림 2의 바이트코드를 그림 4의 중간표현으로 변환한다. 그림 4의 중간표현에는 프로그램의 실행 의미 부분을 제어 흐름에 따라 나타나기 위해 block0와 같은 블록 라벨로 시작하는 기본 블록을 포함하고 from으로 시작하는 제어 흐름을 포함한다. 또한 블록은 바이트코드 명령어에 대응하는 중간표현 명령으로 이루어진다. 그림 5의 중간표현을 사용함으로써 명확한 표현과 명시적인 배정에 대한 정보를 얻을 수 있으며 제어흐름이 구분된다.

그림 5는 그림 4의 중간표현의 문법 중 프로그램의 코드부분을 나타내는 명령어 부분이다. 명령어는 바이트코드의 명령어에 대응되며 값 배정, 분기문, 객체 생성, 배열 생성, 값 반환, 메소드 호출 등을 포함한다.

```
Command ::== Index : Instr
           | assert LogicalExpr
           | assume LogicalExpr
Index ::== Id
Instr ::== assignVal VariableId := ArithExpr
       | referenceObj VariableID := RefExpr
       | modArray VariableId [ArithExpr] := ArithExpr
       | nop
       | affectField(ObjectId.VariableId : x.Type) :=ArithExpr
       | affectStaticField(ObjectId . VariableId : x.Type)
         := ArithExpr
       | goto Index
       | ifd BooleanExpr Index
       | throw ExceptExpr
       | vreturn Expr
       | return
       | new VariableId := new ParamList
       | newArray VariableId := newarray Type [ArithExpr]
       | invokeStatic | invokeVirtual | monitorEnter
       | monitorExit | mayInit | check
```

<그림 5> 중간표현의 명령어 부분의 문법

바이트코드로부터 중간표현으로 변환하는 과정은 다음과 같다. 그림 1의 중간표현 생성부분에서와 같이 중간표현 생성기는 스캐너 및 파서 그리고 변환기 부분으로 나눌 수 있다. 스캐너 및 파서 부분에서는 바이너리 형태의 클래스파일로부터 자바 바이트코드의 클래스, 멤버 변수, 멤버 메소드 부분을 구분하여 읽어들인다. 이때 상수 풀로부터, 호출되는 메소드나 사용되는 클래스 정보를 추출하고 메소드 부분으로부터 지역 변수 정보와 명령어를 추출한다.

변환기에서는 이렇게 스캐너 및 파서로부터 추출한 정보를 이용해 제어흐름그래프를 생성한다. 제어흐름그래프를 생성함으로써 분기문에서 발생하는 제어흐름에 따라 명령어들을 기본 블록으로 구분하고 조건에 따른 제어의 흐름에 대한 정보를 얻을 수 있다. 생성된 제어흐름그래프에 대해 가상 피연산자 스택을 이용해 피연산자 스택을 이용하지 않는 명령어로 변환한다.

<표 1> 자동으로 명세 생성이 가능한 오류 분류

오류 분류	설명
음수 인덱스	배열 참조시 배열 인덱스가 음수일 경우 발생
범위 초과 인덱스	배열 참조시 배열 인덱스가 배열의 범위보다 클 경우 발생
비생성 객체 참조	생성자를 통해 객체 생성시 생성된 객체가 널 값을 참조할 때 발생
널 값 참조	널 값을 갖는 객체를 참조할 때 발생
부적합 캐스트	부적합한 타입으로 캐스트할 때 발생
영으로 나누기	값을 0으로 나눌 때 발생

4.2 오류 분류 및 사용자 명세

중간표현 생성기에 의해 생성된 중간 표현은 검증하고자 하는 성질을 포함하지 않은 순수한 프로그램 코드이다. 따라서 검증조건 생성기가 검증조건을 생성하기 위해서는 사용자 또는 분석가가 검증하고자 하는 성질을 나타낼 수 있는 방법이 필요하다. 자바 소스코드의 검증의 경우 JML을 이용해 검증하고자 하는 성질을 주석형태로 자바 소스코드에 포함시킬 수 있다. 하지만 JML은 자바 언어에 의존적이다. 이로 인해 검증하고자 하는 대상이 자바 언어로 규정되어 있기 때문에 JML이 구조를 포함한다. 따라서 JML을 자바 언어가 아닌 언어로 작성된 프로그램 코드에 적용할 수 없다. 그러므로 중간표현

생성기에 의해 생성된 중간 표현식에 적합한 명세언어를 정의해야 한다.

명세 언어는 술어 논리를 이용해 전조건, 후조건, 불변자 등의 조건 등을 표시할 수 있어야 한다. 이를 위해 Dijkstra의 보호명령(guarded command)[14]의 변형된 형태의 명세언어를 제공한다. 뿐만 아니라 바이트코드의 객체지향언어로서의 특징도 검증할 수 있는 표현 방법을 제공해야 한다. 따라서 명세 언어는 함수 부작용, 클래스 불변자, 정보 은닉, 별칭, 상속 등에 대해서도 명세할 수 있는 구조를 제공해야 한다.

사용자나 분석가의 세분화된 명세를 통한 검증 외에도 기존에 분류된 오류 카테고리에 따른 검증도 제공해야 한다. 표 1에서와 같은 오류들은 이미 오류로 분류되어 있으며 정형화된 오류들이다. 음수 인덱스 오류의 경우 프로그램에서 음수 값을 갖는 변수나 음수 값으로 직접 배열을 참조할 때 발생되는 오류이다. 범위 초과 인덱스 오류의 경우 배열의 범위보다 큰 값을 갖는 변수로 배열을 참조할 때 발생하는 오류이다. 비생성 객체 참조 오류의 경우 생성자를 통해 객체가 생성될 때 생성된 객체가 널 값을 참조할 때 발생한다. 널 값 참조 오류의 경우는 널 값을 참조하는 변수가 사용될 때 발생한다. 그 밖에도 변수를 캐스팅할 때 변수가 가질 수 없는 타입으로 변환하려고 하면 발생하는 부적합 캐스트 오류나 정수 값을 0으로 나눌 때 발생하는 오류등도 반드시 검증되야 하는 오류들이다. 이러한 오류들을 검증하기 위해서는 오류의 종류를 오류 카테고리에 따라 나누고 사용자나 분석가가 검증하고자 하는 오류만 선택하면 그 오류를 검증하기 위한 명세가 자동으로 생성되도록 하는 것이 필요하다.

이러한 명세를 중간표현에 기술할 수 있어야 한다. 그러므로 중간표현의 언어는 프로그램의 코드를 나타내기 위한 구조 뿐만 아니라 명세를 나타내기 위한 구조도 가지고 있어야 한다. 이를 위해 명령어 뿐만 아니라 오류 분류에 따른 검증 명세나 사용자 명세를 포함하기 위한 구조인 술어 정의, assert, assume 등의 구조와 일차 논리식을 표현할 수 있는 구조들과 검증조건을 생성하기 유리한 술어(predicate)와 같은 구조들도 포함한다.

이렇게 함으로써 사용자의 세분화된 명세나 오류 카테고리로부터 선택된 오류를 나타내기 위해 따로 명세 파일을 마련하지 않고 중간표현 생성기에 의해 생성된 중간표현에 같이 명세할 수 있다. 그림 5의 명령어 중 assert와 assume이 함께 포함됨으로써 프로그램의 의미 부분에 추가적으로 명세부분이 같이 표시한다.

```
ErrorCategory ::==
        verifyError ErrorId in ComputationFunctionId

ErrorId ::== null_ref | array_bound_over | div_0 | ...
ErrorId ::== Id
```

<그림 6> 중간표현의 오류 분류에 따른 명세 부분의 문법

표 1에서 미리 정의된 오류 구분에 따라 명세를 자동으로 생성하기 위해 중간표현의 그림 6과 같은 문법을 이용해 검증하고자 하는 오류를 명세할 수 있다.

```
LogicalFunctionDec ::==
        logicalfunction Id :(TypeList) → Type

LogicalExpr ::== T | F | ArithExpr RelOp ArithExpr
        | ArithExpr BoolOp ArithExpr
        | "¬" LogicalExpr
        | LogicalExpr "⇒" LogicalExpr
        | LogicalExpr "≡" LogicalExpr
        | "forall" TypeIdList "." LogicalExp
        | PredicateId(ParamList)
        | "exists" TypeIdList "." LogicalExpr

PredicateDec ::==
    predicate PredicateId (ParamList) := (LogicalExpr)

PredicateId ::== Id
```

<그림 7> 중간표현의 명세 부분의 문법

미리 정의된 오류뿐만 아니라 사용자가 검증하고자 하는 성질을 명세하기 위해 그림 7의 문법을 사용할 수 있다. 그림 7에는 assert, assume, 그리고 그 밖에 명세

시 기술할 속성을 술어 논리로 나타낼 수 있도록 정의된 논리식의 문법이 나타나 있다.

4.3 검증조건 생성기

오류 및 사용자 명세가 포함된 중간 표현을 검증하기 위해서는 중간 표현의 프로그램 부분과 명세부분의 의미를 반영하여 검증 조건을 생성해야 한다. 검증조건 생성기는 그림 1의 검증조건 생성부분과 같이 전처리기와 검증조건 생성기로 나누어진다.

전처리기는 명세와 프로그램 코드 형태를 포함한 중간 표현을 스캔하고 파싱한다.

VC 생성기는 전처리기가 파싱한 코드를 1차 술어 논리식 형태의 검증조건으로 변환한다. 중간표현에는 논리식과 명령어가 동시에 존재하지만 VC 생성기를 통해 논리식과 명령어 부분을 모두 순수한 논리식으로 변환한다. 중간표현의 논리식으로의 변환은 최약 전조건(weakest precondition) 계산과 유사한 형태[15]로 이루어진다. 생성된 논리식은 검증조건 검증기를 통해 검증된다. 생성된 논리식이 만족하는 것은 프로그램의 실행이 잘못된 상태로 가지 않는다는 것을 의미한다.

4.4 검증조건 검증기

검증조건 검증기는 그림 1과 같이 전처리기, 검증엔진, 검증결과보고로 나누어진다.

전처리기는 검증조건 생성기로부터 생성된 1차 술어 논리식 형태의 검증조건 파일을 스캔하고 파싱한다. 파싱된 검증조건은 검증엔진을 통해 검증 조건 해결에 가장 효율적인 해결기와 이론을 고려하여 해결기에 맞게 변환된다. 이는 해결기마다 제공하는 이론이 다르고 해결기 마다 효율적으로 만족가능성을 결정할 수 있는 이론이 다르기 때문이다.

검증엔진은 해결기를 이용하기 위해 검증조건을 SMT-LIB 파일 형태로 변환하거나 해결기가 제공하는 라이브러리를 이용하게 된다. 해결기를 통해 검증조건을 검증하게 되면 검증조건이 만족하는 경우 프로그램이 명세에 맞게 실행된다는 것을 보장할 수 있다. 만족하지 않게 되는 경우에 해결기는 반례(counter-example)을 제공하게 되는데 검증엔진은 해결기가 제공하는 반례를 통해 프로그램의 어떤 부분에서 명세에 맞지 않는 실행을 수행하는지 알 수 있다. 검증엔진이 사용하는 해결기는 현재로서 CVC3[16], Z3[9], Yices[17]가 고려되고 있다.

결과보고기는 해결기가 제공하는 만족가능 여부, 반례 등의 정보를 사용자나 분석가가 알기 쉽도록 결과를 나타낸다.

V. 결론 및 향후 연구

본 논문에서는 자바 바이트코드의 정적 검증을 수행하는 프레임워크를 설계하였다. 이 프레임워크를 사용함으로써 사전에 정해진 오류 분류에 따라 프로그램의 안전성을 검증할 수 있고 필요한 경우 사용자 명세를 통해 추가적으로 검증하고자 하는 성질을 만족하는지 확인할 수 있다.

프레임워크 설계를 위해 프레임워크를 크게 세 부분으로 구성하였다. 중간표현 생성기 부분에서는 자바 바이트코드가 스택 머신에 기반을 두기 때문에 발생하는 단점을 해결하기 위해 자바 바이트코드를 피연산자 스택을 사용하지 않는 코드로 변환한다. 사용자가 명세나 분류된 검증하고자 하는 오류의 선택을 변환된 중간표현에 중간표현식에 반영하고자 할 때, 중간표현식을 수정할 수 있다. 이러한 명세가 반영된 중간코드를 검증조건 생성기를 거쳐 검증조건이 생성된다. 생성된 검증조건은 SMT 해결기를 증명기로 사용하는 검증조건 검증기를 통해 검증된다. 이를 통해 프로그램이 명세한 성질을 만족하는지 만족하지 않는지에 대한 결과가 나오며 만족하지 않을 때에는 반례가 생성된다.

향후 연구과제로는 설계된 자바 바이트코드 검증 프

레임워크의 구현이 필요하다. 추가적으로 중간코드의 효율적 생성, 검증조건의 최적화, 효율적인 SMT 해결기의 사용, 그래픽 사용자 인터페이스 제공 등이 필요하다.

사사(Acknowledgement)

이 논문은 2010년도 정부(교육과학기술부)의 재원으로 한국연구재단의 기초연구사업 지원을 받아 수행된 것임 (No. 2010-0025660).

참고문헌

[1] James, P. R. and Chalin, P., "ESC4: a modern caching ESC for Java," In Proceedings of the 8th international workshop on Specification and verification of component-based systems (SAVCBS '09), ACM, New York, NY, USA, 2009, pp. 19-26.

[2] Spoto, F., "Julia: A Generic Static Analyser for the Java Bytecode," In Proc. of the 7th Workshop on Formal Techniques for Java-like Programs, FTfJP'2005, Glasgow, Scotland, July 2005.

[3] Necula, G. C., "Proof-carrying code," In Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '97). ACM, New York, NY, USA, pp. 106-119.

[4] Gary T. Leavens. 2007. Tutorial on JML, the java modeling language. In Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering (ASE '07). ACM, New York, NY, USA, pp. 573-573.

[5] Barthe, G., et al., "JACK: a tool for validation of security and behaviour of Java applications," In Proceedings of the 5th international conference on Formal methods for components and objects (FMCO'06), Springer-Verlag, Berlin, Heidelberg, 2007, pp. 152-174.

[6] Barnett, M., et al., "The Spec# Programming System: Challenges and Directions," In Verified Software: Theories, Tools, Experiments, Lecture Notes In Computer Science, Vol. 4171. Springer-Verlag, Berlin, Heidelberg, 2005, pp. 144-152.

[7] DeLine, R. and Leino, K. R. M., "BoogiePL: A typed procedural language for checking object-oriented programs," Technical Report MSR-TR-2005-70, Microsoft Research, 2005.

[8] De Moura, L. and Bjørner, N., "Z3: an efficient SMT solver," Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems, March 29-April 06, 2008.

[9] Ahrendt, W., et al., "KeY: a formal method for object-oriented systems," In Proceedings of the 9th IFIP WG 6. 1 international conference on Formal methods for open object-based distributed systems (FMOODS'07), Springer-Verlag, Berlin, Heidelberg, 2007, pp. 32-43.

[10] DeLine, R. and Leino, K. R. M., "BoogiePL: A typed procedural language for checking object-oriented programs," Technical Report MSR-TR-2005-70, Microsoft Research, 2005.

[11] Aydemir, B., Bohannon, A., and Weirich, S., "*Nominal Reasoning Techniques in Coq*," Electronic Notes in Theoretical Computer Science, Vol. 174, No. 5, 2007, pp. 69-77.

[12] Graf, S. and Saidi, H., "Construction of Abstract State Graphs with PVS," Proceedings of the 9th International Conference on Computer Aided

Verification, June 22-25, 1997, pp. 72-83.

[13] Nipkow, T., Paulson, L. C., and Wenzel, M., "Isabelle/HOL," LNCS, Vol. 2283. Springer, Heidelberg, 2002.

[14] Dijkstra, E. W., "Guarded commands, nondeterminacy and formal derivation of programs," Communications of the ACM, Vol. 18, No. 8, Aug. 1975, pp. 453-457.

[15] Barnett, M. and Leino, K. R. M., "Weakest-precondition of unstructured programs," Proceedings of the 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, Lisbon, Portugal, September 05-06, 2005.

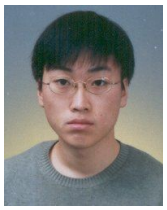
[16] Barrett, C. and Tinelli, C., "CVC3," Proceedings of the 19th international conference on Computer aided verification, Berlin, Germany, July 03-07, 2007.

[17] Dutertre, B. and De Moura, L., "The yices smt solver," Technical report, SRI International, 2006.

[18] 노시춘 · 성중안, "정보보호 기능구조 아키텍처 설계방법," 디지털산업정보학회, 제3권, 제4호, 2007, pp. 65-73.

[19] 김제민 · 김기태 · 유원희, "Mini x86 어셈블리어에서 보안 정보 흐름 분석," 디지털산업정보학회, 제5권, 제3호, 2009, pp. 87-98.

■ 저자소개 ■



김 제 민
Kim, Je Min

2008년 3월~현재 인하대학교 컴퓨터정보공학과 박사과정
2008년 2월 인하대학교 정보공학과(공학석사)
2006년 2월 인하대학교 컴퓨터공학부(공학사)

관심분야 : 프로그래밍 언어, 컴파일러, 프로그램 분석
E-mail : jeminya@hanmail.net

박 준 석
Park, Joon Seok

2006년 3월~현재 인하대학교 컴퓨터정보공학부 조교수
2004년 8월~2006년 삼성전자 SoC 연구소
2004년 7월 미국 남가주대학교 컴퓨터과학과 (공학박사)
2000년 2월 미국 남가주대학교 컴퓨터과학과 (공학석사)

관심분야 : 고성능 컴퓨팅, 병렬 컴파일러, 컴퓨터 구조
E-mail : joonseok@inha.ac.kr

유 원 희
Yoo, Weon Hee

1979년 ~현재 인하대학교 컴퓨터정보공학부 교수
1985년 2월 서울대학교 계산학과(이학박사)
1978년 2월 서울대학교 계산학과(이학석사)
1975년 2월 서울대학교 응용수학과(이학사)

관심분야 : 컴파일러, 프로그래밍언어, 병렬시스템
E-mail : whyoo@inha.ac.kr

논문접수일 : 2011년 4월 18일
수 정 일 : 2011년 5월 9일(1차), 5월 20일(2차)
게재확정일 : 2011년 5월 26일