Journal of The Korea Society of Computer and Information
Vol. 17 No. 8 , August 2012

2012-17-8-1-7

# 자바 바이트코드 검증을 위해 기본경로를 통한 BIRS 코드로부터 검증조건 생성

김 제 민*, 김 선 태*, 박 준 석**, 유 원 희**

# Generating Verification Conditions from BIRS Code using Basic Paths for Java Bytecode Verification

Jemin Kim*, Seontae Kim*, Joonseok Park*, Weonhee Yoo*

## 요 약

BIRS는 자바 프로그램의 검증을 위해 사용되는 중간표현 언어이다[1]. 바이트코드 형태의 자바 프로그램은 검증을 위해 BIRS 코드로 변환된다. 변환된 BIRS 형태의 프로그램을 검증하기 위해서는 BIRS 형태의 프로그램에 대한 검증조건을 생성해야 한다. 본 논문에서는 BIRS 코드에 대한 검증조건을 생성하는 방법을 제시한다. 검증조건 생성은 BIRS 코드에 대한 제어흐름그래프 구성, 제어흐름그래프에 대한 깊이우선 탐색을 통한 기본경로 추출, 기본경로에 대한 최약 전조건(weakest precondition) 계산법의 과정을 통해 이루어진다.

▸ Keywords : BIRS, 검증조건, 최약 전조건

## Abstract

BIRS is an intermediate representation for verifying Java program. Java program in the form of bytecode could be translated into BIRS code. Verification conditions are generated from the BIRS code to verify the program. We propose a method generating verification conditions for BIRS code. Generating verification conditions is composed of constructing control flow graph for BIRS code, depth first searching for the control flow graph to generate basic paths, and calculating weakest preconditions of the basic paths.

▸ Keywords : BIRS, Verification conditions, Weakest preconditions

• 제1저자 : 김제민 • 교신저자 : 김제민
• 투고일 : 2012. 5. 16, 심사일 : 2012. 6. 17, 게재확정일 : 2012. 7. 3.
* 인하대학교 컴퓨터 · 정보공학과(Dept. of Computer and Information Engineering, Inha University)
** 인하대학교 컴퓨터정보공학부(Dept. of Computer Science and Information Engineering, Inha University)

```
1: package main;

2: public class Search{
3: //@ requires 0 <= l && u < a.length;
4: //@ ensures ₩result == (₩exists int j; l<=j && j<=u && a[j]==e);
5: public boolean linearSearch(int[] a, int l, int u, int e){
6:  //@ loop_invariant l<=i && (₩forall int j; l<=j && j<i ==> a[j]!=e);
7:             for(int i=l; i<=u; i=i+1){
8:                 if(a[i]==e)
9:                  return true;
10:            }
11:            return false;
12:    }
13:}
```

그림 1. linearSearch 메소드
Fig. 1. linearSearch Method

## I. 서 론

프로그램 검증 방법 중 Floyd-Hoare 방법[2,3]은 명세와 참 거짓 자동 판별기를 이용하는 방법이다. 프로그래머는 기본적인 명령형 언어로 작성된 코드에 대해 검증하고자 하는 성질을 명세한다. 프로그래머는 Floyd-Hoare 방법을 통해 명세된 성질이 프로그램이 실행 시에 만족하는지 검사할 수 있다.

Floyd-Hoare 방법의 일반적인 검증 과정은 명세와 대상 프로그래밍 언어의 검증조건으로의 변환, 변환된 검증조건의 유효성 검사로 이루어진다. 프로그래머는 검증의 대상이 되는 언어에 프로그램이 실행 시 만족해야하는 성질을 특정 명세언어로 표기한다. 검증조건 생성기는 명세가 표기된 프로그램 코드를 이용해 검증조건을 생성한다. 생성된 검증조건을 DPLL, SAT, SMT-solver와 같은 참 거짓 자동 판별기(decision procedure)를 통해 유효한지 아닌지 판단한다. 생성된 검증조건이 유효한 경우 프로그램이 실행 시에 명세에 맞게 실행된다는 것을 보장한다. 하지만 유효하지 않은 경우 프로그램이 명세에 맞지 않게 실행된다는 것을 의미한다. 검증도구 ESC/Java[4]의 경우 대상언어는 자바이고 명세는 JML(Java Modeling Language)[5]을 통해 표기하며 참 거짓 자동 판별기로서 Simplify[6]를 사용한다.

검증조건 생성을 통한 검증 방법은 Floyd와 Hoare의 방법 이후로 많은 발전을 이루었다. Floyd-Hoare의 방법을 통해 다양한 프로그래밍 언어를 대상으로 한 검증 도구가 존재한다. 대표적인 검증도구는 ESC/Java, Java Applet Correctness Kit(JACK)[7], spec#[8] 등이 있다.

Floyd-Hoare 방법의 대상은 대부분이 고급언어지만 저급언어를 대상으로 한 도구도 필요하다. 왜냐하면 사용자에게 배포되는 대부분의 소프트웨어는 소스코드를 포함하지 않기 때문이다. 자바로 작성된 소프트웨어의 경우 바이트코드 형태로 되어 있기 때문에 바이트코드를 대상으로 하는 도구가 필요하다.

바이트코드를 대상으로 하는 도구로는 JBVF(Java Bytecode Verification Framework)[9]가 존재한다. JBVF에서는 저급언어인 자바 바이트코드를 대상으로 검증을 수행할 수 있다. JBVF에서의 검증과정은 다음과 같다. JBVF에서는 바이트코드 형태의 클래스 파일을 읽어 명세가 가능하며 스택머신 코드가 아닌 BIRS(Bytecode Intermediate Representation with Specifications) 코드로 변환한다. 변환된 BIRS 코드에 분석가는 여러 가지 명세를 표기할 수 있다. JBVF의 검증조건 생성기는 BIRS 코드를 읽어 검증조건을 생성한다. 생성된 검증조건은 검증조건 검증기를 통해 검증된다.

본 논문에서는 JBVF의 여러 과정 중에서 BIRS 코드로부터 검증조건을 생성하는 방법을 제시한다. JBVF에서 BIRS로부터 검증조건이 생성되는 과정은 다음과 같다. BIRS 코드로부터 제어흐름 그래프를 생성한다. 생성된 제어흐름 그래프를 대상으로 기본 경로를 추출해 낸다. 추출된 기본경로를 대상으로 최약 전조건 계산법을 이용해 검증조건을 생성한다.

본 논문의 구성은 다음과 같다. 2장에서는 BIRS 코드로부터 검증조건이 생성되는 과정을 설명한다. 이를 위해 선형 탐색을 하는 BIRS 코드를 대상으로 제어흐름그래프를 생성하는 과정을 보인다. 또한 생성된 제어흐름그래프로부터 기본경로를 생성하고 기본경로로부터 검증조건을 생성하는 과정을 보인다. 3장에서는 BIRS 코드로부터 제어흐름그래프를 구성

하고 기본경로를 추출하는 방법과, 추출된 기본경로로부터 검증조건을 생성하는 방법을 정형적으로 나타낸다. 4장에서는 관련연구에 대해 언급하고 5장에서 본 논문의 결론과 향후연구를 통해 논문을 마무리한다.

## II. 예 제

BML(Bytecode Modeling Language)[10] 형태를 갖는 클래스파일은 명세정보를 포함할 수 있다. 기본적으로 자바 바이트코드에는 명세를 하는 부분이 존재하지 않는다. 또한 클래스 파일 포맷에도 명세를 하는 부분이 포함되어 있지 않다. 하지만 클래스 파일에는 추가적인 정보를 기술할 수 있는 구조를 가지고 있다. 클래스 파일에서 추가적인 정보를 포함할 수 있는 구조를 고려하여 자바 바이트코드도 명세정보를 포함할 수 있도록 확장한 형태가 BML이다. 따라서 BML 형태로 되어있는 클래스파일에는 명세가 포함되어 있다.

설명의 편의를 위해 본 논문에서는 BML코드로부터 변환된 BIRS 코드를 예로서 사용한다. 그림 1에 나타난 JML 코드가 포함된 자바소스코드로부터 BML를 생성하고 생성된 BML로부터 JBVF를 통해 그림 2의 BIRS 코드를 생성한다. 따라서 그림 1과 그림 2의 프로그램은 동일한 의미를 갖는다. 물론 BML로부터 변환된 BIRS 코드를 대상으로 하지 않고 명세가 포함되지 않은 BIRS 코드에 사용자가 직접 명세하고자 하는 성질을 표기한 후 검증조건을 생성하는 작업을 수행할 수도 있다.

그림 1의 예제 코드는 배열의 인덱스 l부터 u까지를 탐색하여 원소 e를 찾는 linearSearch 메소드이다. linearSearch 메소드가 호출되기 위해서는 l은 0보다 크거나 같아야 하고 u는 배열 a의 길이보다 작아야 한다. 이러한 성질을 나타내기 위해 그림 1의 3번 행에서 requires 절을 통해 함수가 호출되기 전에 만족해야 하는 성질을 나타내고 있다. linearSearch 메소드가 호출된 후의 결과값은 l과 u사이의 배열 a에서 원소 e를 찾았을 경우 true이고 찾지 못했을 경우 false이다. 이러한 성질을 나타내기 위해 그림 1의 4번 행에서 ensures 절을 통해 함수가 호출되고 난 후 만족해야 하는 성질을 나타내고 있다. linearSearch 메소드의 for 문을 반복하는 동안 인덱스 변수 i는 l보다 크거나 같아야 하며 l과 i사이의 배열의 원소는 찾고자 하는 원소 e가 아님을 보장해야 한다. 이러한 성질을 나타내기 위해 그림 1의 6번 행에서 루프 불변자를 나타내는 명세를 표기하고 있다.

JBVF를 이용해 그림 1과 동일한 의미의 BIRS 코드가 생성된다. 그림 2에 생성된 BIRS 코드가 나타나 있다. 생성된

```
Class main.Search: (java.lang.Object ) () ()
{computationalFunction linearSearch
( java.lang.Object local0, arrint local1, int local2,
  int local3, int local4 ) -> bool
require ((0 <= local2) && (local3 < local1.length))
ensure (result == (Exist int j. (((local2 <= j) &&
  (j <= local3)) && (local1[j] == local4))))

block 7 is
 7: vretrun 0
end

block 5 is
 5: loopInvariant ((local2 <= local5) &&
     (ForAll int j. (((local2 <= j) &&
     (j < local5)) ==> not((local1[j] == local4)))))
 6: ifd(local5 <= local3) 2
end

block 4 is
 4: assignVal local5 := (local5 + 1)
end

block 3 is
 3: vretrun 1
end

block 2 is
 2: ifd(local1[local5] <> local4) 4
end

block 0 is
 0: assignVal local5 := local2
 1: goto 5
end
block 5 ===> block 7, not local5 <= local3
block 5 ===> block 2, local5 <= local3
block 4 ===> block 5,
block 2 ===> block 4, local1[local5] <> local4
block 2 ===> block 3, not local1[local5] <> local4
block 0 ===> block 5,
returnblock 7 3
end
}
```

그림 2. 그림 1의 BIRS 코드
Fig. 2. BIRS Code of Fig. 1

BIRS 코드는 저급언어로서 goto 문을 가지며 제어구조가 그림1의 자바 코드에 비해 명확하지 않다. 하지만 BIRS 코드는 바이트코드와 달리 연산자 스택머신에 기반을 둔 코드가 아니며 제어흐름이 명확하게 나타난다. 그림 2의 BIRS코드의 require 절과 ensure 절 부분은 그림 1의 requires 절과

ensures 절에 대응하며, 그림 2의 블록 7의 루프불변자 부분은 그림 1의 루프 불변자에 대응한다. 이제 그림 2의 BIRS 코드를 대상으로 검증조건을 생성해야 한다. BIRS 코드를 대상으로 검증조건을 생성하는 것은 기본경로를 추출하는 단계, 추출된 기본경로들에 대해 최약 전조건 계산법을 적용함으로써 검증조건을 생성하는 단계로 나눌 수 있다.

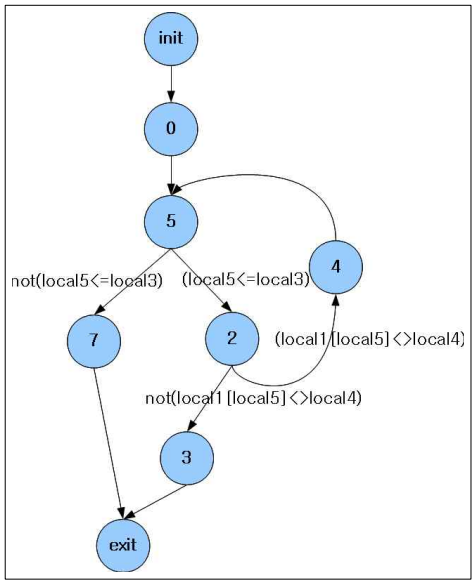


그림 3. linearSearch 메소드의 제어흐름그래프
Fig. 3. Control Flow Graph of linearSearch Method

## 1. linearSearch로부터 기본경로 추출

그림 2의 BIRS 코드를 대상으로 기본경로를 추출하기 위해서는 제어흐름그래프를 구성해야 한다. BIRS 코드는 코드를 제어흐름에 따라 기본블록으로 나누고 기본블록간에 발생하는 제어의 흐름을 코드상에 나타내기 때문에 제어흐름 그래프를 구성하기 용이하게 되어있다. BIRS 코드에서 제공되는 블록 정보와 제어 흐름 정보를 토대로 구성되는 그림 2의 BIRS 코드의 제어흐름그래프는 그림 3과 같다.

그림 3의 제어흐름 그래프는 기본경로 추출에 대해 설명시 설명이 복잡해지는 것을 막기 위해 코드를 생략한다. 그림 3의 제어흐름그래프에서 각 노드의 번호는 그림 2의 기본블록의 번호를 나타낸다. 그리고 ensure 절과 requires 절을 나타내기 위해 init 블록과 exit 블록을 추가한다. 그림 2의 블록 5와 블록 2에 분기문인 if문을 포함하고 있기 때문에 제어흐름그래프의 노드 5과 노드 2에서는 분기가 발생하며 각 간선에는 분기가 발생하여 실행되기 위해 만족해야하는 조건을 나타낸다. 노드 5, 노드 2, 노드 4에서 루프가 발생하는데 이는 그림 1의 코드의 while 문으로부터 변환된 것을 나타낸다.

생성된 제어흐름그래프로부터 기본경로를 추출하기 위해서는 기본경로의 시작점과 끝점을 찾아야 한다. 기본경로의 시작점은 함수 전조건(requires 절)이나 루프 불변자일 수 있고 기본경로의 끝점은 루프 불변자, 단언(assert), 또는 함수 후조건(ensures 절)이 될 수 있다.

기본 경로의 시작점과 끝점의 후보를 대상으로 제어흐름그래프의 init 노드부터 깊이우선탐색(Depth First Search)를 수행하여 기본 경로의 끝점을 만나면 기본 경로하나를 반환하고 새로운 시작점의 후보로부터 끝점을 찾아서 기

| | |
|---|---|
| (a) | PRE ((0 <= local2) && (local3 < local1.length))<br>local5 := local2<br>LOOPINV ((local2 <= local5) && (ForAll int j. (((local2 <= j) && (j < local5)) ==> not((local1[j] = local4))))) |
| (b) | LOOPINV ((local2 <= local5) && (ForAll int j. (((local2 <= j) && (j < local5)) ==> not((local1[j] = local4)))))<br>ASSUME not((local5 <= local3))<br>return_value := 0<br>POST (result = (Exist int j. (((local2 <= j) && (j <= local3)) && (local1[j] = local4)))) |
| (c) | LOOPINV ((local2 <= local5) && (ForAll int j. (((local2 <= j) && (j < local5)) ==> not((local1[j] = local4)))))<br>ASSUME (local5 <= local3)<br>ASSUME (local1[local5] <> local4)<br>local5 := (local5 + 1)<br>LOOPINV ((local2 <= local5) && (ForAll int j. (((local2 <= j) && (j < local5)) ==> not((local1[j] = local4))))) |
| (d) | LOOPINV ((local2 <= local5) && (ForAll int j. (((local2 <= j) && (j < local5)) ==> not((local1[j] = local4)))))<br>ASSUME (local5 <= local3)<br>ASSUME not((local1[local5] <> local4))<br>return_value := 1<br>POST (result = (Exist int j. (((local2 <= j) && (j <= local3)) && (local1[j] = local4)))) |

그림 4. 그림 2의 BIRS 코드로부터 생성된 기본경로
Fig. 4. Basic Paths Generated from BIRS Code of Fig. 2

본경로를 반환할 때까지 깊이우선 탐색을 반복한다. 깊이우선 탐색과정에서 간선에 나타나 있는 조건에 대해서는 assume 문으로 변환함으로써 제어흐름이 발생하는 조건을 나타낸다. 모든 기본경로를 찾아내면 기본경로를 추출하는 과정이 마무리된다.

그림 2에서 기본경로의 시작점은 requires 절과 블록 5의 루프 불변자다. 기본경로의 끝점은 ensures절과 블록 5의 루프 불변자이다. init 블록부터 깊이 우선 탐색을 수행하면 블록 5의 루프 불변자까지의 기본경로가 하나 추출된다. 이 기본경로는 그림 3의 init, 0, 5 노드를 나타내며, 그림 4의 (a)에 해당한다. 이제 루프 불변자로부터 ensure절이 있는 exit 블록까지의 기본경로가 추출된다. 이 기본경로는 그림 3의 5, 7, exit 노드를 나타내며, 그림 4의 (b)에 해당한다. 기본경로 (b)가 생성될 때에, 그림 3의 노드 5와 노드 7 사이의 간선에 나타난 조건 not(local5<=local3)은 그림 4의 (b)의 assume 문으로 대체되고, exit 블록의 ensures 절은 POST 문으로 대체된다. 그림 3의 5, 2, 4, 5 노드를 이루는 기본경로는 그림 4의 (c)에 해당하며, 그림 3의 5, 2, 3, exit 노드를 이루는 기본경로는 그림 4의 (d)에 해당한다. 기본 경로 추출방법을 이용하여 그림 2의 코드로부터 기본경로를 추출하면 그림 4와 같다.

### 2. 기본경로부터 검증조건 생성하기

추출된 기본경로로부터 검증조건을 생성하는 과정은 최약전조건 계산방법을 통해 이루어진다. 기본경로에서 마지막 문장을 후조건으로 두고 마지막에서 두 번째 문장부터 후조건을 만족하는 전조건을 계산한다. 마지막 문장부터 첫 문장까지 순차적으로 최약 전조건을 계산함으로써 하나의 기본경로에 대해 하나의 검증조건을 생성할 수 있다.

그림 4의 기본경로에 대해 최약 전조건 계산방법을 적용하여 나온 결과는 그림 5와 같다. 그림 4와 그림 5의 각 기본경로와 검증조건은 대응된다. 예를 들어, 그림 4의 기본경로 (a)는 PRE 문, 배정문, LOOPINV 문 순으로 나열되어 있는데, 그림 5의 검증조건 (a)와 대응된다. 그림 4의 (a)에 대한 검증조건의 생성과정은 LOOPINV 문부터 역순으로 진행된다. 우선 마지막 문장인 LOOPINV에 대해 LOOPINV 문의 표현식 부분만이 검증조건으로 고려된다. 그 다음 배정문 local5:=local2에 대해 전 과정에서 나온 표현식의 local5 변수를 local2 변수로 치환한다. 그 결과 LOOPINV 문의 표현식 부분 중 local5가 local2로 치환된 결과가 검증조건으로 나온다. 이제 PRE 문에 대해 PRE 문의 표현식 부분과 전 과정에서 생성된 검증조건을 암시(==>)로 연결하면 그림 5의 검증조건 (a)가 결과로 나타난다.

각 기본경로에 대해 나타나는 검증조건을 모두 만족한다면 그림 2의 BIRS 코드는 명세를 한 프로그래머의 의도에 맞게 실행된다는 것을 보장한다. 최종적으로 검증조건은 술어논리식 형태로 나타난다. 술어논리식 형태로 나타난 검증조건에 대해 SMT-solver와 같은 참거짓 자동 판별기를 적용하면 생성된 기본경로에 대한 오류여부를 파악할 수 있다.

## III. BIRS 코드로부터 검증조건 생성

### 1. BIRS 문법

BIRS 코드는 제어흐름을 나타내기 쉽도록 메소드를 블록 단위로 나누며 블록간의 제어흐름을 명시적으로 나타낸다. 따라서 BIRS 코드를 통한 제어흐름 분석이 용이한 형태이다. 본 논문에서는 검증조건을 생성과정을 설명하기 위해 BIRS의 문법 중 일부분만 고려한다. 본 논문에서 다루는 BIRS 코드의 문법은 다음과 같다.

```
Method ::= requires Expr Body ensures Expr
Body ::= Block+ Flow*
Block ::= BlockNum: Instr*
Instr ::= VarId := Expr | assert Expr
```

| | |
|---|---|
| (a) | (((0 <= local2) && (local3 < local1.length)) ==> ((local2 <= local2) && (ForAll int j. (((local2 <= j) && (j < local2)) ==> not((local1[j] = local4)))))) |
| (b) | (((local2 <= local5) && (ForAll int j. (((local2 <= j) && (j < local5)) ==> not((local1[j] = local4))))) ==> (not((local5 <= local3)) ==> (result = (Exist int j. (((local2 <= j) && (j <= local3)) && (local1[j] = local4)))))) |
| (c) | (((local2 <= local5) && (ForAll int j. (((local2 <= j) && (j < local5)) ==> not((local1[j] = local4))))) ==> ((local5 <= local3) ==> ((local1[local5] <> local4) ==> ((local2 <= (local5 + 1)) && (ForAll int j. (((local2 <= j) && (j < (local5 + 1))) ==> not((local1[j] = local4)))))))) |
| (d) | (((local2 <= local5) && (ForAll int j. (((local2 <= j) && (j < local5)) ==> not((local1[j] = local4))))) ==> ((local5 <= local3) ==> (not((local1[local5] <> local4)) ==> (result = (Exist int j. (((local2 <= j) && (j <= local3)) && (local1[j] = local4))))))) |

그림 5. 그림 4의 기본경로로부터 생성된 검증조건
Fig. 5. Verification Conditions Generated from Basic Paths of Fig. 4

| assume Expr | loopInvariant Expr
Flow ::= BlockNum1 ==> BlockNum2 when Expr

Method는 BIRS의 메소드를 나타낸다. Method는 메소드의 전조건인 require 문과, 메소드의 명령어부분인 Body, 메소드의 후조건인 ensures 문으로 구성된다. Body는 여러개의 Block과 Flow로 구성된다. Block은 기본 블록을 나타내며 블록의 번호를 나타내는 BlockNum과 여러개의 Instr을 포함한다. Instr은 명령어를 나타낸다. 명령어는 배정, assert, assume, loopInvariant 중 하나이다. Flow는 기본블록간의 흐름으로서 BlockNum1 블록으로부터 BlockNum2 블록으로의 흐름을 나타낸다. Expr은 표현식을 나타내며 산술연산식, 불리언 연산식 등을 포함한다.

## 2. 제어흐름 그래프

BIRS 코드는 제어흐름그래프 $G = B \times E \times B_{init} \times B_{exit}$로 나타낼 수 있다. B는 제어흐름그래프의 기본블록의 집합을 나타낸다. E는 제어흐름그래프의 간선으로서 기본블록간의 제어흐름을 나타낸다. 간선은 조건간선과 일반간선으로 나뉜다. 조건 간선은 분기를 포함하는 기본블록으로부터 나온 간선으로서 분기의 조건을 나타낸다. 일반간선은 특정 조건없이 순차적 제어의 흐름을 나타낸다. $B_{init}$ 는 제어흐름의 시작을, $B_{exit}$는 제어흐름의 끝을 나타낸다.

## 3. 기본경로 추출

제어흐름그래프를 바탕으로 기본경로를 추출해야 한다. 기본경로는 명령어의 나열이다. 기본경로의 시작 명령어는 함수 전조건, 루프 불변자 중 하나여야 한다. 기본경로의 끝 명령어는 루프 불변자, assert 문, 함수 후조건 중 하나여야 한다. 제어흐름그래프의 위치 중 기본경로를 추출할 수 있는 시작명령어와 끝명령어의 조건을 만족하는 위치를 절단점(cutpoint)라 한다.

제어흐름그래프에서 기본경로를 추출하는 방법은 알고리즘 1과 같다. 기본 경로를 구하기 위해서는 제어흐름그래프의 진입노드인 $B_{init}$부터 깊이 우선 탐색을 수행해야 한다. $B_{init}$ 노드는 함수 전조건을 포함하기 때문에 절단점에 해당한다. $B_{init}$ 부터 시작하여 그래프를 탐색하면서 절단점에 도달했을 때에 절단점까지 찾아낸 명령어 나열을 기본경로로서 인식해야 한다. 절단점의 명령어가 루프불변자인 경우 기본경로로 인식하고 루프불변자를 새로운 기본경로의 시작점으로 간주하여 새로운 기본경로를 인식할 수 있도록 탐색을 수행한다. 도달한 절단점이 assert 문일 경우 절단점까지 찾아낸 명령어 나열에 assert 문을 추가하여 하나의 기본경로로서 인식하고, 찾아낸 명령어 나열을 가지고 추가적으로 탐색을 수행해야 한다. 그래프에서 루프가 발생할 수 있기 때문에 현재 노드가 처음 방문한 노드일 경우 표시를 하고 선행자 노드에 대한 탐색을 수행한다. 이 때 현재 노드와 선행자 노드를 잇는 간선이 조건간선일 경우 assume 문을 기본경로의 끝에 추가한다. 알고리즘 1을 적용하면 제어흐름 그래프로부터 기본경로를 추출할 수 있다.

## 4. 검증조건 생성

추출한 기본경로는 BIRS 명령어의 나열이다. 명령어의 나열 중 끝에 있는 명령어부터 최약 전조건 계산법을 통해 검증조건을 생성한다. 기본경로의 정의에 따라 기본 경로의 끝은 ensures, assert, loopInvariant 중 하나이다. 기본경로의 끝의 명령어의 표현식을 후조건으로 두고 기본경로의 나머지 부분부터 최약 전조건 계산함수 wp를 적용하여 기본경로로부터 검증조건을 생성한다. 기본경로 P에 대해 검증조건을 생성하는 함수 vc는 다음과 같다. 기본 경로 $P = Ins_1; Ins_2; ...; Ins_{n-1} ; Ins_n$ 일 경우:

$$vc(P) = wp(Expr, Ins_1; Ins_2; ...; Ins_{n-1})$$

여기서 $Ins_i$는 ensures Expr 또는 assert Expr 또는 loopInvariant Expr 중 하나이다.

후조건으로부터 명령어에 대한 최약 전조건을 계산하는 함수 wp는 표 1과 같다. assume Expr 명령어는 후조건 Q에 대해 Expr이 참일 경우에만 Q가 참인지 거짓인지 중요하므로 검증조건으로 변환하면 Expr ==> Q와 같다. assert Expr 명령어는 Expr이 참이고 후조건 Q도 참이어야 하므로 Expr && Q로 변환된다. 명령어의 나열 $Ins_1; ... ; Ins_n$의 경우 후조건 Q에 대해 명령어나열의 끝인 Insn에 대한 최약 전조건이 $Ins_n$을 제외한 명령어나열의 후조건이 되어 최약 전조건을 구한다. 배정문 VarId := Expr의 경우 후조건 Q의 VarId를 Expr로 치환함으로써 최약 전조건을 구한다.

표 1. 최약 전조건 계산함수 wp
Table1. Weakest Precondition Function wp

| inst | Q | wp(inst, Q) |
|---|---|---|
| assume Expr | Q | Expr ==> Q |
| assert Expr | Q | Expr && Q |
| requires Expr | Q | Expr ==> Q |
| $Ins_1;... ; Ins_n$ | Q | $wp(Ins_1;...Ins_{n-1}, wp(Ins_n,Q))$ |
| VarId := Expr | Q[VarId] | Q[Expr] |

알고리즘 1. 제어흐름 그래프로부터 기본경로 추출
Algorithm 1. Extracting basic paths from a control flow graph

```
INPUT: G = B × E × B_init × B_exit
OUTPUT : Basic Paths
Method: paths := ∅
          traverse(Binit, ε)
          return paths
USING:
function traverse(Node N, Path P)
{
           for i=1 to N.length do
             if N[i] is loop invariant then
                  P := P ; N[i]
                  paths := paths ∪ {P}
                  P := N[i]
             else if N[i] is assert then
                  tmp := P
                  P := P ; N[i]
                  paths := paths ∪ {P}
                  P := tmp
             else
                  P := P ; N[i]
          if N is not marked then
             N.marked := true
             edges := N.succ
             for all e in edges do
                  if e is conditional edge then
                       P := P ; assume e.cond
                  Succ := e.head
                  traverse(Succ, P)
}
```

## IV. 관련연구

프로그램 소스코드에 프로그램이 실행시 만족해야 하는 명세를 기술하고 정형적인 방법으로 검증하는 방법은 Floyd-Hoare의 방법 [2, 3] 이후로 다양한 언어를 대상으로 연구되고 있다.

Eiffel[11] 프로그래밍 언어의 경우 Design by Contract[12]에 의해 프로그램 언어 구조내에 명세를 할 수 있는 구조를 포함한다. Effiel 언어의 경우 언어의 초기 설계부터 Design by Contract 개념을 기반으로 언어 처리가 구현되었다. 자바의 경우 언어에 명세를 하는 구조가 없기 때문에 주석형태로 명세를 기술할 수 있다. 명세 기술을 위해 JML이 사용된다. JACK[7], ESC/Java[4] 등은 자바와 JML을 이용해 검증조건을 생성하여 프로그램을 검증한다. C#의 경우도 자바와 마찬가지로 명세를 하는 구조를 포함하지 않는다. SPEC#[8]은 이러한 C#의 한계를 해결하기 위해 C#언어에 명세를 기술할 수 있도록 확장한 형태이다. 그 밖에도 SPARK[13]는 Ada 프로그래밍 언어에 명세를 기술할 수 있도록 확장한 형태이다.

고급언어에 대해 직접 검증조건을 생성하지 않고 검증이 용이한 중간언어 코드로 변환한 후 중간언어코드에 대해 검증조건을 생성하는 방법이 있다. SPEC#의 경우 검증을 위한 중간언어인 BoogiePL[14]로 변환 후 검증조건을 생성한다. JBVF의 경우도 자바 바이트코드를 중간언어 형태인 BIRS로 변환한다.

검증조건 생성에 관한연구로는 최약 전조건 계산법[15]을 이용하는 방법과 최강 후조건 계산법을 이용하는 방법이 있다. Flanagan과 Saxe는 루프와 같은 제어구조의 구분이 명백한 구조적인 프로그램에 대한 검증조건을 생성한다[17]. Barnett과 Rustan은 제어구조의 구분이 명백하지 않은 비구조적 프로그램에 대한 검증조건을 생성한다[18]. [17, 18]의 경우 제어흐름 구조에 대한 내용을 반영하기 위해 추가적인 코드를 삽입한다. 이와 달리 제어흐름그래프를 구성한 후 검증조건 생성의 단위가 되는 기본경로를 추출하여 검증조건을 생성하는 방법이 있다[19]. 본 논문은 [17, 18]의 경우와 달리 기본경로를 추출하는 방법을 이용한다. 이렇게 함으로써 추가적인 코드의 삽입이 필요없다.

## V. 결론 및 향후연구

본 논문에서는 자바 바이트코드 형태의 프로그램을 검증하기 위해 기본경로 추출을 통해 BIRS 코드로부터 검증조건을 생성하는 방법을 제시했다. 자바 바이트코드는 검증을 위해 BIRS 코드로 변환된다. 변환된 BIRS 코드에 대해 사용자나 검증전문가는 명세를 기입할 수 있다. 명세가 포함된 BIRS 코드에 대해 제어흐름 그래프를 구성하고 구성된 제어흐름그래프로부터 깊이우선 탐색을 통해 기본경로를 생성한다. 깊이우선탐색시 기본경로의 정의에 따라 기본경로의 시작점과 끝점이 결정된다. 생성된 기본경로에 대해 최약 전조건 계산법을 적용함으로써 프로그램에 대한 검증조건이 생성된다.

BIRS 코드를 이용함으로써 자바 바이트코드에는 기술하기 어려웠던 명세를 프로그램에 포함할 수 있다. 또한 BIRS 코드는 기본경로 추출을 위한 제어흐름그래프 구성에 용이하다. 검증조건 생성과정에서는 기본경로를 이용함으로써 제어흐름그래프 생성 후 루프를 제거하는 방법과 달리 코드를 추가할 필요가 없다.

향후연구로는 생성된 검증조건에 대해 다양한 의사결정 프로시저를 적용하는 방법을 연구할 수 있다. 또한 검증조건에

대한 최적화를 통해 생성되는 검증조건을 최소화하여 검증시에 드는 시간을 단축하는 연구를 수행할 수 있다.

## 사사(Acknowledgement)

이 논문은 2011년도 정부(교육과학기술부)의 재원으로 한국연구재단의 기초연구사업 지원을 받아 수행된 것임(No. 2011-0026495)

이 논문은 인하대학교의 지원에 의하여 연구되었음.

## 참고문헌

[1] Seontae Kim, Jemin Kim, Joonseok Park, and Weonhee Yoo, “Implementation of Stackless Intermediate Representation Language for Java Bytecode,” The Journal of Korean Institute of Information Technology, Vol. 9, No. 9, pp. 129-138, Sep. 2011.

[2] R. W. Floyd, “Assigning meaning to programs,” Mathematical Aspects of Computer Science, Vol. 19, pp. 19-32, 1967.

[3] C. A. R. Hoare, “An axiomatic basis for computer programming,” Communication of the ACM, Vol. 12, No. 10, pp. 576-580, Oct. 1969.

[4] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata, “Extended static checking for Java,” SIGPLAN Not. Vol. 37, No. 5, pp. 234-245, May 2002.

[5] Lilian Burdy, Yoonsik Cheon, David R. Cok, et al. “An overview of JML tools and applications,” International Journal on Software Tools for Technology Transfer, Vol. 7, No. 3, pp. 212-232, June 2005.

[6] David Detlefs, Greg Nelson, and James B. Saxe, “Simplify: a theorem prover for program checking,” Journal of ACM, Vol. 52, No. 3, pp. 365-473, May 2005.

[7] Gilles Barthe, Lilian Burdy, Julien Charles, 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, pp. 152-174, 2007.

[8] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte, “The Spec# Programming System: An Overview,” LNCS, Vol. 3362, 2004.

[9] Jemin Kim, Joonseok Park, Weonhee Yoo, “A Design of Verification Framework for Java Bytecode,” Journal of the Korea Society of Digital Industry and Information Management, Vol. 7, No. 2. pp. 29-37, June 2011.

[10] Lilian Burdy, Marieke Huisman and Mariela Pavlova, “Preliminary Design of BML: A Behavioral Interface Specification Language for Java bytecode,” LNCS, Vol. 4422, pp. 215-229, 2007.

[11] Eiffel Software, http://www.eiffel.com

[12] J. M. Jazequel, B. Meyer, “Design by contract: the lessons of Ariane,” Computer, Vol. 30, No. 1, pp. 129-130, Jan. 1997.

[13] John Barnes, “High Integrity Software: The SPARK Approach to Safety and Security,” Addison Wesley, March 2005.

[14] H. Lehner and P. Müller, “Formal translation of bytecode into BoogiePL,” Electron. Notes Theor. Comput. Sci., Vol. 190, No. 1, pp. 35 - 50, 2007.

[15] E. W. Dijkstra, “A Discipline of Programming,” Prentice Hall, Oct. 1976.

[16] Jimin Kim, Kitae Kim, Jemin Kim, and Weonhee Yoo, “Static Type Inference Based on Static Single Assignment for Bytecode,” Journal of the Korea Society of Computer and Information, Vol. 11, No. 4, pp. 87-96, Sep. 2006.

[17] C. Flanagan and J. B. Saxe, “Avoiding exponential explosion: generating compact verification conditions,” SIGPLAN Not., Vol. 36, No. 3, pp. 193 - 205, 2001.

[18] M. Barnett and Rustan, “Weakest-precondition of unstructured programs,” in PASTE ’05: Proceedings of the 6th ACM SIGPLAN-SIGSOFT

workshop on Program analysis for software tools and engineering, pp. 82 - 87, 2005.

[19] Z. Manna, "Mathematical Theory of Computation," Dover Publications, 2003.

[20] Kyungsoo Kim and Weonhee Yoo, "A Study on Intermediated code for Analyzing Bytecodes," Journal of the Korea Society of Computer and Information, Vol. 11, No. 1, pp. 107-117, Mar. 2006.

## 저 자 소 개

**김 제 민**

2006: 인하대학교 컴퓨터공학부 공학사.
2008: 인하대학교 컴퓨터공학과 공학석사.
2010: 인하대학교 컴퓨터정보공학과 공학박사 수료
현 재: 인하대학교 컴퓨터정보공학과 박사과정
관심분야: 프로그램 분석, 컴파일러
Email : jeminya@daum.net

**김 선 태**

2011: 인하대학교 컴퓨터정보공학과 공학사.
현 재: 인하대학교 컴퓨터정보공학과 석사과정
관심분야: 프로그래밍 언어, 컴파일러, 고성능 컴퓨팅
Email : kst_1205@nate.com

**박 준 석**

2000: 미국 남가주대학교 컴퓨터과학과 공학석사.
2004: 미국 남가주대학교 컴퓨터과학과 공학박사.
2004~2006: 삼성전자 SoC 연구소
2006~현 재: 인하대학교 컴퓨터정보공학부 부교수
관심분야: 고성능 컴퓨팅, 병렬 컴파일러, 컴퓨터 구조
Email : joonseok@inha.ac.kr

**유 원 희**

1975: 서울대학교 응용수학과 이학사.
1978: 서울대학교 계산학과 이학석사.
1985: 서울대학교 계산학과 이학박사
1979~현 재: 인하대학교 컴퓨터정보공학부 교수
관심분야: 컴파일러, 프로그래밍 언어, 병렬시스템
Email : whyoo@inha.ac.kr