중간언어의 VC 생성을 통한 바이트코드 검증

허혜림*, 김제민*, 박준석*, 유원희*
*인하대학교 컴퓨터・정보공학과
e-mail:lena03@naver.com

Verification of Bytecode by Generating Verification Condition for Intermediate Language

Hyerim Hu*, Jemin Kim*, Joonseok Park*, Weonhee Yoo*
Dept of Computer and Information Engineering, Inha University*

요 약

프로그램 신뢰성을 높이기 위한 방법 중 하나로 쓰이는 것이 명세 된 언어의 검증이다. 명세 된 언어를 검증하기 위해 소스 프로그램을 논리식으로 바꾸어 검증하는 방법을 사용한다. 소스 프로그램 뿐만 아니라 바이트코드 역시 프로그램 신뢰성을 높이기 위해서 검증이 필요하다. 본 논문에서는 바이트코드의 검증을 위해 바이트코드의 정보를 가지고 있는 중간언어의 verification condition을 생성하는 방법을 보인다.

1. 서론

프로그램 활용도가 높아질수록 프로그램의 신뢰성도 점점 중요해지고 있다. 프로그램을 실행하기 전 프로그램의 결함을 가능한 많이 발견하여 제거할 수 있다면 그 프로그램의 신뢰성은 높아진다. 프로그램 신뢰성을 높이기 위한 방법은 여러 가지가 있는데 그 중 한 가지 방법이 프로그램 명세를 통한 검증이다[1].

프로그램 신뢰성을 증명하기 위해서는 가능한 많은 오류들을 정적으로 발견할 수 있어야 한다. 일반적으로 실행시간에만 찾을 수 있는 오류들을 추가적으로 정적으로 발견하는 것이 확장된 정적 검증(Extended static checking)[2]의 목표이다. 이 때 사용하는 방법이 verification condition(VC)의 생성을 통한 검증이다. VC 생성을 통한 검증 방법은 소스 프로그램을 논리식으로 변환하여 정리증명기(theorem prover)에 입력으로 넣어 그 결과로 프로그램이 유효한지를 알아보는 방법이다. 프로그램에 문제가 없을 경우 생성된 VC에 대한 결과는 유효하다고 나오고 그렇지 않을 경우 유효하지 않음을 표기하여 주고 정리증명기에 따라서는 반례(counterexample)를 제공해 주기도 한다. 제공된 반례를 이용하여 프로그램의 어떤 위치에 문제가 있는지 확인할 수 있다[3].

소스 프로그램의 검증과 마찬가지로 바이트코드의 검증에 대한 연구 역시 다수 진행 중이다. 자바 언어의 컴파일된 형태인 바이트코드 역시 그 프로그램에 존재하는 문제를 찾아내고 제거하거나 그 프로그램이 정확한지 확인할 필요가 있다. 본 논문에서는 바이트코드의 정확성 검증을 위해 바이트코드를 중간 언어로 변환한 형태를 입력으로 사용한다. 정의된 중간언어를 VC로 생성한 후 생성된 VC를 정리증명기 등을 이용하여 유효성을 검증함으로써 중간언어로 표현된 바이트코드가 유효한지 아닌지를 판단한다. 본 논문에서는 기존의 VC생성 방법[4][5]을 참고하여 중간 코드에 대한 VC생성을 진행한다. 모든 소스 프로그램을 논리식으로 변환하는 것은 현실적으로 무리가 있기 때문에 소스 프로그램 내에서 검증이 필요한 일부를 검증하도록 한다. 검증을 통하여 바이트코드 내의 오류들을 정적으로 가능한 많이 찾아내어 프로그램의 신뢰성을 높일 수 있다.

2. 관련연구

2.1 확장된 정적 검증[2]

프로그램의 신뢰성을 높이기 위해서는 가능한 많은 오류들을 정적으로 검출해야한다. 확장된 정적 검증은 가능한 많은 오류들을 정적으로 잡아내는 것을 목표로 하는 프로그램 검증 방법이다. 이를 수행하기 위하여 명세 된 프로그램을 논리식의 형태로 바꾸어 검증하는 방법을 사용한다.

2.2 2단계 VC 생성

명세 된 프로그램을 논리식의 형태로 바꾼 것을 verification condition(VC)라고 한다. 이전의 VC생성은 weakest precondition을 이용하여 VC를 생성하는 첫 번째 단계와 생성된 VC를 정리증명기와 같은 automatic decision procedure를 이용하여 유효성을 판별하는 두 번째 단계로 구성되고 이를 2단계 VC(two-stage VC) 생성으로 불렀다. 하지만 최근에 사용하는 2단계 VC생성 방법은 중복 문제를 해결하기 위하여 소스 단편(source fragment)을 passive 혹은 assignment-free 형태의 중간

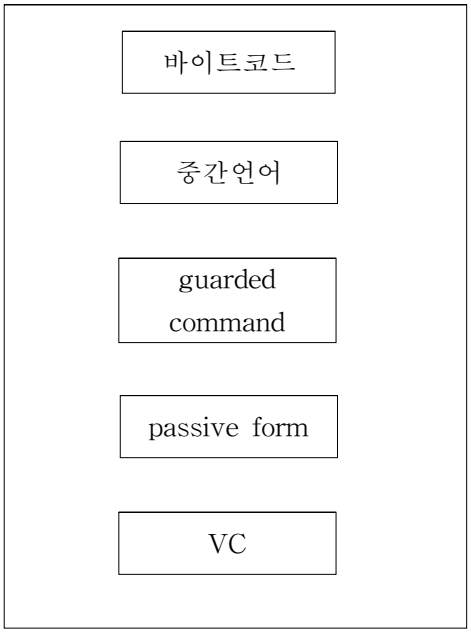
형태로 바꿔주는 첫 번째 단계와 passive form을 VC로 변환해주는 두 번째 단계로 구성되는 방법을 사용하고 있다[4,5].

2.3 명세언어

프로그램을 검증하기 위해서 프로그램이 반드시 만족해야 하는 조건들을 소스 프로그램 안에 삽입하여 검증하는 방법을 사용한다. 이렇게 검증을 위해 삽입하는 조건 등의 내용을 적어 넣는 언어를 명세 언어라고 한다.

3. VC 생성

이 장에서는 중간언어로 변환한 바이트코드를 VC로 변환하는 과정을 보인다. 그 과정은 (그림1)의 다이어그램과 같다.



(그림 1) 바이트코드의 VC생성 과정

변환에 쓰일 바이트코드의 소스 코드는 (그림 2)과 같다.

```
public class TFexample{
        int TF(int x) {
                if(x<10)
                        return 1;
                else
                        return 0;
        }
}
```

(그림 2) 소스코드 형태의 예제 프로그램

(그림 2)의 소스코드는 x가 10보다 작을 경우 참에 해당하는 1을 반환하고 그렇지 않을 경우 거짓에 해당하는 0을 반환해주는 프로그램이다. (그림 2)의 소스 코드가 변환된 바이트코드는 (그림 3)과 같다.

```
public TFexample();
  Code:
    0:    aload_0
    1:    invokespecial    #1;          //Method
java/lang/Object."<init>":()V
    4:    return
int TF(int);
  Code:
    0:    iload_1
    1:    bipush   10
    3:    if_icmpge        8
    6:    iconst_1
    7:    ireturn
    8:    iconst_0
    9:    ireturn
```

(그림 3) 바이트코드 형태의 예제 프로그램

하지만 바이트코드 자체는 (그림2)에 나타난 바와 같이 정적 분석에 필요한 정보를 파악하기가 매우 어렵고 자바 바이트 코드를 역 컴파일 할 경우 기존의 코드를 100% 복구 할 수 없기 때문에 자바 바이트 코드는 중간 언어의 형태로 변환될 필요가 있다. 본 논문에서 사용하는 소스 코드에 대한 바이트코드의 중간언어 변환 형태는 다음의 (그림 4) 와 같다.

```
predicate P(int x) := ( x<10 )
logicalfunction TF : ( int ) -> int
computationfunction TF :( int x ) -> int{
        read( x )
        block 0 : 0 : ifd  x<10  1
        block 1 : 1 : vreturn  1
        block 2 : 2 : vreturn  0

        from 0 to 1 when x<0
        from 0 to 2 when ¬ x<10
        returnblock 1, 2
        ensure result =1 ∨ result = 0
}
```

(그림 4) 바이트코드의 중간언어로 변환된 예제 프로그램

생성된 중간언어에 명세언어를 검증을 위한 명세언어를

추가한다. 그렇게 추가된 것이 중간언어의 마지막 부분에 위치한 ensure에 해당한다. (그림 4)와 같이 생성된 중간 코드를 이용하여 VC를 생성한다. (그림 4)의 중간언어는 비구조화 프로그램이고 기본적인 문법만을 사용하고 있기 때문에 [5]의 규칙만을 가지고도 충분히 VC생성이 가능하므로 [5]의 문법을 이용하여 VC를 생성하였다.

IF : skip; goto Then, Else
Then : assume (x<10); result := 1; goto End
Else : assume ¬(x<10); result := 0; goto End
End : goto After;
After : assert (result =1 ∨ result = 0)

(그림 5) guarded command 형태로 변환된 예제 프로그램

VC를 생성할 때는 보통 two-stage VC 방식으로 생성을 한다. 그 첫 단계에 해당하는 것이 소스프로그램을 guarded command 형태로 바꾼 후 passive form 혹은 assignment free형태로의 변환이다. (그림 4)의 중간언어를 가지고 [5]의 방법을 이용하여 만든 guarded command 형태가 위의 (그림 5)와 같다. 하지만 (그림 5)와 같은 형태는 배정문이 존재하기 때문에 중복 문제가 발생하여 VC가 생성되었을 때 그 크기가 매우 커지는 문제가 발생하게 된다. 이를 해결하기 위한 방법이 passive form 혹은 assignment free 형태로의 변환이다.

IF : skip; goto Then, Else
Then : assume (x_0<10); assume $result_0$ = 1; assume $result_1$ = $result_0$ goto End
Else : assume ¬(x_0<10); assume $result_1$ = 0; assume $result_1$ = $result_0$ goto End
End : goto After;
After : assert ($result_1$ =1 ∨ $result_1$ =0)

(그림 6) passive form으로 변환된 예제 프로그램

(그림 5)에 나온 결과를 passive form으로 변환한 것이 (그림 6)에 나오는 형태이다. 본 코드에서는 passive form에 의해 중복을 제거하여 VC의 크기가 줄어드는 효과를 확연하게 볼 수 없지만 다른 예제를 통하여 그 사실을 확인할 수 있다[4]. 이렇게 나온 결과를 가지고 weakest precondition 계산법을 이용하여 논리식으로 변환한다. 비구조화 프로그램을 VC로 생성하기 위해 [5]에서 사용한 weakest precondition 계산법은 (그림6)과 같다.

wp(assert P, Q) = P∧Q
wp(assume P, Q) = P⇒Q
wp(S;T, Q) = wp(S, wp(T, Q))

(그림 6) VC생성을 위해 사용한 weakest precondition

(그림 6)의 규칙을 이용하여 생성한 VC는 (그림 7)과 같은 결과를 보인다.

IF_{be} : IF_{ok} ≡ $Else_{ok}$ ∧ $Then_{ok}$
$Then_{be}$: $Then_{ok}$ ≡ x_0<10 ⇒ $result_0$=1 ⇒ $result_1$ =$result_0$ ⇒ End_{ok}
$Else_{be}$: $Else_{ok}$ ≡ ¬(x_0<10) ⇒ $result_0$=0 ⇒ $result_1$ =$result_0$ ⇒ End_{ok}
End_{be} : End_{ok} ≡ $After_{ok}$
$After_{be}$: $After_{ok}$ ≡ ($result_1$ =1 ∨ $result_1$ =0) ∧ true

(그림 7) weakest precondition을 적용한 후의 각 block의 verification condition

결과적으로 최종 생성되는 VC는 다음과 같다.

IF_{be} ∧ $Then_{be}$ ∧ $Else_{be}$ ∧ End_{be} ∧ $After_{be}$ ⇒ IF_{ok}

4. 결론

바이트코드의 검증은 프로그램의 신뢰성을 높이기 위한 노력 중 하나이다. 바이트코드의 검증을 하기 위하여 바이트코드에 대한 많은 정보를 가지고 있는 중간언어의 설계에 대한 연구 역시 많이 이루어지고 있다. 이러한 중간언어를 제대로 된 VC로 변환하여 유효성의 검증을 하는 것 역시 매우 중요한 문제이다. 본 논문에서는 설계한 중간언어를 가지고 VC를 생성하는 것을 수행해 보았다. 하지만 비구조화 프로그램을 위한 규칙을 적용하여 VC를 생성할 경우 선택문의 경우에는 중복으로 인한 문제가 발생할 수 있기 때문에 이를 해결하기 위한 방법이 필요하다. 그리고 향후 생성된 VC를 이용하여 프로그램에 문제점이 있는지를 확인하고자 한다.

논문 사사(acknowledgement)

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

참고문헌

[1] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, Erik Poll, “An overview of JML tools and

applications", International Journal on Software Tools for Technology Transfer (STTT), v.7 n.3, p.212-232, June 2005

[2] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, Raymie Stata, "Extended static checking for Java", Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation, June p.17-19, 2002

[3] K. Rustan M. Leino, Todd Millstein, James B. Saxe, "Generating error traces from verification-condition counterexamples", Science of Computer Programming, v.55 n.1-3, p.209-226, March 2005

[4] Cormac Flanagan, James B. Saxe, "Avoiding exponential explosion: generating compact verification conditions", Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, p.193-205, January 2001

[5] Mike Barnett, K. Rustan M. Leino, " Weakest-precondition of unstructured programs", ACM SIGSOFT Software Engineering Notes, v.31 n.1, p.82-87, January 2006

[6] K. Rustan M. Leino, James B. Saxe, Raymie Stata, "Checking Java Programs via Guarded Commands", Proceedings of the Workshop on Object-Oriented Technology, p.110-111, June 14-18