# 자바 바이트코드를 대상으로 하는 메소드 수준 뮤테이션 연산자

[0]신종민* 마유승** 권용래*
* 한국과학기술원 전자전산학과
(jmshin[0],kwon)@salmosa.kaist.ac.kr
** 한국정보통신연구원
ysma@etri.re.kr

# Method-level Mutation Operators for Java Byte-Code

[0]Jong-Min Shin* Yu-Seung Ma** Yong-Rae Kwon*
* Dept. of Electronic Engineering and Computer Science, KAIST
** Electronics and Telecommunication Research Institute

요 약

컴퓨터 성능의 향상으로 고비용의 수행을 요하는 뮤테이션 분석 기법의 적용 가능성이 커지면서, 객체지향 프로그램을 대상으로, 특히 자바 프로그램에 대하여, 뮤테이션 분석 기법에 관한 연구가 수행되었다. 자바의 경우, 바이트 코드를 이용한다면 소스코드 없이 오류 프로그램인 뮤턴트들을 생성할 수 있을 뿐만 아니라, 소스코드를 이용할 때보다 뮤테이션 분석을 빨리 수행할 수 있는 장점이 있다. 하지만 현재 이러한 장점을 효율적으로 활용할 수 있는 바이트 코드 수준의 연산자는 나와있지 않다.

본 논문에서는 자바 바이트 코드를 대상으로 하는 메소드 수준의 뮤테이션 연산자를 정의한다. 개발한 뮤테이션 연산자는 소스코드 수준에서 사용자가 범할 수 있는 오류만을 대상으로 한다. 따라서 소스 코드를 대상으로 하는 뮤테이션 분석의 기능을 모두 보여주면서, 성능향상을 가져다 준다.

## 1. 서 론

효율성 있는 시험 데이터(test case) 선정은 성공적인 소프트웨어 시험을 위한 중요한 요소이다. 뮤테이션 분석(mutation analysis)[1]은 오류를 바탕으로(fault-based) 시험 데이터 효율성을 판단하기 위하여 고안된 방법으로, 주어진 프로그램에 오류를 삽입한 프로그램인 뮤턴트(mutant)들의 검출여부에 의해서 시험 데이터 효율성을 결정한다. 따라서 뮤테이션 분석에서는 오류를 삽입하는 규칙을 정의하는 뮤테이션 연산자(mutation operator)에 대한 적절한 정의 및 선택이 중요하다.

뮤테이션 연산자는 프로그램 언어에 종속적이기 때문에 각기 다른 프로그램 언어마다 상이한 연산자 집합들이 정의된다. 객체지향 프로그램이 대중적으로 사용되면서 객체지향 언어에 대한 뮤테이션 연산자들[2,3]이 개발되었는데, 주로 자바(Java) 언어에 대해 많은 연구가 진행 되어왔다.

자바가 제공하는 바이트코드는 뮤테이션 시험에 다음과 같은 장점을 제공한다. 첫째, 소스코드 없이 오브젝트코드를 이용하여 뮤테이션 시험 수행을 가능하게 한다. 둘째, 바이트코드에 직접 오류를 심어 뮤테이션 분석을 하는 것이 소스코드에 오류를 심어 바이트코드로 변환하여 분석하는 것보다 4배 이상 빠른 수행속도를 갖는다[4]. 하지만 현존하는 뮤테이션 연산자는 소스코드를 대상으로 개발되어 있어, 이러한 장점들을 효율적으로 이용하지 못하고 있는 실정이다.

본 논문에서는 이를 해결하기 위해 바이트코드에 직접 적용시킬 수 있는 뮤테이션 연산자를 제안한다. 본 논문에서 제안하는 뮤테이션 연산자는 프로그램 작성시 프로그래머가 범하는 오류에 해당되는 소스코드와 대응되는 바이트코드 명령어들을 대상으로 이들을 변경, 추가, 혹은 삭제한다. 현재 객체지향 뮤테이션 연산자는 크게 메소드(method) 수준과 클래스(class) 수준으로 나뉘는데, 본 논문에서 제안하는 연산자는 메소드 수준 연산자이다.

본 논문의 구성은 다음과 같다. 2장에서는 객체지향 프로그램에서 사용되는 뮤테이션 연산자에 대해 설명한다. 3장에서는 바이트코드를 대상으로 하는 메소드수준 뮤테이션 연산자를 정의하고, 4장에서는 본 논문에서 제안한 연산자를 소스코드용 뮤테이션 연산자와 비교한 뒤, 이들을 실제 응용 프로그램에 적용해 본다. 마지막으로 5장에서는 결론 및 향후 연구에 대해 기술한다

## 2. 객체지향 프로그램의 뮤테이션 연산자

뮤테이션 분석은 시험 대상 프로그램에 대해서 오류를 인위적으로 삽입한 프로그램인 뮤턴트를 생성해 낸 뒤, 이들을 원(original) 프로그램과 구별 가능 여부로 시험 데이터의 효율성을 측정하는 기법이다. 뮤턴트들은 어떤 오류를 프로그램 상의 어느 위치에 삽입할 것인가를 정의하는 뮤테이션 연산자들에 의해 생성된다.

객체지향 프로그램을 대상으로 할 경우, 뮤테이션 연산자는 크게 클래스 수준과 메소드 수준의 연산자로 나뉜다.

### 2.1. 클래스 수준 연산자 [3]

클래스 수준 연산자는 상속, 다형성 등의 객체지향적 특성을

사용했을 경우 사용자들이 범할 수 있는 오류들을 생성한다. 아래 *super* 키워드를 삭제하는 ISK 뮤테이션 연산자의 예에서는 *super*키워드 삭제에 의해 오버라이딩(overriding) 된 메소드가 잘못 호출되는 경우를 보여준다.

| 원(original) 코드 | ➔ | ISK 뮤턴트 |
|---|---|---|
| *super.add();* | | *add();* |

### 2.2. 메소드 수준 연산자

기존의 뮤테이션 연산자들을 클래스 수준의 연산자들과 구분을 하기 위해 메소드 수준 연산자라 부른다. 메소드 수준 연산자는 프로그램이 기본적으로 제공하는 기능들(예를 들어 기본 연산자, primitive operator)을 사용하였을 경우에 발생할 수 있는 오류들을 생성한다. 아래는 산술 연산자를 다른 산술 연산자로 바꾸는 뮤테이션 연산자인 AOR (Arithmetic Operator Replace)의 적용 예를 보여준다.

| 원(original) 코드 | | AOR 뮤턴트 |
|---|---|---|
| *result = a / b;* | ➔ | *result = a + b;*<br>*result = a % b;*<br>*result = a * b;* |

## 3. 바이트 코드 뮤테이션 연산자 개발

본 장에서는 바이트 코드를 대상으로 하는 메소드 수준 뮤테이션 연산자를 정의한다. 제안하려는 연산자는 바이트코드 명령어들의 변환, 삽입, 삭제 등을 통해 오류를 심는다. 아래 각각의 하위 단락에서는 본 논문에서 어떤 바이트코드 명령들을 대상으로 하며, 대상이 되는 명령어에 어떤 종류의 오류를 심는지를 기술한다.

### 3.1. 적용 대상 바이트코드 명령어

바이트 코드에는 스택 연산 명령, 산술 연산 명령, 저장 명령 등의 다양한 종류의 명령이 존재한다. 하지만 뮤테이션 연산자는 원칙적으로 프로그래머가 프로그램 작성시에 발생시킬 수 있는 오류를 모델링 해야 하므로, 소스 코드에서 발생하는 오류와 직접 연관이 있는 바이트코드 연산자만을 대상으로 한다. 또한 기본 연산자의 사용으로 인해 발생하는 오류에 효과적인 시험 데이터는 메소드 수준의 다양한 오류들(예를 들어, operand 관련 오류)을 검출할 수 있으므로[5], 본 연구에서는 기본 연산자와 관련된 명령어만을 대상으로 한다. [표1]은 자바 바이트 코드 명령어 중 소스코드의 기본 연산자와 직접적 관계가 있는 명령어들을 기능별로 보여준다.

| 분 류 | 바이트코드 명령어 |
|---|---|
| 산술 연산자 | iadd, isub, imul, idiv, irem, ladd, lsub, lmul, ldiv, lrem, fadd, fsub, fmul, fdiv, frem, dadd, dsub, dmul, ddiv, drem, iinc |
| 관계 연산자 | if_icmpeq, if_icmpne, if_icmplt, if_icmpgt, if_icmpge, if_acmpeq, if_acmpne, ifgt, ifge, iflt, ifle, ifeq, ifne, ifnull, ifnonnull |
| 이동 연산자 | ishl, ishr, iushr, lshl, lshr, lushr |
| 논리 연산자 | iadd, ior, ixor, ineq, land, lor, lxor, lneq, fneq, dneq |

[표1] 대상 바이트코드

### 3.2. 바이트코드 뮤테이션 연산자

본 장에서는 본 연구팀에서 [표1]의 분류에 따라 개발한 자바 바이트코드 뮤테이션 연산자를 기술한다. 제안하는 연산자의 이름에서 첫 두문자는 연산자의 종류를 나타내며, 세 번째 문자 는 오류 생성 방식(R:변경, I:삽입, D:삭제)을 나타낸다.

(1) 산술연산자 관련 뮤테이션 연산자

- AOR (Arithmetic Operator Replacement)
  = {AOR_I, AOR_L, AOR_F, AOR_D, AOR_U}
  : 산술연산자를 다른 산술연산자로 변경

AOR은 AOR_I, AOR_L, AOR_F, AOR_D, AOR_U로 더 세분화 될 수 있다. 이때 AOR_ 다음의 문자들 I, L, F, D 는 각각 integer, long, float, double 형식을 대상으로 하는 연산자들을 의미하고 binary 연산자이다. AOR_U는 unary 연산자 iinc의 증감값을 +1혹은 -1로 바꾼다.

- AOD (Arithmetic Operator Deletion) = {AOD_U}
  : 단항 산술 연산자 iinc 삭제

| 소스 수준 | | |
|---|---|---|
| int a, b; | ➔ | int a, b; |
| ***a+b;*** | | ***a-b;*** |

| 바이트 수준 | | |
|---|---|---|
| iload_1 | AOR_I | iload_1 |
| iload_2 | 적용 | iload_2 |
| ***iadd*** | ➔ | ***isub ( or imul or idiv or irem)*** |

(2) 관계연산자 관련 뮤테이션 연산자

- ROR (Relational Operator Replacement)
  = {ROR_I, ROR_R, ROR_Z, ROR_N}
  : 관계연산자를 다른 관계연산자로 변경

ROR_다음의 I, R, Z, N은 각각 integer, reference, 숫자 0, null 값에 관련된 관계연산자를 대상으로 함을 의미한다.

| 소스 수준 | | |
|---|---|---|
| int A; | ➔ | int A; |
| ***if(a==0){ }*** | | ***if(a!=0){ }*** |

| 바이트 수준 | ROR_Z | |
|---|---|---|
| iload_1 | 적용 | iload_1 |
| ***ifeq*** | ➔ | ***ifne*** |

(3) 쉬프트연산자 관련 뮤테이션 연산자

- SOR (Shift Operator Replacement)
  = {SOR_I, SOR_L }
  : 쉬프트 연산자를 다른 쉬프트 연산자로 변경

I, L은 각각 integer, long 형 연산자를 대상으로 함을 의미한다.

| 소스 수준 | | |
|---|---|---|
| int A; | ➔ | int A; |
| ***A>>3;*** | | ***A<<3;*** |

| 바이트 수준 | | |
|---|---|---|
| iload_1 | SOR_I | iload_1 |
| iconst_3 | 적용 | iconst_3 |
| **ishr** | ➔ | **ishl** |
| istore_1 | | istore_1 |

(4) 논리연산자 관련 뮤테이션 연산자

- LOR (Logical Operator Replacement)
  = {LOR_I, LOR_L, LOR_F }
  : 논리 연산자를 다른 논리 연산자로 변경

LOR_다음의 I, L, F는 각각 integer, long, float 형 연산자를 대상으로 함을 의미한다.

소스 수준

| | | |
|---|---|---|
| int A, B; | ➔ | int A, B; |
| ***return A & B;*** | | ***return A \| B;*** |

바이트 수준

| | | |
|---|---|---|
| iload_1 | | iload_1 |
| iload_2 | ILR 적용 | iload_2 |
| ***iand*** | ➔ | ***ior*** |
| ireturn | | ireturn |

## 4. 분석 및 적용

이 장에서는 바이트 코드 뮤테이션 연산자와 소스코드 뮤테이션 연산자의 차이 및 장단점을 논의한다. 또한 바이트 코드 뮤테이션 연산자를 실 응용프로그램에 적용하여 얼마나 많은 뮤턴트들이 생성되는지를 살펴본다.

### 4.1 바이트코드 vs. 소스코드 뮤테이션 연산자

바이트코드 연산자는 소스코드에서 발행시키는 오류를 기본으로 제작되었기 때문에 소스코드 연산자와 거의 유사하다. 하지만 모든 소스코드 명령어들이 바이트코드 명령어에 일대일 관계로 연결되지 않는 부분으로 인해 그 차이가 있다. 이 장에서는 아래의 세가지 차이에 대해서 논의한다.

첫째, 바이트코드 연산자는 변수의 타입에 따라 다른 연산자가 따로 존재한다. 예를 들면, 소스코드 상의 덧셈 연산자는 '+' 한 가지로 존재하나 바이트 코드 상에서는 iadd, ladd, fadd, dadd 의 네 종류가 있다. 둘째, 소스코드 상의 단일 산술 연산자 (예, ++,--)는 바이트 코드상에서 이진 덧셈 혹은 뺄셈 연산자나 단일 연산자인 iinc의 중 하나로 표현된다. 그럼 주의할 점은 전위 후위의 구별이 없음에 주의해야 한다. 셋째, 소스코드상의 관계 연산자와 조건(conditional) 연산자는 바이트코드 상에서 모두 관계 연산자로 처리된다. 즉, 바이트코드에서는 조건 연산자가 존재하지 않는다. 따라서, 소스코드의 경우 관계 연산자와 조건 연산자가 같이 사용될 경우 이들의 조합으로 인해 다수의 뮤턴트가 생성되고 어떤 경우에는 동일한 뮤턴트를 생성해 내는 경우가 있는데, 바이트 코드를 이용할 경우 그러한 문제점이 해결된다. 예를 들어 '(a==0)&&(b!=0)' 과 같은 소스코드의 경우 '==', '&&', '!=' 의 세 연산자에 대해서 뮤테이션이 적용된다. 하지만 위의 소스 코드는 바이트 코드 상에서 'iload_1, ifne *분기점*, iload_2, ifne *분기점*' 으로 표현되므로, 뮤테이션을 적용할 장소가 두 곳의 'ifne' 명령어로 줄어들어, 전체적으로 생성되는 뮤턴트의 개수가 줄어든다.

### 4.2 뮤턴트 개수

본 장에서는 제안된 연산자를 실 응용프로그램에 적용시켜 생성된 뮤턴트의 총 개수를 살펴본다. 적용한 프로그램은 대학 강의 중 과제 목적으로 학생들에 의해 개발된 간단한 ATM 프로그램이며, 총 13개의 클래스 와 195개의 메소드로 구성되어있다. [표2]는 ATM 프로그램을 대상으로부터 생성된 뮤턴트 개수를 보여준다. 총 466개의 뮤턴트가 생성되었는데, 이는 클래스 당 평균 35.8개, 메소드 당 평균 2.39개의 뮤턴트를 의미한다. 프로그램의 특성 상 SOR, LOR 에 해당하는 뮤턴트들은 생성되지 않았다.

| 뮤테이션 연산자 | | 뮤턴트 개수 | |
|---|---|---|---|
| 대분류 | 소분류 | 개수 | 합 |
| AOR | AOR_I | 52 | 64 |
| | AOR_L | 8 | |
| | AOR_F | 0 | |
| | AOR_D | 0 | |
| | AOR_U | 4 | |
| AOD | AOD_U | 4 | 4 |
| ROR | ROR_I | 195 | 398 |
| | ROR_L | 0 | |
| | ROR_Z | 200 | |
| | ROR_N | 3 | |
| SOR | SOR_I | 0 | 0 |
| | SOR_L | 0 | |
| LOR | LOR_I | 0 | 0 |
| | LOR_L | 0 | |
| | LOR_F | 0 | |
| | LOR_D | 0 | |
| 총 합 | | 466 | |

[표2] 뮤턴트 개수

## 5. 결론 및 향후연구

본 논문에서는 바이트코드를 대상으로 하는 메소드 수준 뮤테이션 연산자를 제안하였다. 제안한 뮤테이션 연산자는 소스코드 작성과정에서 실제로 프로그래머들이 발생시킬 수 있는 오류들을 생성해내므로, 소스코드 뮤테이션 연산자과 거의 동일한 효율성을 갖는다. 뿐만 아니라 소스코드상에서 나타나는 관계연산자와 논리연산자의 조합이 바이트코드 상에서는 논리연산자만으로 간단하게 표현되므로, 생성되는 뮤턴트의 개수를 감소 시키는 효과가 있었다.

본 논문에서는 메소드 수준 연산자에 대해서만 연구를 수행하였으나, 향후 클래스 수준 연산자로 확대해보려 한다. 하지만, 클래스 수준의 연산자의 경우 바이트 코드 상에서 객체지향적 특성과 관련된 명령어 집합들을 추출해 내는 작업이 쉽지 않을 것으로 예상된다.

## Reference


[1] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practing programmer," *IEEE Computer*, 11(4), pp.34-41, Apr. 1978

[2] 최병주, "객체지향 프로그램을 위한 뮤테이션 테스트 데이터 선정 기준", *정보과학회논문지(B)*, 제 24권, 제 9호, pp. 966-975, 1997

[3] Y. S. Ma, J. Offutt, and Y. R. Kwon, "MuJava: an automated class mutation system," J. *Software Testing, Verification and Reliability,* published online, Nov. 2004

[4] Y. S. Ma, Y. R. Kwon, and J. Offutt, "Inter-Class Mutation Operators for Java", *Proc. ISSRE 2002,* pp.352-363, Anapolis, MD, U.S.A., Nov. 2002

[5] Jeff Offutt, Ammei Lee, Gregg Rothermel, Roland H. Untch, and Christian Zapf, "An Experimental Determination of Sufficient Mutant Operators", *ACM Transactions on Software Engineering Methodology*, 5(2), pp.99--118, Apr. 1996.