

Mutagen4J: 효과적인 Java 프로그램 변이 생성 도구 (Mutagen4J: Effective Mutant Generation Tool for Java Programs)

전 이 루¹ 김 윤 호² 홍 신³ 김 문 주⁴
(Yiru Jeon) (Yunho Kim) (Shin Hong) (Moonzoo Kim)

요약 프로그램 변이 분석은 분석대상 프로그램의 코드를 변형한 다양한 프로그램 변이를 활용해 분석대상 프로그램의 특성을 분석하는 기법이다. 효과적인 변이 분석을 위해서는 분석대상 프로그램의 동작을 다양하게 변화시키는 유용한 변형 연산자의 사용이 필수적이다. 현재까지 Java 프로그램을 대상으로 제안된 변이 생성 도구들은 변형 연산자의 종류가 제한적이거나, 최근 Java 언어 요소로 작성된 분석대상 프로그램의 경우 올바른 변이 생성을 지원하지 못하는 한계가 있다. 본 논문은 Java 프로그램을 위한 새로운 변이 생성 도구 Mutagen4J를 소개한다. Mutagen4J는 기존 연구를 통해 유용한 것으로 알려진 프로그램 변형 연산자를 추가로 지원하며, 최근 Java 언어요소를 처리함으로써, Java 프로그램에 대한 효과적인 변이 분석을 지원한다. 기존 Java 프로그램 변이 생성 도구와 비교 실험을 수행한 결과, Mutagen4J이 기존 도구보다 유용한 변이를 평균 2.3배 생성하였다.

키워드: 소프트웨어 테스트, 프로그램 변이 분석, 프로그램 변형, 프로그램 변이 테스트, Java 프로그램

Abstract Mutation analysis (or software mutation analysis) generates variants of a target program by injecting systematic code changes to the target program, and utilizes the variants to analyze the target program behaviors. Effective mutation analyses require adequate mutation operators that generate diverse variants for use in the analysis. However, the current mutation analysis tools for Java programs have limitations, since they support only limited types of mutation operators and do not support recent language features such as Java8. In this study, we present Mutagen4J, a new mutant generation tool for Java programs. Mutagen4J additionally supports mutation operators recently shown to generate various mutants and fully supports recent Java language features. The experimental results show that Mutagen4J generates useful mutants for analyses 2.3 times more than the existing Java mutation tools used for the study.

Keywords: Software testing, Mutation analysis, Program mutation, Mutation testing, Java programs

1. 서론

프로그램 변이 분석(변이 분석; mutation analysis)은 분석대상 프로그램(target program)의 코드를 다양하게 변형한 다수의 “프로그램 변이”(mutant; 이하 변이)를 생성한 후, 코드 변화에 따른 프로그램 동작 변화를 분석하는 소프트웨어 분석 기법이다. 변이 분석에서는 분석대상 프로그램 전반에서 다양한 변이를 생성하기 위해 여러 개의 “변형 연산자”(mutation operator)를 활용하는데, 이 때 각 변형 연산자는 분석대상 프로그램 코드의 특정 패턴마다 특정한 형태의 변형을 발생시킨 변이를 자동으로 생성한다. 변이

분석 기법은 테스트 커버리지 메트릭을 평가하는 연구[1], 주어진 테스트 집합(test suite)의 유용성을 정량적으로 측정하는 기법[2], 테스트 오라클 자동 생성[3], 그리고 최근에는 프로그램 결함 위치를 자동으로 추정하는 기법(fault localization)[4] 등 다양한 소프트웨어 테스트 및 분석 기법에 활용도가 높다.

효과적인 변이 분석을 위해서는, 주어진 분석대상 프로그램 코드로부터 각각 상이한 동작을 가지는 다양한 변이를 생성하는 효과적인 변형 연산자를 사용하는 것이 중요하다. 예를 들어, 테스트 집합의 유용성을 판별을 위해 프로그램 변이를 활용할 경우, 테스트 집합의 구성에 따라 프로그램 변이의 테스트 실행 결과가 상이하게 발생하여야 이를 의미 있는 분석이 가능하다. 만약 변이 분석에 사용한 변형 연산자가 분석대상 프로그램 코드로부터 프로그램 변이를 많이 생성하지 못하거나, 혹은 생성된 변이들이 동작이 서로 동일하여 테스트 집합에 따라 특이점을 보이지 못할 경우, 이를 바탕으로 한 의미 있는 분석 결과를 얻기 어렵다.

현재 Java 프로그램을 대상으로 한 변이 분석을 위해 제안된 변이 생성 도구(mutant generation, 변형 연산자 구현의 집합)의 경우, 효과적인 변이 생성에 한계점이 있는 것으로 보인다. 예를 들어, 대표적인 Java 프로그램 변이 분석 도구인 Javalanche[5], Jester[6], Jumble[7], PIT[8]의 경우, 식·단위(expression-level) 변형 연산자만을 지원하고, 최근 연구 결과[9,10], 효과적인 것으로 알려진 “구문

* 본 연구는 한국연구재단 한-아프리카 협력기반조성사업 (NRF-2014K1A3A1A09063167), 미래창조과학부 및 정보통신기술진흥센터의 대학ICT연구센터육성 지원사업 (IITP-2016-H85011610120001002), 미래창조과학부가 지원하는 한국연구재단 신진연구지원사업 과제 (NRF-2015R1C1A1A01055259)의 지원으로 수행되었음.

¹ 학생회원 : KAIST 전산학부 podray@kaist.ac.kr

² 학생회원 : KAIST 전산학부 kimyunho@kaist.ac.kr (Corresponding author임)

³ 정회원 : 한동대학교 전산전자공학부 조교수 hongshin@handong.edu

⁴ 종신회원 : KAIST 전산학부 부교수 moonzoo@cs.kaist.ac.kr

```
boolean isEven(int x, int y){
    int z = ( x + y ) % 2 ;
    return z < 1 ; }

```

(a) Original program

```
boolean isEven(int x, int y){
    int z = ( x * y ) % 2 ;
    return z < 1 ; }

```

(b) Mutant 1

```
boolean isEven(int x, int y){
    int z = ( x + y ) % 2
    return z >= 1 ; }

```

(c) Mutant 2

```
boolean isEven(int x, int y){
    int z = ( x + y ) % 2 ;
    return z != 1 ; }

```

(d) Mutant 3

그림1. 프로그램 변이의 예
Fig. 1. Examples of mutations

삭제”(statement deletion) 변형 연산자와 같은 구문-단위(statement-level) 변형 연산자를 지원하지 않는다. 뿐만 아니라, 일부 도구의 경우, 최신 Java 언어 요소의 올바른 처리를 지원하지 않아 최근에 개발되는 Java 프로그램 분석에 사용할 적합한 도구를 찾는데 한계가 있는 실정이다. 예를 들어, 유명한 공개 소프트웨어인 Apache의 TinkerPop3 프로젝트의 경우, Java8에 포함된 람다 표현식(lambda expression)과 같은 언어 요소를 포함하고 있는데, MuJava[11]와 Major[12] 도구는 Java8으로 작성된 프로그램을 올바르게 처리하지 못한다.

본 논문은 Java 프로그램으로부터 변이 분석에 유용한 프로그램 변이를 효과적으로 생성하는 변이 생성 도구인 Mutagen4J를 소개한다. Mutagen4J는 Java 프로그램 소스코드(source code)를 입력 받아 변형된 소스코드 형태로 변이를 생성한다. Mutagen4J는 기존 도구들이 주로 지원하는 식-단위 변형 연산자는 물론 유용한 구문-단위 변형 연산자로 알려진 구문 삭제(statement deletion) 변형 연산자를 지원한다. Java 프로그램을 대상으로 한 기존의 변이 생성 도구에 대하여, Mutagen4J가 가지는 특징점을 요약하면 다음과 같다:

- Mutagen4J는 구문 삭제 변형 연산자를 통해 분석에 유용한 다양한 변이를 효과적으로 생성한다. 특히 Mutagen4J의 변형 연산자는 단순 구문뿐만 아니라 복합 구문에 대한 변형을 효과적으로 지원한다.
- Mutagen4J는 소스코드를 대상으로 변형을 수행하고 소스코드 형태로 프로그램 변이를 출력함으로써, 바이트코드를 변형하여 출력하는 도구에 비해 사용자가 프로그램 변이를 더 쉽게 이해하고, 추가로 수정하여 사용하기 편하다.

표 1. 그림1의 원본과 변이 프로그램들의 테스트 결과
Table 1. Test Results of Program and Mutants in Fig 1.

No	Input	Original	Mutant1	Mutant2	Mutant3
1	0,2	True	True	False	True
2	0,1	False	True	True	False
3	1,1	True	False	False	True
4	1,0	False	True	True	False
5	2,0	True	True	False	True

- Mutagen4J는 Java8의 구문 요소를 처리할 수 있는 Javaparser[13]를 기반으로 하여, 람다 표현식과 같은 최신 Java 언어요소로 작성된 프로그램에서도 변이를 만들 수 있다.

본 논문의 기여점(contribution)은 다음과 같다.

- 새로운 변이 생성 도구인 Mutagen4J는 변이 분석에 유용한 변이를, 기존의 Java 프로그램 변이 생성 도구보다 효과적으로 생성한다. 실험 결과, Mutagen4J는 기존 도구에 비하여 유용한 변이를 2.3 배 생성하였다.
- 변이 생성 도구의 성능을 평가하기 위하여 유용한 변이와 유용하지 않은 변이를 체계적으로 구별하는 실험을 설계하였고, 이를 통하여 Mutagen4J와 4개 기존 Java 프로그램 변이 생성 도구의 성능을 정량적으로 비교하였다.

본 논문의 구성은 다음과 같다. 2장에서는 현재 사용 가능한 Java 프로그램 변이 생성 도구와 그들의 한계점을 설명한다. 3장에서는 본 논문에서 제안하는 Mutagen4J를 소개한다. 4장에서는 Mutagen4J를 기존의 도구와 비교하는 실험을 통해 Mutagen4J의 효용성을 보이며, 5장에서는 Mutagen4J의 향상과 효과적인 활용 방안을 논의한다. 마지막으로 6장에서 본 연구의 결론을 제시하며 논문을 마무리한다.

2. 변이 분석과 기존 Java 프로그램 변이 생성 도구

본 장에서는 프로그램 변이 분석이 무엇인지 간략히 소개하고, 효과적인 변이 분석을 달성하는데 있어서 기존의 Java 프로그램 변이 생성 도구가 가지는 한계를 설명한다.

2.1 프로그램 변이 분석 기법

변이 분석은 원본 프로그램의 특정 코드 요소(구문, 표현식, 연산자 등)를 다른 코드 형태로 변경한 프로그램 변이(mutant)를 만들고, 이를 주어진 입력 값을 이용해 실행하여 프로그램 동작을 발생시킨 후, 원본 프로그램과 변이의 동작 차이를 분석하는 기법이다. 변이는 원본 프로그램의 소스코드나 바이트코드(혹은 바이너리코드)의 특정한 구문 패턴을 탐지하고 이를 정의된 다른 형태의 구문 패턴으로 치환하는 방식을 통해, 분석대상 프로그램 전반에 대하여 기계적으로 생성하게 된다. 이 때 어떤 문법 패턴을 어떻게 변경할지 정의한 것을 변형 연산자(mutation operator)라고 하며 원본 프로그램에서 변형 연산자를 적용하여 달라진 부분을 변경점(mutation point)이라고

부른다. 변경점을 수정하여 만들어진 변이는 ‘그 변경점에서 생성되었다’고 말한다. 변이에서 그 변경된 부분을 제외한 다른 모든 부분은 원본 프로그램과 동일하다.

프로그램 변이 분석에서 생성된 모든 변이들은 주어진 입력 값에 의해 실행되는데, 이 때 테스트 케이스가 주로 활용된다. 테스트 케이스는 프로그램의 입력 값과 프로그램 실행 결과를 성공(pass) 혹은 실패(fail)로 판별하여 출력하는 테스트 오라클로 구성된다. 변이를 주어진 테스트 집합(테스트 케이스의 집합)에 의해 실행하는 경우, 하나 이상의 테스트 케이스가 변이에 대해 실패했으면 ‘테스트 집합(혹은 테스트 케이스)가 그 변이를 죽였다(kill)’라고 말한다. 만약 어떤 변이를 어떤 테스트 케이스도 죽이지 못했다면, 그 변이를 ‘살았다’고 말한다.

그림1은 원본 프로그램과 원본 프로그램 코드에서 산술/비교 연산자를 치환하는 변형 연산자를 적용하여 생성한 세 가지 변이의 예를 보여준다. 원본 프로그램은 두 정수를 입력 받아, 그 합이 음수가 아닌 짝수인지를 판별한 값을 반환하는 함수이다. 이 때, 변이3의 경우, 비교연산자가 변경되어 원본 프로그램과 구조(syntax)는 다름에도 불구하고, 프로그램 의미(semantics)가 원본 프로그램과 동일함을 알 수 있다. 변이3과 같은 프로그램 변이를 무효 변이(equivalent)라고 부른다. 표 1은 원본 프로그램과 세 가지 변이를 5개의 테스트 케이스로 테스트한 결과이다. 변이1의 경우 1번, 5번 테스트 케이스 실행 결과는 원본과 같은 반면, 나머지 테스트 케이스에서 다른 실행 결과를 가진다. 변이2의 경우, 모든 테스트 케이스에 대해 원본 프로그램과 다른 결과를 보인다. 변이3의 경우, 원본과 프로그램 의미가 같으므로, 모든 테스트 케이스에서 원본과 같은 출력 값을 가진다.

2.2 유용하지 않은 프로그램 변이

변이 분석은 프로그램의 특정 코드 요소를 변화시키에 따라 어떠한 프로그램 동작 변화가 발생하는지를 분석하는 기법으로, 분석대상 프로그램으로부터 다양한 동작을 보이는 프로그램 변이를 많이 생성할 수록 변이 분석에 유리하다. 반면, 원본 프로그램과 동작이 동일한 무효 변이나, 서로 간의 동작이 동일한 동작을 보이는 변이는 분석이 유용하지 않다. 따라서, 변이 분석에는 유용하지 않은 프로그램을 변이를 적게 생성하는 변형 연산자가 사용되는 것이 적합하다. 특히, 생성되는 변이의 수가 많으며, 이를 큰 규모의 테스트 집합에 대해 실행하게 될 경우 많은 분석 비용이 소모되는 규모가 큰 실제 소프트웨어 대상으로 변이 분석을 수행할 경우에는 유용하지 않은 변이의 생성이 테스트 비용 측면에서도 문제가 발생할 수 있다.

주어진 테스트를 실행한 결과를 바탕으로 유용하지 않은 변이를 정의할 수 있으며, 이를 다음과 같은 유형으로 분류할 수 있다.

- 잘못된 변이(invalid mutants): 변형 연산자가 프로그램 코드의 문법적(syntactic) 제약 사항을 위반하여 올바른 프로그램이 생성되지 못한 경우이다. 컴파일 에러가 나거나 문법 규칙 위반 오류가 발생하여 프로그램 실행이 불가능하므로, 변이 분석이 유용하지 않다.
- almost-always-killed(이하 AAK) 변이: 대부분의 테스트 케이스로 인하여 변이가 죽는 경우를 뜻한다(예: 전체 테스트 케이스의 70% 이상). 이 경우, 대부분의

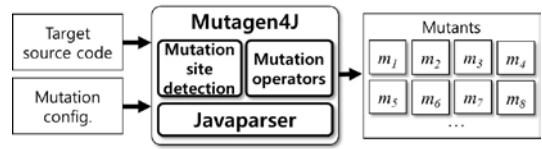


그림2. Mutagen4J의 흐름
Fig. 2. Workflow of Mutagen4J

실행에서 원본 프로그램과 상이한 결과가 발생하므로, 프로그램 입력값이나 실행 경로에 따른 동작 변화를 분석하는데 유용하지 않다.

- almost-always-killed-when-reached (이하 AKR) 변이: 대부분의 변경점을 실행하는 테스트 케이스로 인하여 변이가 죽는 경우를 뜻한다(예: 전체 테스트 케이스의 70% 이상). 이러한 변이의 경우, 변경점에 대한 커버리지 조건과 같이 변경점 실행 유무를 판별하는 것 외에 다른 분석에 유용성이 낮다.
- 무효 변이(equivalent mutant): 모든 입력에 대해 원본 프로그램과 동일한 테스트 결과를 갖는 변이다. 따라서 무효 변이는 모든 테스트 케이스의 결과가 원본과 결과가 같다.
- 중복(redundant) 변이: 모든 입력에 대하여, 코드가 다른 변이 프로그램과 동일한 테스트 결과를 갖는 경우다. 중복 변이의 수가 늘어도, 추가되는 프로그램 동작 정보가 늘지 않으므로 유용하지 않은 변이로 볼 수 있다.

그림1의 변이2와 변이3은 유용하지 않은 변이로, 각각 AAK 변이와 무효 변이에 해당한다. 일부 기법[3]에서는 유용하지 않은 변이의 정보로부터 프로그램의 특성의 일부를 유추하기도 하지만, 유용하지 않은 경우가 대량 생성될 경우, 전체 변이 분석의 효율성이 떨어질 수 있다. 따라서, 변이 생성 도구는 유용하지 않은 변이를 가능한 생성하지 않을 수록 효과적이라고 볼 수 있다.

2.3 기존 Java 프로그램 변이 생성 도구의 한계점

효과적인 변이 생성을 위해 변형 연산자에 따라 유용한 변이 생성 정도를 측정한 연구들[9,10]이 수행되어왔다. 여러 변형 연산자를 실험적으로 연구한 결과에 따르면, 분석 대상 프로그램에서 구문 하나를 삭제하는 SDL(Statement Deletion) 연산자는 유용한 변이를 효과적으로 생성하는 대표적인 변형 연산자로 알려졌다. 연구 결과에 따르면, SDL가 생성한 변이의 경우, 프로그램 동작을 상당히 변화하여 무효 변이의 발생이 적으며, 프로그램 코드 전반에 적용 가능하므로 다양한 변이를 발생시킬 수 있다.

하지만, 현재 도구로서 사용 가능한 변이 생성 기법들(Javalanche[5], Jester[6], Jumble[7], PIT[8])을 조사한 결과, 이들 중 SDL이나 구문-단위 변형 연산자를 효과적으로 지원하는 도구가 없으며, 비교적 간단한 식-단위 변형 연산자만을 제공하여 유용한 변이 생성에 한계가 있음을 확인할 수 있었다. Javalanche, Jumble, PIT는 Java 바이트코드 상에서 변형을 수행하므로, SDL를 포함하여 구문-단위의 변형 연산자를 제공하지 않는다. 소스코드에 변형을 생성하는 Major는 SDL 연산자를 제공하나 단순 구문을 대상으로만 적용 가능한 변형 연산자를 제공하여, 실제 Java 프로그램을

//Java source code	//bytecode of m()
01 public	0: iload_1
class test{	1: lookupswitch
02 int a, b;	...
03 void m(int x){	20: iload_1
04 switch(x){	21: iconst_1
05 case 0:	22: iadd
06 x = x + 1;	23: istore_1
07 default:	24: aload_0
08 b = x;	25: iload_1
09 }	26: putfield#2
10 a = x;	29: aload_0
11 }	30: iload_1
12 }	31: putfield#3
	34: return

그림 3. switch 문의 Java 소스 코드와 바이트코드

Fig. 3. Java source code and bytecodes of a switch statement example

대상으로 적용할 경우 다양한 변이를 생성하는데 한계가 있었다.

이에 더하여, Major와 MuJava는 최근에 추가된 언어 요소를 올바르게 처리하지 못하여 Java8 환경에서 개발된 프로그램으로부터 올바른 변이 생성이 불가능하므로, 사용성이 제한적인 상황을 파악할 수 있었다.

3. Mutagen4J 도구

3.1 개요

Mutagen4J는 그림 2에서 보는 바와 같이, Java 소스코드들을 입력 받아, 소스코드 수준에서 변형을 수행한 후, 소스코드 형태로 변이를 출력한다. Mutagen4J는 소스코드 변환을 위하여 Javaparser[13]를 기반으로 개발된 변경점 결정 모듈과 변형 연산자로 구성되어 있다. 변경점 결정 모듈은 Javaparser가 생성한 구문 구조 트리(abstract syntax tree)에서 Mutagen4J가 지원하는 변형 연산을 수행할 수 있는 코드 위치를 추출한 후, 입력 받은 변형 설정의 내용을 바탕으로 변형을 수행할 변경점을 선택하는 역할을 한다. 변형 설정에는 총 생성되는 변이의 수를 조절하기 위해 코드 당 변경점 개수의 상한선, 사용할 변형 연산자의 종류, 변이를 발생시키지 않을 코드 라인을 기술할 수 있다. 변형 연산자는 선택된 변경점에 실제 변형 연산을 적용하여 변이 프로그램을 생성하는 기능을 수행한다.

Mutagen4J는 바이트코드가 아닌 소스코드를 대상으로 변형을 수행함으로써, SDL을 포함한 구문-단위 변형 연산자를 효과적으로 구현/지원하도록 하였다. Mutagen4J는 소스코드 변형을 구현하기 위해 Java 구문 구조 트리(AST) 분석 프레임워크인 Javaparser[13]를 기반으로 변이 도구를 개발하였다. 이와 같이 Mutagen4J는 Java 구문 구조 트리를 직접 분석/처리하도록 개발되어 향후 새로운 Java 문법 변화에 효과적으로 대응할 수 있도록 하였다. 반면 MuJava와 Major의 경우, 소스코드를 기반으로 하지만, Java 구문에 대한 매크로(macro)나 스크립트(script)를 거쳐서 분석을 수행하므로 Java 문법 변화에 대한 적응성이 낮다는 한계가 있다. 특히, Mutagen4J는 단순/복합 구문에 모두 적용 가능한 일반적인 SDL 변형 연산자를 제공하여 유용한 변이를 효과적으로 생성하는 것을 목표로 하였다. SDL에 더하여,

Mutagen4J는 이전 연구[14] 결과 유용한 것으로 알려진 4종류의 변형 연산자들을 구현하고 있으며, 각각의 대략적인 설명은 다음과 같다[15]:

- OAAN(Operator, arithmetic to arithmetic, non-assignment): 산술연산자(+, -, *, /, %)를 다른 산술연산자로 변경한다.
- ORRN(Operator, relational to relational, non-assignment): 비교연산자(<, <=, ==, ...)를 다른 비교 연산자로 변경한다.
- OLLN(Operator, logical to logical, non-assignment): 논리연산자(&&, ||)를 다른 논리연산자로 변경한다. e.g. $x \&\& y \rightarrow x || y$
- OLNG(Operator, logical negation): 논리연산자를 포함한 표현식 $a \&\& b, a || b$ 를 다음과 같이 변경한다.
 - 불리안(Boolean) 표현식 a, b 를 부정(negate) (예: $a \&\& b$ 를 $!(a \&\& b)$ 혹은 $a \&\& (b)$ 로 변경)
 - 전체 표현식 $(a \&\& b)$ 혹은 $a || b$ 를 부정 (예: $a \&\& b$ 를 $!(a \&\& b)$ 로 변경)
- SDL(Statement deletion): 구문 하나를 삭제한다.

3.2 변형 연산자 설계

Agrawal이 C언어에 맞게 설계한 변형 연산자 중 3개(OAAN, OLLN, ORRN)는 Java 언어에서도 그대로 사용할 수 있었다. 나머지 2개인 SDL과 ORRN은 Java 언어에 맞게 수정되었다. SDL은 Java가 갖고 있는 구문들을 삭제할 수 있고 ORRN은 Java에서 기본 타입과 래퍼런스 타입 사이의 비교가 일어나는 것을 막도록 설계되었다.

3.2.1. SDL 변형 연산자

Mutagen4J의 SDL 연산자는 이전 연구에서 명시한 SDL 연산자[15]처럼 복합 구문도 삭제할 수 있다. 이 SDL 연산자는 C 언어에 맞게 설계되었는데, Mutagen4J의 SDL은 Java 문법에 맞게 확장되었다. Mutagen4J의 SDL은 C 언어에는 없지만, Java가 제공하는 예외처리문(try-catch), 동기문(synchronized block) 등들도 삭제할 수 있다. Mutagen4J는 Javaparser[13]를 사용해서 구문에 해당하는 AST 노드를 변경하여 구문 삭제를 수행한다.

바이트코드를 수정하는 방법으로 SDL 변형 연산자를 구현하기는 쉽지 않다. 우리는 SDL을 구현하기 위해서 소스코드를 수정하여 변이를 만들었다. Java 변이 생성 도구들은 변이를 소스 코드를 변경하거나 바이트코드를 변경하여 생성한다. 소스 코드를 수정하는 방법은 바이트코드를 수정하는 방법보다 느리고 복잡하지만, 바이트코드를 고치는 방법으로는 할 수 없는 변형을 수행할 수 있다.

그림3는 Java 소스 코드에서의 switch문과 switch 문의 컴파일 되어 생성된 바이트코드를 나타낸다. 소스 코드 수준에서는 switch 구문 전부(04~11)를 삭제해 변이를 생성할 수 있다. 하지만 바이트코드에서는 default case안에 있는 명령들(24~26)과 switch 이후의 명령들(29~31)을 구분할 수 없다. 따라서 소스 코드를 수정할 때와는 달리 29~31 명령들만 가진 변이를 생성할 수 없다. 이는 소스코드에 있던 정보들이 저수준인 바이트코드로 컴파일 되면서 사라졌기 때문이다. 따라서 우리는 소스


Original	Mutant1	Mutant2	Mutant3
if (cond) { ... } else { ... }	if (cond) { ... } else { ... }	if (cond) { ... } else { ... }	

그림 4. SDL이 if문에서 생성한 세 변이

Fig. 4. Examples of mutants by SDL

Original	Mutant1	Mutant2	Mutant3
int a; ... if (a > 40)	int a; ... if (a >= 40)	int a; ... if (a < 40)	int a; ... if (a == 40)

그림 5. 기본 타입 비교 연산자에 ORRN를 적용한 예

Fig. 5. Examples of mutants by ORRN

표 2. 변이 생성 결과

Table 2. Overview of mutant generations by 5 tools

	Mutagen4J	Jumble	PIT	Major	MuJava
#. total generated mutants	1,675	862	1,610	1,621	948
# code lines with mutant sites	614 (77.5%)	535 (67.6%)	705 (89.0%)	479 (60.5%)	164 (20.7%)

코드를 수정하여 변이를 만들고 SDL 변형 연산자를 제공하는 새로운 도구 Mutagen4J를 구현하였다.

구문의 형태에 따른 SDL의 동작은 다음과 같다.

- **단일 구문:** 단일 구문을 빈 구문(;)으로 대체한다.
- **지역 변수 정의 구문:** 선언 구문을 삭제하면 이후 문장에서 정의되지 않은 지역 변수를 사용하기 때문에 컴파일 에러가 발생한다. 따라서, 구문 전체를 지우는 것이 아니라 초기화에 해당하는 식만 제거하도록 설계 하였다.
- **if, for, for-each, while, do-while, synchronized 복합 구문들:** 전체 구문을 제거한 변이와, 복합 구문 내 각각의 구문 블록(들)을 제거한 변이를 생성한다. 그림4는 if 구문에서 then-body를 삭제한 변이1과 else-body를 삭제한 변이2, 그리고 if문 전체를 삭제한 변이3을 보여준다. for 구문에서 body block을 삭제한 변이와 구문 전체를 삭제한 변이만을 만든다.
- **try-catch 구문:** try-catch 구문을 전체를 삭제한 변이를 생성할 수 있다. 만약에 원본 프로그램에 finally 블록이 있다면 finally 블록만 삭제한 변이도 생성할 수 있다. try-catch 구문에는 여러 catch 절들이 있을 수 있는데 이들을 하나씩 제거한 변이를 생성한다.

3.2.2. ORRN 변형 연산자

Java 언어에서는 비교 연산자를 두 종류로 나눌 수 있다. 첫 번째는 기본 타입의 값들을 비교하는 >, >=, <, <=이고 두 번째는 기본 타입과 참조(reference) 타입의 값들을 비교하는 ==, !=이다. 첫 번째 종류의 연산자를 두 번째 종류의 연산자들로 변형하는 것은 컴파일 에러를 만들지 않지만 두

번째 종류의 연산자를 첫 번째 종류의 연산자로 바꾸는 것은 컴파일 에러를 만들 수 있다.

그림5는 정수형 타입에 사용된 비교 연산자(>)가 다른 5개의 비교 연산자로 변형된 변이의 예이다. 참조 타입 변수의 경우, Mutagen4J은 다음과 같이 변형을 수행한다.

- 비교 연산자가 !=면 ==으로 변경한다.
- 비교 연산자가 ==와 !=가 아니면, <, <=, >, >=, ==, != 중 원본과 다른 5개로 변형한다. 즉 5개의 변이를 생성한다.

4. 실험

Mutagen4J이 유용한 변이를 효과적으로 생성하는 지 평가하기 위하여, Mutagen4J를 잘 알려진 4개의 Java 프로그램 변이 생성 도구(Major, MuJava, Jumble, PIT)와 비교하는 실험을 수행하였다.

4.1 연구질문

Mutagen4J이 생성하는 변이들이 프로그램 변이 분석에 효과적인지 평가하기 위해 3개의 연구 질문을 세우고 실험 결과를 통해 답하였다. 각각의 연구 질문과 질문에 답하기 위한 실험 측정 기준은 다음과 같다.

- 연구질문 1: Mutagen4J은 얼마나 많은 변이를 생성할 수 있는가?

효과적인 변이 분석을 위해서는 다양한 변경점에서 다양한 변이를 생성할 필요가 있다. 연구질문 1에 답하기 위해, Mutagen4J과 기존 도구가 생성하는 총 변이의 수를 측정하고, 변경점이 포함된 소스코드 라인 수를 측정하여 비교하였다.

- 연구질문 2: Mutagen4J은 얼마나 많은 유용한 변이를 만드는가?

연구질문2에 답하기 위해, Mutagen4J와 기존 도구가 생성한 변이 중 유용하지 변이들(useless mutants)을 제외한 후, 유용한 것으로 판별되는 변이(useful mutants)의 개수를 확인하였다. 유용한 변이의 수를 측정하기 위해, 각 도구가 생성한 총 변이 중에서, 앞서 정의한 (2.2절) 잘못된 변이, AAK 변이, AKR 변이를 제거하였다. 이 때 AAK 변이와 AKR 변이는 총 테스트 케이스의 70%와 도달 테스트 케이스의 70% 이상에서 죽는 변이로 각각 정했다. 이에 더하여, 본 실험에서 사용하는 테스트 케이스를 사용하여 원본과 동작의 차이를 보이지 않은 변이의 경우 무효 변이로 간주하여, 이를 추가로 제거한 이후의 수를 유용한 변이의 수로 측정하였다.

- 연구질문 3: Mutagen4J의 SDL 변형 연산자가 Mutagen4J의 다른 변형 연산자들보다 얼마나 더 효과적인가?

프로그램 변이 분석을 효과적으로 수행하기 위해서는 유용한 변이를 만드는 효과적인 변형 연산자를 사용하는 것이 중요하다. 연구질문 3에 답하기 위해 SDL 변형 연산자가 Mutagen4J에서 얼마나 많이 유용한 변이를 만드는지 확인하였다.

4.2. 비교 대상 도구들

Mutagen4J이 만드는 변이들의 효과성을 확인하기 위해, 현재 개발/관리가 되고 있으며 공개적으로 접근이 가능한 Java 변이 생성 도구인 Major[12], MuJava[11], Jumble[7], PIT[8]을 비교 대상 도구로 선정하여 실험하였다. Jumble, PIT는 바이트코드를 수정해 변이를 생성하고 Major, MuJava는 소스 코드를 수정해 변이를 생성한다.

이 외에도 Java 프로그램 변형 도구로 Javalanche[5]가 있으나, 이를 이용한 변이 생성 비교 실험에 어려움이 있어서 제외했다. Javalanche의 경우, 변이 중 하나만 테스트 케이스라도 실패할 경우, 나머지 테스트 케이스들을 실행하지 않으므로, 모든 변이의 모든 테스트 케이스들의 결과를 알 수 없었기 때문이다. 이러한 이유로, Javalanche를 직접 이용하는 실험이 불가능하였다.

4.3 실험 대상

본 실험에서는 소프트웨어 테스트 비교 분석 연구에 널리 사용되고 있는 SIR 벤치마크[16] 중 Jtopas 프로그램(Java tokenizer and parser tools)을 분석대상 프로그램으로 사용하였고 SIR에서 제공하는 테스트 집합을 실험에 사용하였다. Jtopas는 총 1,924 라인의 코드로, 21개의 Java 파일로 이루어져 있다. 129개의 JUnit 테스트 케이스로 이루어진 테스트 집합을 사용해서 프로그램 변이 분석을 수행하였다. 각 테스트 케이스는 실행 결과의 정확성 유무를 판단하여 성공(pass) 혹은 실패(fail)로 결과를 반환한다. 실험에 사용한 Jtopas 프로그램의 경우 129개 테스트 케이스 실행에서 모두 성공을 결과 값으로 가진다. Jtopas의 테스트 집합이 실행한 전체 코드 라인 수는 792였다.

4.4 도구의 구현

Mutagen4J의 변형 연산자는 Javaparser[13]가 분석대상 프로그램에 대하여 생성한 구문 구조 트리(Abstract Syntax Tree)를 탐색하는 Visitor 형태로 일종으로 구현되었다. Mutagen4J의 변이 생성 모듈은 생성된 구문 구조 트리를 대상으로, 먼저는 변경점의 개수와 위치를 파악하기 위한 탐색을 수행하고, 이를 바탕으로 변형을 발생할 변경점을 선택한다. 두 번째로, Mutagen4J는 다시 구문 구조 트리를 탐색하며 선택된 한 변경점에 도달할 경우, 변이 연산의 적용하여 해당 변이점에 대한 변이 프로그램 코드를 생성한다. Mutagen4J는 사용자로 입력 받은 설정에 따라(3.1절) 변이 연산이 적용 가능한 변이점 중 일부만을 선택하여 변이 생성을 수행하게 된다.

각 변이 연산자는 Javaparser의 추상화 구조를 활용하여 계층적으로 변이를 위한 코드 패턴을 적용하는 구조로 구현됐다. 현재 Mutagen4J에 포함된 변이 연산자는 구문 단위 연산자인 SSDL 변이 연산의 경우, Javaparser 구문 구조 트리에서 Java의 구문에 대응되는 Statement 타입의 노드에 대해 동작을 하는 Visitor 구현되었다. 하지만, SSDL 변이 연산자의 경우 Java의 구문 종류에 따라 적합한 구문 삭제 패턴이 상이하므로, Statement 구문에 대해서도 네 가지 세부 타입(ReturnStmt, TryStmt, ExpressionStmt, BlockStmt)에 따라 각각 다른 변이 패턴을 적용하는 코드를 호출하는 하는 형태로 구현됐다. 이와 유사하게, 식 단위 변형 연산자인 OAN, OLLN, OLNG, ORRN도 모두 BinaryExpr 노드에 대한 Visitor 형태로 구현되었지만, BinaryExpr 세부 타입에

따라 적용 가능한 변이연산자, 변이 패턴이 상이하다. 따라서, 각 변이 연산자에서는 노드의 세부 타입에 따라 서로 다른 변이 연산자 코드가 실행되도록 개발되었다.

4.5 실험 과정

Mutagen4J와 비교 대상인 4개의 변이 생성 도구에 Jtopas의 소스코드 혹은 바이트코드를 입력하여 변이를 최대한 생성하였다. 그리고 생성된 변이 중 변경점이 129개의 테스트케이스가 최소한 한 번 도달하는 경우만 한정하여 수집하였다. 그 이외의 변경점에서 생성된 변이는, 항상 원본 프로그램과 동일한 실행 결과를 가지기 때문에, 실험에 불필요하다. 수집된 변이를 모두 129개의 테스트 케이스에 대하여 실행하고, 실행 결과를 기록하였다.

4.6 결과

4.6.1 연구질문1에 대한 결과

표 2는 Mutagen4J과 각 Java 변이 생성 도구들이 Jtopas의 792 라인에 대해 생성한 변이의 수와 변경점이 포함된 라인의 수를 나타낸다. Mutagen4J은 가장 많은 변이를 생성하였으며 총 614 라인에서 변이를 생성하였다. Mutagen4J은 PIT를 제외한 나머지 3개 도구들보다 평균 2.1배 더 많은 코드 라인에서 변이를 생성하였다. 이 결과로 미루어볼 때, Mutagen4J이 기존 도구보다 더 많은 코드에서 더 많은 변이를 생성할 수 있음을 알 수 있다.

Mutagen4J이 변형을 하지 못한 178 라인을 추가로 조사한 결과, 그 중 132 라인(74.2%)은 return 구문이었다. 현재 Mutagen4J는 return 구문에 대한 변형 연산자를 포함하지 않고 있다. 반면 PIT의 경우, return 구문에 있는 값을 상수로 바꾸는 변형 연산자를 제공하기 때문에 Mutagen4J보다 더 많은 코드 라인에서 변이를 생성함을 알 수 있었다.

MuJava의 경우, Jtopas의 소스코드 중 일부 언어 요소를 처리 못해 특정 소스코드 파일(AbstractTokenizer.java, 1,157 라인)가 변이 생성에서 배제되어, 다른 도구에 비하여 적은 코드 라인만 변형하는 문제점이 있었다.

4.6.2 연구질문2에 대한 결과

표3은 각 도구가 생성한 변이 중 유용한 변이와 유용하지 않은 변이를 유형 별로 분류한 수를 나타낸다. 표3의 각 칸의 숫자는 각 도구 별로 총 생성된 변이에 대한 비율이다. Mutagen4J은 다른 도구보다 더 많은 유용한 변이를 생성하며, 다른 4개 도구들보다 평균 2.3배의 유용한 변이를 생성함을 확인할 수 있다. 그리고 생성한 전체 변이 개수에 대한 비율을 생각하였을 때도 Mutagen4J이 다른 도구보다 평균 1.5배 더 높은 빈도로 유용한 변이를 만들 수 있음을 알 수 있다.

잘못된 변형의 경우, 소스코드 변형을 기반으로 하는 Mutagen4J, Major, MuJava의 경우 발생하였다. MuJava는 전체 생성한 변이 중 21.8%에 해당하는 변이가 잘못된 변이인 반면, Mutagen4J와 Major는 각각 4.1%와 1.2%로 더 적은 수의 잘못된 변이를 생성하였다.⁵ 바이트코드를 수정해서 변이를 만드는 Jumble, PIT는 잘못된 변이들을 생성하지

⁵이 때, Mutagen4J를 포함하여 소스코드 변형 도구가 생성한 잘못된 변이는 컴파일 단계에서 판별이 가능하므로 테스트 실행 효율성이나 비용 면에서 추가적인 문제가 발생하지는 않는다.

표 3. 도구별 생성된 변이의 분류

Table 3. Mutants generated by 5 mutation tools

	Mutagen4J	Jumble	PIT	Major	MuJava
Useful mutants	710 (42.4%)	282 (32.7%)	554 (34.4%)	541 (33.4%)	174 (18.4%)
Useless mutants	Invalid mutants	74 (4.4%)	0 (0.0%)	20 (1.2%)	207 (21.8%)
	AAK mutants	105 (6.3%)	117 (13.6%)	91 (5.7%)	22 (2.3%)
	AKR mutants	390 (23.3%)	287 (33.3%)	528 (32.8%)	218 (23.0%)
	Equiv. mutants	396 (23.6%)	176 (20.4%)	437 (27.1%)	327 (34.5%)
Total generated mutants	1675 (100.0%)	862 (100.0%)	1610 (100.0%)	1621 (100.0%)	948 (100.0%)

않음을 알 수 있었다.

AAK 변이는 모든 도구에서 대체로 유사한 비율로 생성되었다. 반면, AKR 변이의 경우, Jumble과 PIT가 상대적으로 많은 비율로 발생함을 확인할 수 있었다(각각 33.3%, 32.8%). Major의 경우, 본 실험에서는 상대적으로 무효 변이를 많이 만드는 현상이 관찰되었다(38.2%).

4.6.3. 연구질문3에 대한 결과

표 4는 Mutagen4J의 변형 연산자들이 생성한 유용한 변이의 수와 그 비율(괄호)을 나타낸다. Mutagen4J에서 유용한 변이를 가장 많이 생성한 변형 연산자는 SDL이었다. SDL은 다른 네 연산자보다 평균 5.3배 더 많은 유용한 변이를 만들었다. 이 결과는 Mutagen4J이 다른 도구들보다 더 많은 효과적인 프로그램 변이를 만드는데 SDL이 큰 역할을 하였음을 알 수 있다.

5. 논의

실험 결과에 비추어 볼 때, 효과적인 변이 생성을 위해서 각 도구를 특징점에 따라 복합적으로 사용하는 것이 유용할 것으로 예상된다. SDL에 대한 실험결과(연구질문3)에 비추어볼 때, 구문-단위 변형 연산자를 많이 확보할 경우 유용한 변이 생성에 유리함을 알 수 있다. Mutagen4J와 같이, 소스코드 변형 기반 도구의 경우, 소스코드의 정보를 직접 활용하여 구문-단위 연산자를 정확하게 구현할 수 있는 반면 바이트코드(바이너리) 기반 변형 도구는 구문-단위 수정을 정확하게 구현하기 어렵다는 단점이 있다. 반면에, 바이트코드 기반 변형 도구의 경우, 더 많은 코드에서 변형을 발생시킬 수 있다는 장점이 있다. 예를 들어, PIT의 경우 가장 많은 코드 라인에서 변이를 생성하였는데(표2), 이는 복잡한 소스코드 요소도 간단하고 정규화된 바이트코드로 변환된 이후에는 쉽게 변경점을 찾을 수 있기 때문이다. 따라서

표 4. 변형 연산자 별 효과적인 변이

Table 4. Useful mutants per mutation operator

	SDL	OAAN	OLLN	OLNG	ORRN
Useful mutants	289 (40.7%)	129 (18.2%)	21 (3.0%)	75 (10.6%)	196 (27.6%)

구문-단위 변형 연산자는 소스코드 기반 도구를 활용하고 소스코드 기반 도구가 변경점을 찾지 못한 부분에 대해서는 바이트코드 기반 도구를 적용하는 것이 유용할 것으로 보인다.

이에 더하여, Mutagen4J에서 구문-단위 변형 연산자의 설계에서 분석대상 코드에 대한 추가적인 분석을 수행할 경우, 그 유용성을 더 향상시킬 수 있을 것으로 보인다. 예를 들어, 구문-단위 변형 연산자인 SDL은 실험에서 총 74개의 잘못된 변이 중에 73개를 생성하였는데, 이는 return 구문을 포함한 복합 구문, 변수 초기화를 하는 구문에 대한 예외처리가 부족하였기 때문이다. 향후 제어/데이터 흐름 분석 결과를 활용하는 구문-단위 변이 연산자의 정확도를 개선할 경우, 그 유용성이 더 증대될 것으로 기대된다.

6. 결론

본 논문에서는 Java 프로그램을 대상으로 유용한 변이 프로그램을 효과적으로 생성하는 Mutagen4J 도구를 소개하였다. Jtopas를 대상으로 기존 Java 프로그램 변이 생성 도구와 Mutagen4J가 생성한 변이를 체계적으로 분석한 실험 결과, Mutagen4J은 다른 도구들보다 평균 2.3배 더 많은 유용한 프로그램 변이를 만들었다. 특히, Mutagen4J이 생성한 유용한 변이 중 40.7%는 구문 삭제 변형 연산자를 통한 것으로, Mutagen4J이 제공하는 구문 삭제 변형 연산자가 유용함을 확인할 수 있었다. 향후에는 Mutagen4J에 추가적인 구문-단위 변형 연산자를 보강하여 더 다양한 변이 생성이 가능하도록 하며, 사용자로부터 소스코드 변형 패턴을 입력 받아서 사용자 정의 변형 연산자로 활용하는 기법을 추가하여 Mutagen4J의 확장성을 개선할 계획이다.

참조문헌

- [1] J. H. Andrews, L. C. Briand, Y. Labiche, A. S. Namin, "Using Mutation Analysis for Assessing and Comparing Test Coverage Criteria", IEEE Transactions on Software Engineering, 32(8), pp.608—624, 2006
- [2] R. A. DeMillo, R. J. Lipton, F. G. Sayward, "Program mutation: a new approach to program testing", Infotech State of the Art Report on Software Testing 2, pp. 107-126, 1979
- [3] G. Fraser, A. Zeller, "Mutation-Driven Generation of Unit Tests and Oracles", IEEE Transactions on Software Engineering, 38(2), pp. 278—292, 2012
- [4] S. Moon, Y. Kim, M. Kim, S. Yoo, "Ask the mutants: Mutating faulty programs for fault localization", Proceedings of the International Conference on Software Testing, Verification and Validation, 2014

- [5] D. Schuler, A. Zeller, "Javalanche: efficient mutation testing for Java", Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, 2009
- [6] I. Moore, "Jester-a JUnit test tester", Proceedings of the International Conference on Extreme Programming and Flexible Processes (XP), 2001.
- [7] S. A. Irvine, T. Pavlinic, L. Trigg, J. G. Cleary, S. Inglis, M. Utting, "Jumble Java byte code to measure the effectiveness of unit tests", Proceedings of Testing: Academic and Industrial Conference Practice and Research Techniques (TAICPART)-MUTATION, 2007.
- [8] PIT Mutation Testing, <http://pitest.org>
- [9] R. H. Untch, "On reduced neighborhood mutation analysis using a single mutagenic operator", Proceedings of the Annual Southeast Regional Conference Article, 2009
- [10] L. Deng, J. Offutt, N. Li, "Empirical Evaluation of the Statement Deletion Mutation Operator", Proceedings of the International Conference on Software Testing, Verification and Validation, 2013
- [11] MuJava, <https://cs.gmu.edu/~offutt/mujava>
- [12] R. Just, "The Major mutation framework: Efficient and scalable mutation analysis for Java", Proceedings of the International Symposium on Software Testing and Analysis (ISSTA), 2014
- [13] Javaparser, <https://github.com/javaparser/javaparser>
- [14] J. Offutt, A. Lee, G. Rothermel, R. Untch, C. Zapf, "An experimental determination of sufficient mutation operators", ACM Transactions on Software Engineering Methodology, 5(2), pp.99—118, 1996
- [15] H. Agrawal et al., "Design of mutant operators for the C programming language", Technical Report SERC-TR-120-P, Software Engineering Research Center, Purdue University, 1989
- [16] SIR Benchmark, <http://sir.unl.edu/portal/usage.php>



홍 신

2007년 KAIST 전산학부 학사
 2010년 KAIST 전산학부 석사
 2015년 KAIST 전산학부 박사
 2016년~현재 한동대학교 전산전자
 공학부 조교수

관심분야는 소프트웨어 테스트/디버깅 자동화, 소프트웨어 변이 분석, 동시성 소프트웨어, 임베디드 소프트웨어

김 문 주

정보과학회논문지: 소프트웨어 및 응용 제 43권 제 6호



전 이 루

2015년 KAIST 전산학과 학사
 2015년~현재 KAIST 전산학부 석사과정 재학
 관심분야는 프로그램 변이 분석, 자동 프로그램 결함 위치 추정

김 윤 호

정보과학회논문지: 소프트웨어 및 응용 제 43권 제 6호