Symbolic Execution 을 통한 Code Coverage 의 향상

김진현*, 박선우*, 박용수*
*한양대학교 컴퓨터 소프트웨어학부
e-mail : sunwoo0428@hanmail.net

Code Coverage Improvement through Symbolic Execution

Jin-Hyun Kim*, Sun-Woo Park*, Yongsu Park*
*Dept. of Computer Science, Hanyang University

요 약

프로그램의 코드에 있어서 실행되지 않은 영역은 미지의 영역으로써 각종 에러와 오류의 잠재적 가능성을 지니고 있다. 개발자는 이러한 영역을 모두 검증, 테스팅 해봐야 이후 프로그램의 실행에서 예상치 못한 치명적 오류들에 대응할 수 있을 것이다. 우리는 본 논문에서 소프트웨어 테스팅의 두 가지 기법에 대하여 소개를 하고 이 두 가지를 이용하여 미실행된 영역을 실행시킬 수 있는 방법론을 제안하고자 한다. 실험에서 JaCoCo 와 SPF 를 사용하여 방법론을 적용하였고 이를 통하여 미실행 영역이 커버되는 테스트 케이스를 자동으로 얻어 낼 수 있었다.

1. 서론

IT 산업에 대한 정부의 관심이 높아짐에 따라 프로그래밍의 중요성 또한 대두되고 있다. 프로그래밍을 경험해 본 사람이라면 알겠지만, 코드의 어느 부분이 수행되지 않았는지 파악하는 것은 쉬운 일이 아니다. 코드의 길이가 짧으면 직관적으로 한눈에 파악할 수 있지만 길이가 길고 복잡해 지면 이를 파악하는 것은 상당히 어려워 질 수 있다. 미실행된 코드의 영역은 개발자가 테스팅 해보지 못한 영역으로 어떠한 잠재적 오류가 있는지 알 수가 없다. 이를 방지하고자 최대의 코드 커버리지를 얻으려는 것이고 이를 위해서 이 논문에서는 대중적으로 사용되는 JAVA 에서 코드 커버리지의 측정에서부터 심볼릭 수행을 사용하여 해당 영역이 실행되기 위한 입력 값을 찾아내는 방법론을 제안하고, 이를 JaCoCo 와 SPF 를 이용하여 검증하였다.

2. 코드 커버리지

코드 커버리지는 소프트웨어 테스팅의 척도 지표 중 하나로써 특정 테스트 케이스에 대하여 프로그램의 어느 소스 코드가 실행되었는지 확인하는 것이다. 높은 코드 커버리지를 갖는 프로그램은 소스 코드가 더 많은 부분 실행되었다는 것을 의미한다. 코드 커버리지를 측정하는 기준에 여러 가지가 존재하는데 그중 몇 가지를 간략하게 소개하도록 하겠다.

2.1 Statement 커버리지

Statement 커버리지는 각 Statement 가 테스트 되는 수준을 정의하는 척도이다. Statement 또한 세부적으로 크게 4 가지로 분류될 수 있다. 타입이나 루프가 없는 Simple Statement, ‘if, then, else’ 와 같은 Two Way Statement, Switch 문과 같은 Multi Way Statement, 반복문과 같은 Loop Statement 이다.[1]

2.2 Branch(Decision) 커버리지

Branch 커버리지 는 각 branch 의 컨트롤 구조(if 나 case 와 같은 구문)이 실행되었는지 확인하는 측정 방식이다.[2] Statement 커버리지의 단점을 보완하고자 개별 조건식이 참, 거짓인 경우 모두 테스트하고자 하는 것이다.

2.3 Path 커버리지

Path 커버리지 테스트는 개별 Statement 나 Branch 가 아닌 프로그램의 가능한 모든 논리적 경로를 커버하기 위해 사용 된다.[2] 경로는 가능한 논리적 조건에 대한 모든 조합을 시도해야 함을 의미한다. 일반적으로 메소드에 N 개의 decision 이 있다면 2^N 개의 경로를 갖게 된다.

2.4 Condition 커버리지

Conditional statement 내부에 존재하는 각각의 조건에 대해 참인지 거짓인지 체크하는 방법을 말한다. 예를 들어 if(a || b)와 같은 구문이 있다면, a 가 참, 거짓과 b 가 참, 거짓일 때를 체크하면 된다.

2.5 Modified Condition / Decision 커버리지

Condition 커버리지와 Decision 커버리지를 보완해서 만든 커버리지 기법이다. Condition statement 내부에 존재하는 각각의 조건들에 대해 가능한 모든 조합을

만든 후 한 개의 조건이 변경 됐을 때, 전체의 Condition 결과가 달라지는 조합을 찾아낸다.[3] 예를 들어 if(a || b)가 있다면,

<표 1> if(a || b)에 대한 MC/DC 커버리지

Case	a	b	Result
1	True	True	True
2	True	False	True
3	False	True	True
4	False	False	False

a 조건을 기준으로 MC/DC 를 만족하는 경우는 (2, 4)이고 b 조건을 기준으로 MC/DC 를 만족하는 경우는 (3, 4)이다. 따라서 위의 케이스에서는 MC/DC 조건이 TF,FT,FF 가 된다.

3. 코드 커버리지 Tools

3.1 EvoSuite

EvoSuite [4] 는 자바 코드로 작성된 클래스에 대한 어설 션을 사용하여 테스트 케이스를 자동 생성하는 툴이다.

3.2 Java Code Coverage(JaCoCo)

Java Code Coverage [5]는 자바 소프트웨어의 테스트 커버리지 분석을 위한 바이트 코드 분석기 툴이며 별도의 문법이나 소스 코드가 필요하지 않다.

3.3 Open Code Coverage Framework (OCCF)

기존 툴의 다양성을 극복하기 위해, 다양한 언어에 대한 커버리지를 측정하는 새로운 접근법이 개발되었는데 이를 오픈 코드 커버리지 프레임워크라고 부른다.[6]

4. 코드 커버리지 Tools 비교 분석

4.1 지원 가능 언어

모든 도구는 프로그래밍 언어를 지원한다. <표 2> 는 지원 언어 목록을 보여준다.

<표 2> 툴들의 지원 가능 언어 비교

툴 이름	자바	그 외
EvoSuite	O	X
JaCoCo	O	X
OCCF	O	O

4.2 커버리지 측정 방식

Statement, Branch, Function 등 다양한 커버리지 측정법이 있다. Statement 커버리지를 통해 우리는 실행된 Statement 와 코드가 실행되지 않은 부분을 식별할 수 있다. Decision 커버리지는 참 및 거짓 조건을 다룬다. Function 커버리지[2]는 디자인한 기능에 대한 측정값이다. Function 커버리지는 디자인 의도와 관련이 있으며 코드 커버리지는 디자인 구현도를 측정한다.

<표 3> 툴들의 커버리지 측정 방식 비교

툴 이름	Statement /Line	Branch /Decision	Method /Function
EvoSuite	X	O	X
JaCoCo	O	O	O
OCCF	O	O	X

5. 심볼릭 수행

심볼릭 수행은 소프트웨어 테스팅 기법의 하나로써 어떠한 입력 값이 프로그램의 어느 부분을 동작시키는지 분석하는 방식이다.[7] 심볼릭 수행은 수행 방식에 따라 다양하게 존재하는데 그 중 몇 가지를 간단히 소개해보겠다.

Generalized 심볼릭 수행은 기본 방식에 멀티스레딩과 재귀의 데이터값 처리가 추가된 형태이다. Directed Automated Random Testing(DART) 는 동적으로 수행되는 방식이다. Execution Generated Testing 은 Concrete/Symbolic 두 가지 방식을 함께 사용하는 기법이다.

6. 심볼릭 수행 Tools

6.1 SPF

SPF(Symbolic PathFinder) 는 자바 바이트코드와 statechart 모델을 모두 분석할 수 있고, 정수와 실수가 섞여 있는 제약조건, 복잡한 수식의 조건들을 경험적 방식을 통하여 풀 수 있다.[8]

6.2 DART

DART 툴은 Directed Automated Random Testing 을 구현한다.[9].

6.3 CUTE and jCUTE

이 두가지 툴은 Illinois 대학에서 개발된 것으로 C 와 자바 프로그램용으로 개발되었다. 이들은 Concolic 테스팅과 입력 데이터 구조와 멀티스레딩의 처리를 목적으로 구현되었다.

6.4 CREST

CREST 는 오픈소스 툴로 C 프로그램의 Concolic 테스팅의 용도로 개발되었다.

6.5 Pex

Pex 는 동적 심볼릭 수행을 구현한 것으로 C#, VisualBasic, F#과 같은 .NET code 를 위해 개발되었다.[10]

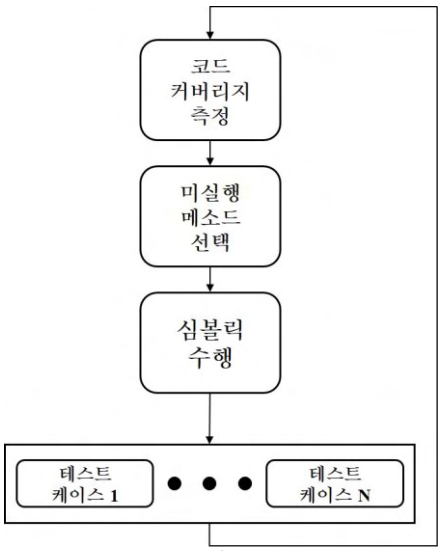
6.6 KLEE

KLEE 는 LLVM[11]을 기반으로 만들어진 완전한 EXE 툴의 재설계 형태이다. EXE 처럼 KLEE 도 콘크리트와 심볼릭의 융합의 형태로 동작하며 메모리를 비트 단위의 정확성으로 모델링하고 다양한 Solver 최적화를 지원한다.

7. 제안 방법

본 절에서는 이 논문에서 진행하고자 하는 실험에 대한 방법론을 제시하고 이에 해당하는 구체적 진행

방식을 소개하고자 한다. 앞서 설명한 소프트웨어 테스팅의 큰 두 가지 방법을 접목해 미실행된 코드를 탐색하고 이를 커버시킬 적절한 입력값을 찾고자 하는 방법이다.



(그림 1) 제안 방법론

(그림 1)은 실험에서 진행하고자 하는 방법론을 간단한 플로우 차트의 형식으로 표현한 것이다. 코드 커버리지 툴을 사용하여 코드의 미실행 영역을 우선적으로 탐색한다. 그 뒤 해당 영역을 포함하고 있는 메소드를 찾아서 이를 심볼릭하게 수행한다. 마지막으로 생성된 여러 개의 테스트 케이스를 통하여 해당 영역이 실행되기 위한 적절한 테스트 케이스를 찾는다. 심볼릭 수행을 통해 생성되는 테스트 케이스는 해당 영역으로 접근하는 모든 경로가 나오기 때문에 테스트 케이스는 다수의 결과값이 나온다.

위 관련 연구들을 통해서 소개한 다양한 툴들 중 코드 커버리지 측정용 툴로는 JaCoCo, 심볼릭 수행용 툴로는 Java PathFinder 의 SPF 를 사용할 것이다.

(그림 1)에 대하여 자세히 설명하자면 우선 원하는 자바 코드를 JaCoCo 를 사용하여 코드 커버리지를 측정한다. JaCoCo 는 다양한 형식으로 분석 결과를 생성하는데, 이번 실험에서는 편의상 직관적인 html 파일을 사용하겠다. JaCoCo 를 실행하면 html 파일로 커버되지 않은 영역이 붉은색으로 하이라이트 된다. 이 영역을 커버하기 위하여 다음 과정은 이 영역을 포함하고 있는 메소드를 찾는 것이다. 해당 영역을 포함하고 있는 메소드를 찾아서 이를 심볼릭 수행의 타겟으로 삼는 .jpf 파일을 생성한다. 그 뒤 생성된 .jpf 파일을 기반으로 심볼릭 수행을 진행하여 출력로그의 형태로 여러 개의 테스트 케이스를 받는다. 결과로 나온 테스트 케이스를 바탕으로 코드 커버리지를 재수행하면서 미실행 된 영역이 수행되는 적절한 테스트 케이스를 찾으면 된다.

8. 실험 결과

자바 프로그램의 코드 커버리지를 JaCoCo 로 측정하고, 커버 되지 않은 영역에 대하여 SPF 로 가능한 경로에 대한 입력 값을 얻고 이를 통해서 코드 커버리지가 향상되는지를 확인하였다.

JaCoCo Ant Example > default > TestPaths.java

TestPaths.java

```
1.  public class TestPaths {
2.
3.    public static void main (String[] args){
4.      System.out.println("----------Test Start----------");
5.      (new TestPaths()).exampleMethod(90,true);
6.    }
7.
8.    public void exampleMethod (int x, boolean b) {
9.       System.out.println("----------First Step----------");
10.        if (b) {
11.          if (x <= 100){
12.              System.out.println("  <= 100");
13.          }
14.          if(x <= 50){
15.              System.out.println("  <= 50");
16.          }
17.        }
18.     }
19.
20. }
```

(그림 2) 초기 코드 커버리지 결과

JaCoCo 를 이용하여 exampleMethod 에 (90, true)를 테스트 세트로 전달했을 시 코드 커버리지를 측정한 모습이다. 15 번째 행이 커버되지 않았음을 알 수 있다.

```
target=TestPaths

classpath=${jpf-symbc}/build/examples

sourcepath=${jpf-symbc}/src/examples

symbolic.method= TestPaths.exampleMethod(sym#con)

symbolic.min_int=0

symbolic.max_int=200

listener = gov.nasa.jpf.symbc.sequences.SymbolicSequenceListener

vm.storage.class=nil
```

(그림 3) SPF 실행을 위한 .jpf 파일

커버되지 않은 부분을 해결하기 위해서 심볼릭 수행 툴인 SPF 를 이용하여 커버되지 않은 부분을 커버할 수 있는 입력값을 얻어낸다. 먼저 SPF 실행을 위한 .jpf 파일을 작성한다. exampleMethod 의 1 번째 인자를 심볼릭하게, 2 번째 인자를 콘크리트하게 설정하였고, int 의 범위를 0~200 으로 설정해주었다.

```
================== search started:
----------Test Start----------
----------First Step----------
  <= 100
  <= 50
constraint # = 2
x <= 50 &&
x <= 100
constraint # = 2
x > 50 &&
x <= 100
constraint # = 1
x > 100
================== Method Sequences
[       testpaths.exampleMethod(0,true);]
[       testpaths.exampleMethod(51,true);]
[       testpaths.exampleMethod(101,true);]
```

(그림 4) SPF 실행 로그 일부

실행 결과는 (그림 4)과 같고, 두 가지 조건인 '<= 100', '<= 50' 에 대해서 3 가지 경로가 도출되었다. 각 경로에 대한 예시로 (0, true), (51, true), (101, true)가 도출되었고 실제로 도출된 값으로 커버리지를 측정해

보았다.

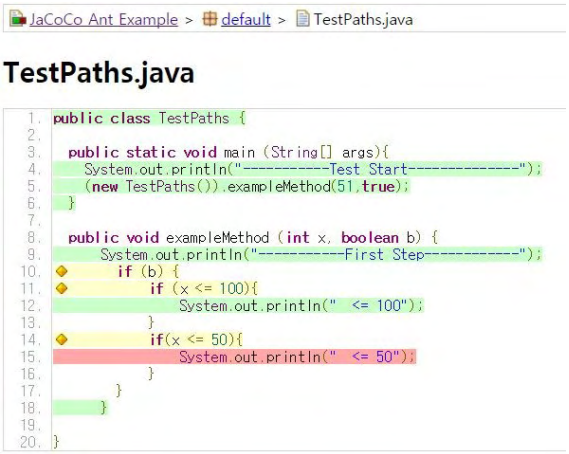
JaCoCo Ant Example > default > TestPaths.java

TestPaths.java

```
1.  public class TestPaths {
2.
3.    public static void main (String[] args){
4.      System.out.println("----------Test Start----------");
5.      (new TestPaths()).exampleMethod(101,true);
6.    }
7.
8.    public void exampleMethod (int x, boolean b) {
9.        System.out.println("----------First Step----------");
10.       if (b) {
11.           if (x <= 100){
12.               System.out.println("  <= 100");
13.           }
14.           if(x <= 50){
15.               System.out.println("  <= 50");
16.           }
17.       }
18.   }
19.
20. }
```

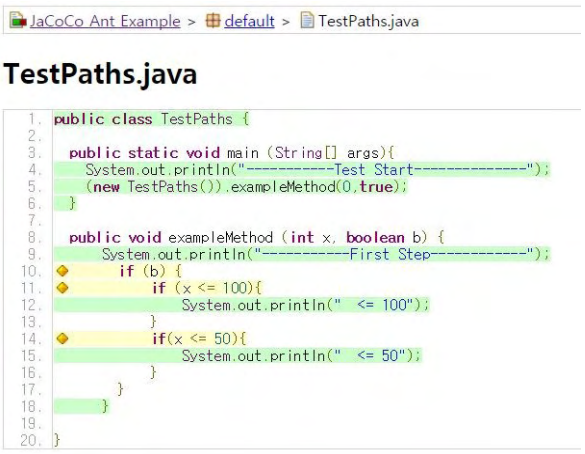
(그림 5) (101, true) 에 대한 코드 커버리지

(101, true)로 인자를 전달했을 시 커버리지 비율이 더 악화되는 경로를 제시했음을 알 수 있다.



(그림 6) (51, true) 에 대한 코드 커버리지

(51, true)로 인자를 전달했을 시 지난 실행과 같은 커버리지 비율을 보임을 알 수 있다.



(그림 7) (0, true) 에 대한 코드 커버리지

(0, true)로 인자를 전달했을 시 지난 실행에서 커버되지 않았던 부분이 커버 되었음을 확인할 수 있다.

9. 결론

우리는 자바 언어에서 코드커버리지를 심볼릭 수행을통해 넓힐 수 있는 방법론을 제시한다. 제시한 방법론의 실효성을 검증하기 위하여 위에서 제시한 많은 연구 방식과 툴들을 사용하여 실험을 진행해 보았다. 심볼릭 수행은 가능한 모든 경로에 대한 입력 값을 도출하므로, 제시된 입력 값 중에서 적절한 선택을 하여 커버리지 비율을 높일 수 있음을 확인할 수 있다. 이처럼 두 가지 소프트웨어 테스팅 기법을 접목하여 코드의 커버리지를 향상 시킬 수 있었다. 이를 통해 서론에서 언급했던 테스팅 하지 못한 영역에 대한 잠재적 오류를 줄여나갈 수 있을 것이다.

참고문헌

[1] S. Pathy “A Review on Code Coverage Analysis” International Journal of Computer Science & Engineering Technology (IJCSET)

[2] https://en.wikipedia.org/wiki/Code_coverage

[3]https://en.wikipedia.org/wiki/Modified_condition/decision_coverage

[4] G. Fraser and A. Arcuri, “Evolutionary Generation of Whole Test Suites,” Proc. 11th Int’l Conf. Quality Software, pp. 31-40, 2011

[5] R. Lingampally, A. Gupta, P. Jalote. "A Multipurpose Code Coverage Tool for Java," In Proceedings of the 40thAnnual Hawaii International Conference on System Sciences, IEEE Computer Society, 261b, 2007

[6] Kazunori Sakamoto, et al.,”A Framework for Measuring TestCoverage Supporting Multiple Programming Languages”,First Software Engineering Postgraduates Workshop (SEPoW 2009; In conjunction with APSEC 2009), 2009. Sakamoto

[7] https://en.wikipedia.org/wiki/Symbolic_execution

[8] M. Staats and C. S. Pasareanu. Parallel symbolicexecution for structural test generation. In ISSTA’10, July 2010.

[9] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In PLDI’05, 2005.

[10] N. Tillmann and J. de Halleux. Pex - white box test generation for .NET. In TAP’08, Apr 2008.

[11] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In CGO’04, Mar 2004.