결함 위치 추적을 위한 테스트 케이스 자동 생성 기법

박창용*, 김준희**, 류성태**, 윤현상*, 이은석*
성균관대학교 전자전기컴퓨터공학과*
성균관대학교 IT융합학과**
e-mail:ckddyd777@gmail.com, comemail@skku.edu, xenoz0718@gmail.com,
wizehack@gmail.com, eslee@ece.skku.ac.kr

Test Case Automatic Generation for Fault Localization

Changyong Park*, Junhee Kim**, Sungtae Ryu**, Hyunsang Youn*, Eunseok Lee*
Dept of Computer Electrical and Engineering, Sungkyunkwan University*
Dept of IT Convergence, Sungkyunkwan University**

요 약

오늘날 소프트웨어가 가지는 규모와 복잡성은 날로 심화되고 있으며, 소프트웨어 개발 시 결함을 찾아내기 위한 테스트에 많은 시간이 소모되고 있는 실정이다. 이러한 문제점을 해결하기 위해 중요한 기술 중 하나가 결함 위치 추적(Fault Localization)이다. 이 기법을 이용하여 결함을 추적하기 위해서는 다량의 테스트 케이스를 필요로 하며, 추가로 테스트 케이스를 작성하는 것은 또 다른 개발 부하이다. 본 논문에서는 이를 해결하기 위해서 분기별 입력 조합 기반 테스트 케이스 생성방법과 시드 결과 기반 테스트 케이스 생성방법을 제안하였다. 개발자는 본 생성방법을 통해 테스트 케이스 생성에 대한 비용 절감을 기대 할 수 있다. 제안하는 내용의 효용성을 검증하기 위해 실제 예제 코드에 적용하여 평가하였다. 두 가지 방법 모두 무작위 생성한 테스트 케이스에 비해 개발자가 직접 생성하는 것과 유사한 테스트 케이스를 생성하고, 제안 방법으로 생성한 테스트 케이스의 신뢰성을 확인하였다.

키워드 : 소프트웨어 테스팅, 결함 위치 추적, 코드 커버리지, 테스트 케이스 자동 생성

1. 서론

현대의 소프트웨어가 가지는 규모와 복잡성은 날로 심화되고 있다. 소프트웨어를 개발하기 위해 소스코드를 작성하고 테스트하기 위한 과정에 많은 시간이 소모된다. 특히 구조가 복잡하거나 규모가 큰 소프트웨어의 경우 테스트 과정 중 발생한 결함의 정확한 위치 및 상세한 원인을 파악하기 어려우며 많은 시간을 소비하게 하여 개발 비용 및 기간 증가의 원인이 되기도 한다.

특히 소프트웨어 개발 프로세스 중 테스팅 과정에서 결함이 발생할 경우 당시의 발생 조건을 재연하기 어려운 경우가 많다. 만약 테스트 케이스를 재연하여 결함이 발생하는 것을 확인하게 되더라도 원인이 되는 부분의 정확한 위치를 찾아내는 것은 쉽지 않다. 현실적으로 이러한 원인을 규명하기 위해 개발자는 코드 리뷰라는 과정을 거쳐야 하는데 이것은 많은 시간을 소비시키며 전체 개발 시간을 증가시키는 큰 원인이 된다.

이러한 문제점을 해결하기 위해 제안된 방법 중 하나가 결함 위치 추적이다. 결함을 추적하기 위해서는 결함을 찾기 위한 복수의 테스트를 거쳐야 한다. 테스트를 위해서는 입력값과 그에 상응하는 출력값으로 구성되어 있는 테스트 케이스가 필요하다. 테스트 케이스를 개발자가 직접 작성하는 방법은 비교적 정확성은 높다는 장점이 있지만 작성 시간이 많이 소모된다는 치명적인 단점이 있다.

테스트 케이스를 자동으로 생성하는 방법은 이 문제점을 해결하는 데 도움을 줄 수 있다. 하지만 이 경우에 잠재적인 결함이 있는 대상 소프트웨어의 소스 코드를 통해 생성된 테스트 케이스의 결과값은 신뢰하기 어려울 수밖에 없다는 추가적인 문제점이 발생한다.

따라서 본 논문에서는 특정 조건 하에서 자동으로 생성하는 테스트 케이스의 결과값을 기대값에 최대한 근사하게 추정하는 방법에 대해 제안한다. 이를 통해 개발자는 비교적 신뢰성이 높은 테스트 케이스를 자동으로 생성할 수 있고 이를 통해 작성 시간을 줄일 수 있다.

본 논문의 구성은 다음과 같다. 먼저 2장에서 결함 위치 추적과 관련된 연구와 테스트 케이스 자동생성 기법에 대해 소개하고 3장에서는 기존 테스트 케이스 생성과정의 문제점을 정리한다. 4장에서 본 연구가 제안하는 테스트 케이스 자동 생성 과정에서 비교적 정확한 결과값을 추론하는 방법론에 대해서 소개한다. 5장에서는 실제 테스트 케이스 자동 생성 실험 결과를 분석하고 마지막으로 6장에서 본 연구의 결론과 향후 연구 방향에 대해 제시한다.

2. 관련 연구

결함 위치 추적이란 소프트웨어 내에 실행되는 각 구문을 테스트하여 버그가 포함되어 있을 확률을 계산하고 가장

본 연구는 정보통신산업진흥원의 IT/SW 창의연구과정의 연구결과로 지식경제부와 주관기업명에 의해 지원된 과제로 수행되었음.
(NIPA-2011-C1810-1104-0013)

높은 확률을 가진 구문을 결함 발생 가능 위치로 찾아내는 것을 의미한다.[1]

본 장에서는 결함위치추적과 관련하여 현재까지 제안된 방법 중 코드 커버리지 기반 결함 위치 추적과 자료 흐름 커버리지 기반 결함 위치 추적에 대해 정리하고, 테스트 케이스 생성 방법론에 대해 소개한다.

2.1. 코드 커버리지 기반 결함 위치 추적

코드 커버리지 기반 위치 추적은 테스트의 대상을 소스코드로 하여 각 소스코드에 대한 커버리지를 계산하여 결함을 추적하는 방식이다.[2] 이 방식은 소스코드를 어떻게 분할하여 분석할 것인지에 따라 결함 추적 결과가 달라질 수 있다. [10]은 코드 커버리지의 대상을 전반적인 프로세스, 유닛, 함수, 시스템 4가지로 구분하여 각 대상의 코드 커버리지를 계산하는 방법을 제안하였다. 또 [11]은 대상을 상태와 분기로 나누어 코드 커버리지를 계산하여 결함의 위치 추적하는 방법을 연구하였다.

2.2. 자료 흐름 커버리지 기반 결함 위치 추적

자료 흐름(Data-Flow) 커버리지 기반 결함 위치 추적 기법 역시 테스트 대상이 소스코드라는 점에서 코드 커버리지 기반의 결함 위치 추적과 유사하다. 하지만 실행되는 모든 항목을 자료의 흐름으로 나누어 각 구성 요소에 대한 커버리지를 따로 계산하는 것에서 그 차이가 있다.[3][7]은 각 커버리지에 대해 계산하는 방법과 각 커버리지를 조합하여 데이터의 흐름을 추적하여 커버리지를 높이는 방법에 대해 연구하였다. [6]은 각 코드의 실제 사용범위에 따라 데이터를 분할하여 결함의 위치 추적하는 방법에 대해 제안하였다.

2.3. 테스트 케이스 자동 생성 방법론

테스트 케이스를 자동생성 하는 방법에는 일반적으로 무작위 생성 방식을 기본으로 한다. 대부분의 테스트 케이스 자동 생성과 관련한 연구들은 테스트 케이스의 개수를 최소화하면서 모든 분기를 거칠 수 있도록 하는 데 그 중심을 두고 있다. 다시 말해, 코드 커버리지를 최대화하는 테스트 케이스의 자동 생성에 그 목적이 존재한다고 할 수 있다. [1][8][9]는 분기 조건을 이용해 코드 커버리지를 최대한 높이면서 테스트 개수는 최소화 하는 결함 위치 추적용 테스트 케이스 생성 방법을 제안하고 웹 어플리케이션 개발 프로세스를 대상으로 검증하였다. 하지만 테스트 케이스의 입력값을 무작위로 생성하고 이를 결함 검출 대상이 되는 소스코드를 통하여 실행한 결과값을 테스트 케이스로 구성하여 사용하기 때문에 정확성이 떨어진다는 단점이 있다.

3. 문제점

결함 위치를 정확하게 추적하기 위해서는 신뢰할 수 있는 테스트 케이스를 생성해야 한다. 테스트 케이스를 생성하는 방법은 개발자가 입력값에 대한 정확한 출력값의 조합을 수작업으로 생성하는 방법이 있다. 이 방법은 입출력값의 정확성이 높아 결함 위치 추적을 성공할 가능성이 높지만 개발자가 이를 생성하는 것에 비용이 많이 소모된다. 테스트 케이스 생성 비용을 줄이는 방법으로 테스트를 해야 할 소프트웨어에 맞는 입력값을 통해 출력되는 값을 테스트 케이스 집합으로 사용하는 것이다.

```
1.  getMid() {
2.      int x,y,z,m;
3.      read("Enter 3 numbers : ",x,y,z);
4.      m=z;
5.      if(y<z)
6.          if(x<y)
7.              m = y;
8.          else if(x<z)
9.              m=y;    // ***bug***
10.     else
11.         if(x>y)
12.             m=y;
13.         else if(x>z)
14.             m=x;
15.     print("Middle number is:", m);
16. }
```

(그림 1) 결함이 포함된 예제 소스 코드

하지만 잠재적인 결함이 있는 소프트웨어를 대상으로 하여 생성되는 결과값은 신뢰할 수 없는 결과값이 된다. 그러므로 앞의 방법으로 생성된 테스트 케이스로 결함을 추적하여 생성된 결과는 신뢰할 수 없는 값이 될 수 있다.

그림1의 결함이 있는 소스코드로 테스트 케이스를 생성하면 입력값 2, 1, 3과 출력값 1이라는 잘못된 테스트 케이스를 생성하게 되므로 사용할 수 없는 테스트 케이스가 된다. 이 결과를 통해 잠재적인 결함이 있는 소프트웨어를 대상으로 생성된 테스트 케이스의 일부는 신뢰할 수 없다는 것을 확인할 수 있다.

다시 말해 정확하지 않은 출력값을 테스트 케이스로 사용하는 결함 위치 추적은 정확성이 떨어지므로 새로운 방법을 적용하여 테스트 케이스를 자동 생성 시 출력값에 대한 신뢰성을 최대한 높일 수 있도록 해야 한다.

4. 테스트 케이스 생성 방법

본 논문에서 제안하는 방법은 총 2가지이다. 첫 번째는 분기별로 입력값이 조합될 수 있는 모든 경우의 수를 조합하여 결과에 대한 집합을 구성한다. 그 중 가장 출현 횟수가 많은 값들의 집합을 기대되는 결과 값으로 추정하여 정확도가 높은 테스트 케이스가 생성되도록 하는 방법이다. 두 번째는 주어진 시드 값과의 차이가 가장 적은 출력값의 집합을 기대되는 결과값으로 추정하여 정확도를 높이는 테스트 케이스 생성 방법을 제안한다.

4.1. 분기별 입력 조합 기반 테스트 케이스 생성

결함의 위치를 추적하고자 하는 대상 소스 코드가 많은 분기로 나뉘어져 있어 다양한 경우의 수가 존재할 때 본 제안 방법이 유용하다. 이 테스트 케이스 생성 방법을 사

```
<Method 1>
Condition : Input value's property is equal
Parameter : Program P, Number of input

Input[] =: MakeRandNumber();
NumberOfMaxCombination =: !NumberOfInput;
foreach i=0...NumbeerOfMaxCombination do
  Output[i] = getCombination(Input);
BestOccur =: 0;
BestOutput =: 0;
while(Output[] is not empty) do
  OutputOccurs[]=:CompareCombination(Output[])
  if OutputOccurs[i] > BestOccur then
    BestOccur =: OutputOccurs[i];
return BestOccur;

<Method 2>
Condition : Seed Input, Seed Output was entered.
Parameter : Program P, Seed Input, Seed Output

Input[] =: MakeRandNumber();
Count =: getCounter(Input);
Differ[i] = {0};
BestOccur =: 0;
foreach i=0...Count do
  Output[i] = getCombination(Input);
  Differ[i] = Output[i] - Seed Output;
  if BestOccur>Differ[i] then
    BestOccur =: Output[i];
return BestOccur;
```

(그림 2) 제안 방법의 의사코드

용하기 위한 구체적인 전제 조건은 다음과 같다.

- 입력값의 부분집합에 속하는 원소(변수 또는 객체)들에게서 대등한 관계가 발견된다.
- 구조가 참/거짓의 분기를 포함하며 위 조건의 원소와 관계가 있다.

제안하는 방법은 먼저, 하나의 입력값의 집합으로 가능한 모든 조합을 생성한다. 각각의 조합에서 결과값들의 집합 중 가장 많은 경우의 수를 보이는 결과값을 신뢰할 수 있는 것으로 보고 이를 테스트 케이스로 생성한다. 수식(1)은 기대하는 결과값에 가장 가까운 값을 취하기 위한 제안 방법을 수식으로 표현한 것이다.

$$output = most(\cup T(i_n)) \\ i_n \subset combination(input) \\ n = !count(input) \qquad (1)$$

위 수식은 입력받은 값들의 개수를 n이라 하고. 입력값들의 집합을 T라고 했을 때 ${}_nT_n$개 만큼의 조합을 생성한다. 조합에 포함된 값 중 가장 빈도가 높은 값을 신뢰할 수 있는 출력값으로 사용한다. 예를 들어 x, y, z라는 변수가 입력값일 때 3개의 변수의 자료형을 검사하여 동일한 자료형의 집합일 경우 첫 번째 방법을 사용한다. 변수의 집합 T에 대한 6개의 조합을 생성하고 그 조합을 통해 나오는 6개의 결과값 중 가장 빈도가 높은 결과값을 테스트 케이스의 검증된 결과값으로 사용한다. 그림2에는 첫 번째 제안 방법의 의사 코드가 나타나 있다.

4.2. 시드 결과 기반 테스트 케이스 생성

결함 위치 추적을 위한 두 번째 테스트 케이스 자동 생성 방법은 대상 소스 코드가 1회 이상의 반복문을 포함하며, 그 결과가 복수의 원소를 가진 배열 형태를 가질 때 유용한 방법이다. 이 테스트 케이스 생성 방법을 사용하기 위한 전제 조건은 다음과 같다.

- 개발자가 시드(Seed) 입력값과 시드 출력값을 입력한다.
- 시드 입력값과 시드 출력값의 상관관계 성립한다.

이 방법은 시드 입력값과 시드 출력값을 필요로 한다. 이를 기반으로 테스트 케이스 자동 생성을 하는데 무작위로 생성된 입력값들의 집합으로 생성된 결과값을 시드 출력값과 비교하여 가장 유사한 형태와 구조를 띄는 값을 최종 결과값으로 취한다. 수식(2)는 기대하는 결과값에 가장 가까운 값을 취하기 위한 제안 방법을 수식으로 표현한 것이다.

$$output = MAX_{i=0}^{n}(S-combination(I_n)) \quad (2)$$

위 수식은 시드 출력값을 S라고 한다. 생성된 입력값을 I라고 하고 입력받은 값들의 개수를 n이라 했을 때 입력값에 대한 조합을 $n!$개만큼 생성하고 생성된 조합 중 시드 출력값과의 차이가 가장 적은 값은 신뢰할 수 있는 출력값으로 사용한다. 예를 들어 시드 입력값은 1, 3, 5, 7일 때 자동 생성된 값이 3, 2, 9, 4일 경우 입력값들의 조합을 생성하여 시드 출력값과의 차이를 계산한다. 예를 들어 시드 출력값 1, 3, 8, 10으로 생성된 값 2, 3, 9, 4와의 차이를 계산한다. |1-2| + |3-3| + |8-9| + |10-4|은 8이고 생성된 조합 중 2, 3, 4, 9와의 차이는 6이므로 시드 출력값과의 차이가 6인 2, 3, 4, 9를 신뢰할 수 있는 출력값으로 추정한다. 그림 2의 두 번째 단락은 실제로 두 번째 실험을 위해 작성된 의사코드이다.

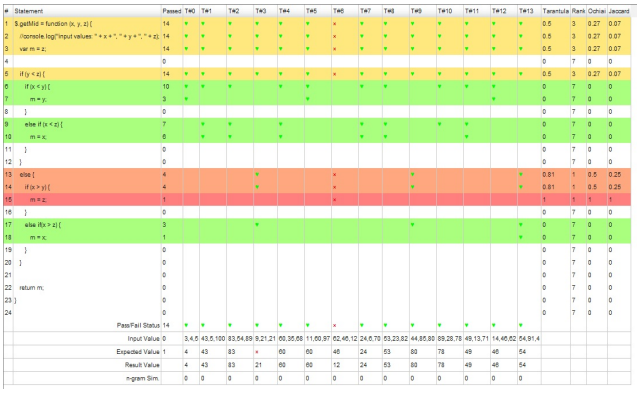
5. 평가

분기별 입력 조합 기반 테스트 케이스 생성방법과 시드 결과 기반 테스트 케이스 생성방법을 그림2의 의사코드를 기반으로 Javascript 코드를 이용하여 실제로 구현하였다. 본 실험에 사용된 예제는 총 7가지의 예제로서 그림 3은 이 예제 중 하나인 그림 1의 예제 코드를 이용한 결함 위치 추적 결과를 나타낸다.

본 예제는 세 개의 입력값을 통해 그 숫자 중 중간 값을 출력해주는 메소드이다. 하지만 각 예제에는 정확한 실

<표 3> 예제 소스코드

함수명	설명
getMid()	중간값 구하는 함수
getMax()	최대값 구하는 함수
getMin()	최소값 구하는 함수
getTri()	삼각형 종류 구하는 함수
Sort()	정렬 함수
Reverse()	문자열 뒤집는 함수
getScore()	학생 성적 읽어오는 함수



(그림 3) Javascript를 기반으로 작성한 결함 위치 추적 지원 도구

험을 위해서 임의로 결함을 발생시켰다. <표 2>은 제안하는 첫 번째 테스트 케이스 생성 방법을 평가하기 위해 총 6가지의 대상 코드를 결함 추적을 위한 테스트 코드를 직접 생성한 경우, 무작위로 생성한 경우, 제안 방법을 이용하여 생성한 경우를 비교한 것이다. 정확도는 다음과 같이 정의하여 측정하였다.

$$Accuracy = suspicousness\ of\ rank1 - suspicousness\ of\ rank2 \quad (3)$$

<표 2>에서 직접 생성은 개발자가 결함 추적을 위해 테스트 케이스를 실제로 작성한 것이지만 예제 소스코드에 의존적이므로 정확도가 높지 않고 작성 시간이 많이 들기 때문에 비용 효율성이 떨어지는 편이다. 무작위 생성의 경우에는 결함 발견을 해내는 정확도가 상대적으로 떨어진다. 하지만 제안하는 방법을 이용하여 직접 생성하는 것보다 높은 정확도를 보이므로 제안 방법이 유용하다고 평가할 수 있다.

두 번째 테스트 케이스 생성 방법론을 평가하기 위해 테스트 케이스의 생성 결과와 실제 정상적인 결과를 비교 분석하였다. <표 3>에서 두 번째 방법의 경우 입력값에 특정한 연관성이 있을 경우 완벽에 가깝게 테스트 케이스를 생성할 수 있지만 연관성이 없는 경우 유사도를 측정하기 어렵다. 결함 추적 과정에서 실제 결과와 방법론을 통해 생성된 추정 결과의 유사도를 통해 테스트 통과 여부를 판단해야 한다는 추가적인 테스팅 방법을 필요로 한다는 점과 시드입력과 시드출력 사이의 상관관계가 존재하지 않는 값들에서는 유사도의 정확성이 낮아지는 한계점이 존재한다.

6. 결론 및 향후 과제

본 연구에서는 결함 위치 추적을 하기 위한 테스트 케이스 자동 생성 할 때 발생하는 테스트 케이스의 정확성을 높이는 방법을 제안하였다. 제안한 두 가지 방법을 통하여 일반적인 테스트 케이스 생성 방법보다 정확성이 높아질 수 있음을 확인하였다. 하지만 두 방법 모두 제약 조건이 존재하고 한계점이 존재하므로 상황에 맞는 수정이 필요하다.

향후에는 입력값의 속성이 다른 다양한 경우에도 테스

<표 2> 첫 번째 생성 방법의 정확도 측정 (단위:Accuracy)

대상 코드	직접 생성	무작위생성		제안방법		
		시도 1	시도 2	시도 1	시도 2	시도3
예제1	0.21	0.44	0.32	0.19	0.09	0.20
예제2	0.23	0.46	0.44	0.23	0.12	0.18
예제3	0.21	0.40	0.38	0.00	0.24	0.12
예제4	0.22	0.42	0.38	0.12	0.10	0.22

<표 3> 두 번째 생성 방법의 결과값 정확도 비교

대상 코드	제안방법				
	생성결과1	생성결과2	생성결과3	생성결과4	생성결과5
	실제결과1	실제결과2	실제결과3	실제결과4	실제결과5
	n-gram 유사도	n-gram 유사도	n-gram 유사도	n-gram 유사도	n-gram 유사도
예제5	1,3,6,8,9	1,6,7,10	2,6,8,9	1,4,7	1,9
	1,3,6,8,9	1,6,7,10	2,6,8,9	1,4,7	1,9
	1	1	1	1	1
예제6	3,7,6,9	5,4,8,2	6,7,9	2,6,9,1	4,6,3,8
	3,7,6,9	5,4,8,2	6,7,9	2,6,9,1	4,6,3,8
	1	1	1	1	1
예제7	92,88	92,88,95,99 ,85,81	95,85,97,81 ,99,100,65	92,88,76,81	92,88,95,97 ,85
	92,88	99,85,95,92 ,88,81	97,65,85,10 0,95,99,81	92,88,81,76	92,85,95,99 ,85
	1	0.22	0.22	0.5	0

트 케이스를 생성하는 방법과 입출력 데이터간의 상관관계가 없는 경우에 사용할 수 있는 테스트 케이스 생성방법을 연구할 예정이다.

참고문헌

[1] Shay Artzi, "Directed Test Generation for Effective Fault Localization", In ISSTA'10

[2] Brian Marick "How to Misuse Code Coverage", In Reliable Software Technologies

[3] Hiralal Agrawal "Fault Localization using Execution Slices and Dataflow Tests", In IEEE

[4] James A. Jones and Mary Jean Harrold " Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique", In ASE'05

[5] Shay Artzi, "Fault Localization for Dynamic Web Applications", In IEEE '10

[6] Xiangyu Zhang "Experimental Evaluation of Using Dynamic Slices for Fault Location", In AADEBUG '05

[7] J. R. Horgan "Data Flow Coverage and the C Language", In ACM'91

[8] Shay Artzi, "Finding Bugs in Web Applications Using Dynamic Test Generation and Explicit-State Model Checking", In IEEE '10

[9] Shay Artzi, "Fault Localization for Dynamic Web Applications", In IEEE'10

[10] Paul Piwowarski "Coverage Measurement Experience During Function Test" In IEEE'93

[11] John Joseph Chilenski, "Applicablility of modified condition/decision coverage to software testing", In Software Engineering Journal '94