자동 유닛 테스팅을 활용한 효과적인 지속적 자동 테스팅 시스템

김도윤 김성민 홍신

한동대학교 전산전자공학부
{21700083, 21300109, hongshin}@handong.edu

Effective Continuous Testing with Automated Unit Test Generation Technique

요약

본 논문은 유닛 테스트 생성 기법을 연속적으로 개발되는 프로젝트의 지속적인 테스팅에 효과적으로 적용하는 CTG-Randoop 프레임워크를 제시한다. CTG-Randoop은 버전별로 유닛 테스트 생성 결과를 상호 비교하여 특정 버전에서 새롭게 발견된 오류만을 사용자에게 보고함으로써 유닛 테스트 생성이 갖는 오경보 문제를 제한한다. 유닛 테스트 생성 기법인 Randoop을 활용하는 CTG-Randoop을 구현하고 Defects4J의 여섯 개 결함 사례를 사용한 실험을 통해, Randoop을 단독으로 사용한 경우에 발생하는 오경보의 최대 66%를 제거하면서도 효과적인 오류 검출을 성취할 수 있음을 확인하였다.

1. 서론

유닛 테스트를 활용한 회기 테스팅(regression testing)은 소프트웨어 프로젝트의 품질을 보장하는 주요 방법으로 오늘날 많은 소프트웨어 프로젝트에서 개발 초기부터 유닛 테스트 케이스를 작성한 후, 프로젝트 형상 관리 및 지속적 통합 과정의 일부로서 매 코드 변경마다 유닛 테스트 케이스를 실행하는 개발 방법론을 시행하고 있다. 이러한 회기 테스팅의 수행은 잘못된 코드 변경을 단시간에 발견함으로써 결함으로 인한 비용과 위험을 줄일 수 있다는 장점이 있는 반면, 지속적 회기 테스팅을 위해 코드 추가/변경마다 새로운 유닛 테스트를 추가해야 하며, 다양한 프로그램 기능을 면밀히 검사하기 위해서는 세분화된 다량의 유닛 테스트를 작성해야 하는 비용이 따른다는 문제점을 갖는다.

유닛 테스트 생성 기법[1][2]은 테스트 대상 프로그램의 정적, 동적 정보를 활용하여 해당 코드에 대한 유닛 테스트 케이스를 생성하는 기법이다. 자동 유닛 테스트 생성 기법을 연속적으로 개발되는 프로젝트 상황에서 적용하여 회기 테스팅에 필요한 유닛 테스트 케이스를 자동으로 구축하는데 사용할 수 있으나, 현재 개발된 유닛 테스트 생성 기법의 경우, 테스트 대상 프로그램에 대한 기능 요구사항을 올바르게 고려할 수 없어 많은 오경보(false alarm)을 발생시킨다는 문제를 가지고 있어, 실효성 있는 적용에 한계가 큰 상황이다(2.1절 참고).

본 연구에서는 특정 프로젝트에 연속되는 버전별로 유닛 테스트를 자동 생성하고 테스팅을 수행한 결과를 누적한 후, 버전 간에 발생한 오류 결과를 비교하여 중복해서 발견되는 결함을 오경보로 추정하여 제거함으로써, 특정 버전에서 새롭게 발견된 오류만을 사용자에게 보고하는 효과적인 지속적 자동 테스팅 기법인 CTG-Randoop (Continuous Unit Test Generation with Randoop) 프레임워크를 제안한다.

본 논문에서 제안한 CTG-Randoop을 Java를 대상으로 한 테스팅 도구로 구현하여 Defects4J[3] 벤치마크에 포함된 6개 실제 결함을 기반으로 한 테스팅 사례에 적용한 결과, 제안한 기법은 단일 버전에서 Randoop을 적용하였을 때 발견되는 오경보를 최대 66%를 제거하면서도 검출 가능한 모든 결함을 검출함을 확인할 수 있었다.

본 연구는 과기정통부의 소프트웨어중심대학 지원사업(2017-0-00130)과 신진연구지원사업(NRF-2020R1C1C1013512)의 지원을 받았음.

2. 연속적인 유닛 테스트 생성 프레임워크

2.1. 배경: 유닛 테스트 생성 기법

유닛 테스트 생성 기법은 테스트 대상 프로그램의 구조적 정보를 활용하여 테스트 대상 프로그램 유닛(예: Java 의 클래스)을 다양한 조건에서 실행하는 테스트 코드를 자동으로 생성하는 기법이다. Java 프로그램을 대상으로 한 기법[1][2]의 경우, 테스트 대상으로 지정된 클래스의 객체를 생성한 후, 다양한 입력 값을 사용하여 해당 객체의 여러 메소드를 순차적으로 실행하는 코드를 임의로 생성한 후, 정적/동적 정보를 바탕으로 유닛 테스트 용도로 적합한 것으로 추정되는 코드를 생성된 유닛 테스트 코드로 사용자에게 제공한다.

높은 테스트 커버리지 달성 성능에 불구하고, 현재까지 개발된 유닛 테스트 생성 기법을 연속적 테스팅 상황에 사용하는 데에 여러 한계점이 존재한다.

첫째 한계점은 효과적인 자동 테스트 생성이 수행될 수 있도록, 테스트 대상 프로그램의 특성에 맞는 테스트 환경 설정을 사용자가 직접 제공하여야 한다는 점이다. 예를 들어, 테스트 대상 클래스의 메소드 호출 시 입력되는 객체 혹은 값을 난수적으로 생성할 경우, 테스트 수행에 의미 있는 경로를 탐색할 확률이 매우 낮다. 효과적인 유닛 테스트 생성을 위해서, 사용자가 테스트 대상 클래스의 특성에 따라 입력 객체의 종류, 값의 범위나 특징적 상수(seed)를 테스트 환경 설정으로 지정하는 작업이 필요하다[4].

유닛 테스트 생성 기법의 또다른 한계점은 다량의 오경보 발생이다. 자동으로 생성된 유닛 테스트의 경우, 테스트 대상 프로그램에서 의도한 메소드 호출 순서, 메소드 입력 값의 제한 조건 등의 프로그래밍 규칙을 올바르게 따르지 않아 유효하지 않은 동작에서 예외(Exception) 발생을 일으키는 우가 많다. 또한, 테스트 대상 프로그램에서 정상동작으로서 발생시킨 예외를 오류와 구별하지 못하여 오경보가 발생할 수 있다. 아래는 `Apache Commons CSV` 에 대한 유닛 테스트로 서 오경보에 해당하는 사례이다.

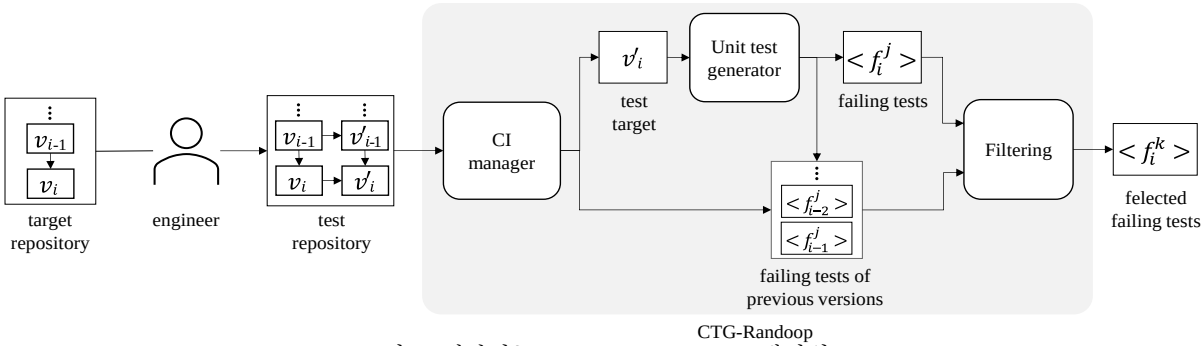

그림1. 제안하는 CTG-Randoop 프레임워크

```
01   public void test1 () {
02     Reader in = new FileReader("path/to/file.csv");
03     CSVParser parser = new CSVParser(in) ;
04     parser.close() ;
05     parser.getRecords() ; }
06   public void test2 () {
07     CSVFormat f = null ;
08     CSVParser parser = CSVParser.parse("…", f) ; }
```

위 코드의 `test1`를 실행할 경우 `IOException`이 발생하는데, 이는 `CSVParser`의 Javadoc에 명세되어있는 API 호출 규칙을 위반하여 `close` 호출한 후 `getRecords`를 호출한데 따른 정상적인 결과이다. `test2`의 경우 `CSVPaser.parse` 메소드의 두 번째 인자에 `null`이 입력되어 `IllegalArgumentStatement`가 발생하는 데, 이 역시 `CSVPaser`의 Javadoc에 잘못된 사용으로 명시된 경우로 오경보에 해당한다.

이와 같이, 현재 제안된 유닛 테스트 생성 기법은 테스트 대상 프로그램에 대해 유효한 테스트와 유효하지 않은 테스트로 인한 오경보를 구별할 수 있는 방법이 없다. 이로 인해 유닛 테스트 생성 기법이 도출한 오류 정보를 사용자가 수작업으로 판별하는 등의 추가적인 비용이 소요되는 한계점을 갖는다. 또한, 유닛 테스트 생성 기법이 도출하는 유닛 테스트 코드는 특정 버전에 종속되기 때문에, 프로그램 수정으로 프로그램 구조나 식별자가 변경된 버전에 대하여는 직접적인 사용이 불가능하다는 한계가 있다.

2.2. 프레임워크

그림 1 은 본 논문이 제안하는 연속적 유닛 테스트 생성 기법과 이를 활용한 테스팅 과정을 나타낸다. 유닛 테스트 생성 기법의 효과적인 적용을 위해서는 테스팅 대상 프로그램의 특징에 맞게 테스팅 환경을 필수적으로 설정해주어야 한다. 자동 유닛 테스팅의 효과적인 연동을 위하여, 본 논문에서는 테스트 대상 프로젝트 저장소에 개발 버전(v_i)과 병행하여 이에 대한 자동 테스팅 환경 설정이 저장되는 테스트 버전(v'_i)을 사용자(engineer)가 갱신하는 방식을 제안한다. 새로운 개발 버전이 등록될 경우, 사용자(engineer)는 직전 테스트 버전(v'_{i-1})과 새로 등록된 개발 버전(v_i)을 통합(merge)함으로써 새로운 개발 버전에 대한 테스트 버전을 작성할 수 있다. 특정 프로그램에 대한 테스팅 환경은 여러 버전에 걸쳐 재사용이 가능하므로, 직전 테스트 버전과 새로운 개발 버전의 통합 과정은 점층적으로 진행할 수 있으며, Git 과 같은 다중 버전을 지원하는 코드 형상 관리체계를 통해 효율적으로 이루어질 수 있다.

새로 등록된 테스트 버전에 대하여 제안한 프레임워크는 자동으로 테스트 생성과 실행을 수행한다. 코드 저장소로에 새로운 테스트 버전이 등록되면, 지속적 통합 시스템(CI manager)이 이를 내려 받아 유닛 테스트 생성 모듈(Unit test generator)에 전달하며, 이와 동시에 테스트 결과 데이터베이스에 새로운 버전에 대한 정보를 저장한다. 유닛 테스트 생성 모듈은 테스트 대상 프로그램 코드와 이에 대한 환경설정을 입력 받은 후, 테스트 대상 프로그램에 대한 유닛 테스트를 자동으로 생성한다. 제안한 CTG-Randoop 프레임워크가 유닛 테스트 생성 모듈로 사용하는 Randoop [1]은 Java 프로그램의 클래스별 소스코드를 입력으로 받아, 해당 클래스에 대한 유닛 테스트 집합을 생성한 후 테스트 대상에 실행하여 오류가 검출된 결과(오류 정보)를 출력한다. 이 때, 테스트 실패(failure)가 발생한 유닛 테스트 집합(failing)은 오류 정보 판별(Filtering) 모듈로 전달된다.

오경보 판별(Filtering) 모듈은, 자동으로 생성된 유닛 테스트의 오경보로 인한 문제를 완화하기 위한 방법으로, 유닛 테스트 생성 모듈이 전달한 다수의 오류 정보들 중 새로운 개발 버전(v_i)의 결함 검출에 해당하는 오류 정보를 구별하는 것을 목표로 한다. 제안하는 기법은 테스트 대상 프로젝트의 직전 버전에 대한 유닛 테스트 생성 및 실행 결과를 저장하여 활용함으로써, 직전 버전에서도 대응되는 오류는 오경보로 간주하여 걸러내어 직전 버전에는 관찰되지 않은 오류만을 사용자에게 제공하는 방식으로 오경보를 추정한다. 제안하는 프레임워크는 서로 다른 버전 간에 오류를 대응하는 총 10 개의 휴리스틱을 제공하며, 사용자는 프로젝트의 특성을 고려하여 한 가지 휴리스틱을 적용할 수 있다. 오경보로 추정되는 오류 정보가 제거된 오류 정보는 사용자에게 테스팅 결과로 제공된다.

2.3. 오경보 판별 휴리스틱

코드 변경으로 프로그램 구조와 기능이 변경된 두 버전 사이에 동일한 이유로 발생한 오류들을 대응하는 작업은 정확도의 한계를 갖는다. 제안한 기법에서는 Java 프로그램에서 발생하는 오류와 관련된 동적 정보와 정적 정보를 활용하는 10 가지 서로 다른 휴리스틱을 제공하여 사용자가 테스트 대상 프로그램의 특성을 고려하여 이를 선택적으로 사용할 수 있도록 지원한다. 본 기법이 제공하는 10 가지 휴리스틱의 판별 기준은 다음과 같다:

- **H1.** 오류에서 발생한 Exception 의 타입과 오류 메시지 및 스택 트레이스(stack trace) 전체의 메소드명과 연관 소스코드 줄 번호(line number)가 동일한 경우 대응
- **H2.** 오류에서 발생한 Exception 의 타입과 오류 메시지가 동일하고 스택 트레이스의 최상위 단 메소드명과 연관 소스코드 줄 번호가 동일한 경우 대응
- **H3.** 오류에서 발생한 Exception 의 타입과 오류 메시지 및 스택 트레이스 전체의 메소드명이 동일한 경우 대응
- **H4.** 오류에서 발생한 Exception 의 타입과 오류 메시지 및 스택 트레이스 최상위 단 메소드명이 동일한 경우 대응

표1. 오경보 판별 휴리스틱에 따른 오류 검출 실험 결과

	No filter		H1		H2		H3		H4		H5		H6		H7		H8		H9		H10	
	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T
Time-8	34168 (100%)	2 (100%)	33642 (98%)	2 (100%)	17006 (49%)	2 (100%)	33641 (98%)	2 (100%)	15219 (44%)	2 (100%)	13898 (40%)	2 (100%)	33694 (98%)	2 (100%)	23178 (67%)	2 (100%)	33694 (98%)	2 (100%)	20529 (60%)	2 (100%)	1 (0%)	0 (0%)
Time-14	32715 (100%)	53 (100%)	32159 (98%)	53 (100%)	18370 (56%)	53 (100%)	32159 (98%)	53 (100%)	16872 (51%)	53 (100%)	15477 (47%)	53 (100%)	32159 (98%)	53 (100%)	25102 (76%)	53 (100%)	32159 (98%)	53 (100%)	23795 (72%)	53 (100%)	0 (0%)	0 (0%)
Gson-12	66910 (100%)	492 (100%)	66799 (99%)	492 (100%)	24168 (36%)	492 (100%)	66798 (99%)	492 (100%)	23438 (35%)	492 (100%)	3154 (4%)	0 (0%)	66802 (99%)	492 (100%)	56954 (85%)	492 (100%)	66802 (99%)	492 (100%)	56256 (84%)	492 (100%)	4 (0%)	0 (0%)
JSoup-8	14238 (100%)	1656 (100%)	14229 (99%)	1656 (100%)	4980 (34%)	1656 (100%)	14229 (99%)	1656 (100%)	4955 (34%)	1656 (100%)	3024 (21%)	0 (0%)	14234 (99%)	1656 (100%)	7941 (55%)	1656 (100%)	14234 (99%)	1656 (100%)	7366 (51%)	1656 (100%)	0 (0%)	0 (0%)
Codec-5	103638 (100%)	5 (100%)	100710 (97%)	5 (100%)	45459 (43%)	5 (100%)	100681 (97%)	5 (100%)	40613 (39%)	5 (100%)	35418 (34%)	0 (0%)	101848 (98%)	5 (100%)	71754 (69%)	5 (100%)	101848 (98%)	5 (100%)	70134 (67%)	5 (100%)	0 (0%)	0 (0%)
Codec-13	49586 (100%)	75 (100%)	49273 (99%)	75 (100%)	30220 (60%)	75 (100%)	49273 (99%)	75 (100%)	28121 (56%)	75 (100%)	26519 (53%)	17 (22%)	49273 (99%)	75 (100%)	33722 (68%)	75 (100%)	49273 (99%)	75 (100%)	33020 (66%)	75 (100%)	1 (0%)	17 (22%)

H5. 오류에서 발생한 Exception 의 타입과 오류 메시지가 동일한 경우 대응

H6. 오류에서 발생한 Exception 의 타입이 같고 스택 트레이스 전체의 메소드명과 연관 소스코드 줄 번호가 동일한 경우 대응

H7. 오류에서 발생한 Exception 의 타입이 같고 스택 트레이스의 최상위 단 메소드명과 연관 소스코드 줄 번호가 동일한 경우 대응

H8. 오류에서 발생한 Exception 의 타입이 같고 스택 트레이스 전체의 메소드명이 동일한 경우 대응

H9. 오류에서 발생한 Exception 의 타입이 같고 스택 트레이스 최상위 메소드명이 동일한 경우 대응

H10. 오류에서 발생한 Exception 의 타입이 같은 경우 대응

제안한 10 가지 휴리스틱 중 H1 이 오경보 대응 범위가 가장 좁고 H10 이 오경보 대응 범위가 가장 넓으므로, H1 의 사용할 때 사용자에게 제공되는 오류가 가장 많으며, H10 은 가장 적도록 설계하였다.

3. 실험 평가

본 연구에서 제안한 CTG-Randoop 의 효용성을 평가하기 위해 Defects4J 결함 벤치마크를 활용한 실험을 수행하으며, 구체적으로 두 가지 질문에 대한 답을 얻기 위하여 실험을 설계하였다:

질문 1. CTG-Randoop 은 프로그램 내에 존재하는 결함을 얼마나 효과적으로 탐지하여 사용자에게 보고하는 가?

질문 2. CTG-Randoop 은 휴리스틱 사용에 따라 오경보를 얼마나 효과적으로 제거하는 가?

오류 검출과 오경보 제거 능력을 평가하기 위하여 본 실험에서는 Defects4J 의 Jsoup, Codec, Joda-Time, Gson 대상 결함 사례 중, 결함 버전에 대하여 Randoop 을 단독으로 실행할 경우, 해당 결함으로 인한 오류가 검출되는 6 개 결함 사례를 수합하여 실험 대상을 구성하였다. 이 때, 각 결함 사례에서 결함이 수정된 올바른 버전을 직전 버전(v_{i-1})으로, 결함이 발견된 버전을 새로운 버전(v_i)으로 사용하였다. 표 1 의 첫 번째 행에 나타난 여섯 개의 테스트 대상 프로그램을 나타낸다.

각 테스트 대상 프로그램에 대하여 CTG-Randoop 은 총 30 분간 테스트 생성을 수행하였다. Randoop 실행에 있어 난수적 요소의 영향을 고려하여, 본 실험에서는 총 5 회의 실험을 반복한 데이터를 평균한 값을 실험 결과로 사용하였다.

CTG-Randoop 은 CI 시스템으로 Jenkins 를 사용하였으며, 형상 관리는 Github 과 Git 을 이용하는 상황을 상정하여 구성하였다. 유닛테스트 생성 기법으로는 본 실험을 위해 개선된 Randoop 을 사용하였다. 본 실험에서는 유닛 테스트 생성 도구로 Randoop 이 특정 프로젝트에 대해 보다 효과적으로 유닛 테스트를 생성할 수 있도록 하기 위하여, 테스트 입력 생성 시 유용한 초기 값(seed)을 테스트 환경으로부터 입력 받아 사용할 수 있는 기능을 추가로 구현한 후 CTG-Randoop 에 적용하였다. 또한, CTG-Randoop 결과 도출된 오류 정보는 Gerrit 을 통하여 사용자에게 리뷰 리프토 형태로 전달되도록 인터페이스를 구성하였다.

표 1 은 각 실험 대상에 대하여 CTG-Randoop 을 서로 다른 휴리스틱을 사용하여 적용할 때 발생한 오경보(F 행)와 실제 결함 탐지 오류(T 행)의 개수를 나타낸다. 'No filter'로 표시된 결과는 휴리스틱을 사용하지 않고 Randoop 의 결과를 그대로 보고하였을 때의 결과로, 본 실험의 비교대상이 된다. 각 셀의 괄호 안의 숫자는 'No filter' 경우에 대비한 오경보와 실제 결함 탐지 경보의 비율을 나타낸다.

표 1 의 실험 결과를 볼 때 CTG-Randoop 은 실제 결함 탐지 결과를 그대로 유지하면서도 오경보를 최대 66% (Jsoup 에 대한 H4 결과)까지 자동으로 제거함을 확인할 수 있다. 총 10 개의 오경보 판별 휴리스틱의 성능을 비교해 볼 때, H4(오류에서 발생한 Exception 의 타입과 오류 메시지 및 스택 트레이스 최상위 단 메소드명이 동일한 경우 대응)를 사용할 경우, 가장 높은 정확도를 보임을 확인할 수 있다.

4. 결론 및 향후연구

본 논문에서는 유닛 테스트 생성 기법을 연속적으로 개발되는 프로젝트의 지속적인 테스팅에 효과적으로 적용하기 위한 방법으로 프레임워크를 제시하고, Java 대상 유닛 테스트 생성 도구인 Randoop 을 활용하여 이를 구현한 CTG-Randoop 프레임워크를 소개하였다. 본 연구 결과를 바탕으로 향후에는 코드 변경에 따른 소스코드 대응 관계 정보를 활용하여 보다 정확하고 효과적으로 오경보를 판별하는 방법과 EvoSuite [2] 등 보다 다양한 자동 유닛 테스트 생성기법을 활용하도록 CTG-Randoop 를 확장하는 연구를 수행할 계획이다.

참조문헌

[1] C. Pacheco, S. K. Lahiri, M. D. Ernst, T. Ball, "Feedback-directed Random Test Generation", ICSE, 2007

[2] G. Fraser and A. Arcuri, "1600 Faults in 100 Projects: Automatically Finding Faults While Achieving High Coverage with EvoSuite," Empirical Software Engineering, 2013

[3] R. Just et al., "Defects4J: a database of existing faults to enable controlled testing studies for Java programs", ISSTA 2014

[4] G. Fraser and A. Arcuri, "The Seed is Strong: Seeding Strategies in Search-Based Software Testing", ICST 2012