유닛 테스트 케이스 반복 실행을 통한 자바 프로그램에서의 메모리 블롯 오류의 효과적 검출: Apache Commons VFS 대상 적용 사례

이재훈 홍신
한동대학교 전산전자공학부
{21400575, hongshin}@handong.edu

Detecting Memory Bloats of Java Programs by Monitoring Repeated Unit Test Executions: A Case Study with Apache Commons VFS

요 약

본 논문은 Java 프로그램에서 발생할 수 있는 메모리 누실 오류의 한 종류인 메모리 블롯을 효과적으로 탐지하는 새로운 동적 분석 기법을 제안하고, 이를 Apache Commons VFS에 적용하여 실제 결함을 검출한 사례 연구를 소개한다. 제안한 동적 분석 기법은 기존에 개발된 유닛 테스트 케이스를 활용하여, 같은 동작을 반복 수행 하는 가운데 메모리 증가가 발생하는 지 자동으로 검사함으로써 Java 프로그램에서의 메모리 블롯을 효과적으로 탐지한다. 제안한 기법을 Apache Commons VFS에 적용한 결과, 실제적인 메모리 블롯을 자동으로 검출할 수 있었으며, 이를 분석하여 3개의 신규 메모리 블롯 결함을 발견하였다.

1. 서론

메모리 블롯(memory bloat)이란 프로그램이 특정 시점 이후에 기능(functionality) 제공에 불필요한 메모리 자원을 반환하지 않음으로써 실행 중 메모리 사용량이 불필요하게 증가하는 문제를 일으키는 일종의 *메모리 누실*(memory leak) 오류를 뜻한다[1, 2]. 메모리 블롯 발생으로 인한 메모리 누실은 일차적으로 메모리 자원 관리 비용 증대로 인한 성능저하를 일으킬 수 있으며, 장기간 누적될 경우, 메모리 부족 현상을 발생시켜 서비스 실패 등 심각한 소프트웨어 안정성 문제를 야기할 수 있다. 할당된 메모리 주소의 상실로 발생하는 메모리 상실(memory loss) 결함과 달리, 메모리 블롯 결함은 Java와 같이 불용 메모리 수거(garbage collection)가 제공되는 언어/실행환경에서도 발생할 수 있으며, 일반적인 테스팅에서 오류 증상을 명확히 식별하기 어려워 디버깅이 난해한 결함으로 알려져, 효과적인 메모리 블롯 검출 기법의 개발이 요구되는 상황이다.

기존 연구에서는 시스템 테스트(system testing) 중 메모리 사용을 프로파일링(profiling)함으로써, 메모리 블롯 발생을 통계적으로 추정하는 방식이 제안되어 왔다[2, 3, 4]. 이러한 프로파일링 기반 기법은 개발자가 메모리 블롯이 두드러지게 발생하는 시스템 테스트를 제공해 주지 않을 경우, 효과적인 오류의 탐지가 불가능하므로 오류 검출 효용성과 민감성에 한계를 가지고 있다. 뿐만 아니라, 프로파일링 기반 방식은 실행 과정에서 어떠한 메모리가 기능에 불필요한지 판별하기 어려우므로, 정확한 오류 분석에 한계가 있다는 점에서 또 다른 한계점을 가진다.

본 논문은 Java 프로그램 실행에서 발생한 메모리 블롯을 효과적으로 탐지하기 위해 유닛 테스트 케이스를 활용하여 메모리 블롯 검출에 유용한 프로그램 실행을 자동으로 생성하고, 실행 중 체계적으로 메모리 사용을 추적한 기록을 분석함으로써 메모리 블롯을 민감하게 탐지, 분석하는 새로운 동적 분석 기법을 제안한다. 제안하는 동적 분석 기법은 유닛 테스트 케이스에 따른 실행에서 메모리 블롯을 효과적으로 관찰하기 위해, 각 유닛 테스트 케이스를 의도적으로 수차례 반복 재생하고, 각 재생마다 메모리 사용량과 메모리 상태를 비교하여, 동일한 기능을 수행함에도 메모리 소비가 지속적으로 증가하는 현상을 메모리 블롯 오류로 검출한다. 본 연구에서 제안한 기법을 실제적인 Java 프로그램에 대해 동작하는 도구로 구현하여, Apache Commons VFS [5] 에 적용한 사례 연구 결과, 이전에 발견되지 않은 3개의 실제적인 메모리 블롯 결함을 검출할 수 있었다.

본 논문이 제안하는 기법은 사용자가 설계한 시스템 테스팅 대신, 유닛 테스트 케이스로부터 메모리 블롯 검출에 효과적인 실행을 능동적으로 생성하여 활용한다는 점에서, 수동적 프로파일링에 의존하는 기존 연구와 차별성을 갖는다.

본 논문의 구성은 다음과 같다. 2장에서는 본 연구의 주제와 접근 방법에 대한 이해를 돕기 위해 유닛 테스트 케이스의 기능과 구조, 메모리 블롯 결함의 예제를 소개한 후 Apache Commons VFS 프로젝트에서 발견된 실제 결함 사례를 소개한다. 3장에서는 본 논문이 제안하는 새로운 메모리 블롯 검출 기법을 소개하고, 이어서 4장에서는 Apache Commons VFS를 대상으로 제안한 기법을 적용하여 메모리 블롯을 실제적으로 검출하고 3개의 신규 결함을 발견한 사례 연구를 소개한다. 5장에서는 향후 연구 계획과 함께 논문을 마무리한다.

2. 연구배경(Background)

2.1 유닛 테스트 케이스의 기능

*유닛 테스팅*은 검증대상 프로그램의 특정 모듈 혹은 객체를 테스트 프로세스 내에서 생성한 후 일련의 명령/메소드를 호출하고 그 반환 값을 검사하는 과정들, 즉 *유닛 테스트 케이스*들의 집합으로 구성된다. 검증대상 프로그램의 완성된 코드를 실행시킨 후 실제 사용자가 발생시킬 수 있는 데이터를 입력하여 여러 기능 간의 상호작용을 검사하는 시스템 테스트 케이스와 달리, 각 유닛 테스트 케이스는 검증대상 프로그램의 구체적인 기능을 특정하여 수행하고 검사하는 것을 목표로 설계되는 것이 일반적이다. 유닛 테스트 케이스는 개발 과정에서 지속적인 회귀 테스팅(regression testing)을 가능하게 하고 각 모듈 API의 용례로 사용될 수 있기 때문에, 오늘날 소프트웨어 개발 프로젝트에서 품질관리와 협업의 중요한 매체로 인식되고 있으며, 대부분의 실제 프로젝트에서 소프트웨어 주요 요소로 체계적으로 개발/관리되고 있다.

본 연구는 한국연구재단 신진연구자지원과제(NRF-2017R1C1B1008159) 지원으로 수행됨.

```
01 class Stack {              08 public Object pop() {
02  Object [] arr;           09  if (!isEmpty()) {
03  int size;                10   Object e=arr[--size];
04  Stack() {                11 /*FIX: arr[size]=null;*/
05   arr=new Object[MAX];    12   return e;
06  }                        13  }
07  ...                      14 }
```

그림 1. 메모리 블롯 결함 예제

본 연구에서는 이와 같은 유닛 테스트 케이스에 주목하여, 검증대상 프로그램에 대해 개발자가 올바르게 작성한 유닛 테스트 케이스가 충분히 제공되는 상황을 가정하였다. 또한 각 유닛 테스트는 특정 기능을 올바르게 실행하는 코드로 작성되었으므로, 이를 반복 실행할 경우, 항상 동일한 기능/동작이 수행되게 되며, 그 과정에서 불필요한 메모리의 증대는 없어야 한다고 추정하였다.

2.2. 유닛 테스트 스위트의 구조

일반적으로, 특정 모듈에 관련된 유닛 테스트 케이스들은 하나의 *유닛 테스트 스위트*(unit test suite) 형태로 작성된다. 유닛 테스트 스위트는 해당 모듈의 유닛 테스트 케이스들과 유닛 테스트 케이스의 구동에 필요한 일체 *테스트 구동 코드*로 구성된다. 테스트 구동 코드는 각 유닛 테스트 케이스의 실행 전후에 호출되어 테스트 케이스 실행에 필요한 객체를 생성/소멸하거나, 자원을 할당/해제함으로써 유효한 실행 환경을 구성하는 역할을 한다.

유닛 테스팅 프레임워크는 개발자가 유닛 테스트 스위트를 체계적으로 작성할 수 있는 구조를 제공하고, 구조에 따라 작성된 테스트 구동 코드와 테스트 케이스를 일련의 순서로 실행시켜, 테스트 케이스가 올바르게 구동되도록 지원한다. Java에서 표준적으로 사용되는 JUnit 프레임워크의 경우, 하나의 유닛 테스트 스위트를 한 개의 TestClass 클래스로, 각 테스트 케이스를 한 개의 메소드로 작성하도록 지원한다. 또한, @BeforeClass, @AfterClass, @Before, @After와 같은 메소드 속성 선언을 통해서 테스트 스위트 전체 초기화, 전체 마무리, 각 테스트 케이스 실행 직전 초기화, 실행 직후 마무리에 필요한 테스트 구동 코드를 개발자가 작성하도록 지원한다.

2.3 메모리 블롯 결함

메모리 블롯은 실행 중 특정 시점 이후로 다시 사용하지 않을 객체를 제거하지 않는 결함으로 인하여, 소멸(reclaim)되지 않는 불필요한 객체가 지속적으로 발생하게 되는 오류적 상황을 뜻한다. Java 프로그램에서는 불필요한 객체에 대한 참조(hard reference)를 모두 소멸함으로써, 해당 객체를 불용 객체(eligible object)로 선언하여 불용 메모리 수거 루틴에 의해 자동으로 소멸되도록 할 수 있다. Java 프로그램에서 메모리 블롯은 결함으로 인해 불필요해진 객체를 향한 모든 참조 제거를 놓치거나 실패할 경우 발생하며, 주로 배열이나 리스트(list), 맵(map)과 같은 *컨테이너*(container) 관리를 잘못 처리 하여 발생하는 경우가 빈번하다.

그림1은 Java로 스택을 구현한 코드에서 발생한 메모리 블롯의 예이다. 본 프로그램은 배열을 통해 스택 데이터 구조를 구현하고 있다. `Stack`은 데이터 객체를 입력 받아 `arr` 배열(2행)에 저장한다. `Stack`은 `pop()` 명령이 실행될 경우, `arr` 배열에 가장 최근에 저장된 데이터를 반환하고, `size`를 줄여 더 이상 스택에서 사용할 수 없게 한다. 이때, `arr[size]`에 불필요한 객체에 대한 참조가 남아있으므로, 해당 객체가 메모리에서 사라지지 않고 남아있게 된다. 이는 결과적으로 필요이상의 메모리 사용을 발생시키는 이상 증상, 즉 메모리 블롯을 야기한다. 이 결함은, 코드의 11행의 주석에 표시한 바와 같이, 불필요한 참조를 제거하는 코드를 추가함으로써 수정(bug fix)할 수 있다.

표1. Apache Commons VFS에서 보고된 11개 메모리 블롯 사례

Bug ID	Severity	오류 발생 상황	결함 원인
VFS-134	Major	컨테이너에 포함 (가)	잘못된 컨테이너에서 제거 (C)
VFS-140	Minor	컨테이너에 포함 (가)	소유자 객체의 정리 실패 (A)
VFS-142	Major	컨테이너에 포함 (가)	쓰레드 종료 전 미제거(B)
VFS-143	Major	컨테이너에 포함 (가)	약한 참조 미사용(D)
VFS-227	Major	컨테이너에 포함 (가)	쓰레드 종료 전 미제거(B)
VFS-287	Major	다른 객체가 참조 (나)	소유자 객체의 정리 실패 (A)
VFS-309	Blocker	컨테이너에 포함 (가)	쓰레드 종료 전 미제거(B)
VFS-480	Critical	컨테이너에 포함 (가)	소유자 객체의 정리 실패 (A)
VFS-544	Minor	컨테이너에 포함 (가)	잘못된 컨테이너에서 제거 (C)
VFS-545	Major	컨테이너에 포함 (가)	소유자 객체의 정리 실패 (A)
VFS-633	Major	컨테이너에 포함 (가)	소유자 객체의 정리 실패 (A)

2.4 Apache Commons VFS에 발생한 실제 메모리 블롯 사례

본 연구에서는 실제 소프트웨어 발생한 메모리 블롯 실태를 이해하기 위해, 오픈소스 프로젝트인 Apache Commons VFS 저장소를 탐색하여, 이전에 실제 발생한 메모리 블롯 관련 사례를 수집하고 정리하였다. Apache Commons VFS(이후 VFS)는 Java 어플리케이션에서 여러 파일 시스템에 접근할 수 있는 API를 제공하는 라이브러리 프로그램으로, 복잡한 파일 관련 객체를 지속적으로 생성/해체하는 코드가 복잡한 편이다.

VFS 프로젝트의 이슈 관리 시스템(issue tracker)에서 해결된 결함(bug)으로서 'memory leak' 혹은 'memory bloat' 이라는 키워드와 연관된 이슈를 모두 검색하였다. 검색 결과, 2005년 1월부터 2017년 12월까지(총 13년간) 보고된 총 406개의 해결된 결함 중 관련 키워드를 포함하는 이슈는 총 15개였으며, 각 사례를 정성적으로 분석한 결과. 총 11개의 메모리 블롯 실제 사례를 파악할 수 있었다.

표1은 11개 사례에 대해 관리자(maintainer)가 부여한 심각도(severity)와, 정성적 분석을 통해 파악한 메모리 블롯 발생 상황과 결함 원인을 분류한 결과다. 표에서 알 수 있듯이 11개 중 9개 결함이 Critical, Blocker, Major와 같이 주요 결함으로 개발자들에게 인식되었음을 확인할 수 있다.

정성적으로 분석 결과, 11개 메모리 블롯 결함은 '오류 발생 상황'에 따라 다음 두 가지로 종류로 분류 할 수 있다.

가. 맵(Map), 리스트(List)와 같은 컨테이너에서 불필요한 객체 참조 (총 10건)

나. 다른 객체의 멤버가 불필요한 객체를 참조 (총 1건)

11개 사례에 대해 오류를 일으킨 결함의 원인을 정성적으로 조사한 결과, 다음 4개의 유형이 있음을 파악할 수 있었다.

A. 소유자 객체(owner)가, 재사용의 목적으로 소유하고 있던 데이터 객체를 정리(clear) 과정에서 실수로 특정 데이터 객체에 대한 참조를 지우지 않아, 불필요한 데이터 객체가 메모리 블롯을 일으키는 경우. (총 5건)

B. 쓰레드(thread)가 기능 수행을 위해 공역 데이터에 등록하였으나, 종료 시점에 불필요해 진 객체를 제거하지 않고 쓰레드를 종료하게 됨. (총 3건)

C. 불필요하게 된 객체를 컨테이너에서 제거하기 위한 코드가 있으나, 실수로 해당 객체와 상관없는 컨테이너에 대해 제거 명령을 수행 한 경우. 결과적으로 불필요한 객체가 제거되지 않음. (총 2건)

D. 약한 참조(weak reference)를 사용 해야 하는 상황에서 실수로 해당 객체에 대한 일반 참조(strong reference)를 사용해 예기치 않게 해당 객체에 대한 소멸이 발생하지 않게 된 경우. (총 1건)

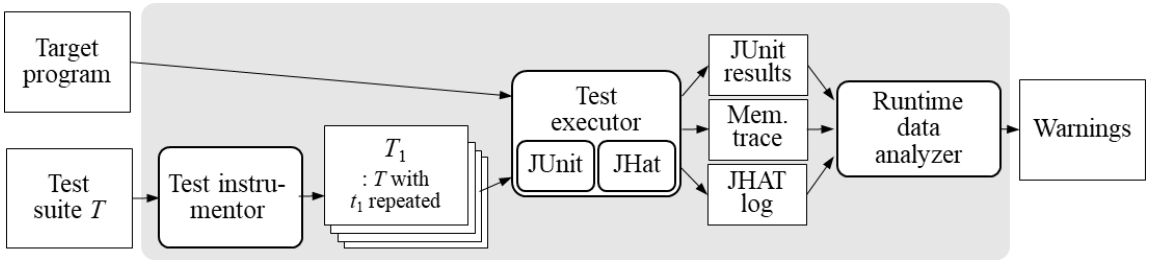

그림 2. 동적 분석 기법의 구성과 실행 흐름

3. 메모리 블롯 검출 기법

3.1 동적 분석 과정

메모리 블롯 결함을 효과적으로 탐지하기 위하여, 본 논문에서는 유닛 테스트 케이스를 반복 실행 시켜 동일한 동작을 지속적으로 발생시키고, 이 때 프로그램이 사용하는 메모리 사용량의 추세를 분석하는 동적 분석(dynamic analysis) 기법을 제안한다. 일반적으로 개발자가 작성한 유닛 테스트 케이스는 분석대상 프로그램의 특정 모듈을 시동한 후 특정 기능을 수행시키고, 해당 모듈을 종료하는 일련의 과정으로 구성된다. 따라서, 올바르게 작성된 유닛 테스트 케이스의 경우, 여러 차례 반복 시행을 할 경우 동일한 기능을 시연하게 된다. 본 논문에서 제안하는 동적 분석 기법은 유닛 테스트 케이스의 반복 실행에서 동일한 동작이 발생함에도, 반복에 따라 메모리 사용량이 지속적으로 증가하는 유닛 테스트 케이스를 메모리 블롯 의심 증상으로 추정하고, 해당 메모리 사용량 증가에 기여한 코드 요소를 결함 의심 요소로 판별하여 경보 정보를 사용자에게 제시한다.

그림 2는 제안하는 동적 분석 기법의 구성 요소와 동작 흐름을 보여준다. 본 기법은 사용자로부터 분석대상 프로그램(target program) 코드와 JUnit 상에서 작성된 유닛 테스트 스위트(test suite) 소스코드를 입력으로 받아 분석을 수행하며, 그 결과는 메모리 블롯을 발생시키는 것으로 의심이 되는 객체 정보(warnings)다. 테스트 스위트 T 에는 테스트 케이스 집합 $testcases(T)=\{t_1, t_2, \ldots, t_n\}$과 테스트 케이스의 실행에 필요한 일체의 부속코드(test fixture)로 구성된다. 이 때 모든 테스트 케이스 결과는 성공(pass)으로 정상적으로 동작 해야 하며, 실패(fail) 테스트 케이스는 분석에서 제외하고 사용해야 한다.

테스트 편집기 모듈(test instrumentor)은 입력 받은 테스트 스위트에서 각 테스트 케이스 t_i별로 (1) t_i가 k회 반복 실행되도록 테스트 케이스 구동코드를 추가하며, (2) 테스트 케이스 반복에 메모리 사용량 기록과 JHAT 을 통해 수집한 메모리 상태 정보를 출력하도록 탐침(probe)를 삽입한 n개의 수정된 테스트 스위트 집합 $\{T_1, T_2, \ldots, T_n\}$을 생성한다. T_i 는 T와 동일한 순서로 테스트 코드를 실행하되, i 번째 테스트 케이스 t_i와 관련된 코드는 k번 연속 반복 수행하게 된다.

테스트 실행 모듈(test executor)은 JUnit을 각각의 수정된 테스트 스위트 T_i 를 입력 받아 테스트 스위트를 실행하고, 그 결과 생성되는 JUnit 결과, 메모리 사용량 기록, JHAT (Java Heap Analysis Tool) [6]의 메모리 덤프 해석을 수집한다. JUnit 실행 결과는 각 테스트케이스 실행의 성공/실패 여부를 나타내며, 메모리 사용량 기록은 t_i가 최초 실행되기 직전, 매번 t_i실행이 마친 직후, 그리고 T_i 전체의 종료 직전에 테스트 실행 프로세스의 전체 메모리 공간과 해당 시점에서 가용(free memory) 메모리 양의 차를 구한 k+2 길이의 순열이다. 메모리 상태 기록(JHAT logs)은 테스트 스위트 종료 시점에 메모리 덤프를 통해 프로세스 공간에 존재하는 객체의 종류, 개수, 주소, 참조관계를 포착한 정보다.

런타임 데이터 분석 모듈(*runtime data analyzer*)은 테스트 실행에서 추출한 결과로부터 다음과 같은 분석을 수행한다:

- **분석1. 테스트 케이스 반복 실행의 유효성 판별**: 각 T_i 의 JUnit 결과를 분석하여 모든 테스트 케이스 결과가 성공인지 확인한다. 만약 실패한 테스트 케이스가 있을 경우, 테스트 케이스의 재실행으로 인해 동일한 동작이 반복되지 않고 의도하지 않은 다른 동작이 발생한 경우로 볼 수 있으므로, 동적 분석에서 제외한다.
- **분석2. 메모리 블롯 발생 유무 판별**: 분석1을 통과한 각 T_i 에 대하여, 메모리 사용량 측정 결과로 얻은 메모리 사용량 순열에 선형 회기(linear regression)를 사용해 추세선을 얻고, 추세선의 계수가 1000 byte/time이상이며, R^2 결정계수가 0.9 이상인 경우 메모리 블롯이 발생한 것으로 판별한다. 그 이외의 경우는 메모리 블롯이 발생하지 않은 것으로 이후 분석에서 제외한다.
- **분석3. 메모리 블롯 객체 판별**: 분석2에서 메모리 블롯이 발생한 것으로 판별된 각 T_i의 메모리 상태 기록을 분석하여 메모리 블롯에 기여한 것으로 의심되는 순서로 객체를 판별한다. JHAT으로 포착한 메모리 상태로부터 측정 시점의 프로세스 내에 유효하게 존재하는 객체, 각 객체가 생성(instantiation) 된 시점의 스택(stack trace), 그리고 각 객체의 멤버가 다른 객체를 참조하는 관계를 파악할 수 있다. 메모리 블롯에 기여한 객체를 파악하기 위해서, 메모리 상태 기록에 존재하는 모든 객체를 클래스가 같고 생성 시점의 스택(stack trace)이 같은 객체끼리 묶어서 객체군 집합을 정의하고, 각 객체군에 속한 객체의 수를 구하고, 객체군에 속한 객체 수가 많은 순서대로 정렬한다.

런타임 데이터 분석 모듈은 메모리 블롯 발생이 탐지된 각 테스트케이스에 의심도에 따라 정렬한 객체 정보를 분석의 최종 결과로 사용자에게 제공하여, 사용자가 분석 결과 결함의 기여가 큰 것으로 보이는 객체와 객체 관련 코드를 우선적으로 검토하도록 지원한다.

3.2 구현

제안한 기법에서 간단한 소스코드 편집과 JHAT 덤프 분석, 메모리 사용량 기록 분석은 Python 스크립트로 구현했고, 구조적인 소스코드 편집에는 JavaParser, 선형 회귀 분석에서는 SciPy 라이브러리를 활용해 계산하였다. 테스트 편집기가 삽입하는 탐침 코드에는 매 테스트 케이스 실행 직후 메모리 사용량을 정확히 측정하기 위해 `System.gc()`와 `System.sleep()`을 각각 호출하여 불용 메모리 수거를 유도 후, `Runtime.totalMemory()`, `Runtime.freeMemory()`를 호출하여 메모리 사용량을 측정하였다. 메모리 상태 기록을 얻기 위해서 테스트 실행 모듈에서는 JUnit을 구동할 때 JVM 옵션으로 종료시점에 메모리 덤프를 출력하도록 설정하였고, 테스트 실행 직후 JHAT을 호출해서 덤프를 해석하게 했다.

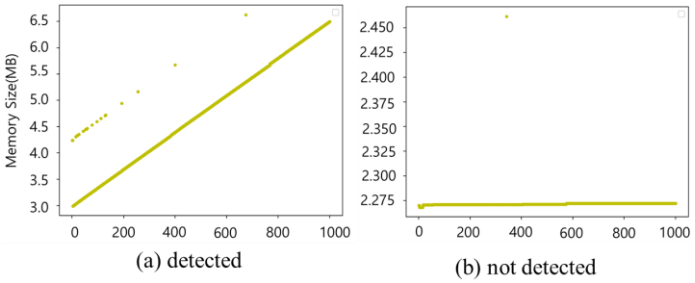

그림3. 메모리 블롯이 탐지된 실행(a)과 메모리 블롯이 탐지되지 않은 실행(b)에서 메모리 사용 추세 비교

4. Apache Commons VFS 대상 사례 연구

제안한 기법을 Apache Commons VFS에 적용하여 실제 결함을 검출하는 사례 연구를 수행하였다. 적용대상으로는 사례연구 수행 당시 최신 버전인 Apache Commons VFS 2.2-rc2가 사용됐다. 해당 프로젝트는 총 382개의 Java 파일로 구성 되어 있으며, Java 소스코드는 총 26376 LOC 규모이다. 프로젝트 내에는 JUnit으로 작성된 총 2341개 테스트 케이스(총 106개의 테스트 스위트)가 작성 돼 있는데, 이들은 정상적인 빌드 상황에서 모두 성공하는 테스트 케이스이다. 본 사례 연구에서는 2341개의 테스트 케이스를 각각 총 1000번 반복 실행하도록 설정하여 제안한 기법을 적용하였다. 실험은 Intel Core i5-6600 CPU와 64G DDR4 RAM이 설치된 Ubuntu OS환경에서 수행됐으며, 실행에는 OpenJDK 1.8를 사용하였다.

총 2341개의 테스트 케이스에 대해 동적 분석 기법을 적용한 결과, 총 3개의 테스트 케이스에 대해서 메모리 블롯 발생이 검출됐다. 나머지 2338 개의 테스트 케이스의 경우, 모두 반복 실행은 유효하게 실행 되었지만 메모리 블롯 발생이 탐지되지는 않았다. 그림3은 메모리 블롯이 검출된 `JunctionTests.testAncestor()` 실행의 경우와, 이와 반대로 실행 과정에서 메모리 블롯 발생이 검출 되지 않은 `UserAuthenticationDataTestCase.testCustomType()` 경우를 예시로 하여 반복 횟수에 따른 메모리 사용량 기록을 그래프로 표현한 결과다. 그림3-(a) 에서는 반복실행에 따라 메모리 사용량 증가가 뚜렷한 반면, 그림3-(b)에서는 사용량이 일정하여, 오류 발생 여부가 분명히 구별됨을 볼 수 있다.

표2는 메모리 블롯이 탐지된 3개의 테스트 케이스의 이름과, 1000번의 반복 실행 전 후 메모리 사용량 증가량, 선형 회귀 계수와 R^2 값을 나타낸다. 표2의 결과로 볼 때, 메모리 블롯이 탐지된 3개 경우에서는 테스트 케이스 반복에 따른 메모리 사용량의 선형적 증가가 분명하게 관찰됨을 확인할 수 있다.

탐지된 3개의 경보가 실제 결함에 의한 것인지 판별하기 위하여, 제시된 결함군 정보와 관련 소스코드를 정성적으로 분석하여 결함 유무를 분석하였다. 그 결과, `JunctionTets.testAncestor()` 케이스와 관련된 결함은 실제 VFS 라이브러리 코드 내에 `LocalFileSystem` 클래스에서 소유자 객체가 정리되는 과정에서 컨테이너 내에 불필요한 `WeakRefFileListener` 객체를 제거하지 않아 발생하는 결함임을 확인할 수 있었다. `JunctionTests.testAncestor()` 분석 결과로 도출된 180개의 객체군에서 `WeakRefFileListener`는 4번째로 가장 높은 의심에 위치하였다. `WeakRefFileListener`보다 의심도가 높은 3개의 객체군은 테스트 케이스의 특성 상 반복실행에 따라 정상적으로 추가 할당되는 자원에 해당하는 객체로, 검토 결과 메모리 블롯과 관련이 없는 것을 확인했다.

메모리 블롯 객체가 발생하는 원인을 분석하기 위해, 메모리 덤프로부터 `WeakRefFileListener` 객체군에 속한 객체의 참조 관계를 분석한 결과, `WeakRefFileListener` 객체가 `LocalFileSystem` 내 컨테이너에 한번 추가가 된 이후 테스트 종료 전까지 제거되지 않음을 확인할 수 있었다. 해당 객체가 불필요한 객체인지 판별하기 위해, `JunctionTests.testAncestor()`가 검사하는 시나리오를 분석한 결과, 테스트 케이스 구동 코드에서 `WeakRefFileListener` 객체와 관련된 다른 객체는 제거되는 반면 `WeakRefFileListener` 객체가 제거되지 않음을 알 수 있었다. 메모리 블롯 여부를 확인하기 위해, 관련 객체의 제거 시 `WeakRefFileListener` 객체가 컨테이너에서 제거되도록 코드를 수정하자, 테스트 케이스는 여전히 성공하면서도 메모리 블롯 증상이 사라짐을 확인할 수 있었고, 이를 바탕으로 분석 결과가 실제 메모리 블롯 문제를 탐지하였음을 확인할 수 있었다.

`CustomRamProviderTest.testReadyEmptyFileByByte()`와 `FileLockTestCase.testResolveAndOpenCloseContent()` 탐지 결과, 역시 테스트 케이스 내에서 불필요한 메모리 사용을 발생 시키는 메모리 블롯과 연관되어 있음을 확인하였다. `JunctionTets.testAncestor()` 경우와 달리, 이 2개 경우는 결함이 메모리 블롯이 테스트 케이스 코드에 위치하므로, 상대적으로 위험도가 낮은 결함으로 볼 수 있다. 다만, 테스트 케이스 코드는 개발자들에게 표준적 용례를 나타내는 용도로 쓰인다는 측면에서, 해당 결함 역시 수정되어야 하는 실제적 결함으로 볼 수 있다.

표2. 메모리 블롯 검출 결과

Test case	Mem증가	선형회귀계수	R^2
JunctionTets.testAncestor()	2.32MB	2370.4	0.979
CustomRamProviderTest. testReadyEmptyFileByByte()	2.26MB	3605.4	0.969
FileLockTestCase. testResolveAndOpenCloseContent	50.00MB	50952.2	0.999

5. 결론

본 논문에서는 Java 프로그램의 메모리 누실 오류의 한 종류인 메모리 블롯의 발생을 효과적으로 탐지하기 위해 유닛 테스트 케이스 반복 실행을 통한 새로운 동적 분석 기법을 제안 하였고, Apache Commons VFS를 대상으로 한 사례연구를 통해 제안한 기법의 결함 검출 효용성을 논의하였다. 제안한 기법을 Apache Commons VFS에 적용한 결과, 라이브러리 코드에서 1개의 메모리 블롯 결함과 테스트 코드에서 2개의 메모리 블롯 결함을 신규로 탐지할 수 있었다.

향후에는 Apache Commons VFS 이외에 다양한 Java 프로그램에 개발한 동적 분석 기법을 적용하여 일반적 유용성을 실험할 계획이다. 또한 본 기법을 유닛 테스트 케이스 자동 생성과 연계하여, 개발자로부터 주어진 유닛 테스트 케이스가 없거나 부족한 경우에도 능동적으로 테스트 케이스를 생성하며 메모리 블롯을 검출하는 방법을 연구할 계획이다.

참조문헌

[1] N. Mitchell et al., Four Trends Leading to Java Runtime Bloat, IEEE Software, Jan/Feb, 2010
[2] G. Xu et al., Go with the Flow: Profiling Copies to Find Runtime Bloat, PLDI, 2009
[3] M. D. Bond and K. S. McKinley, Bell: Bit-Encoding Online Memory Leak Detection, ASPLOS, 2006
[4] M. Jump and K. S. McKinley, Cork: Dynamic Memory Leak Detection for Garbage-collected Languages, POPL, 2007
[5] Commons Virtual File System, Apache Software Foundation, https://commons.apache.org/proper/commons-vfs
[6] JHat – Java Heap Analysis Tool, Oracle https://docs.oracle.com/javase/7/docs/technotes/tools/share/jhat.html