동적 심볼릭 테스팅(Concolic 테스팅): 효과적인 오류 검출을 위한 실용적인 Whitebox 테스트 입력 생성 기법

한국과학기술원 ▌김문주*·김윤호**
한동대학교 ▌홍 신**

1. 서 론

제 4차 산업혁명 사회의 핵심 기능이 SW 기술로 구현됨에 따라, 오늘날 개발되는 SW의 규모와 복잡도가 폭발적으로 증가하고 있다. 이러한 SW의 양적 팽창과 달리, 생산되는 SW의 안전성을 진단하고 보장하는 품질 생산성의 향상은 미미한 수준에 그치고 있어, SW 품질 관리에 소요되는 개발 비용이 빠른 속도로 가중되고 있다. 2018년 발표된 World Quality Report는 2018년 기준 조사대상 기업의 총 IT 지출 중 약 26%가 SW 품질관리 및 테스팅에 소요되고 있다고 보고하고 있다(그림 1 참조). 이는 SW 개발 과정에서 품질관리 및 테스팅에 많은 비용이 소요되고 있음을 뜻한다[1]. 빌 게이츠가 2002년 OOPSLA 기조연설에서 예견한대로, IT 기업의 주업무는 SW 품질관리 및 테스팅으로 전환되고 있으며, IT 산업이 경제 성장을 이끌어 가고 있는 오늘날 글로벌 산업 구조에서 SW 테스팅은 가장 결정적인 생산활동 중 하나로 자리 매김하고 있다.

SW 테스팅은 SW개발 과정에서 산출된 프로그램 코드에 여러 값을 입력하여 실제 동작을 발생시킨 후, 해당 동작을 관찰하여 목표 품질(예: 정확성, 시간성능)의 달성 여부를 판별하는 품질관리 방법론이다. SW 테스팅은 SW 개발 과정에서 필연적으로 이루어지는 활동이며, 구체적이고 실제적인 프로그램 실행 결과에 기반하기 때문에, 검출된 오류의 유효성이 높다는 장점이 있다(예: 시스템 테스팅은 오경보(false alarm)이 없음).

SW 테스팅을 통해 프로그램 오류를 효과적으로 색출하기 위해서는 검증대상 프로그램의 다양한 실행 상황을 발생시키는 여러 가지 프로그램 입력 값, 즉 테스트 입력 집합의 확보가 필수적이다. 하지만, SW 개발자가 SW 요구사항으로부터 개별 테스트 입력 값을 수작업으로 도출하는 전통적인 SW 테스트 입력 작성 방법은 많은 인력 비용이 소요되기 때문에 시장 출시가 긴박한 오늘날 IT 개발 현장에서 효과적인 품질 관리를 구현하는데 심각한 한계점을 가진다. 수많은 개발자가 수많은 코드변화를 동시다발적으로 발생시키며 개발이 진행되는 대규모 SW 프로젝트에서 노동집약적 SW 테스팅 관행은 개발 프로세스의 병목현상을 발생시켜 생산성에 문제를 일으킬 뿐만 아니라, 궁극적으로는 품질보증 미달을 발생시키는 SW 품질 사고의 주요 위협 원인으로 부각되고 있다.

동적 심볼릭 테스팅 기술(Concolic 테스팅[1])은 프로그램 소스코드를 입력 받은 후, 해당 프로그램의 구조와 의미를 분석하고, 이를 바탕으로 프로그램의 모든 실행 경로를 각각 도달하도록 테스트 입력 값을 연속적으로 생성하는 Whitebox 테스팅 기법이다. 동적 심볼릭 테스팅의 기반이 되는 심볼릭 실행(symbolic execution) 기법은 프로그램의 상태를 논리식으로 표현함으로써 프로그램 동작에 대한 연역적 분석이 가능하게 하게 하는 방법으로, 1976년부터 이론적 토대가 정립[2]되었다. 그리고 2000년 초반 논리식의 해답을 자동으로 찾는 SMT 해법기(Satisfiability Module Theory Solver)[3]의 비약적 성능 향상과 더불어

* 종신회원
** 정회원
† 본 연구는 정부(과학기술정보통신부)의 재원으로 한국연구재단의 지원(NRF-2016R1A2B4008113, NRF-2017R1C1B1008159, NRF-2017M3C4A7068177, NRF-2017M3C4A7068179)과 정부(교육과학기술부)의 재원으로 한국연구재단의 지원(NRF-2017R1D1A1B03035851)을 받아 수행됨

1) Concolic 은 CONCrete + symbOLIC의 합성어다.

2000년 중반에 여러 연구자들이 프로그램의 정적 정보에 런타임 데이터(동적 정보)를 접목하여 실제 프로그램에서 발생하는 복잡한 논리식을 효과적으로 근사하는 방법을 제시함으로써, 실용적 테스트 입력 자동 생성 기법으로 주목받아 지난 15년간 비약적 발전이 이루어졌다. 이러한 발전에 힘입어, 동적 심볼릭 테스팅 기법은 다양한 산업체에서 유닛 레벨 테스팅을 위한 테스트 입력 값 생성에 성공적으로 활용되고 있다[4-5]. 또한, 동적 심볼릭 테스팅을 시스템 레벨 테스팅에도 효과적으로 적용하기 위하여 기법의 규모 확장성(scalability)을 개선하는 연구가 현재에도 활발하게 이루어지고 있다[6].

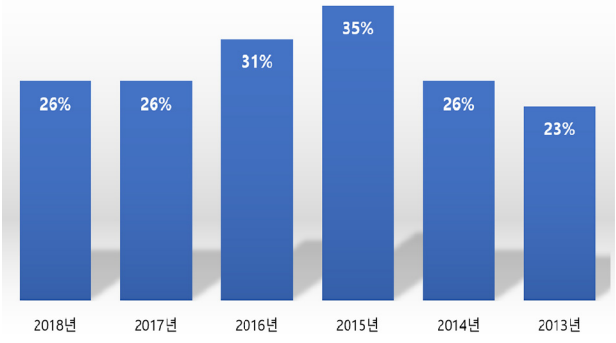

그림 1 기업의 총 IT비용 지출 중 QA/테스팅 비용 [World Quality Report 2018-2019[1]]

본 글은 오늘날 동적 심볼릭 테스팅 기법의 발전에 대한 이해를 돕기 위해, 동적 심볼릭 실행의 핵심 기술을 설명한 후, 임베디드 SW를 대상으로 동적 심볼릭 테스팅을 성공적으로 적용한 사례와 현재 활발히 연구가 진행되고 있는 주요 발전 방향을 소개하도록 한다. 2장에서는 예제를 통해 동적 심볼릭 테스팅의 원리와 주요 기술 요소를 개관하고, C 프로그램을 대상으로 기존에 개발되어 온 오픈소스 도구를 소개한다. 3장에서는 임베디드 SW를 대상으로 동적 심볼릭 테스팅을 성공적으로 산업 현장에 적용 사례를 소개한다. 4장에서는 현재 진행되고 있는 주요 연구 방향을 소개한 후, 5장에서 향후 연구 방향을 소개하며 마무리한다.

2. 동적 심볼릭 테스팅 기술의 개요

2.1 예제

동적 심볼릭 테스팅이 어떻게 테스트 대상 프로그램의 모든 가능한 수행경로를 실행하는 테스트 입력을 생성하는지 아래의 예제를 통해 설명한다.

```
// Test input a, b, c
1 void f(int a, int b, int c) {
2   if (a == 1) {
3     if (b == 2) {
4       if (c == 3*a + b) {
5         Error();
6 }}}}
```

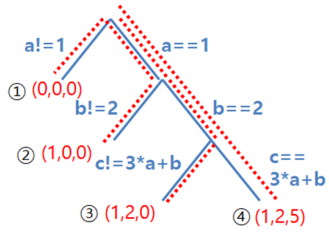

그림 2 동적 심볼릭 테스팅 설명 예제

위 예제에서 테스팅 대상 함수 f는 입력으로 변수 a, b, c를 받는다. 함수 f 의 2라인, 3라인, 4라인의 조건을 모두 만족해야 실행되는 5라인에 오류(즉, Error())가 있는 상황이다.

이 때 랜덤 테스팅, 즉 난수(random) 값으로 입력값(a, b, c 값)을 생성하는 방식으로 5라인의 오류를 발견할 가능성은 매우 낮다. 이는 a 값을 랜덤하게 생성해서 a==1을 만족할 확률은 $1/2^{32}$, 동시에 b 값을 랜덤하게 생성해서 b==2를 만족할 확률은 $1/2^{64}$ (= $1/2^{32}$ × $1/2^{32}$), 그리고 동시에 c 값을 랜덤하게 생성해서 c==5를 만족할 확률은 1/296 (= $1/2^{32}$ × $1/2^{32}$ × $1/2^{32}$)이기 때문이다.

동적 심볼릭 테스팅 기법은 테스트 대상 코드의 모든 분기문 실행의 조합, 즉 실행 가능한 모든 경로를 복합적으로 수행(combinatorial execution)하는 테스트 입력 값 집합을 자동으로 생성한다. 위 예제에서는 다음의 총 4개 테스트 입력을 생성하여 5라인의 오류를 검출하게 된다:

- **테스트 입력1 (t_1):** 첫번째 테스트 입력은, 프로그램 실행 정보가 없기 때문에 임의의 값으로 정한다. 설명의 편의를 위해 (a,b,c)=(0,0,0)이 생성됐다고 가정하자. 이 때 t_1은 a!=1의 경로를 수행했으므로, t_1의 심볼릭 경로 논리식 (symbolic path formula (SPF)) ϕ_1는 a!=1이다.
- **테스트 입력2 (t_2):** 새로운 수행 경로를 실행하는 테스트 입력값을 생성하기 위해, 직전에 생성한 테스트 입력의 SPF의 제일 마지막 조건 구문을 부정 (negation) 한 논리식 ψ_1=!(a!=1)을 생성한다. ψ_1을 풀면 해답은 (1,0,0) 이 되고, 이 해답을 2번째 테스트 입력 t_2로 사용하여 프로그램 실행한다. 그 결과 프로그램은 2라인, 3라인, 6라인을 차례로 실행하고, 이 실행의 SPF ϕ_2는 a==1 && b!=2가 추출된다.
- **테스트 입력3 (t_3):** 마찬가지로, 직전에 생성한 테스트입력의 SPF의 제일 마지막 조건구문을 부정 (negation) 한 논리식 ψ_2 (즉 a==1 && !(b!=2)) 을

생성한다. ψ_2을 풀면 해답은 (1,2,0)이 되고, 이 해답을 3번째 테스트 입력 t_3로 사용하여 t_3의 SPF ϕ_3는 a==1 && b==2 && (c!=3*a +b) 이다.

- **테스트 입력4 (t_4):** 직전에 생성한 테스트입력의 SPF의 제일 마지막 조건구문을 부정 (negation) 한 논리식 ψ_3 (즉 a==1 && b==2 && (c==3*a +b)) 을 생성한다. ψ_3을 풀면 해답은 (1,2,5)가 되고, 이 해답을 4번째 테스트 입력 t_4로 사용하여 테스트 하면, 드디어 5라인에 위치한 오류를 발견한다.

2.2 주요 기술 요소

동적 심볼릭 테스팅 기법을 구성하는 주요 기술 요소는 다음과 같다.

2.2.1 심볼릭 환경 모델링 (symbolic environment modeling)

심볼릭 환경 모델링이란 테스트 입력 중 어떠한 변수를 심볼릭 변수로 사용할 지 결정하는 방법이다. 한정된 테스팅 시간 안에 최대한의 오류를 검출하기 위해, 실제 프로그램의 수행 동작을 정확히 표현하면서도 심볼릭 경로 공간을 (symbolic path space) 작게 유지하도록, 심볼릭 환경 모델링이 필요하다. 즉, 수 많은 입력 변수를 모두 심볼릭 변수로 사용하는 대신, 핵심적인 변수만을 선별함으로써 테스트 입력 생성에 유용하면서도 간단한 심볼릭 경로 논리식이 생성되도록 하여야 한다. 또한, 실제 프로그램의 수행동작을 정확히 표현하기 위해, 심볼릭 변수로 선언된 변수들 간의 제약 조건을 명시하는 것이 필요하다. 예를 들어, 이진 탐색(binary search) 함수 bin_search(int *arr, int size)를 테스팅하는 경우, bin_search의 입력인 정수 배열 arr 안의 모든 원소가 정렬되어 있어야 테스팅 결과값이 제대로 도출된다. 따라서, 사용자는 bin_search의 입력으로 전달되는 arr의 모든 원소가 심볼릭 변수이면서도 정렬되어 있도록 심볼릭 환경을 모델링해 주어야 한다. 이러한 심볼릭 환경 모델링은 특히 동적 심볼릭 테스팅을 통한 자동 유닛 테스팅에서 거짓 경보(false alarm)를 줄이기 위해 중요하다.

동적 심볼릭 테스팅은 테스트 입력을 자동으로 생성하는 기술이지만, 일반적으로 심볼릭 환경 모델링은 개발자/테스터가 수작업으로 이루어진다. 따라서, 동적 심볼릭 테스팅을 성공적으로 적용하기 위해서는, 테스팅 대상 SW의 도메인 지식과 동적 테스팅 기술 양쪽 모두에 능통한 전문가가 필요하다.

2.2.2 심볼릭 경로 논리식 (symbolic path formula) 생성

심볼릭 경로 논리식(SPF)을 추출하기 위해서는 테스팅 대상 프로그램의 실제 실행을 관찰하여 이에 해당하는 심볼릭 경로 논리식을 구성하는 과정이 필요하다. SPF를 추출하는 방식은 크게 가상머신을 기반으로 하는 방법(예: KLEE[7])과 테스트 대상 프로그램 코드에 탐침(probe)을 삽입하는 방법(예: CROWN[8]) 이 있다. 이 때, 실제 수행 경로의 논리식을 부울연산식(Boolean Expression, Bit-vector Fomula) 수준으로 정교하게 표현하는 방법이 널리 쓰이고 있다. 또한, 선형 정수연산식(Linear-Integer Arithmetic (LIA)) 수준으로 추상화하여 표현하는 방법도 유용하게 쓰이고 있는데, 이는 LIA로 표현된 SPF가 상대적으로 단순하여 보다 빠르게 SPF를 추출하고 풀 수 있기 때문이다.

2.2.3 새로운 테스트 입력을 생성하기 위해, 부정할 SPF 조건구문 선택 (symbolic search strategy)

동적 심볼릭 테스팅은, 현재 추출한 SPF안의 어떤 조건 구문을 부정(negate) 하느냐에 따라, 생성되는 테스트 입력 값의 순서가 크게 달라질 수 있다. 예를 들어 "2.1예제" 섹션에서는, SPF ϕ의 가장 마지막 if 조건을 부정하는 방식, 즉 깊이 우선 선택(DFS)이라고 불리는 심볼릭 탐색 전략(symbolic search strategy)을 사용하였다. 또한, 부정할 조건 구문을 무작위로 선하는 전략도 높은 테스팅 커버리지를 달성하는 입력값을 생성하는 것으로 알려져 있다. 그 외에도, 아직 도달하지 못한 분기를 빨리 도달하기 위해 제어 흐름 그래프(control-flow graph)에 나타난 정적 정보를 활용하는 여러 가지 심볼릭 탐색 전략들이 개발되고 있다(예: CFG[9], CGS[10], SGS[11])

2.2.4 심볼릭 경로 논리식 (symbolic path formula) 해법기

"2.2.3절"에서 생성된 SPF ψ를 SMT 해법기한테 입력으로 넣어 주면, SMT 해법기가 SPF ψ의 해를 구한다. 현재, 다양한 장단점을 갖는 여러 종류의SMT 해법기들이 존재하며, 최근에 오픈소스로 전환된 Microsoft의 Z3 해법기가 여러 동적 심볼릭 테스팅 도구에 널리 사용되고 있다. 이 때 SPF ψ의 해를 보다 빠르게 구하기 위해 불필요한 조건을 제거하는 SPF 슬라이싱(slicing) 기법[12], 또는 이전에 구했던 SPF의 해를 재활용하는 SPF 캐싱(caching) [13] 등의 기법이 쓰인다.

2.3 C 프로그램을 대상으로 하는 오픈소스 도구

2.3.1 KLEE [7]

KLEE는 C 프로그램에서 컴파일된 LLVM 중간언어(bitcode)를 대상으로 하는 동적 심볼릭 테스팅 도구이다(즉, LLVM 가상머신 위에서 동작한다). KLEE는 COREUTILS, Busybox 같은 UNIX 유틸리티들[7], 그리고 컴퓨터 비전 코드[14] 등을 포함한 다수의 SW 시스템의 결함과 취약점들을 찾는데 성공적으로 사용

되었다. KLEE는 높은 테스트 커버리지를 달성하는 회귀 테스트 케이스(regression test suite)를 생성에도 효과적으로 사용되었다. 예를 들어, KLEE는 89개의 프로그램으로 구성된 COREUTILS 유틸리티에 대해 평균 90%이상의 구문 커버리지를 달성하는 테스트 케이스를 성공적으로 자동 생성했다[7].

2009년 6월부터 KLEE는 200명 이상의 개발자가 참여하는 오픈소스 프로젝트 형태로 발전되고 있으며, 다수의 연구 그룹들이 무선 센서 네트워크[15]부터 자동 디버깅[16], 바이너리 코드 역공학(reverse engineering)[17], 온라인 게임[18], GPU를 위한 테스팅과 검증[19] 등을 포함한 다양한 분야에서 연구에 이용되고 있다.

2.3.2 CROWN [8]

CROWN은 C 프로그램 소스코드를 대상으로 하는 오픈소스 동적 심볼릭 테스팅 도구이다[8]. CROWN은 SPF 추출을 위하여 테스트 대상 소스 코드에 탐침을 삽입하는 방식을 사용한다. CROWN은 가상머신 기반으로 동작하는 KLEE에 비해 테스트 입력 생성 속도가 빠르다. 또한, CROWN은 실제 프로그램 실행과 병렬적으로 구동되므로 보다 실제적인 런타임 정보를 활용하기 때문에, 심볼릭 분석이 불가능한 외부 라이브러리 함수 호출 등에 보다 효과적으로 대응할 수 있다.

CROWN은 삼성전자[4], LG전자[20], 현대모비스[5] 등 한국 산업체와의 밀접한 산학연구를 통해 10여년간 산업체 SW의 수많은 오류를 검출하면서 (“4. 사례연구” 참조) 산업체 사용자 피드백을 통해 개선되어 왔다. 그 결과, 임베디드 SW에 많이 사용되는 비트 단위 구조체(bit-field)나 비트 단위 연산자(bit-wise operator) 등 C의 세부적인 요소를 충실하게 지원한다. 또한, 사용자가 수작업으로 생성한 테스트 입력 값을 초기 테스트 입력(seed input)으로 사용가능하고, 심볼릭 경로에 대한 사용자가 이해하기 쉬운 형태로 변환 출력하는 등 실제 산업체에서 동적 심볼릭 테스팅을 적용하는데 편리한 기능들을 제공하고 있다.

3. 임베디드 SW 테스팅을 위한 동적 심볼릭 테스팅

SW 시대가 도래함에 따라 기존 하드웨어 제품에 다양한 기능을 구현하기 위한 임베디드 SW의 개발이 활발하다. 임베디드 SW가 제품의 핵심 기능은 물론 복잡한 부가 기능을 담당함에 따라 임베디드 SW의 복잡도가 증가하고 있으며, 복잡한 임베디드 SW를 효과적으로 테스트하기 위한 테스트 기법이 점차 요구되고 있다. 본 장에서는 스마트폰 SW와 자동차 제어 SW와 같은 복잡한 임베디드 SW를 효과적으로 테스트하기 위해 동적 심볼릭 테스팅을 적용한 산업체 사례 연구를 소개한다.

3.1 사례1. 스마트폰 SW 유닛 테스팅[4]

삼성전자는 수 백만 라인 규모의 임베디드 C 프로그램을 체계적이고 자동화된 방법으로 테스트하기 위해 KAIST SWTV 연구실과 공동으로 동적 심볼릭 테스팅 도구인 CONBOL[4]을 개발하였다. CONBOL은 대규모 임베디드 C 프로그램의 각 함수의 유닛 테스팅을 자동으로 수행하기 위한 테스트 드라이버(test driver), 테스트 스텁(test stub), 테스트 입력 값 일체를 동적 심볼릭 테스트 생성 기술을 사용하여 자동으로 생성한다.

임베디드 SW는 특정 하드웨어에서만 수행되기 때문에 테스트 환경 구축이나 자동화된 테스트 생성 기술을 적용하기 어렵다. 따라서 임베디드 SW에 동적 심볼릭 테스트 기술을 적용하기 위해서는 하드웨어 의존적인 코드를 유닛 테스트 스텁 함수(유닛 테스트 수행 시 원래 함수가 수행할 역할을 단순화한 테스트 목적의 대체 함수)로 치환하여 특정 하드웨어에 의존하지 않는 유닛 테스트 환경을 구축해야 한다. 이를 위하여, CONBOL은 하드웨어 인터럽트, 메모리 매핑 하드웨어 I/O 등 하드웨어를 직접 제어하는 함수 코드를 스텁 함수로 자동으로 치환한다.

CONBOL을 삼성전자가 자체 개발한 400만 라인 규모의 임베디드 C 프로그램에 적용한 결과, 비정상 종료나 보안 취약점을 야기할 수 있는 24개의 심각한 크래시 버그(crash bug)를 발견하였다. CONBOL에는 유닛 테스팅 과정에서 발생할 수 있는 거짓 경보를 효과적으로 제거하는 휴리스틱 알고리즘이 적용되었다. 실제 사례 연구에서 거짓 경보 알고리즘을 전혀 적용하지 않았을 경우 CONBOL은 총 5,884개의 경보를 보고한 반면, 거짓 경보 제거 알고리즘 적용 시 5,607개의 알람이 제거되었고, 남은 277개를 삼성전자 SW 엔지니어가 분석하여 24개의 실제 크래시 버그를 디버깅할 수 있었다.

3.2 사례2. 자동차 SW 테스팅[5]

현대모비스는 SW 테스트 엔지니어가 수작업으로 작성하던 기존의 SW 테스트 방법론을 대신하여 동적 심볼릭 테스팅 기법을 통한 자동화된 SW 테스트 방법론을 구축하기 위하여 KAIST SWTV 연구실 공동으로 MAIST[5] 도구를 개발하였다. MAIST는 현대모비

스가 개발하는 자동차 제어 SW 모듈의 구조적 특성을 고려하여 실제적인 유닛 테스트 환경을 구축하는 자동 심볼릭 환경 모델링 기능을 제공한다. 또한, MAIST는 자동차 임베디드 SW에서 자주 사용되는 공용체(union)와 비트 필드(bit-field)를 효과적으로 지원하기 위하여, 공용체와 비트 필드를 사용하는 C 소스 코드가 주어지면 비트 수준 연산(C 언어의 &, |, << 등)을 사용하여 해당 프로그램과 동일한 의미를 가지면서 공용체와 비트 필드를 사용하지 않는 C 소스 코드를 생성하는 기능을 제공함으로써 동적 심볼릭 테스팅이 임베디드 C 프로그램에 대해 충실히 적용되도록 지원한다.

MAIST를 통합형 차체제어시스템(IBU)과 써라운드 뷰모니터링 시스템(SVM) SW 테스트 프로세스에 적용한 결과, IBU와 SVM의 SW에 들어가는 SW 테스트 엔지니어 공수(Man-month)를 각 53%, 70% 절감하는 효과를 거두었다. 이러한 성공을 바탕으로 현대모비스는 MAIST를 전장 SW 개발 전반에 보급하여, 제동과 조향 등 핵심 제어 SW는 물론 자율주행/첨단 운전자 지원 시스템(ADAS) 테스팅에도 적용할 예정이다.

4. 최신 연구 동향

4.1 대규모 SW의 오류를 효과적으로 검출하기 위해, 심볼릭 경로 폭발 문제 해결을 위한 연구노력

동적 심볼릭 테스팅은 테스팅 대상 SW의 모든 가능한 수행 경로를 실행하는 테스트 입력 값을 만들기 때문에, 1000여줄의 SW에 대해서도 수백만 개의 테스트입력을 생성하는 등, 높은 계산 비용이 소요된다. 따라서, 현실적으로 주어진 테스팅 시간 안에 최대한 많은 오류를 검출하기 위한 다양한 휴리스틱 탐색 알고리즘이 연구되고 있다. 본 장에서는 광대한 심볼릭 경로 공간(symbolic path space)안에서 중복되는 탐색을 최소화하며(“4.2 조립식 동적 심볼릭 테스팅”), 모든 탐색 공간을 살펴보기보다 특정한 코드 영역을 집중하여 탐색하고(“4.3. 목표지향적 탐색 알고리즘”), 또한 동적 심볼릭 테스팅 기술의 한계를 탐색 기반 테스팅 기술과 결합하여 극복하는 기술(“4.4 동적 심볼릭 테스팅과 탐색 기반 테스팅 기술을 혼합 적용”) 및 분산 시스템을 활용한 동적 심볼릭 테스팅의 속도를 향상하기 위한 기술(“4.5 분산 동적 심볼릭 테스팅”)을 소개한다.

4.2 조립식 동적 심볼릭 테스팅

조립식(compositional) 동적 심볼릭 테스팅 기술은 복잡한 프로그램을 효율적으로 분석하기 위해 제안된 기술로, 크고 복잡한 프로그램 전체에 동적 심볼릭 테스트 생성 기술을 적용하는 대신 프로그램을 구성하는 각 컴포넌트 (주로 함수) 단위로 동적 심볼릭 테스팅을 적용하고 그 결과를 합성하여 전체 프로그램에 대한 테스트를 생성하는 기술이다.

- SMASH [21]는 정적 분석과 동적 분석 기법을 혼합하여 함수 요약 정보를 생성하고 이를 바탕으로 효율적으로 프로그램 분석을 수행하는 기법이다. 정적 분석 기법을 적용해서 생성한 함수 요약 정보를 사용하여 해당 함수가 주어진 항상 만족하는지 먼저 검사하고 만약 항상 만족하지 못한다면 동적 심볼릭 테스트 기법을 적용하여 생성한 함수 요약 정보를 사용하여 해당 함수가 요구사항을 만족하지 못하는 실제 테스트 입력 값을 생성한다. SMASH를 69개의 Windows OS 디바이스 드라이버에 적용한 결과 조립식 동적 심볼릭 테스팅 기술을 적용하지 않았을 때 보다 3배 이상 더 빠른 수행 결과를 보였다.
- ALTER [22]는 함수 요약 정보를 사용하여 실행하고자 하는 특정 코드 위치를 동적 심볼릭 테스트 기술로 빠르게 실행할 수 있게 하는 기법이다. 먼저 실행 대상 코드 위치가 포함된 함수의 요약 정보를 사용해서 실행 대상 코드 위치를 실행할 수 있는지 검사하고 실행 가능하면 실행 대상 코드 위치를 포함한 함수를 호출하는 호출자(caller) 함수들을 순차적으로 따라 올라가면서 해당 코드 구문이 실행 가능한지 검사한다. 호출자 함수를 따라 올라가는 과정에서 불필요한 함수 호출 경로를 탐색하는 것을 피하기 위해 크레이그 보간식(Craig interpolant)을 사용한 최적화를 수행한다.

4.3 목표지향적 탐색 알고리즘

동적 심볼릭 테스팅에서 목표지향적(directed) 탐색 알고리즘은 실행하고자 하는 특정 코드 위치(주로 코드 라인)를 입력으로 받아서 해당 코드 위치를 최대한 빠르게 실행하도록 탐색을 전개하는 기법이다. 주로 버그 재현, 패치 검증 등에 많이 사용한다.

- Mix-CCBSE[23]는 후방 (backward) 함수 호출 체인 탐색과 전방 (forward) 동적 심볼릭 테스트 기법을 결합하여 목표한 코드 위치를 빠르게 실행하는 기법이다. 먼저 목표 코드 위치가 포함된 함수에서 동적 심볼릭 테스트를 적용하여 함수 시작점에서 목표 코드 위치를 실행하는 경로를

찾는다. 해당 경로를 찾으면 목표 코드 위치가 포함된 함수를 호출하는 호출자 함수들에서 동적 심볼릭 테스트를 수행하여 호출자 함수에서 목표 코드 위치를 포함된 함수를 호출하는 실행 경로를 찾고 이 과정을 프로그램 시작 함수(예: main())까지 도달하도록 반복한다.

- Cilocnoc[24]은 목표 코드 위치에서부터 후방 동적 심볼릭 테스팅 기법을 적용하여 프로그램 시작점에서 도달하는 경로가 있는지 탐색하는 기법이다. 후방 동적 심볼릭 테스팅 기법을 적용하기 어려운 구문 (복잡한 반복문, 포인터 연산 등)을 만나는 경우 전방 탐색 기반 테스트 생성 기법을 적용하여 해당 경로를 실행할 수 있는지 검사한다.
- KATCH[25]는 프로그램 패치를 집중적으로 테스트하기 위한 기법이다. 회귀 테스트 입력(regression test inputs)과 프로그램 패치를 받아서 프로그램 패치가 수정한 코드 라인을 실행하는 테스트 입력을 만드는 기법이다. 동적 분석과 정적 분석 기법을 사용하여 각 코드 구문과 변수 사이의 의존 관계를 계산하고 의존 관계 정보를 활용하여 수정한 코드를 실행하는 테스트를 생성하도록 심볼릭 경로 탐색 과정을 유도한다.

4.4 동적 심볼릭 테스팅과 탐색 기반 테스팅 기술을 혼합 적용

논리식 형태의 표현이 어려운 프로그램 요소(예: 소스 코드가 없는 외부 라이브러리 함수 호출, 복잡한 포인터 연산)에 대한 동적 심볼릭 테스팅의 한계를 극복하기 위하여 탐색 기반(search-based) 테스팅 기술을 혼합 적용하는 기술이 개발되고 있다.

- UNL ESQUARED연구실과 KAIST SWTV 연구실은 동적 심볼릭 테스팅과 탐색 기반 테스팅 기술의 비교 연구를 통해 두 기법이 쉽게 커버리지를 높일 수 있는 코드의 특징이 서로 다름을 보였고[26-27], 두 기법을 일정 시간 간격으로 번갈아가며 실행함으로써 각각의 기술보다 더 높은 커버리지를 달성하는 새로운 기법을 제안하였다[28-29].
- 보안 취약점을 효과적으로 찾기 위해 동적 심볼릭 테스팅과 탐색 기반 테스팅 기술인 퍼징(fuzzing)을 혼합하는 기술이 개발되고 있다. Driller[30]는 바이너리 실행 파일의 보안 취약점을 찾기 위해, 우선 퍼징 기법으로 넓은 범위의 다양한 실행 경로를 탐색한 후 각각의 실행 경로를 더 면밀히 탐색하기 위해 동적 심볼릭 테스팅을 적용하였다. QSYM[31]은 동적 심볼릭 테스팅 기법을 퍼징에 더 적합하게 최적화를 하여 Driller 보다 더 빠른 속도로 보안 취약점을 찾도록 성능을 향상시켰다.
- NEC 미국 연구소는 C/C++ 프로그램을 테스트하기 위해 동적 심볼릭 테스팅 기법과 탐색 기반 테스팅 기술을 결합하였다[32]. 탐색 기반 테스팅 기술을 결합하여 유닛 테스트 드라이버 함수가 호출할 함수 호출 순서를 정하고 해당 함수의 입력 값을 생성하기 위해 동적 심볼릭 테스팅 기법을 적용하여 테스트 성능을 향상시킨 기법을 제시하였다.

4.5 분산 동적 심볼릭 테스팅

Amazon AWS와 같은 클라우드 컴퓨팅 환경이 대세가 됨에 따라, 적은 비용으로 분산 동적 심볼릭 테스팅을 위한 분산 시스템을 구축할 수 있다. 본 절에서는 분산 시스템을 활용한 분산 동적 심볼릭 테스팅 기술을 소개한다.

- NASA JPL연구소에서는 Symbolic Java PathFinder [33]를 확장하여 분산 심볼릭 실행 기술을 개발하였다[34]. 이 방법은 제한된 범위 내에서 심볼릭 실행을 수행하여 초기 작업량을 계산하고 초기 작업량을 주어진 작업 노드(worker node)에 분산하여 심볼릭 실행을 수행하는데, 처음에 한 번 작업 분산을 수행한 이후로는 작업 분산을 수행하지 않기 때문에 작업이 오래 걸리는 노드를 작업이 먼저 끝난 노드가 도와주지 않기 때문에 규모 확장성(scalability)이 크게 제한적이다.
- Cloud9[35]은 KLEE를 확장하여 개발된 C 프로그램을 위한 분산 동적 심볼릭 테스팅 도구이다. 효과적인 작업 분산을 위하여, Cloud9은 워크 스틸링(work stealing)을 활용한 온-디맨드(on-demand) 작업 분산 기능을 구현하였다. 마스터 노드가 주기적으로 각 작업 노드와 교신하면서 각 노드의 남은 작업량 정보를 관리하고 유휴 노드 발생시 작업량이 많은 노드에서 유휴 노드로 처리해야 할 잡을 분산해준다.
- SCORE[36]는 CROWN을 기반으로 한 분산 동적 심볼릭 테스팅 도구이다. Cloud9과 같이 워크 스틸링을 활용한 온-디맨드 작업 분산 기능을 구현하고 있어 높은 수준의 규모 확장성을 보인다. Cloud9의 경우 마스터 노드가 주기적으로 각 작업 노드와 교신해야 하기 때문에 노드 숫자가 늘어날수록 교신 비용이 커져서 규모 확장성이 제

한적인 반면 SCORE는 유휴 노드가 발생할 경우에만 온-디맨드로 작업 노드와 마스터 노드의 교신이 이뤄지기 때문에 규모 확장성이 더 좋다. Cloud9의 경우 48개 작업 노드 수준에서 규모 확장성의 한계를 보이지만 SCORE는 256개 이상의 작업 노드에서도 규모 확장성을 보인다.

5. 결론 및 향후 연구

동적 심볼릭 테스팅 기법은 검증대상 프로그램의 각 실행경로를 입력 변수에 대한 논리식으로 표현한 후 이를 SMT 해법기로 자동 해결함으로써, 프로그램의 각 실행경로를 도달하는 테스트 입력을 체계적으로 도출해내는 Whitebox 테스팅 기법이다. 지난 15년간 많은 연구 개발을 통해, 오늘날 동적 심볼릭 테스팅 기법은 산업 현장에서 개발되는 복잡한 C 프로그램에 대해서도 효과적인 테스트 입력 생성을 수행하고 있으며, 실제 임베디드 SW 개발 현장에 적용되어, 기존의 노동집약적 테스팅 관행을 기술 중심적 테스팅으로 혁신하는데 활용되고 있다. 지금도 전세계의 많은 연구자들은 대규모 SW에 대해 높은 결함 검출력을 갖는 완전 자동화된 테스트 입력 생성을 목표로 동적 심볼릭 테스팅 기술 향상을 위한 다방면의 연구를 활발히 전개하고 있다.

한동대 ARISE 연구실과 KAIST SWTV 연구실은 동적 심볼릭 테스팅에 뮤테이션 테스팅 과정에서 추출한 동적 정보를 활용함으로써 테스트 커버리지 달성을 향상시키는 파괴적 심볼릭 테스팅(invasive SW testing) 기술을 연구 중이다[37]. 또한 효과적인 유닛 테스팅 자동화를 위하여, 함수 간의 연관성을 고려한 새로운 심볼릭 환경 모델링 및 심볼릭 탐색 알고리즘을 개발함으로써[38], 기존 유닛 테스팅 보다 결함 검출력과 거짓 경보율이 동시에 개선된 동적 심볼릭 테스팅 기법 개발을 연구하고 있다.

참고문헌

[1] Capgemini, World Quality Report 2018-2019, 2018

[2] J. King, Symbolic Execution and Program Testing, (CACM), 19(7), 1976

[3] C Barrett, R Sebastiani, S Seshia, and C Tinelli, Satisfiability Modulo Theories, Handbook of Satisfiability, vol. 185 of Frontiers in Artificial Intelligence and Applications, IOS Press, 2009

[4] Y. Kim, Y, Kim, T. Kim, G. Lee, Y. Jang, and M. Kim, Automated Unit Testing of Large Industrial Embedded Software using Concolic Testing, ASE Experience track, 2013

[5] Y. Kim, D. Lee, J. Baek, and M. Kim, Concolic Testing for High Test Coverage and Reduced Human Effort in Automotive Industry, ICSE SEIP track, 2019

[6] Y. Kim, S. Hong, and M. Kim, Target-driven Compositional Concolic Testing with Function Summary Refinement for Effective Bug Detection, TECH-REPORT, 2019

[7] C. Cadar, D. Dunbar, and D. Engler, KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. OSDI, 2008

[8] CROWN, https://github.com/swtv-kaist/CROWN

[9] J. Burnim and K. Sen, Heuristics for Scalable Dynamic Test Generation, ASE, 2008

[10] H. Seo and S. Kim, How We Get There: A Context-guided Search Strategy in Concolic Testing, FSE, 2014

[11] Y. Li, Z. Su, L. Wang, and X. Li, Steering Symbolic Execution to Less Traveled Paths, OOPSLA, 2018

[12] CREST, https://github.com/jburnim/

[13] W. Visser, J. Geldenhuys, and M. Dwyer, Green: Reducing, Reusing and Recycling Constraints in Program Analysis, FSE, 2012

[14] P. Collingbourne, C. Cadar, and P. H. J. Kell, Symbolic crosschecking of floating-point and SIMD code, ACM Conference on Computer systems, 2011

[15] R. Sasnauskas and J. A. B. Link: Kleenet, automatic bug hunting in sensor network applications, SensSys, 2008

[16] C. Zamfir and G. Candea, Execution synthesis: a technique for automated software debugging. ECCS, 2010

[17] V. Chipounov and G. Candea, Reverse engineering of binary device drivers with RevNIC. ECCS, 2010

[18] D. Bethea, R. A. Cochran, and M. K. Reiter, Server-side verification of client behavior in online games. TISSEC, 2011

[19] G. Li, P. Li, G. Sawaya, I. Ghosh: GKLEE, Concolic verification and test generation for GPUs, PPoPP, 2012

[20] Y. Park, S. Hong, M. Kim, D. Lee, and J. Cho, Systematic Testing of Reactive Software with Non-deterministic Events: A Case Study on LG Electric Oven, ICSE SEIP track, 2015

[21] P. Godefroid, A. V. Nori, S. K. Rajamani, and S. D. Tetali, Compositional may-must program analysis: Unleashing the power of alternation, POPL, 2010

[22] N. Sinha, N. Singhania, S. Chandra, and M. Sridharan,

Alternate and learn: Finding witnesses without looking all over, CAV, 2012
[23] K. Ma, K. Phang, J. Foster, and M. Hicks, Directed symbolic execution, SAS, 2011
[24] P. Dinges and G. Agha, Targeted test input generation using symbolic-concrete backward execution, ASE, 2014
[25] P. Marinescu and C. Cadar. 2013. KATCH: High-coverage Testing of Software Patches, ESEC/FSE, 2013
[26] Z. Xu, Y. Kim, M. Kim, G. Rothermel, and M. Cohen, Directed Test Suite Augmentation: Techniques and Tradeoffs, FSE, 2010
[27] Z. Xu, Y. Kim, M. Kim, M. Cohen, and G. Rothermel, Directed Test Suite Augmentation: An Empirical Investigation, STVR, 25(2), 2015
[28] Z. Xu, Y. Kim, M. Kim and G. Rothermel, A Hybrid Directed Test Suite Augmentation Technique, ISSRE, 2011
[29] Y. Kim, Z. Xu, M. Kim, M. Cohen, and G. Rothermel, Hybrid Directed Test Suite Augmentation: An Interleaving Framework, ICST, 2014
[30] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, Driller: Augmenting fuzzing through selective symbolic execution, NDSS, 2016
[31] I. Yun, S. Lee, and M. Xu, Y. Jang, T. Kim, QSYM : A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing, USENIX SEC, 2018
[32] P. Garg, F. Ivancic, G. Balakrishnan, N. Maeda, and A. Gupta, Feedback-directed unit test generation for C/C++ using concolic execution, ICSE, 2013
[33] C. Pasareanu and N. Rungta, Symbolic PathFinder: symbolic execution of Java bytecode, ASE, 2010
[34] M. Staats and C. Pasareanu, Parallel symbolic execution for structural test generation, ISSTA, 2010
[35] S. Bucur, V. Ureche, C. Zamfir, and G. Candea, Parallel symbolic execution for automated real-world software testing, EuroSys, 2011.
[36] M. Kim, Y. Kim and G. Rothermel, A Scalable Distributed Concolic Testing Approach: An Empirical Evaluation, ICST, 2012
[37] Y. Kim, S. Hong, B. Ko, L. Phan and M. Kim, Invasive Software Testing: Mutating Target Programs to Diversify Test Exploration for High Test Coverage, ICST, 2018
[38] Y. Kim, Y. Choi, and M. Kim, Precise Concolic Unit Testing of C Programs Using Extended Units and Symbolic Alarm Filtering, ICSE, 2018

약 력

김 문 주

1995 KAIST 전산학부(학사)
2001 Univ. of Pennsylvania, Computer and Information Science (박사)
2002~2004 삼성 SECUi.COM 차장
2004~2006 POSTECH 연구원
2006~2011 KAIST 전산학부 조교수
2012~현재 KAIST 전산학부 부교수
관심분야 소프트웨어 공학, 정형검증 기술, 소프트웨어 테스팅, 소프트웨어 디버깅
Email: moonzoo.kim@gmail.com

김 윤 호

2007 KAIST 전산학부(학사)
2017 KAIST 전산학부(박사)
2017~2018 KAIST 정보전자연구소 연수연구원
2018~현재 KAIST 전산학부 연구조교수
관심분야: 소프트웨어 테스팅, 소프트웨어 디버깅, 임베디드 시스템
Email: yunho.kim03@gmail.com

홍 신

2007 KAIST 전산학부(학사)
2010 KAIST 전산학부(석사)
2015 KAIST 전산학부(박사)
2015 KAIST 정보전자연구소 연수연구원
2016~현재 한동대학교 전산전자공학부 조교수
관심분야: 소프트웨어 공학, 소프트웨어 테스팅, 소프트웨어 디버깅
Email: hongshin@handong.edu