(KSC2018 우수논문) 국방 무기 체계 SW 품질 향상을 위해 Concolic 테스팅을 통한 테스트 자동 생성

(Automatic Test Case Generation through Concolic Testing to Improve SW Quality of Defense Weapon System)

박 건 우 ‡ 이 주 현 § 송 형 곤 § 조 규 태 § 김 윤 호 ‡ 김 문 주 ¶
(Kunwoo Park) (Joohyun Lee) (Hyunggon Song) (Kyu Tae Cho) (Yunho Kim) (Moonzoo Kim)

요 약 국방 무기 체계 SW 품질 향상을 위해 노동집약적 수작업 SW 테스트 관행이 아닌, 테스트 입력을 자동으로 그리고 체계적으로 생성하는 것이 필요하다. 본 연구는 concolic 테스팅을 국방 무기 체계 SW에 적용해 높은 커버리지의 테스트 입력값을 효과적으로 생성하고, 결함을 발견하여 SW의 품질 향상에 기여하였다. 프로그램의 복잡성이 크고 전체 실행 경로가 많은 프로그램의 경우, concolic 테스팅의 효율을 높일 수 있는 방법 (4개의 탐색 전략, LIA 로직)을 제안하였다. 또한, 실무자들이 concolic 테스팅을 확장 적용할 수 있도록 심볼릭 모델링 방법을 예시로 제안하였다.

키워드 : Concolic 테스팅, 심볼릭 모델링, 테스트 자동 생성, 탐색 전략, 로직

Abstract To improve SW quality of defense weapon system, automatic and systematic generation of test cases is necessary; however, that is not the case in the traditional practice of labor-intensive and manual SW testing. The paper applies concolic testing to the defense weapon system SW, effectively generates test cases that achieve high coverage, and discovers defects which contributes to the improvement in SW quality. Also, two methods are proposed using 4 search strategies in concolic testing and using LIA logic, to increase the efficiency of concolic testing for a program with high complexity. In addition, a symbolic modeling method is proposed as an example to extend concolic testing for practitioners.

Key words: Concolic testing, Symbolic modeling, Automatic test case generation, Search strategy, Logic

† 이 논문은 LIG넥스원, 과학기술정보통신부의 재원으로 한국연구재단의 지원(NRF-2019R1A2B5B01069865), 과학기술정보통신부의 재원으로 한국연구재단-차세대 정보 컴퓨팅기술개발사업의 지원(NRF-2017M3C4A7068177), 교육부의 재원으로 한국연구재단의 지원(NRF-2017R1D1A1B03035851)을 받아 수행한 연구임

‡ 정 회 원 : KAIST 전산학부
kunwoo1209@kaist.ac.kr, yunho.kim@gmail.com
§ 비 회 원 : LIG Nex1
{joohyunlee, hyunggon.song, kyutae.cho}@lignex1.com
¶ 종신회원 : KAIST 전산학부 교수(KAIST)
moonzoo@cs.kaist.ac.kr

논문접수 : 2019년 3월 27일
심사완료 : 2019년 6월 28일

1. 서 론

국방 무기/비무기체계 (전력지원체계)의 첨단화가 이루어지는 현 상황에서 무기체계 SW 의 품질 관리 능력은 필수가 되고 있다[1][2]. 그러나 SW 테스팅 전문 인력의 부족 및 노동집약적 수작업 SW 테스트 관행은 국방 무기체계 SW 품질 향상에 걸림돌이 되고 있다.

테스팅의 목적은 SW 내부에 존재하는 결함을 발견하는 것이고, 이를 위해서는 SW 의 실행 가능한 경로를 가능한 한 많이 실행하는 테스트 케이스가 필요하다. 수작업으로 만든 테스트 케이스는 SW 의 복잡성이 클 경우 SW 의 가능한 모든 동작을 실행한다는 보장이 없기 때문에 SW 에 존재하는 결함을 발견하기 어렵다.

Concolic (CONCrete + symbOLIC) 테스팅[3][4]은 상기 문제를 해결하고 테스트 커버리지를 효과적으로 증가시키는 자동화 테스팅 기법이다. Concolic 테스팅이 자동으로 테스트 케이스를 생성하기 위해서 대상 프로그램의 입력값을 포함한 환경에 대한 심볼릭 모델 작성이 필요하다. 본 연구가 기여한 바는 다음과 같다.

1. Concolic 테스팅을 통해 수많은 효과적인 테스트 입력값을 생성하여, 고신뢰성이 있어야 하는 국방 무기 체계의 품질을 향상하였다. (평균 분기 커버리지 76.6% 달성 및 2 개의 크래시 버그 검출)
2. 프로그램의 복잡성이 크고 전체 실행 경로가 많은 프로그램의 경우, Concolic 테스팅의 효율을 높일 수 있는 방법 (4 개의 탐색 전략 사용, LIA 로직 사용)들을 제안하였다.
3. 국방 무기체계 미들웨어 도메인에 Concolic 테스팅을 확장 적용하는 실무자들이 참고할 수 있도록, 국방무기체계 미들웨어 도메인에 Concolic 테스팅을 적용할 때 필요한 심볼릭 모델링 방법을 예시로 제안하였다.
4. 국방 무기체계 미들웨어 도메인에 Concolic 테스팅의 성공적 적용을 위한 추가 논의 사항들을 정리하였다.

본 논문의 구성은 다음과 같다. 1 장의 서론에 이어 2 장에서는 Concolic 테스팅과 본 연구에 사용된 Concolic 테스팅 툴에 대하여 설명하고, 3 장에서는 타겟 프로그램과 프로그램을 테스트하기 위한 실험 설계에 대해 살펴본다. 4 장에서는 테스트를 자동으로 생성하는 Concolic 테스팅 기술의 효과를 보여주는 실험 결과를 보인다. 5 장에서는 Concolic 테스팅 적용 시 논의할 사항들을 정리하고, 마지막으로 6 장에서는 본 연구의 결론과 향후 연구의 방향에 대해 서술한다.

2. Concolic 테스팅 및 Concolic 테스팅 툴 소개

Concolic 테스팅은 dynamic concrete 분석과 static symbolic 분석을 결합하여 높은 커버리지를 갖는 테스트 입력값을 자동으로 생성하는 기법이다. Concolic 테스팅은 테스트 입력값을 심볼릭 변수로 설정해주면 주어진 프로그램의 실행 경로를 체계적으로 찾고 심볼릭 경로 수식을 생성해, 각 경로를 따라 실행할 수 있는 테스트 입력값을 SMT solver 로 자동으로 계산해서 만들어준다[5][6][7][8]. Concolic 테스팅은 모든 실행 가능한 경로를 찾아 테스트하기 때문에 수작업으로 입력값을 만드는 것보다 효과적으로 코드 커버리지를 올릴 수 있다.

CROWN[9]과 CREST[10]는 C 프로그램을 위한 Concolic 테스팅 도구로, SMT formula 를 푸는 로직에 차이가 있다. CREST 는 LIA(Linear Integer Arithmetic)[11] 로직을 사용하기 때문에 linear integer 연산만 지원한다. 반면 CROWN 은 CREST 을 확장하여 BV (Bit Vector) 로직[12]을 사용하기 때문에 linear integer 연산뿐만 아니라 bitwise 연산, floating point 연산, 비트 필드와 같은 복잡한 형태의 C 연산을 지원한다.

Concolic 테스팅은 대상 프로그램을 실행하면서 아직 탐색하지 않은 분기를 방문하는 것을 목표로 입력값을 생성한다. 새로운 입력값을 생성하기 위해 다음 iteration 에서 어떤 분기를 방문할지 결정하여야 그에 알맞은 입력값을 생성할 수 있는데, 어떤 분기를 방문할지 결정하는 전략을 탐색 전략이라고 한다. CROWN 과 CREST 에서는 dfs, rev-dfs, cfg, random, random_input 중 하나를 선택할 수 있다. dfs (깊이 우선 탐색, depth first search)는 새로운 심볼릭 경로 수식을 만들 때 뒤에 있는 조건부터 부정한다. 한다. rev-dfs 는 dfs 의 역순으로 앞에 있는 있는 조건부터 부정한다. random 은 새로운 심볼릭 경로 수식을 만들 때 부정할 조건을 무작위로 선택한다. cfg 는 프로그램의 CFG (Control Flow Graph)를 분석하여 커버되지 않은 분기를 먼저 커버하도록 새로운 입력값을 생성하는 휴리스틱이다. 마지막으로 random_input 은 random 테스팅처럼 심볼릭 변수에 무작위 값을 넣어 테스트를 수행한다.

3. Concolic 테스팅 기술 적용 연구

3.1 대상 프로그램 코드 정보

본 연구는 LIG 넥스원에서 개발한 미들웨어 프레임워크의 16 개 프로그램을 테스팅하였다. 표 1 은 각 타겟 프로그램에 대한 code 정보이다. 마지막 열은 기존 수작업 테스트 케이스로 달성한 분기 커버리지이다. 일반 프로그램과 달리 대상 SW 는 무기체계 SW 개발 및 관리 매뉴얼[16]에 명시된 소스코드 수준의 SW 신뢰성 시험을 수행한 상태이다.

표 1. LIG 넥스원 타겟 프로그램 코드 정보
Table 1. Code Statistics of Target Programs from LIG nex1

Target Program	#Func	LOC	#Branch	Achieved Branch Cov.
Prog1	2	29	40	35.0%
Prog2	15	184	133	10.5%
Prog3	28	149	46	32.6%
Prog4	7	123	64	64.1%
Prog5	32	588	390	54.9%
Prog6	21	347	260	68.7%
Prog7	15	271	110	40.4%
Prog8	28	424	284	42.7%
Prog9	20	294	203	4.9%
Prog10	9	159	81	8.5%
Prog11	4	77	80	25.0%
Prog12	10	136	50	36.0%
Prog13	51	726	581	60.2%
Prog14	18	309	165	6.1%
Prog15	26	395	151	53.5%
Prog16	26	461	316	43.7%
Average	**19.5**	**292**	**184.6**	**36.7%**

3.2 연구 문제 (Research Questions)

RQ1. CROWN 이 생성한 테스트 케이스가 같은 수의 random 테스팅으로 생성한 테스트 케이스보다 분기 커버리지를 얼마나 높이 달성했는가?

RQ2. CROWN 이 생성한 테스트 케이스가 같은 시간 동안 random 테스팅으로 생성한 테스트 케이스보다 분기 커버리지를 얼마나 더 높이 달성했는가?

RQ3. 4 개의 탐색 전략 (dfs, rev-dfs, cfg, random) 이 같은 수의 1 개의 탐색 전략 (dfs)으로 생성한 테스트 케이스보다 분기 커버리지를 얼마나 더 높이 달성했는가?

RQ4. LIA 로직을 사용하는 Concolic 테스팅 (CREST)이 BV 로직을 사용하는 Concolic 테스팅(CROWN)보다 동일 테스팅 시간에 분기 커버리지가 얼마나 차이 났는가?[1)]

RQ1 과 RQ2 는 Concolic 테스팅이 random 테스팅보다 얼마나 효과적으로 타겟 프로그램의 분기 커버리지를 높일 수 있는지 확인한다. RQ3 은 Concolic 테스팅에서 1 개(dfs)의 탐색 전략만 사용했을 때 제한된 테스트 케이스 수로 커버하지 못하는 분기를 4 개의 탐색 전략을 사용했을 때 커버할 수 있는지 확인한다. RQ4 는 Concolic 테스팅이 사용하는 로직이 다를 때 달성한 분기 커버리지가 얼마나 차이 나는지 확인한다. LIA 로직은 bitwise operator (예. <<) 및 *, / 를 지원하지 못해, 프로그램의 의미를 정확히 표현하지 못하는 단점 때문에, 생성한 테스트 케이스의 커버리지가 낮을 수 있다. 하지만, LIA 로직은 BV 로직보다 훨씬 간단하기 때문에, LIA 로직으로 표현된 심볼릭 경로 수식을 만들고 푸는 시간이 BV 로직보다 훨씬 빠르기에, 동일 시간에 BV 로직보다 더 많은 테스트 케이스를 자동 생성한다.

3.3 실험 설계

RQ1, RQ2: 각 타겟 프로그램에 대해 CROWN 을 사용해 생성된 테스트 케이스 수와 걸린 시간을 측정한 후, 같은 수의 테스트 케이스를 생성하도록 (RQ1) 혹은 같은 시간만큼 (RQ2) random 테스팅을 적용해 커버리지를 비교하였다. CROWN 이 생성할 최대 테스트 케이스 수를 10 만개로 설정하였고, Concolic 테스팅에 대해서 dfs 탐색 전략을, random 테스팅에 대해서 random_input 탐색 전략을 사용하였다. 동일한 입력 변수(즉, 동일한 심볼릭 변수)를 대상으로 concolic 테스팅과 random 테스팅을 진행하였고, random 테스팅의 경우 자체적으로 불확실성을 갖기 때문에 10 번 실험을 진행해 달성 커버리지를 평균 내었다.

RQ3: RQ1 에서 Concolic 테스팅으로 생성한 10 만 개 테스트 케이스로 전체 실행 경로를 탐색하지 못한 3 개 타겟 프로그램(Prog9, Prog12, Prog14)를 실험 대상으로 하였다. 생성한 테스트 케이스 수를 같게 하기 위해 4 개의 탐색 전략 (dfs, rev-dfs, cfg, random) 각각 2 만 5 천 개 테스트 케이스를 생성하도록 설정하였고, 커버리지를 측정해 RQ1 의 커버리지 결과와 비교하였다.

RQ4: RQ2 에서 각 타겟 프로그램 마다 측정한 테스트 케이스 생성 시간만큼 CREST 로 달성한 분기 커버리지를 RQ2 의 결과와 비교하였다. 심볼릭 변수 설정과 탐색 전략은 RQ2 와 동일하다. [2)]

---

1) CREST가 모든 경로를 커버한 함수의 경우, BV 로직보다 LIA 로직이 심볼릭 경로 수식을 생성하고 푸는 속도가 빠르기에, CROWN의 테스팅 시간보다 짧은 테스팅 시간이 소요되었다.

2) CREST가 지원하지 않는 longlong 타입 변수의 경우 integer 타입 심볼릭 변수로 선언해서 진행하였다.

실험은 Ubuntu Linux 16.04.5 LTS가 설치된 i5-4670K @3.40GHz의 CPU와 8GB RAM을 가진 서버에서 진행하였다.

3.4 심볼릭 변수 설정 예시

심볼릭 변수는 기본적으로 프로그램의 파라메터 변수, 전역 변수, 그리고 스텁의 리턴 값을 변수/리턴 값의 타입에 맞게 모델링하였다.

3.4.1 primitive 타입

Primitive 타입 변수는 CROWN이 제공하는 API 함수를 사용해서 심볼릭 설정한다. 예를 들어 SYM_int(x); 구문은 integer 타입 x를 심볼릭 변수로 설정하겠다는 의미이다.

3.4.2 구조체 타입

구조체 타입 안의 멤버 변수는 멤버 변수가 primitive인 경우 각각에 대해 심볼릭 설정한다. 멤버 변수가 구조체인 경우 그 구조체의 멤버 변수를 다시 한번 재귀적으로 심볼릭 선언한다.

3.4.3 enum 타입

enum 타입 변수는 정의한 값 이외의 값을 가질 경우 false alarm[13][14]을 일으킬 수 있기 때문에 되도록 정의한 값을 갖도록 심볼릭 설정을 하는 것이 중요하다. 그림 1은 enum 타입 변수인 level을 enum에서 정의한 값과 예외 처리를 위해 정의가 안된 값 중 하나를 가지도록 심볼릭 설정한 모습이다.

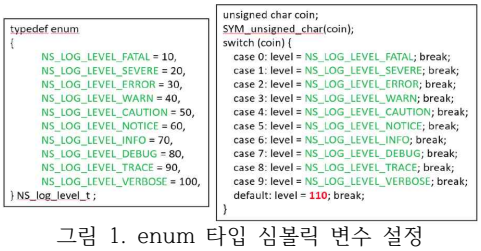

그림 1. enum 타입 심볼릭 변수 설정

Figure 1. Symbolic Modeling of enum-type variables

3.4.4 스텁의 리턴 값

대상 프로그램 코드뿐 아니라 스텁의 리턴값도 심볼릭 선언해야 한다. 스텁의 고정된 리턴 값이 분기 조건에 사용되는 경우 분기 조건의 값이 바뀌지 않기 때문에 부득이하게 탐색하지 못한 경로가 생긴다. 스텁의 리턴 값을 개발자가 설정한 가능한 값의 범위 내에서 심볼릭 설정을 해서 이 문제를 해결할 수 있다. 그림 2는 가능한 값이 2개인 스텁의 리턴 값을 심볼릭 설정한 모습이다.

```
ret_t function() {
  unsigned char coin;
  SYM_unsigned_char(coin);
  if (coin) return SUCCESS;
  else return FAIL;
}
```

그림 2. 스텁의 리턴 값 심볼릭 설정 예시

Figure 2. Symbolic Modeling of Stub Function's Return Value

4. 실험 결과

표 2는 각 타겟 프로그램에 대해 Concolic 테스팅, 같은 테스트 케이스 수의 random 테스팅, 그리고 같은 시간 동안의 random 테스팅 분기 커버리지 결과이다. 10만 개 미만 테스트 케이스로도 모든 경로가 커버된 경우, 테스트 케이스 생성을 중단하였다.

4.1 RQ1에 대한 결론

같은 테스트 케이스 수로 Concolic 테스팅이 random 테스팅보다 효과적으로 분기 커버리지를 높였다. Concolic 테스팅이 생성한 평균 25644.9개 테스트 케이스는 평균 76.6%의 분기를 달성하여, random 테스팅이 달성한 분기 커버리지보다 29.4%p 더 높이 달성했다. 또한, 16개 프로그램 모두 Concolic 테스팅이 최소 1.8%p 더 높은 분기 커버리지를 달성했다.

표 2. Concolic 테스팅(CROWN, BV 로직) 및 random 테스팅 달성 커버리지

Table 2. Branch Coverage Results achieved by Concolic Testing and Random Testing

Target Program	#TC	Time (s)	Concolic Branch Cov. (%)	Random Branch Cov. (Same # TC) (%)	Random Branch Cov. (Same Time) (%)
Prog1	6,144	111	100.0	77.5	77.5
Prog2	312	6	91.0	60.2	60.2
Prog3	27,841	366	87.0	2.2	2.2
Prog4	96	3	82.8	71.8	71.8
Prog5	43,974	2,361	82.3	55.4	55.4
Prog6	7	1	78.5	64.2	66.5
Prog7	35	16	78.2	50.0	50.0
Prog8	14	1	74.3	59.7	60.3
Prog9	100,000	4,900	73.1	4.9	4.9
Prog10	960	17	72.8	66.2	70.4
Prog11	20	1	72.5	32.5	45.0
Prog12	100,000	2,427	72.0	54.0	54.0
Prog13	2660	34	71.1	61.7	59.5
Prog14	100,000	4,637	70.3	28.5	28.5
Prog15	4	55	61.6	59.8	61.8
Prog16	28,252	2,833	57.6	7.3	7.3
Average	**25644.9**	**1110.6**	**76.6**	**47.2**	**48.5**

4.2 RQ2 에 대한 결론

같은 시간 동안 Concolic 테스팅이 random 테스팅보다 효과적으로 분기 커버리지를 높였다. Concolic 테스팅이 평균 1110.6 초 동안 생성한 테스트 케이스는 평균 76.4%의 분기를 달성하여, random 테스팅이 달성한 분기 커버리지보다 28.1%p 더 높이 달성했다. 또한, Prog15 타겟 프로그램을 제외한 15 개 프로그램 모두 Concolic 테스팅이 최소 1.4%p 더 높은 분기 커버리지를 달성했다.

4.3 커버하지 못한 분기

16 개 타겟 프로그램 중 Concolic 테스팅이 모든 경로를 탐색했지만 달성 분기 커버리지가 100%가 아닌 프로그램이 13 개이다. 결과적으로 실행 불가능한 dead 분기(예. 그림 3), 소스 코드 없는 외부 함수 사용으로 인해 심볼릭 변수들을 concrete 화해 커버하지 못한 분기(예. 그림 4), 예외 사항 처리 구문 (예. 메모리 소진)에 있는 분기(예. 그림 5), 그리고 assert 문을 violate 하지 않아 커버하지 못한 분기(예. 그림 6) 등이 존재했다.

```
1 SYM_unsigned_long(ctx->state);
...
2 ctx->state = 1;
3 ...
4 while (ctx->state) {... break; }
```

Cannot cover the case where the branch condition in line 4 becomes false because the symbolic variable ctx->state has a constant value during program execution.

그림 3. 결과적으로 실행 불가능한 dead 분기

Figure 3. Dead Branches

```
1 index = atoi(<symbolic string>)
2 if (0 > index) {...}
```

Cannot cover the case where last line of the branch condition is true when the symbolic string is not a numeric string (atoi function always return 0 in that case).

그림 4. 외부 함수 사용으로 인한 심볼릭 변수 concrete 화

Figure 4. External Binary Library Functions Concretizing Symbolic Variables

```
1 frw_node_t *n =(frw_node_t *)malloc(...);
2 ...
3 if (NULL == n) {/* 8 branches */}
```

Cannot cover 8 branches in the if body of line 3 because the branch checks for memory exhaustion of variable n which is dynamically allocated in line 1.

그림 5. 메모리 소진 처리 구문

Figure 5. Branches that handle Memory Exhaustion

```
1 assert(*p == '>');
```

Cannot create TC that satisfies the condition (*p != '>')

그림 6. assert 문을 violate 하지 않는 경우

Figure 6. The Case where Program Never Violates Assertion

4.3.1 분기 커버리지가 70% 이하인 프로그램

2 개의 타겟 프로그램 Prog15 와 Prog16 에 대해 Concolic 테스팅으로 달성한 분기 커버리지가 각각 61.6%와 57.6%이다. Prog15 의 커버리지가 낮게 나온 이유는 테스트 케이스를 1 개씩 만드는 iteration 마다 생성되는 심볼릭 경로 수식이 그림 7 과 같은 형태로 커져서 4 시간이 지나도 5 번째 iteration 의 수식을 생성하지 못하기 때문이다. Prog16 의 커버리지가 낮게 나온 이유는 타겟 프로그램의 코드가 설정한 심볼릭 변수들을 concrete 값들로 rewrite 함에 따라 심볼릭 경로를 생성하지 못하기 때문이다.

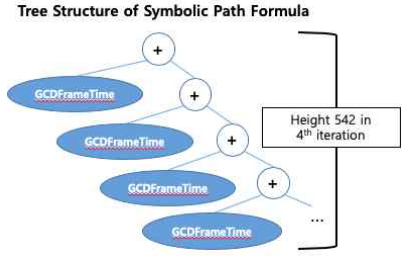

그림 7. 4 번째 iteration 에서 심볼릭 경로 수식의 형태

Figure 7. Format of Symbolic Path Formula in 4th Iteration

4.4 발견한 결함

Concolic 테스팅으로 Prog6 과 Prog9 에서 크래시 버그를 발견하였다. Random 테스팅으로 발견 못 했고 LIG 넥스원에서 그동안 발견 못 했던 새로운 결함이다.

4.4.1 Prog6 결함

Prog6 은 트리 자료 구조에 저장된 XML 파일의 파싱 데이터를 다시 읽는 프로그램이다. Concolic 테스팅이 달성하지 못한 분기를 분석하면서, XML 파일에 닫는 태그 없이 64 개 이상의 여는 태그(예. <a>만 있는 경우) 버퍼 오버플로우가 발생하여 크래시가 발생하는 버그를 검출하였다.

4.4.2 Prog9 결함

Prog9 는 입력값으로 받은 string 형태의 XML 파일을 파싱해서 트리 자료 구조에 저장하는 프로그램이다. Concolic 테스팅이 생성한 10 만 개의 테스트 케이스 중 9164 개가 모두 같은 line 에서 크래시 버그를 일으켰다. 결함 분석 결과, XML 파일에 여는 태그 없이 닫는 태그만 있는 경우(예.) NULL 포인터 역참조가 발생하여 크래시가 발생하는 버그를 검출하였다.

4.5 RQ3 에 대한 결론

표 3 은 Concolic 테스팅으로 생성한 10 만 개 테스트 케이스로 전체 실행 경로를 탐색하지 못한 3 개 타겟 프로그램(Prog9, Prog12, Prog14)에 대해 생성한 테스트 케이스 수, DFS 탐색 전략을 사용했을 때 걸린 시간과 달성한 분기 커버리지, 그리고 4 개의 탐색 전략을 사용했을 때 걸린 시간과 달성한 분기 커버리지 결과이다. 나머지 타겟 프로그램들은 DFS 로 모든 실행 경로를 탐색하였기에, 4 개 탐색 전략 사용한 경우와 비교를 생략했다.

DFS 탐색 전략과 비교 시, 동일하게 10 만 개 테스트 케이스를 생성하는 4 개 탐색 전략이 더 높은 분기 커버리지를 달성했다. 타겟 프로그램 3 개에 대해 4 개의 탐색 전략이 평균 78.0%의 분기를 달성해 DFS 탐색 전략이 달성한 분기보다 6.0%p 더 높이 달성했다. Prog12 의 경우 달성한 분기에 차이가 없지만 걸린 시간을 비교했을 때 4 개의 탐색 전략이 DFS 탐색 전략보다 더 적은 시간을 소모하여 효율적으로 분기 커버리지를 높였다. 또한 모든 타겟 프로그램에서 4 개전략을 사용한 테스팅이 DFS 한 전략을 사용한 것보다 빨랐다.

표 3. DFS 와 4 개 전략 비교

Table 3. Comparison between DFS and 4 search strategies

Target Program	#TC	DFS Time (s)	DFS Branch Cov. (%)	4 Strategy Time (s)	4 Strategy Branch Cov. (%)
Prog9	100,000	4,900	73.1	2,597	77.6
Prog12	100,000	2,427	72.0	1,924	72.0
Prog14	100,000	4,637	70.3	3,917	84.2
Average	**100,000**	**3,988**	**72.0**	**2,813**	**78.0**

4.6 RQ4 에 대한 결론

표 4 는 CROWN 이 모든 경로를 탐색하지 못한 3 개의 프로그램에 대한 CROWN(BV 로직)과 CREST(LIA 로직)의 실험 결과이다. 표 5 는 CROWN 이 모든 경로를 탐색한 프로그램에 대한 CROWN 과 CREST 의 실험 결과이다.

표 4. CROWN 이 모든 경로를 탐색하지는 못한 프로그램 3 개에 대한 BV 로직과 LIA 로직 비교

Table 4. Comparison of BV logic and LIA logic for 3 programs where CROWN did not execute all the paths

Target Program	BV Logic			LIA Logic		
	DFS #TC	DFS Time (s)	DFS Branch Cov. (%)	DFS #TC	DFS Time (s)	DFS Branch Cov. (%)
Prog9	100,000	4,900	73.1	2,583,433	4,900	74.1
Prog12	100,000	2,427	72.0	1	1	50.0
Prog14	100,000	4,637	70.3	42,526	71	24.5
Average	**100,000**	**3,988**	**72.0**	**875,320**	**1,657**	**50.0**

표 5. CROWN 이 모든 경로를 탐색한 프로그램 13 개에 대한 BV 로직과 LIA 로직 비교

Table 5. Comparison of BV logic and LIA logic for 13 programs where CROWN executed all the paths

Target Program	BV Logic			LIA Logic		
	DFS #TC	DFS Time (s)	DFS Branch Cov. (%)	DFS #TC	DFS Time (s)	DFS Branch Cov. (%)
Prog1	6,144	111	100.0	9	1	65.0
Prog2	312	6	91.0	62	4	90.2
Prog3	27,841	366	87.0	27,841	48	87.0
Prog4	96	3	82.8	96	1	82.8
Prog5	43,974	2,361	82.3	43,974	103	82.3
Prog6	7	1	78.5	7	1	78.5
Prog7	35	16	78.2	35	1	78.2
Prog8	14	1	74.3	14	1	74.3
Prog10	960	17	72.8	1	1	5.0
Prog11	20	1	72.5	1	1	3.8
Prog13	2660	34	71.1	2,660	4	71.1
Prog15	4	55	61.6	4	55	54.1
Prog16	28,252	2,833	57.6	28,252	93	57.6
Average	**8,486**	**447**	**78.0**	**7,920**	**24**	**64.0**

4.6.1 분기 커버리지 비교

CROWN 이 모든 경로를 탐색 못한 프로그램들의 경우 (표 4), BV 로직이 LIA 로직보다 평균적으로 22.0%p 더 높은 분기 커버리지를 달성했고, CROWN 이 모든 경로를 탐색한 프로그램들 (표 5) 역시 BV 로직이 평균적으로 14.0%p 더 높이 달성했다. 이는 LIA 로직이 bitwise 연산, floating point 연산, 비트 필드와 같은 복잡한 형태의 C 연산을 지원하지 않아 프로그램의 실행 경로가 제한적이기 때문이다. 가장 커버리지

차이가 많이 나는 Prog11 의 경우, 달성 분기 커버리지가 68.7% 더 낮았고, 1 개 테스트 케이스만 생성되었다. Prog11 프로그램의 모든 분기 조건이 그림 8 과 같이 LIA logic 이 지원하지 않는 bitwise 연산을 사용하기 때문이다.

```
CREST_unsigned_int(EventTypes);
...
if (EventTypes & EVENT_TYPE_UE0) {...}
if (EventTypes & EVENT_TYPE_UE1) {...}
if (EventTypes & EVENT_TYPE_UE2) {...}
...
if (EventTypes & EVENT_TYPE_MSG) {...}
```

그림 8. Prog11 프로그램 구조

Figure 8. Program Structure of Prog11

그러나 예외적으로 표 4 에서 xml 의 경우 LIA 로직이 BV 로직보다 달성 커버리지를 1.0% 더 높였다. 그 이유는 LIA 로직의 연산 속도가 BV 로직보다 빨라 동일 시간에 약 248 만 개 더 많은 테스트 케이스를 생성하였고, 추가 생성한 테스트 케이스가 BV 로직으로 커버하지 못한 분기를 커버하였기 때문이다.

4.6.2 테스트 케이스 생성 속도 비교

LIA 로직이 BV 로직보다 더 빠르게 분기 커버리지를 높였다. BV 로직과 LIA 로직의 달성 커버리지가 같은 프로그램 8 개(Prog3, Prog4, Prog5, Prog6, Prog7, Prog8, Prog13, Prog16; 평균 76.0% 분기 커버리지)에 대해 평균 12,860 개 테스트 케이스 생성 시간이 LIA 로직 (평균 32 초)이 BV 로직 (평균 702 초)보다 21.9 배 (=702 초/32 초) 더 빠르다.

5. 추가 논의 사항

국방 무기체계 미들웨어 도메인에 Concolic 테스팅을 적용하는 실험을 통해 발견한 내용은 다음과 같다.

5.1 수작업 심볼릭 변수 설정의 어려움

테스터가 소스 코드 상에서 직접 모델링할 변수의 정의와 타입을 찾아야 하는 번거로움이 있다. 특히, Prog7 의 경우, 구조체가 4 중 구조체로 정의돼 있어 (구조체 A 의 멤버 b 가 구조체 B 타입이고, B 가 구조체 멤버 c 를 갖고 등등) 14,416 개 변수에 대해 수작업으로 심볼릭 설정을 진행했다. 따라서, 수작업을 줄이기 위해 심볼릭 변수 설정을 자동화하는 연구가 진행된다면 이러한 수고를 줄일 수 있을 것이다.

5.2 경로의 폭발적 증가

경로의 폭발적 증가(path explosion)으로 인한 Concolic 테스팅의 확장성(scalability) 문제가 발생한다. 3 개 타겟 프로그램 (Prog9, Prog12, Prog14)의 경우 복잡한 loop 과 recursion 으로 인해 10 만 개 테스트 케이스로도 전체 실행 경로를 탐색하지 못하였다. 이러한 경로의 폭발적 증가를 해결하기 위한 연구가 필요하다.

5.3 국방 미들웨어 동적 할당 디버깅의 어려움

미들웨어 프레임워크 특성 상 프로그램 당 평균 7.3 번의 메모리 동적 할당이 사용되고 있어 메모리 누수나 의도치 않은 메모리 할당 실패로 인한 내부적인 결함이 발생할 수 있다[15]. 그러나 Concolic 테스팅은 구체적인 성능 관련 assert 문이 있지 않으면, 이러한 동적 성능 결함을 자동으로 발견하지 못한다.

5.4 LIA 로직과 BV 로직의 혼용 유용성

표 6 에서 보듯이, 16 개 중 8 개의 프로그램에서 LIA 로직이 BV 로직보다 더 빠르게 분기 커버리지를 달성했다 (즉, BV 로직과 LIA 로직의 달성 커버리지가 같을 때 LIA 로직이 BV 로직보다 21.9 배 더 빠르다). 따라서, 테스팅 시작단계에서는 LIA 로직으로 Concolic 테스팅을 빠르게 수행하고, 커버리지가 미흡한 프로그램에 대해서만 BV 로직을 적용하면 테스팅 효율을 향상시킬 수 있다.

6. 결론 및 향후 연구

Concolic 테스팅을 LIG 넥스원에서 개발한 국방 무기 체계 미들웨어 도메인에 적용해 높은 커버리지를 달성하고 결함을 발견하여 SW 의 품질을 향상시켰다. 또한, 실무자들이 확장 적용할 수 있게 심볼릭 모델링 방법을 예시로 제안하였다. 5 장에서 추가 논의 사항을 정리했기에 이들을 해결하고 확장하는 노력이 필요하다.

참 고 문 헌

[1] P. Carnes, "Software reliability in weapon systems." *Proceedings The Eighth International Symposium on Software Reliability Engineering-Case Studies-*, pp. 95-100, 1997.

[2] H. G. Stuebing, "A Software Engineering Environment (SEE) for Weapon System Software," *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 4, pp. 384-397, 1984

[3] R. Kannavara, C. J. Havlicek, B. Chen, M. R. Tuttle, K. Cong, S. Ray, F. Xie, "Challenges and opportunities with concolic testing." *2015 National Aerospace and Electronics Conference (NAECON)*, pp. 374-374, 2015.

[4] C. Cadar, D. Dunbar, D. R. Engler, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs." *8th USENIX Symposium on Operating*

Systems Design and Implementation, pp. 209-224, 2008.
[5] M. Kim, Y. Kim, Y. Jang, "Industrial application of concolic testing on embedded software: Case studies." *IEEE Fifth International Conference on Software Testing, Verification and Validation*, pp. 390-399, 2012.
[6] Y. Kim, M. Kim, Y. J. Kim, Y. Jang, "Industrial application of concolic testing approach: A case study on libexif by using CREST-BV and KLEE." *34th International Conference on Software Engineering*, pp. 1143-1152, 2012.
[7] M. Kim, Y. Kim, G. Rothermel, "A scalable distributed concolic testing approach: An empirical evaluation." *IEEE Fifth International Conference on Software Testing, Verification and Validation*, pp.340-349, 2012.
[8] Y. Kim, Y. Kim, T. Kim, G. Lee, Y. Jang, M. Kim, "Automated unit testing of large industrial embedded software using concolic testing." *28th IEEE/ACM International Conference on Automated Software Engineering*, pp. 519-528, 2013.
[9] Y. Kim, CROWN [Online]. Available: https://github.com/swtv-kaist/CROWN
[10] J. Burnim, CREST [Online]. Available: https://github.com/jburnim/crest
[11] B. Dutertre, L. De Moura, "A fast linear-arithmetic solver for DPLL(T)." *International Conference on Computer Aided Verification*. pp. 81-94, 2006.
[12] C. W. Barrett, D. L. Dill, J. R. Levitt, "A decision procedure for bit-vector arithmetic." *Proceedings 1998 Design and Automation Conference. 35th DAC*. pp. 522-527, 1998.
[13] P. Godefroid, N. Klarlund, K. Sen, "DART: directed automated random testing." *ACM Sigplan Notices*, Vol. 40, No. 6, pp. 213-223, 2005
[14] Y. Kim, Y. Choi, M. Kim, "Precise concolic unit testing of C programs using extended units and symbolic alarm filtering." *40th International Conference on Software Engineering*, pp. 315-326, 2018.
[15] L. Szekeres, "Sok: Eternal war in memory." *IEEE Symposium on Security and Privacy*, pp. 48-62, 2013.
[16] Defense Acquisition Program Administration, "Manual of Software Development and Management in Weapon System Software Development," *DAPA Manual 2016-4*, pp. 100, 2016. (in Korean)

Kunwoo Park
2018년 연세대학교 컴퓨터과학과 졸업(학사). 2018년 ~ 현재 KAIST 전산학부 석사 과정. 관심 분야는 자동화된 Concolic 유닛 테스팅

Joohyun Lee
2007년 한양대학교 전자전기컴퓨터학부 졸업(학부). 2007년 ~ 현재 LIG넥스원 SW연구소 수석연구원. 2017년 고려대학교 소프트웨어공학과 졸업(석사). 관심 분야는 소프트웨어 테스팅, 소프트웨어 검증, 머신러닝

Hyunggon Song
2009 QUT(Queensland University of Technology) Software Architecture 학사. 2011 ~ 현재 LIG넥스원 SW연구소 선임연구원. 관심 분야는 소프트웨어 설계, 소프트웨어 테스팅

Kyu Tae Cho
2002 숭실대학교 컴퓨터학부 학사. 2004 한국과학기술원 전산학과 석사. 2007 한국과학기술원 전산학과 박사수료. 2007 ~ 현재 LIG넥스원 SW연구소 수석연구원. 관심 분야는 소프트웨어 설계, 머신러닝

Yunho Kim
2007년 KAIST 전산학과 학사. 2007년 ~ 2009년 KAIST 전산학과 석사. 2009년 ~ 2017년 KAIST 전산학부 박사. 관심분야는 자동화된 Concolic 유닛 테스팅, 변이 테스팅, 자동 오류 위치 추정, 정형검증

Moonzoo Kim
1995년 KAIST 전산학과 학사. 2001년 Univ. of Pennsylvania 박사. 2002년 ~ 2004년 SECUi.COM 차장. 2004년 ~ 2006년 POSTECH 연구원. 2006년 ~ 2012년 KAIST 전산학과 조교수. 2012년 ~ 현재 KAIST 전산학부 부교수. 관심분야는 Concolic 테스팅, 자동 오류 위치 추정, Concurrency 테스팅, 변이 테스팅, 정형 검증, 내장형 소프트웨어