

프로그램 합성 기법을 이용한 소프트웨어 검증용 스텝 자동 생성 연구

컴퓨터학부 석사과정 김요엘

지도교수: 최윤자 교수님

개요

외부 라이브러리 모듈 (데이터베이스, 파일 시스템, 하드웨어 등)과 상호작용하는 소프트웨어의 검증:

- 외부 모듈을 직접 가동하면서 테스트 → 검증 비용 증가
- 따라서, 외부 모듈을 단순하게 시뮬레이션 하는 스텝 (Stub) / 목 객체 (Mock object)을 사용

스텝/목 객체 자동 생성 프레임워크의 현주소:

- 프레임워크 (예: Mockito*)는 다수 존재. 그러나, 뼈대만 제공하고 내부 코드는 사용자가 직접 작성
- Alshahwan 외 연구**에서 내부 코드도 자동 생성하는 연구 진행. 그러나, 몇몇 입력 케이스에 대해서만 처리할 수 있는 등, 스텝의 재사용이 거의 불가능

→ PBE (Programming-By-Example) 프로그램 합성 기법을 통해, C 프로그램을 대상으로 정확하게 재사용 가능한 스텝을 자동 생성하는 방법 – 입출력 예제 수집 및 합성 방법을 제안

*<https://site.mockito.org>

**[Automock: Automated synthesis of a mock environment for test case generation](#)

연구 배경: 검증 환경의 구성

오른쪽 위 그림:

- 임베디드 소프트웨어에서 전형적인 하드웨어와 상호작용하는 코드 → 외부 라이브러리 함수인 `nxt_motor_get_count`을 호출
- `turnSensor` 함수를 검증하기 위해선, 관련 하드웨어 환경도 모두 갖춘 후에만 가능 → 검증 비용 증가

오른쪽 아래 그림: 고정된 행동을 수행하는 스텝 코드의 예시

- 실제 하드웨어를 수행하지 않음 → 검증 비용 감소
- 그러나, 다른 목적의 테스트를 수행하기 위해선 변경된 목적에 맞게 스텝을 수작업으로 재작성해야 함

```
1 #include <math.h>
2 #include "ecrobot_interface.h"
3 void turnSensor(int edge){
4     int offset = 10;
5     int s_speed, deg;
6     while(fabs(offset) > 3){
7         deg = nxt_motor_get_count(MOTOR_S);
8         offset = edge - deg;
9         if(offset > 0){
10             s_speed = BASE_SPEED_S + SENSOR_P*offset;
11         }
12         else {
13             s_speed = -BASE_SPEED_S + SENSOR_P*offset;
14         }
15         s_speed = (s_speed > 100) ? 100 : s_speed;
16         s_speed = (s_speed < -100) ? -100 : s_speed;
17         ecrobot_set_motor_speed(MOTOR_S, s_speed);
18     }
19     ecrobot_set_motor_speed(MOTOR_S, 0);
20 }

1 int nxt_motor_get_count_stub(int fake_param){
2     return 0;
3 }
```

정형 검증 시 검증 환경의 구성

소프트웨어 모델 체킹 (Software Model Checking) 등 소스 코드 수준의 정형 검증 기법 적용 시:

- 외부 라이브러리 함수의 소스 코드가 미공개 상태일 경우, 일반적으로 오른쪽 그림과 같이 처리

```
1 int nxt_motor_get_count_stub(int fake_param){  
2     int ret;  
3     SYM_int(ret); // or, ret = nondet();  
4     return ret;  
5 }
```

- 소프트웨어 모델 체킹: 검증 대상 코드를 논리식으로 변환하고, 모든 가능한 상태를 탐색하여 주어진 명세를 만족하는지 자동으로 증명하는 검증 기법
- 명세를 위반하는 경우가 존재할 경우, 반례 실행 트레이스를 반환
- 위 스텝 코드는 미공개된 외부 라이브러리 코드와 실제 하드웨어의 동작을 전혀 반영하지 못함
→ 실제 하드웨어를 포함하여 검증한 경우에는 재현이 불가능한 트레이스 (오탐)일 가능성이 매우 큼

PBE기반 프로그램 합성 기법

- 프로그램 합성 기법: 사용자의 의도 (제약 조건)을 만족하는 프로그램을 찾아내는 기법
- **PBE기반 합성 기법**: 대상 프로그램의 입출력 예제를 제약 조건으로 받아 들이는 합성 기법

PBE 합성 기법을 통해...

1. 소스 코드가 공개되지 않더라도 입출력 예제만 있으면 스텝을 자동 생성할 수 있고,
2. 다양한 입력에 대해 비교적 정확한 처리가 가능한, 즉 재사용 가능한 스텝을 생성할 수 있음

입출력 예제 (제약 조건)

$$f(0) = 3$$

$$f(1) = 5$$

$$f(2) = 7$$



PBE
합성도구



합성된 프로그램

$$f(x) = 2x + 3$$

SyGuS (Syntax-Guided Synthesis) 스펙

- SyGuS: 의미적 제약 뿐만 아니라, 합성될 프로그램이 사용할 수 있는 파라미터, 연산자, 상수값을 제한할 수 있는 문법 제약도 포함

→ 프로그램 탐색 공간의 크기를 효율적으로 줄일 수 있음

- 함수형 언어와 유사한 형태. ‘ $3 + 1 - 2$ ’는 ‘ $(+ 3 (- 1 2))$ ’로 표현
- 오른쪽 SyGuS 스펙을 합성 도구에 전달하면
‘ $(ite (< b a) b (ite (= b (+ a a)) a (/ b (+ 3 2))))$ ’인 프로그램을 합성

```
1 (constraint (= (gcd_test 12 30) 6))
2 (constraint (= (gcd_test 60 5) 5))
3 (constraint (= (gcd_test 20 100) 20))
4 (constraint (= (gcd_test 16 32) 16))
```

```
1 (set-logic LIA)
2 (synth-fun gcd_test ( (a Int) (b Int)) Int
3 ((Start Int (ntInt))
4 (ntInt Int (
5   -1 0 1 2 3
6   a b
7   (+ ntInt ntInt)
8   (- ntInt ntInt)
9   (* ntInt ntInt)
10  (/ ntInt ntInt)
11  (ite ntBool ntInt ntInt)
12 ))
13 (ntBool Bool (
14   true false
15   (not ntBool)
16   (and ntBool ntBool)
17   (or ntBool ntBool)
18   (< ntInt ntInt)
19   (= ntInt ntInt)
20 ))))
```

PBE 합성 기법의 종류

- 나열식 (Enumerative) 합성 기법: 생성할 수 있는 가장 작은 프로그램부터 나열하여, 후보 프로그램의 크기를 조금씩 늘려가며 모든 제약 조건을 만족하는 프로그램을 찾는 bottom-up 방식

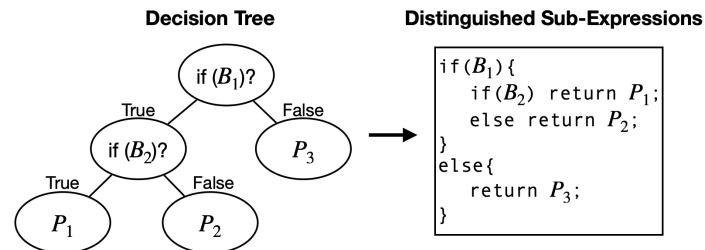
→ 사이즈가 작은 프로그램은 매우 잘 찾으나, 확장성 (scalability) 문제가 존재

- **EuSolver***: 여러 후보 서브 프로그램들이 각각 만족하는 제약 조건들의 합집합이 전체 집합이 되는 경우를 빠르게 찾고, 이를 구분해주는 조건식들을 사용해 적절히 결합하는 방식

- **DUET****: 작은 프로그램 부품(컴포넌트)을 만들어내고, 전체의 큰 문제를 작은 문제들로 나눠 작은 문제를 해결하는 컴포넌트들을 서로 결합하여 최종적으로 모든 제약 조건을 만족하는 프로그램을 찾아내는 방식

→ SyGuS-Comp 2019 에서 우승을 차지했던 CVC4 및 EuSolver보다 벤치마크 문제들을 더 빠르고 정확하게 풀어내어 다른 PBE기반 합성 기법들보다 우위에 있음이 입증.

따라서, 본 연구에서는 스텝 합성을 위해 DUET을 사용



[*Scaling enumerative program synthesis via divide and conquer](#)

[**Combining Top-Down Propagation and Bottom-Up Enumeration for Inductive Program Synthesis](#)

PBE 합성 기법의 과적합 문제

- PBE 합성 기법은 주어진 제약 조건에는 100% 만족하나, 다른 입력에 대해서는 정확함을 보장할 수 없다는 한계가 명확
- 특히 이러한 문제가 심화될 경우, 주어진 제약 조건에만 맞추게 되는 과적합 (**over-fitting**) 문제가 발생
- 일반적인 해결 방법: 주어지는 제약 조건의 수를 증가시키기
 - 그러나, 아무렇게 제약 조건을 추가한다고 해서 정확도가 항상 증가하는 것이 아님 (오히려, 복잡도 및 합성 시간만 증가)
 - 즉, 최소한의 제약 조건 수를 유지하면서 최적의 결과 (높은 정확도, 넓은 출력 범위 등)를 달성할 수 있는 제약 조건들의 조합을 찾아내야 함
- 그러나, 대부분의 연구는 합성 기법 자체의 알고리즘만 관심을 둘 뿐, 사용자 입장에서 합성 도구를 다루는 연구는 거의 존재하지 않음
 - 따라서, 본 연구에서는 최적의 결과를 찾아낼 수 있도록 하는 제약 조건들의 선택 방식을 제안

입출력 예제 수집을 위한 TC 생성 도구

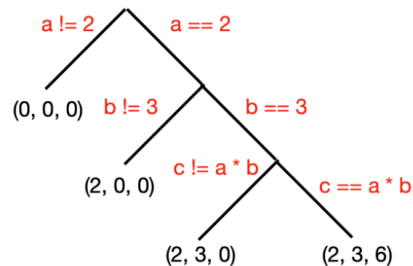
AFL++^{*}:

- 그레이박스 퍼징 (Fuzzing) 도구로, 정상적으로 수행되는 입력값 집합인 시드(seed)를 뮤테이션하여 새로운 입력값을 생성
- 분기 커버리지 정보를 뮤테이션에 이용하며, 서로 다른 실행 경로를 수행하는 TC 집합을 반환
- 바이너리 코드 수준에서도 그레이박스 퍼징 적용 가능 (즉, 소스 코드가 없는 외부 라이브러리도 적용 가능)

CROWN^{**}:

- 콘콜릭 (Concolic) 테스트 도구로, 프로그램의 모든 실행 경로를 각각 도달할 수 있는 TCs을 생성하는 기법
- 프로그램을 논리식으로 변환하여, 특정 실행 경로에 도달할 수 있는 논리식의 해 (입력값)를 SMT (Satisfiability Modulo Theory) solver을 통해 계산

```
1 void f(int a, int b, int c){  
2   if (a == 2) {  
3     if (b == 3) {  
4       if (c == a * b) {  
5         Assert(0);  
6       }  
8     }  
9   }  
10 }
```



^{*}[AFL++: Combining Incremental Steps of Fuzzing Research](#)

^{**}[CROWN: Concolic Testing for Real-wOrld softWare aNalysis](#)

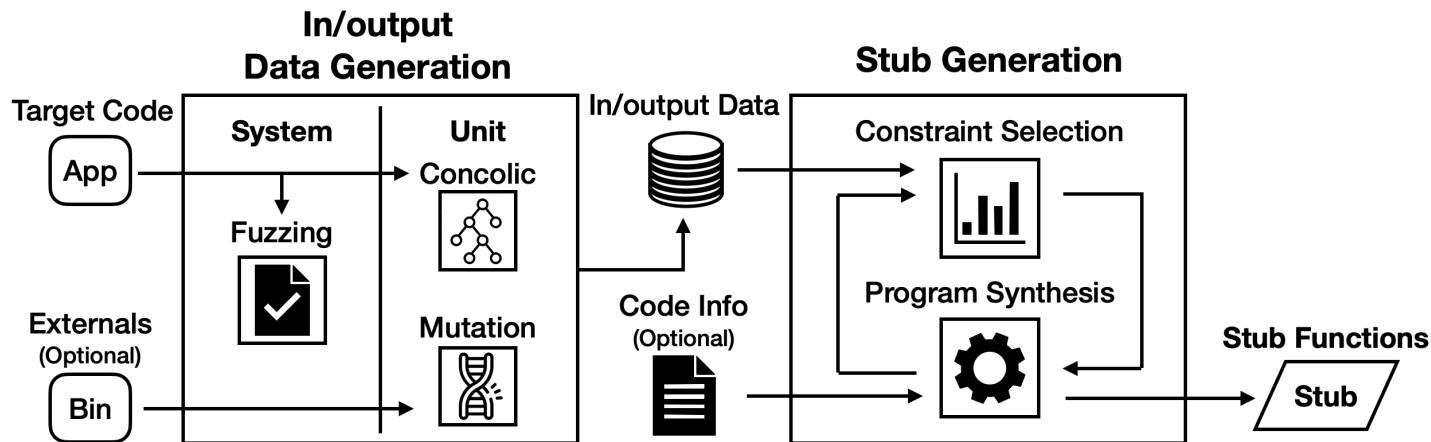
관련 연구: 스텝 자동 생성 방식에 대해

- **capture-replay**: 실제로 수행된 입출력 값들을 저장해 두고, 이를 그대로 스텝 코드로 사용하는 방식 (old)
- **Automock***: 심볼릭 실행 (Symbolic Execution)을 이용하여, 단위 테스트 코드의 코드 커버리지를 최대화하는 입출력값들을 스텝 코드로 사용하는 방식을 제안
 - capture-replay 방식보다 더 다양한 테스트를 수행할 수 있으나, 여전히 몇몇 입력에만 처리할 수 있는 등, 미탐 (false positive) 비율이 높고 재사용이 거의 불가능한 스텝을 생성
- **MOKA****: 최초로 프로그램 합성 기법을 이용해, 모바일 앱을 대상으로 다양한 입력에 대해 처리가능한 (재사용 가능한) 스텝을 자동 생성하는 방식을 제안
 - 그러나, 아이디어 수준의 연구로서 구체적인 스텝 생성 결과에 대해서는 아직 발표하지 않음

[*Automock: Automated synthesis of a mock environment for test case generation](#)

[**A Framework for Automated Test Mocking of Mobile Apps](#)

입출력 예제 수집 및 합성 프로세스 개요



입출력 예제 수집 단계:

- 시스템 퍼징 + (소스 코드 존재 시) 단위 콘콜릭 테스트 + 입력값 뮤테이션
- 입출력 예제 집합의 높은 분기 커버리지 + 각 분기를 커버하는 충분한 예제 수를 모두 충족하도록

스텝 합성 단계:

- (소스 코드 존재 시) 맞춤 문법 제약 생성 + 매 합성된 스텝의 정확도, 출력/분기 커버리지 측정 후 제약 조건 선택
- 최대한의 출력/분기 커버리지 달성 + 그 중에서 정확도가 가장 높은 스텝을 최종 선택

확장된 분기 커버리지 (소스 코드 존재 가정)

- 재사용 가능한 스텝은 다양한 입력에 대해 비교적 정확한 처리가 가능해야 함
- 즉, 다양한 실행 경로, 다양한 분기를 커버한 입력들에 대해 처리할 수 있어야 함
- 그러므로, 수집되는 입출력 예제 집합의 분기 커버리지가 높아야 함
- 그러나, 분기 커버리지를 기준으로 삼을 경우, 특정 분기를 커버하는 예제가 1개만 존재해도 커버리지가 증가 → 각 분기를 커버하는 예제 수가 충분히 존재해야 합성 및 평가가 원활하게 이루어질 수 있음
- 따라서, 이러한 고찰을 반영하는 평가 지표로서 확장된 분기 커버리지를 정의:

$$(|B_{full}(X)| \cdot 2 + |B_{scarse}(X)| \cdot 1 + |B_{not}(X)| \cdot 0) / |B| \cdot 2$$

$$B_{full}(X) = \{b \in B \mid C(b) \geq \tau\},$$

$$B_{scarse}(X) = \{b \in B \mid C(b) < \tau \text{ and } C(b) > 0\},$$

$$B_{not}(X) = \{b \in B \mid C(b) = 0\},$$

$$C(b) = \sum_{(x, f(x)) \in X} \mathbb{I}[b \in B(x)]$$

X : 입출력 예제 집합,
 B : 전체 분기 집합,
 τ : 임의의 목표 개수,
 $C(b)$: X 에서 b 를 커버하는 예제 수

시스템/단위 수준에서의 입출력 예제 생성

	시스템 수준 그레이박스 퍼징	유닛 수준 콘콜릭 테스트
장점	• 다양한 함수들의 입출력 예제를 빠르게, 동시에 수집 가능 • AFL++의 경우, 소스 코드가 존재하지 않아도 바이너리 수준에서 적용 가능	• 다양한 실행 경로를 도달한 입출력 예제를 거의 확정적으로 수집 가능 • 시스템 수준 퍼징에서 호출하지 못했던 함수를 따로 수집 가능
단점	• 분기 조건의 복잡함에 따라 도달하지 못한 함수가 존재할 수 있음 • AFL++의 경우, 서로 다른 실행 경로를 가진 실행 트race만을 제공하는 점이 오히려 충분한 수의 예제를 수집하지 못할 수 있음	• 소스 코드가 존재해야 함 • 애초에 실행 경로 종류가 적다면, 충분한 수의 예제를 수집하지 못함

- 시스템 수준 퍼징과 단위 수준 콘콜릭 테스트는 서로의 장점과 단점을 상호보완하는 관계 → 해당 두 기법을 모두 사용하는 것이 적절
- 단, 공통적으로 상황에 따라 충분한 수의 예제를 수집하지 못할 수 있다는 단점이 존재 → 두 방법으로 수집한 입력을 기준으로 따로 뮤테이션하여 부족한 입출력 예제를 추가적으로 생성

합성 프로세스: 소스 코드 미공개 시

f : 대상 함수, X : 입출력 예제 집합,
 t_{entire} : 전체 제한 시간,
 t_{unit} : 매 합성 제한 시간

```

1: procedure GETNEWCONSTRAINT    $W$ : 틀린 입출력 예제 집합,
2:    $T \leftarrow \emptyset$             $O_{not}$ : 커버하지 못한 출력값 집합
3:   for each  $(x, f(x)) \in W$  do
4:     if  $f(x) \in O_{not}$  then  $T \leftarrow T \cup \{(x, f(x))\}$ 
5:   if  $T = \emptyset$  then return  $RandomSelection(W)$ 
6:   return  $RandomSelection(T)$ 
    
```

- 라인 10: 고정된 문법 사용
- 라인 16: 합성된 스텝 f_{stub} 의 유효한 출력 범위 $O_{covered}$ 을 통해 출력 커버리지 cov 계산
- 라인 19-21: cov 을 더 높이거나, 같은 커버리지라도 더 정확한 경우 ($acc > acc_{best}$) f_{best} 을 업데이트
- 라인 1-6, 24: 틀린 예제 집합 W 중에서, 아직까지 커버하지 못한 출력값 집합 $O - O_{covered}$ 을 가진 예제 중 임의로 하나를 선택하여 C 에 추가
- 라인 22-23: cov 가 최대치이나, 정확도가 더 떨어진 경우 제약 조건 집합 C 에서 임의로 하나의 제약 조건을 제외

```

7: procedure ITERATIVESYNTHESIS
8:    $C \leftarrow RandomSelection(X)$  // initial constraint set.
9:    $f_{best} \leftarrow \emptyset, acc_{best} \leftarrow 0.0, cov_{best} \leftarrow 0.0, O \leftarrow GetOutputRangeFrom(X)$ 
10:   $op_{basic} \leftarrow \{+, -, *, /, ite, <, >=, =, and, or, not\}, const_{basic} \leftarrow \{-1, 0, 1, 2, 3\}$ 
11:  while  $t_{elapsed} < t_{entire}$  do
12:     $f_{spec} \leftarrow GenerateSpec(op_{basic}, const_{basic}, C)$ 
13:     $f_{stub} \leftarrow Synthesize(f_{spec}, t_{unit})$ 
14:    if  $f_{stub} \neq \emptyset$  then
15:       $acc, W, O_{covered} \leftarrow Evaluate(f_{stub}, X)$ 
16:       $cov \leftarrow |O_{covered}|/|O|$ 
17:      if  $W = \emptyset$  then
18:        return  $f_{stub}$ 
19:      if  $cov > cov_{best}$  or ( $cov = cov_{best}$  and  $acc > acc_{best}$ ) then
20:         $acc_{best} \leftarrow acc, cov_{best} \leftarrow cov$ 
21:         $f_{best} \leftarrow f_{stub}$ 
22:      else if  $cov = 1.0$  then
23:         $C \leftarrow C - RandomSelection(C)$ 
24:         $C \leftarrow C \cup GetNewConstraint(W, O - O_{covered})$ 
25:      else
26:         $C \leftarrow C - RandomSelection(C)$ 
27:      if  $C = \emptyset$  then
28:         $C \leftarrow C \cup RandomSelection(X)$ 
29:  return  $f_{best}$ 
    
```

원본 함수의 검증 복잡도 예측

- 함수 f 의 복잡도 $cmplx(f)$ 는 다음과 같이 정의:

$$cmplx(f) = unit(f) + \sum_{C_k^g \in down(f)} cmplx(g)$$

- $unit(f)$: 함수 f 내부의 조건 (branch), 할당 (assignment), 함수 호출 (call), 반환 (return) 문장들의 총 합
- $down(f)$: f 가 호출하는 함수 호출들의 집합. C_k^g 는 f 에서 호출되는 함수 g 의 k 번째 호출

검증 복잡도 예측의 필요성:

- 소스 코드가 존재하는 함수의 경우, 굳이 스텝을 생성해서 대체할 이유가 없음
- 단, 향후 연구로 원본 함수보다 덜 복잡하면서도 비슷한 행동을 수행할 수 있는 스텝을 합성 및 대체한 후 정형 검증 기법을 적용하는, 추상화 기법을 고려할 수 있음
- 이를 사전에 평가하기 위해, 본 실험에서는 소스 코드가 존재할 경우에 원본 함수의 복잡도를 넘지 않는 스텝을 생성하도록 하는 제약을 추가하였음

합성 프로세스: 소스 코드 존재 시

f : 대상 함수, X : 입출력 예제 집합,
 t_{entire} : 전체 제한 시간,
 t_{unit} : 매 합성 제한 시간

1: **procedure** GETNEWCONSTRAINT

2: $T \leftarrow \emptyset$

3: **for each** $(x, f(x)) \in W$ **do**

4: **if** $f(x) \in O_{not}$ and $B(x) \cap B_{scarse} \neq \emptyset$ **then** $T \leftarrow T \cup \{(x, f(x))\}$

5: **if** $T = \emptyset$ **then return** $RandomSelection(W)$

6: **return** $RandomSelection(T)$

W : 틀린 입출력 예제 집합,

O_{not} : 커버하지 못한 출력값 집합,

B_{scarse} : 부족하게 커버된 분기 집합

- 라인 12: 함수 f 가 사용하는 연산자 및 상수값을 문법 스펙에 사용
- 라인 14: 합성된 스텝 스텝 f_{stub} 의 복잡도가 원본 함수의 복잡도를 넘지 않도록 함
- 라인 17-20: 출력 커버리지를 50%, 제약 조건 집합에 대한 확장된 분기 커버리지를 50% 반영
- 라인 4, 28: 부족하게 커버된 분기를 가진 예제를 선택하도록 하는 제약이 추가

7: **procedure** ITERATIVESYNTHESIS

8: $C \leftarrow RandomSelection(X)$ // initial constraint set.

9: $f_{best} \leftarrow \emptyset, acc_{best} \leftarrow 0.0, cov_{best} \leftarrow 0.0, O \leftarrow GetOutputRangeFrom(X)$

10: $B \leftarrow GetToTalBranchesFrom(f)$

11: **while** $t_{elapsed} < t_{entire}$ **do**

12: $f_{spec} \leftarrow GenerateSpec(Op(f), Const(f), C)$

13: $f_{stub} \leftarrow Synthesize(f_{spec}, t_{unit})$

14: **if** $f_{stub} \neq \emptyset$ and $cmplx(f_{stub}) < cmplx(f)$ **then**

15: $acc, W, O_{covered} \leftarrow Evaluate(f_{stub}, X)$

16: $cov_O \leftarrow |O_{covered}|/|O|, cov_B \leftarrow 1.0$

17: **if** $|B| > 0$ **then**

18: $B_{full}(C), B_{scarse}(C) \leftarrow DivideBranches(C, \tau)$

19: $cov_B \leftarrow (|B_{full}(C)| \cdot 2 + |B_{scarse}(C)|)/|B| \cdot 2$

20: $cov \leftarrow cov_B \cdot 0.5 + cov_O \cdot 0.5$

21: **if** $W = \emptyset$ **then**

22: **return** f_{stub}

23: **if** $cov > cov_{best}$ or $(cov = cov_{best}$ and $acc > acc_{best})$ **then**

24: $acc_{best} \leftarrow acc, cov_{best} \leftarrow cov$

25: $f_{best} \leftarrow f_{stub}$

26: **else if** $cov = 1.0$ **then**

27: $C \leftarrow C - RandomSelection(C)$

28: $C \leftarrow C \cup GetNewConstraint(W, O - O_{covered}, B_{scarse}(C))$

29: **else**

30: $C \leftarrow C - RandomSelection(C)$

31: **if** $C = \emptyset$ **then** $C \leftarrow C \cup RandomSelection(X)$

32: **return** f_{best}

실험 예제 및 합성 대상 함수 선정

1. Lego Mind Storms NXT에서 수행되는 임베디드 로봇 자동차인 Object Follower (TS1),
 2. 임베디드 시스템인 엘리베이터 컨트롤러의 시뮬레이션 프로그램 (TS2),
 3. SV-COMP 2022의 벤치마크 일부 (TS3).
- TS1은 측정된 물체와의 거리를 가지고 여러 전처리 과정을 수행하는 함수를,
 - TS2는 엘리베이터의 버튼이 눌렸는지 확인하는 함수, 몇 층으로 가는 버튼이 눌렸는지 확인하는 함수를,
 - TS3는 다양한 종류의 벤치마크에서 정의된 다양한 함수들을 선정하였음
-
- 각 실험 예제별로 후보 함수의 복잡도 상위 5개까지 합성 대상 함수로 선정
 - 단, 함수 내부에서 동적할당, 부동 소수점 (floating point)의 사용, 여러 개의 전역 변수가 동시에 사용 및 할당, scanf 등 외부 입력 함수의 사용이 존재할 경우 여러 이유로 합성 대상에서 제외
 - 모든 대상 함수는 소스 코드가 모두 존재하나, 합성 프로세스의 비교를 위해 소스 코드가 공개되지 않았다고 가정한 실험도 진행

실험 옵션

입출력 예제 수집 단계:

- 퍼징 도구 AFL++을 전체 시스템을 대상으로 1시간동안 실행. 이 때의 시드는 콘콜릭 테스트 도구 CROWN을 통해 5개의 입력을 생성하여 사용
- 소스 코드가 존재한다고 가정한 경우, CROWN을 통해 각 함수별로 최대 600초씩, 최대 iteration 횟수는 조건문 수의 두 배로 설정
- 위 방법으로 수집한 입출력값들의 일부 파라미터를 뮤테이션하여 입출력 예제를 최대 10,000개까지 생성
- 확장된 분기 커버리지에 필요한 임의의 목표 개수 $\tau = 10000/|B|$ 로, 각 함수의 분기 수마다 다르게 설정

스텝 합성 단계:

- PBE 합성 도구인 DUET의 실행 옵션은 'max_size 256 max_height 10'으로, 전체 합성 시간은 1시간, 매 합성 도구 최대 실행 시간은 300초로 설정
- $\tau = 2$ 로 하여 각 분기를 커버하는 제약 조건 수가 최소 2개 이상은 가질 수 있도록 설정
- 정확도는 소스 코드가 공개/미공개 시 각각 수집했던 입출력 예제 집합을 합집합한 것을 기준으로 측정

실험 대상 프로그램 및 시스템 퍼징 수행 결과

Program	# of Traces	Branch Cov (%)	# of Cand Func.	Cand Func. Cov (%)	# of I/O ex.
Object Follower	30	55.4	6	100	70
Elevator Simulator	5	1.9	3	0	0
gcd.3+newton.1.4	17	87.5	1	100	10
email_spec27_productSimulator.cil	139	42.3	26	100	360
Fibonacci05	17	83.3	1	100	13
pals.lcr.7.1.ufo.UNBOUNDED.pals	59	55.8	2	50	48
token_ring.05.cil-2	153	69.9	8	100	63

- Elevator 예제: 무한 루프가 존재하여 일반적인 상황에서는 종료되지 않아 AFL++을 적용할 수 없었음
- pals 예제: init 함수의 조건식들을 만족하지 못하고 그대로 종료되어 다른 함수에 접근할 수 없었음
- AFL++에서 반환하는 실행 트레이스는 모두 서로 다른 실행 경로를 가져야 하기 때문에, 상대적으로 실행 경로 수가 적은 프로그램의 경우 입출력 예제를 많이 수집할 수 없었음

입출력 예제 생성 결과 (콘콜릭 + 뮤테이션)

- 소스 코드가 존재하는 경우, 분기가 존재하는 15개의 프로그램 중에서 커버리지가 더 높은 경우가 7개, 서로 같은 경우가 7개였음
- init 함수의 경우, 50개의 조건문을 모두 참으로 통과해야 1을 반환하는 함수임
- init 함수를 대상으로 콘콜릭 테스트를 수행한 결과, 600초 안에 1을 반환하는 경우를 찾지 못하였음

Program	Function	B	With Source Code		Without Source Code	
			Ex. Cov (%)	# of I/O ex.	Ex. Cov (%)	# of I/O ex.
Object Follower	getbiggestrect	6	100	9,399	91.7	9,362
	getWidth	0	N/A	9,308	N/A	9,331
	getHeight	0	N/A	9,305	N/A	9,287
	getArea	0	N/A	9,325	N/A	9,369
	fisqrt	4	50	568	50	530
Elevator Simulator	isButtonPressed	6	100	9,888	91.7	9,867
	floorIsOrdered	32	100	8,688	56.3	8,638
gcd_3+newton_1_4	gcd_test	6	100	5,674	91.7	5,367
email_spec27_productSimulator.cil	getEmailFrom	4	75	9,527	75	9,756
	isReadable	8	100	9,890	87.5	9,890
	isReadable_role_Encrypt	6	100	9,576	75	9,709
	findPublicKey	18	100	9,934	83.3	9,923
	isEncrypted	4	75	9,496	75	9,787
Fibonacci05	fibonacci	4	50	548	50	569
pals_lcr.7.1.ufo.UNBOUNDED.pals	check	10	85	9,280	85	9,401
	init	100	57	9,094	64.5	9,119
token_ring.05.cil-2	is_master_triggered	4	62.5	6,614	62.5	7,466
	is_transmit1_triggered	4	62.5	6,564	62.5	7,400

스텝 합성 결과

주목할 만한 케이스:

- is_master_triggered
- fisqrt
- findPublicKey
- init
- floorIsOrdered
- gcd_test

Function	# Params	O	Original Cmplx	With Source Code			Without Source Code		
				Acc (%)	Cmplx	C	Acc (%)	Cmplx	C
getbiggestrect	43	9	405	47.69	22	34	48.85	18	33
getWidth	42	251	3	55.85	2	2	78.19	66	90
getHeight	42	247	3	56.76	2	2	81.86	48	78
getArea	42	700	11	76.6	10	6	71.32	52	36
fisqrt	1	23	67	62.87	44	26	16.64	22	11
isButtonPressed	12	2	1,683	93.5	16	11	89.87	14	14
floorIsOrdered	14	2	2,587	56.13	94	63	86.22	6	10
gcd_test	2	102	14	6.41	12	10	8.58	24	13
getEmailFrom	3	199	8	100	6	6	100	6	4
isReadable	4	2	27	99.85	12	17	99.3	16	10
isReadable_role_Encrypt	3	2	18	99.86	14	13	99.76	10	9
findPublicKey	14	227	29	72.19	16	25	44.96	88	73
isEncrypted	3	113	8	100	6	5	100	6	4
fibonacci	1	18	225	97.56	10	5	100	32	19
check	8	2	11	76.5	10	10	95.38	6	12
init	29	1	102	100	2	1	100	2	1
is_master_triggered	2	2	5	99.39	4	5	100	6	6
is_transmit1_triggered	2	2	5	99.18	4	4	100	6	5

합성 결과 분석 (1)

- **is_master_triggered**: 소스 코드가 존재할 때 원본 코드의 복잡도를 넘지 않아야 된다는 제약 때문에 정확도에서 손해를 봄
- 소스 코드가 존재할 때의 스텝이 "if(m_pc==E_M) return 1; else return 0;"이고, 소스 코드 미공개 시의 합성 결과는 "if(1==m_pc) if(m_pc==E_M) return 1; else return 0; else return 0;"
- **getWitdh, getHeight** 역시 마찬가지로, 합성 스펙에서 표현하지 못하는 인덱스 접근 수식이 존재하여 더 작은 사이즈의 스텝을 만들기 어려움
- **fisqrt**: 약 650개의 출력의 근사치 값들을 미리 저장해두고 사용함. 소스 코드를 알 수 없다면 당연히 해당 상수들도 사용할 수 없으므로, 정확도 면에서 큰 차이를 볼 수 밖에 없음 (62% vs 16%)
- **init**: 앞서 언급했듯이 1을 반환하는 예제가 존재하지 않기 때문에 표면상으로는 정확도 100%이지만, 실질적으로는 재사용 불가능한 스텝을 생성하였음

합성 결과 분석 (2)

- **findPublicKey**: 출력 범위가 상대적으로 넓으면서 (227개), 분기 역시 18개로 비교적 많음.
 - 소스 코드가 공개되지 않았을 때는 단순히 출력 범위를 넓혀야 하기 때문에 출력 범위가 넓으면 정확도에 안 좋은 영향을 미침
 - 반면 소스 코드가 존재하는 경우, 출력 범위에 더해 확장된 분기 커버리지 정보를 이용하여 더 나은 방향으로 가이드할 수 있었음
-
- **floorIsOrdered**: 소스 코드 정보를 확인하지 않고 수집된 8,648개의 예제 중, 오직 241개만이 1를 반환하는 경우였음
 - 소스 코드가 존재하는 경우에서 합성을 하게 될 경우, 새로운 분기를 커버하는 예제는 모두 1을 반환하는 예제이고, 알고리즘에 따라 해당 예제 모두 제약 조건에 추가되어 합성됨
 - 이렇게 합성된 스텝은 분명히 다양한 조건에서 다양한 행동을 수행할 수 있음에도 불구하고, 테스트 집합은 이를 과소평가

합성 결과 분석 (3)

- **gcd_test**: 두 방법 모두 매우 낮은 정확도를 기록
- 분석 결과, 출력/분기 커버리지를 최우선 사항으로 두었기 때문
- 매 합성마다 정확도와 커버리지를 추적한 결과, 커버리지가 85%에서 94%로 증가할 때 정확도가 44%에서 8%로 매우 급격하게 감소하는 경우가 존재
- 하지만 그렇다고 해서 정확도를 일정 부분 반영하게 될 경우, 이 역시 상충 관계 (trade-off) 문제를 겪을 수 밖에 없음
- 예를 들어, 스팸 메일 분류기 A는 항상 스팸 메일이 아니다 (false)만을 반환하여 정확도가 99%인 경우이고, 분류기 B는 스팸 메일을 적절히 분류하나 정확도가 50%이라면, 재사용 가능한 스텝을 생성하기 위해선 B를 선택하는 것이 정답

결론 및 향후 연구

결론:

- PBE기반 프로그램 합성 기법을 이용해 소스 코드가 공개되지 않은 상황이라도 비교적 높은 정확도와 높은 출력/분기 커버리지를 가진 스텝 코드를 합성할 수 있었고, 이를 통해 재사용 가능한 스텝을 자동 생성할 수 있었음

향후 연구:

- 소스 코드가 공개된 상황이라도, 원본 함수보다 비슷한 동작을 수행하면서 더 단순화된 스텝 코드를 합성하여 이를 대체 후 정형 검증을 수행하는, 추상화 기법을 고려
- 이를 위해서는 현재 실험 결과보다 더 정확하고 덜 복잡한 스텝을 합성할 수 있어야 함