



저작자표시-비영리-변경금지 2.0 대한민국

이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.

다음과 같은 조건을 따라야 합니다:



저작자표시. 귀하는 원저작자를 표시하여야 합니다.



비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.



변경금지. 귀하는 이 저작물을 개작, 변형 또는 가공할 수 없습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 [이용허락규약\(Legal Code\)](#)을 이해하기 쉽게 요약한 것입니다.

[Disclaimer](#)

경북대학교 공학석사학위논문

프로그램 합성 기법을 이용한 소프트웨어 검증용 스텝 자동 생성 연구

대학원 컴퓨터학부

김 요 엘

2022년 12월

경북대학교 대학원

프로그램 합성 기법을 이용한 소프트웨어 검증용 스텝 자동 생성 연구

이 논문을 공학석사 학위논문으로 제출함

대학원 컴퓨터학부

김 요 엘

지도교수 최 윤 자

김요엘의 공학석사 학위논문을 인준함

2022년 12월

위원장 _____이우진 (인)

_____최윤자 (인)

_____권영우 (인)

경북대학교 대학원위원회

목 차

제 1 장	서론	1
1.1	연구의 필요성	1
1.2	제안: 프로그램 합성 기법을 이용한 스텝 자동 생성	2
1.3	본 논문의 구성	3
제 2 장	연구 배경 및 관련 연구	5
2.1	소프트웨어 검증 환경 구성	5
2.2	프로그램 합성 기법	7
2.2.1	SyGuS 기반 합성 스펙	7
2.2.2	프로그램 합성 기법의 비교	10
2.3	소프트웨어 검증 및 테스트 기법	13
2.3.1	퍼징	14
2.3.2	콘콜릭 테스트	15
2.3.3	소프트웨어 모델 체킹	17
2.4	관련 연구	18
제 3 장	스텝 자동 생성 방식	20
3.1	전체 프로세스 개요	20
3.2	입출력 예제 수집 방법	21
3.3	점증적 합성 프로세스	24
제 4 장	실험	30
4.1	소스 코드 공개 시 입출력 예제 수집 및 합성 방법	31

4.1.1	유닛 콘콜릭 테스트를 통한 입출력 예제 생성	31
4.1.2	소스 코드가 존재할 때의 합성 프로세스	33
4.2	실험 방법	35
4.2.1	합성 대상 함수 선정	35
4.2.2	실험 옵션	36
4.3	실험 결과	37
제 5 장	결론 및 향후 계획	44

표 목차

표 4.1	실험 대상 프로그램의 시스템 퍼징 실행 결과	37
표 4.2	소스 코드 공개된/공개되지 않은 상황에서 각 합성 대상 함수의 입출력 예제 수집 결과	39
표 4.3	소스 코드 공개된/공개되지 않은 상황에서 각 합성 대상 함수의 스텝 생성 결과	40

그림 목차

그림 2.1	임베디드 소프트웨어의 전형적인 함수 코드	5
그림 2.2	nxt_motor_get_count 함수의 단위 테스트용 스텝 코드	6
그림 2.3	nxt_motor_get_count 함수의 정형 검증용 스텝 코드	7
그림 2.4	SyGuS 기반 합성 스펙 중 문법 제약 부분	8
그림 2.5	SyGuS 기반 합성 스펙 중 의미 제약 부분 (PBE)	10
그림 2.6	EuSolver의 합성 방식	11
그림 2.7	입출력 간의 관계식 기반 합성 기법의 제약 조건 예시	12
그림 2.8	콘콜릭 테스트 설명 예제	16
그림 2.9	CBMC가 대상 프로그램을 논리식(모델)으로 변환시키는 과정	17
그림 3.1	입출력 예제 생성 및 검증용 스텝 합성 과정 개요	20

제 1 장 서론

1.1 연구의 필요성

소프트웨어 검증 [1]은 소프트웨어의 신뢰성 (reliability)과 안전성 (safety)을 향상시키기 위해, 소프트웨어가 주어진 명세 (specification)를 만족하는지 검토하고 확인하는 작업으로, 이를 달성하기 수단으로 크게 소프트웨어 테스트 기법과 정형 검증 기법이 존재한다. 소프트웨어 테스트는 대상 소프트웨어의 입력값들을 생성하여, 대상 소프트웨어에 실제로 수행함으로써 반환된 출력값들이 기대 출력값과 일치하는지 확인하는 과정으로 설명된다. 정형 검증 기법 [2]은 소프트웨어를 실제로 수행하지 않는 대신 주어진 명세를 수학적인 모델로 변환하고 대상 소프트웨어가 명세를 만족하는지의 여부를 수학적으로 증명하는 기법이다. 대표적으로 소프트웨어 모델 체킹 기법 [3]이 존재한다.

소프트웨어 검증을 효율적으로 수행하기 위해서는, 검증 대상 소프트웨어와 상호작용하는 외부 라이브러리 모듈/컴포넌트에 대한 검증 환경 구성이 필수적이다 [4]. 예를 들어, 단위 테스트 시 검증 대상 함수가 접근하는 외부 라이브러리 컴포넌트/모듈을 실제로 가동하면서 테스트를 수행하기에는 검증 비용이 많이 소요되는 문제가 존재한다. 따라서, 테스트 환경 구축을 위해 일반적으로 사용되는 방식은 스텝 (stub) [5]의 사용이다. 스텝¹은 검증 비용을 줄이기 위해 외부 라이브러리 모듈을 대체하고 더 단순하게 시뮬레이션하는 코드로, 주로 특정 행위 및 상태만을 검증하기 위해 고정된 값을 반환한다.

이러한 스텝을 자동 생성하는 도구로 목 객체 프레임워크가 존재한다. 대표적으로 Java를 타겟으로 한 Mockito [7], EasyMock [8]가 있으며, C언어를 지원하는

¹비슷한 용어로, 목 객체 (mock object) [6]가 있다. 본 논문에서는 스텝과 목 객체를 서로 같은 의미로 가정하고 사용한다.

Mimicc [9], Cgreen [10], CMocka [11]등이 있다. 이러한 프레임워크는 스텝/목 객체를 생성하기 위한 뼈대 (필요 클래스/메서드 정의, 제어 API 등)를 제공할 뿐, 실제로 수행되는 행위는 사용자가 직접 작성해야 하는 등, 여전히 추가적인 노력이 필요하다. 목 객체의 행위를 심볼릭 실행 (symbolic execution)을 통해 자동 생성하는 연구 [12, 13]가 진행되었으나, 여전히 특정된 몇몇 입력에 대해서만 처리할 수 있는 등, 오직 특정 테스트에서만 수행 가능하다. 즉, 생성된 목 객체의 재사용이 거의 불가능하고 새로운 테스트에 맞춰 다시 생성해야 하는 문제가 존재한다.

이러한 검증 환경 구성의 어려움은 소스 코드 수준의 정형 검증 기법을 적용할 때도 마찬가지이다. 해당 기법을 적용하기 위해서는 검증 대상 소프트웨어의 모든 소스 코드를 정형 검증 기법이 수학적 모델로 변환할 수 있어야 한다. 그러나, 대상 소프트웨어와 상호작용하는 외부 라이브러리의 경우 대상 소프트웨어와의 소스 코드와는 다른 언어로 작성되었을 가능성이 있으며, 특히 소스 코드가 공개되지 않은 경우 (즉, 실행 파일만 제공하는 경우) 수학적 모델로 변환할 수 없어 정형 검증 기법을 적용하기 어렵다. 이를 해결하기 위해선 일반적으로 외부 라이브러리 모듈에 대해 임의의 값을 반환하는 스텝을 생성 및 대체함으로써 검증할 수 있다. 다만, 이 경우 외부 환경에 대한 어떠한 제약도 존재하지 않기 때문에 오탐 (실제 시스템에서는 발생할 수 없는 오류)만을 찾아낼 가능성이 매우 크다 [14].

1.2 제안: 프로그램 합성 기법을 이용한 스텝 자동 생성

본 논문에서는, C 프로그램을 대상으로 PBE (Programming-By-Example) 기반 프로그램 합성 기법 [15]을 이용해 다양한 입력을 처리할 수 있고 원본 함수/모듈과 비슷한 행위를 수행하는 코드를 합성함으로써, 다양한 입력에 대해 처리 가능한, 즉 재사용 가능한 스텝을 생성하는 기법을 제안한다. PBE 합성 기법은 주어진 입출력 예제로부터 일정한 패턴을 찾아 일반화된 프로그램 코드를 찾아내는 기법으로, 샘플로 주어진 입출력 예제에 대해서 만큼은 100%의 정확도를 보장하나, 샘플로 주

어지지 않은 다른 입출력 예제에 대한 정확도는 보장할 수 없다. 따라서 합성 도구에 어떠한 입출력 예제 집합을 제공하는지에 따라 합성되는 스텝 코드의 정확도, 출력 범위, 복잡도 (코드 사이즈)에 큰 영향을 주며, 사용자는 최적의 해 (최대의 정확도 및 출력 범위, 최소의 복잡도를 가진 코드를 합성하도록 하는 입출력 예제 집합)를 일반적으로 알 수 없다.

이러한 단점에도 불구하고, 소스 코드의 정보를 알 필요가 없이 오직 입출력 예제만으로 프로그램을 만들어낼 수 있기 때문에 스텝 자동 생성에 있어 큰 장점이 될 수 있다는 생각이 들었고, 또한 단순히 고정된 값을 반환하거나 몇 개의 입력에만 처리 가능한 일반적인 스텝보다는 더 정확한 행위를 수행하는 스텝을 합성할 수 있다는 생각에서 본 연구를 진행하게 되었다. 프로그램 합성 기법을 사용하여 목 객체를 자동 생성하는 방식을 아이디어 수준에서 제안한 연구 [16]를 근거로 시작하여, 본 연구는 (1) 합성 대상 함수의 다양한 행위를 커버할 수 있도록, 여러 테스트 케이스 생성 기법 (퍼징 [17] 및 입력값 뮤테이션)을 이용하여 다양한 수준 (시스템, 단위 함수)에서의 입출력 예제를 수집하는 방법을 설명하고, (2) 수집된 입출력 예제를 사용해 합성 도구에 제공하는 예제의 수를 점증적으로 늘려가며 합성을 시도하여 비교적 높은 정확도를 가지고 다양한 입력에 대해 처리 가능한, 재사용 가능한 스텝을 자동 생성하는 알고리즘을 제안하며, (3) 2개의 임베디드 소프트웨어 예제와 SV-Comp 2022 [18]에 사용된 벤치마크 중 몇 개를 골라 합성 대상 함수를 선정하고 제안한 합성 방식을 적용한 실험 결과를 보여준다.

1.3 본 논문의 구성

나머지 논문 내용은 다음과 같이 구성되어 있다: 2장은 먼저 소프트웨어 검증 환경 구축을 위해 스텝이 실제로 어떻게 적용되는지 그 예시를 보여준다. 그다음 프로그램 합성 기법에 대해 더 자세히 설명하고, PBE에 사용될 입출력 예제를 생성하는 데 사용된 소프트웨어 테스트 (테스트 케이스 생성) 기법들에 대해서도 각각 설명한다.

마지막으로, 관련 연구에 대해서도 설명한다. 3장에서는 먼저 본 논문에서 제안하는 전체적인 프로세스에 대해 설명한다. 그다음 입출력 예제를 수집하는 방법에 대해서 설명한다. 시스템/단위 수준에서 입출력 예제를 수집했을 때의 장단점을 설명하고 구체적으로 어떻게 수집하는지 설명한다. 마지막으로 높은 출력 커버리지를 가지고 일정 수준의 정확도를 가지는, 재사용 가능한 스텝을 생성하기 위한 합성 프로세스를 설명한다. 4장에서는 2개의 임베디드 프로그램과 SV-Comp 2022의 벤치마크 중 일부 프로그램들을 대상으로, 각 프로그램에 정의되어 있는 합성 대상 함수를 선정하고 이전 장에서 설명한 예제 생성 및 합성 방법을 적용하여 실험한 결과를 보이고 소스 코드가 공개되었을 때 추가적으로 사용할 수 있는 정보를 이용한 예제 수집 및 합성 방법을 통해 비교/분석한다. 5장에서는 결론 및 향후 연구에 대해 설명한다.

제 2 장 연구 배경 및 관련 연구

2.1 소프트웨어 검증 환경 구성

```
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 }
```

그림 2.1: 임베디드 소프트웨어의 전형적인 함수 코드

그림 2.1은 임베디드 소프트웨어에서 사용되는 전형적인 함수의 예시를 보여주

고 있다. `turnSensor` 함수를 완전히 검증하기 위해서는 특정 하드웨어와 종속되어 있는 `nxt_motor_get_count` 함수와 (라인 7), `ecrobot_set_motor_speed` 함수 (라인 17, 19) 함수를 포함하여 검증해야 한다. 그러나, 해당 외부 라이브러리 함수들을 호출하면서 검증하기에는 실제 접근되는 하드웨어의 환경까지도 포함하여 검증해야 하기 때문에, 단위 테스트를 수행하고 있다는 점을 고려했을 때 검증 비용이 많이 소요된다고 볼 수 있다. 이러한 문제는 소스 코드 수준의 정형 검증 기법을 적용할 때도 존재한다. 해당 외부 라이브러리 함수들이 하드웨어에 접근하지 않는다고 가정하더라도, 애초에 소스 코드가 공개되지 않았다면 해당 라이브러리 함수들을 수학적 모델로 변환할 수 없기 때문에, 완전한 의미에서의 검증을 수행했다고 볼 수 없다.

```

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

```

그림 2.2: `nxt_motor_get_count` 함수의 단위 테스트용 스텝 코드

이러한 검증 환경 구성과 관련된 문제를 해결하는 방식 중 하나로, 스텝 코드를 생성하여 외부 라이브러리 함수 대신 호출되어 더 간단하게 시뮬레이션 되도록 하는 방법이 일반적이다. 예를 들어, 그림 2.2의 경우 특정한 단위 테스트를 위해 만들어진 스텝 코드로, 단순히 0을 고정적으로 반환한다. 이러한 스텝은 실제 외부 라이브러리 함수를 호출하는 대신 간단하게 수행되는 코드가 수행됨으로써 검증 비용을 감소시킬 수 있다. 그러나, 이렇게 작성된 스텝은 특정 상태를 검증하기 위해 항상 고정된 행동만 수행하기 때문에, 다른 목적을 가진 테스트에 맞는 행동을 수행할 수 없다. 즉, 해당 테스트의 목적을 달성하기 위해선 수작업으로 스텝 코드를 변경해 주어야 하는 번거로움이 발생한다.

그림 2.3은 소스 코드 수준의 정형 검증 기법을 적용할 때 일반적으로 사용되는 스텝 코드의 예시이다. 라인 3에서 `SYM_int` 함수는 `SYMbolic`에서 따온 것으로, 검증기가 변수 `ret`에 임의의 정수값을 할당하여 시뮬레이션할 수 있다는 의미이다.

```

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

```

그림 2.3: `nxt_motor_get_count` 함수의 정형 검증용 스텝 코드

또한, 소프트웨어 모델 체킹에서는 `nondet` 함수가 사용되는데, 이는 **NONDETerministic** (비결정적인)의 약자이다. 만약 검증기를 통해 `turnSensor` 함수가 정상적으로 종료될 수 있는지 그 여부를 확인하고 싶다면, 검증기는 `turnSensor` 함수의 파라미터인 `edge`와 변수 `ret`에 임의의 값을 할당해 보면서 시뮬레이션을 수행하고, 만약 가능하다면 `turnSensor` 함수가 종료된 실행 트레이스를 반환할 것이다. 그러나, 이렇게 반환된 트레이스는 해당 외부 라이브러리의 코드와 실제 하드웨어의 동작까지 반영할 수 없기 때문에, 하드웨어를 포함하여 검증한 경우에는 재현이 불가능한 트레이스 (오탐)일 가능성이 존재한다 [14].

2.2 프로그램 합성 기법

프로그램 합성 기법 [19]은 사용자가 다양한 형태로 제공하는 의도 (혹은, 제약 조건)을 파악하여 이를 만족하는 프로그램을 찾아내는 기법이다. 이러한 의도에는 대표적으로 입출력 예제 [20], 입출력 간의 관계식 [21], 미완성된 프로그램 [22]이 대표적이며, 그중에서 입출력 예제 기반 (PBE) 합성 기법과 입출력 관계식 기반 합성 기법을 서로 비교하여 스텝을 합성하는 데 있어서 PBE 합성 기법이 더 효율적인 이유를 설명하도록 한다.

2.2.1 SyGuS 기반 합성 스펙

```

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 ))))

```

그림 2.4: SyGuS 기반 합성 스펙 중 문법 제약 부분

우선 두 합성 기법의 명확한 비교를 위해, SyGuS (Syntax-Guided Synthesis) [23]을 기반으로 하는 합성 기법을 기준으로 설명한다. SyGuS는 위에서 설명한 사용자의 제약 조건뿐만 아니라, 합성될 프로그램이 사용할 수 있는 파라미터 (변수), 연산자, 상수값들을 명시하여 합성기의 프로그램 탐색 공간을 줄여줌으로써 더 효율적으로 합성할 수 있도록 하는 문법 제약을 제시한다. 거기에 더해, SyGuS 기반 합성 스펙 (specification)을 표준화하여 PBE 합성 기법과 입출력 관계식 기반 합성 기법 모두 동일한 형식의 합성 스펙을 사용할 수 있어 비교가 용이하다.

그림 2.4는 SyGuS 기반 합성 스펙 중 문법 제약 부분만을 명시하여 보여준다. 라인 1은 Linear Integer Arithmetic의 약자로, 정수와 관련된 문법을 사용하겠다는 의미이다. 다른 종류로는 문자열과 비트 벡터 (bit vector)가 있다. 라인 2는 gcd.test라는 이름의 함수를 합성하겠다는 것을 명시하고, 해당 함수는 각각 a와 b라는 정수형의 파라미터 변수를 가지며, 정수형의 값을 반환한다는 정보를 명시한다. 라인 3-12는 정수형의 상수값, 파라미터, 연산자를 명시하고 있으며, 라인 13-19는 Boolean 형식을 명시하고 있다. 라인 5는 사용 가능한 상수값들을 명시하며, 라인 6은 사용 가능한 파라미터를, 라인 7-11은 각각 사용 가능한, 연산의 결과가 정수인 연산자를 명시하고 있다. 라인 11의 ite는 if-then-else의 약자로, ntBool이 참이면 두 번째 ntInt 식을, 거짓이면 세 번째 ntInt 식을 반환한다. 이러한 연산자의 형태는 함수형 프로그래밍 언어와 유사하다. 예를 들어 $(3 + 1 - 2)$ 라는 식을 위 스펙에서 표현하고 싶다면 $(+ 3 (- 1 2))$ 로 표현할 수 있다. 또한 라인 7의 더하기 연산의 경우, '+'라는 기호를 가지고, 첫 번째 ntInt 식과 두 번째 ntInt 식을 더한 값을 반환하는 의미를 가진 함수라고 생각할 수 있다.

그림 2.5는 SyGuS 기반 합성 스펙 중 의미 제약 부분, 특히 입출력 예제를 제약 조건으로 표현한 것을 보여준다. 각 라인마다 서로 다른 입출력 예제를 명시하였으며, 라인 1을 살펴보면 앞서 설명한 문법 형식을 따른다는 것을 알 수 있다. $(= (\text{gcd.test } 12 \ 30) \ 6)$ 의 의미는 다시 풀어쓰면 $(\text{gcd.test}(12, 30) == 6)$ 와 동일하다. SyGuS 기반 합성 도구는 그림 2.4의 문법 제약이 명시되어 있는 대로 만들어낼 수 있는 프로그

```

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))

```

그림 2.5: SyGuS기반 합성 스펙 중 의미 제약 부분 (PBE)

램 후보 중에서, 이 4개의 제약 조건을 만족하는 프로그램을 찾아 반환하는 것으로 설명할 수 있다. 이러한 제약 조건을 가지고 PBE 합성 도구인 DUET [15]에 합성시키면 '(define-fun gcd_test ((a Int) (b Int)) Int (ite (< b a) b (ite (= b (+ a a)) a (/ b (+ 3 2)))))'라는 프로그램이 반환된다. define-fun gcd_test는 말 그대로 gcd_test라는 이름의 함수를 정의하고 있다는 의미이고, 파라미터 a, b 및 반환 타입 Int는 앞서 문법 제약 그림 2.4과 동일한 형식이며, 그다음부터는 실제로 합성된 프로그램이다.

2.2.2 프로그램 합성 기법의 비교

그림 2.5처럼, 입출력 예제를 제약 조건으로 받아들이는 것을 PBE 기반 프로그램 합성 기법이라 한다. SyGuS-Comp [24]가 개최된 이래로, 2014년 및 2015년에는 [23]에서 소개된 나열식 (Enumerative) 합성 기법이 우승을 차지했다. 나열식 합성 기법은 bottom-up 방식으로, 문법 제약으로부터 생성할 수 있는 가장 작은 프로그램부터 시작하여, 제약 조건을 모두 만족하는 프로그램을 찾을 때까지 후보 프로그램의 사이즈를 늘려가며 탐색하는 방식을 채택한다. 나열식 기법은 작은 크기의 프로그램들을 매우 빠르게 나열할 수 있어 필요 사이즈가 작은 프로그램은 매우 쉽게 찾아낼 수 있다는 장점이 존재하나, 후보 프로그램의 크기가 커질수록 탐색 공간이 기하급수적으로 넓어지기 때문에 큰 규모의 프로그램은 찾기 매우 힘들어지는 단점이 존재한다.

이러한 나열식 합성 기법의 단점을 극복하기 위해, Alur 외의 연구 [25]에서는 하나의 표현식이 모든 제약 조건을 만족하는 프로그램을 찾는 기존 나열식 방법에

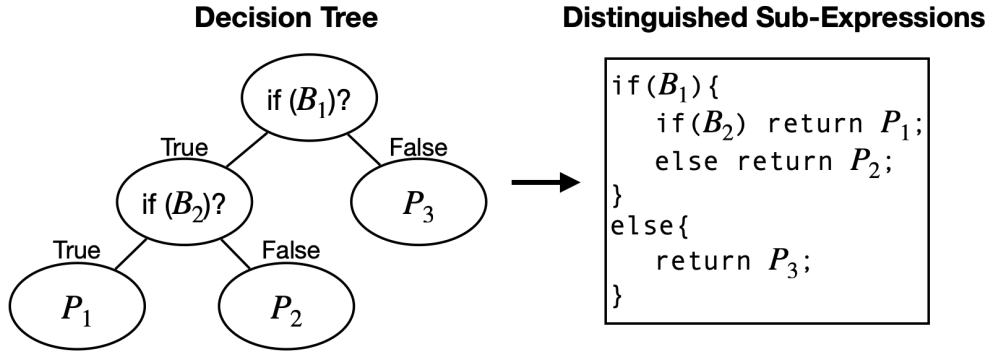


그림 2.6: EuSolver의 합성 방식

서, 여러 서브 표현식들이 각각 만족하는 제약 조건들의 합집합이 전체 제약 조건 집합이 되는 것을 찾고, 서브 표현식들을 구분하는 조건식을 따로 찾아 이를 결합하는 방법을 소개하고 있다. 해당 합성 도구의 이름은 EuSolver로, 그림 2.6에서 합성 방식을 설명하고 있다. 기존 나열식 방법에서는 필요한 조건식 B_1, B_2 와 서브 표현식 P_1, P_2, P_3 들이 결합된 형태를 한 번에 찾아야 했지만, EuSolver는 조건식과 서브 표현식들을 각각 찾아냄으로써 작은 프로그램을 빠르게 찾아낼 수 있는 나열식 방법의 장점을 이용하면서도 이들을 효과적으로 결합하는 방법을 제시함으로써 더 효율적으로 큰 규모의 프로그램을 찾아낼 수 있다. 그 결과, EuSolver는 2016년 및 2017년 SyGuS-Comp에서 우승을 차지했다.

Lee의 연구 [15]에서는 먼저 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, DUET이라는 이름을 가진 합성 기법을 제안하였다. EuSolver와 유사한 내용을 지니고 있으나, DUET은 조건식을 기준으로 프로그램을 나누는 방법 이외에도 3가지의 추가적인 추론 규칙을 정의하였다. 2019년 SyGuS-Comp에서 우승을 차지했던 CVC4 및 EuSolver보다 벤치마크 문제들을 더 빠르고 정확하게 풀어내어 다른 PBE 기반 합성 기법들보다 우위에 있음이 입증되었다.

```

1 (define-fun fare((meter Int) (second Int)) Int (
2   + 3300 (+ (* 100 (/ meter 131)) (* 100 (/ second 31)))
3 ))
4
5 (constraint (= (taxi meter second time) (
6   ite (or (and (<= 0 time) (<= time 4)) (= time 24)) (
7     + (fare meter second) (* 2 (/ (fare meter second) 10)))
8     (fare meter second)) // else
9 ))

```

그림 2.7: 입출력 간의 관계식 기반 합성 기법의 제약 조건 예시

입출력 간의 관계식을 기반으로 하는 합성 기법은 대표적으로 DryadSynth [21]가 존재한다. 일반적으로, 입출력 관계식 기반 합성 기법은 연역적 (deductive) 합성 기법이라 불린다. 반면 PBE 기반 합성 기법은 귀납적 (inductive) 합성 기법이라 불리는데, 이는 주어진 입출력 예제에 대해서는 100% 만족하지만, 주어지지 않은 입력값에 대해서는 정답을 보장할 수 없기 때문이다. 즉, 연역적 합성 기법은 입출력 간의 관계식을 받아 이를 재구성하는 것으로, 전제 조건인 제약 사항이 틀리지만 않았다면 연역적이기 때문에 정확한 프로그램을 합성할 수 있다. 그림 2.7은 연역적 합성 기법의 제약 조건의 예시를 보여준다. 라인 1-3은 taxi 함수를 합성하기 위해 사용할 수 있는 매크로 함수인 fare을 정의한 것이고, 라인 5-9에서 taxi 함수의 제약 조건을 나타내고 있다. 연역적 합성 기법은 대상 함수의 소스 코드나 요구사항 문서, 주석으로 된 설명 등을 가지고 특징을 파악한 뒤에 사용자가 직접 제약 조건을 작성해야 하기 때문에, 스텝을 자동 생성한다는 본 연구의 목적과는 맞지 않다. 이에 대한 자세한 논의는 [26]에서 다루고 있다.

결국 스텝을 자동 생성하기 위해서는, 자동화된 방법을 수집할 수 있는 입출력 예제를 기반으로 하는 PBE 기반 프로그램 합성 기법을 선택해야 한다. 단순히

랜덤 입력 값들을 생성해서 실행만 해도 쉽게 수많은 입/출력 예제들을 수집할 수 있으며, 소스 코드가 없이 실행 파일만 존재하더라도 예제들을 수집할 수 있는 장점이 존재한다. 따라서, 본 연구에서는 PBE 기반 프로그램 합성 도구 중 비교적 성능 우위를 보여주었던 DUET [15]을 통해 재사용 가능한 스텝을 자동 생성하고자 한다.

그러나, 앞서 언급했듯 PBE 합성 기법은 주어진 입출력 예제에 대해서는 100% 정확도를 보장하나, 그 외의 다른 예제에 대한 정확도는 보장할 수 없다. 특히 이러한 문제가 심화될 경우, 일반화가 되지 못한 채 제약 조건으로 주어진 입출력 예제만을 맞추게 되는 과적합 (over-fitting) 문제가 발생할 수 있다. 이는 단순히 특정 행동을 수행하지 못하는 것 (under-approximation) 뿐만 아니라, 의도치 않은 행동 (over-approximation)을 유발할 수도 있다는 의미이다.

과적합된 스텝을 사용하여 소프트웨어를 검증할 경우, (실제로는 오류가 존재하는데도) 오류 자체를 찾지 못하게 되거나, 오류를 찾아내더라도 오탐이 될 수 있는 가능성이 존재한다. 일반적으로 주어지는 제약 조건의 수가 증가할수록 과적합 문제가 해결될 수는 있으나, 이번엔 반대로 합성된 코드의 사이즈 (복잡도)가 증가하거나 제한 시간 안에 합성하지 못하는 등의 문제가 발생한다. 따라서, 스텝을 합성하는데 있어서 영향을 줄 수 있는 부분 (사용 가능한 입출력 예제 집합, 제약 조건을 추가하는 방법, 합성된 스텝의 평가 지표 등)을 잘 정의할 필요가 있다. 그러나, 보통 합성 도구 자체의 알고리즘을 연구하는 데에 관심이 있을 뿐, 합성 도구를 사용하는 사용자 입장에서 합성 도구를 다루는 연구는 거의 존재하지 않는다.

2.3 소프트웨어 검증 및 테스트 기법

본 섹션에서는 소프트웨어 테스트를 위해 테스트 케이스를 생성하는 기법 (퍼징, 콘콜릭 테스트)과, 소프트웨어 검증을 위한 모델 체킹 기법에 대해 설명한다. 테스트 케이스를 생성하는 방식에 따라 크게 랜덤/퍼징 (random/fuzzing) [17,27,28] 방식과 심볼릭/콘콜릭 (symbolic/concolic) [29–31] 방식으로 구분할 수 있다.

2.3.1 퍼징

퍼징이란, 대상 프로그램을 퍼즈 (fuzz) 입력을 통해 테스트하는 것으로서, 퍼즈 입력은 대상 프로그램에서 예상된 입력 범위를 넘어서는, 부정확하거나 의도하지 않은 행동을 유발하는 입력을 의미한다 [32]. 일반적으로 사용자로부터 대상 프로그램에서 정상적으로 수행되는 입력값들의 집합인 시드를 받아 이를 유지하고 적절히 변화시킴으로써 퍼즈 입력을 만들어내려고 시도한다.

Algorithm 1 퍼징 알고리즘

Input: t_{limit} , C // fuzz configurations

Output: B // a set of bugs

```
1:  $B \leftarrow \emptyset$ 
2: while  $t_{elapsed} < t_{limit} \wedge Continue(C)$  do
3:    $conf \leftarrow Schedule(C, t_{elapsed}, t_{limit})$ 
4:    $tcs \leftarrow InputGen(conf)$ 
5:    $B', execinfos \leftarrow InputEval(conf, tcs, O_{bug})$ 
6:    $C \leftarrow ConfUpdate(C, conf, execinfos)$ 
7:    $B \leftarrow B \cup B'$ 
8: return  $B$ 
```

알고리즘 1은 전반적인 퍼징 알고리즘에 대해 설명한다. 입력으로 제한 시간 t_{limit} 과 퍼즈 구성 (configuration) 집합 C 을 입력으로 받아 찾아낸 버그들의 집합 B 을 반환한다. 퍼즈 구성은 퍼징 알고리즘에 영향을 주는 여러 파라미터나 시드 정보를 가지고 있다. 먼저 라인 2에서는 주어진 시간 내에서, *Continue* 함수를 호출하여 해당 알고리즘을 계속 이어갈지를 결정한다. 보통 후술할 화이트박스 퍼징에서 모든 실행 경로를 탐색했을 경우 종료할 때 사용된다.

라인 3-7을 통틀어서 퍼즈 반복 (iteration)이라 부르며, 먼저 *Schedule* 함수를 통해 현재 반복에서 사용할 퍼즈 구성 $conf$ 을 선택한다 (라인 3). 선택한 퍼즈 구성

conf을 통해 테스트 케이스 집합 tcs을 생성하며 (라인 4), tcs을 통해 대상 프로그램을 실행하여 버그 오라클 O_{bug} 을 기준으로 새롭게 발견된 버그 집합 B' 을 찾아낸다. 또한, 이때 실행하면서 얻어낸 정보들인 execinfos도 수집한다 (라인 5). 수집된 실행 정보를 토대로 *ConfUpdate* 함수를 통해 퍼즈 구성 집합 C 을 업데이트하며 (라인 6), 새롭게 찾아낸 버그를 B 에 추가한다 (라인 7).

퍼징의 종류로는 크게 블랙박스 (blackbox), 그레이박스 (greybox), 화이트박스 (whitebox)가 존재하며, 블랙박스 퍼징은 대상 프로그램의 소스 코드 정보를 사용하지 않고 오직 입력 대비 출력 결과만을 확인하는 방식이다. 대표적으로 [28]가 존재하며, 버그를 많이 찾아낸 입력값을 위주로 뮤테이션하는 방식을 사용한다. 그레이박스 퍼징은 간단한 소스 코드 분석을 통해 (보통 코드 커버리지 정보를 사용) 수행 속도를 크게 줄이지 않고 퍼징을 수행할 수 있는 방식이다. 대표적으로 AFL++ [17]가 존재한다. AFL++는 새로운 커버리지를 달성한 입력값을 위주로 뮤테이션하는 방식을 사용한다. 특히 본 연구에서는 C 프로그램을 대상으로 다양한 실행 경로를 수행할 수 있는 테스트 케이스들을 빠르게 찾아낼 수 있기 때문에 PBE 합성에 사용될 입출력 예제를 수집하기 위해 AFL++을 사용한다. 마지막으로 화이트박스 퍼징은 소스 코드 정보를 가장 많이 사용하는 방식으로, 콘콜릭 테스트가 화이트박스 기법에 포함된다. 자세한 설명은 콘콜릭 테스트를 설명하는 부분에서 진행한다.

2.3.2 콘콜릭 테스트

콘콜릭 테스트는 동적 심볼릭 (Symbolic) 테스트으로도 불리며, 대상 프로그램 소스 코드의 구조를 분석하고 이를 토대로 프로그램의 모든 실행 경로를 각각 도달할 수 있는 테스트 케이스들을 생성하는 화이트박스 테스트 기법이다. 콘콜릭 테스트 기법은 심볼릭 실행 기법 [29]을 기반으로 한다. 심볼릭 실행 기법은 프로그램을 논리식으로 변환하여 수학적으로 분석할 수 있는 기법으로, 특정 실행 경로에 도달할 수 있게 하는 논리식의 해 (입력값)를 SMT 해법기 (Satisfiability Modulo Theory Solver) [33]을 통해 찾아낸다.

```

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       }
3     }
2   }
1 }

```

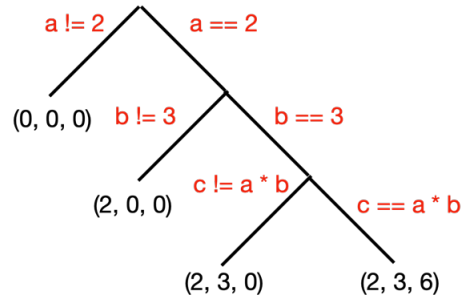


그림 2.8: 콘콜릭 테스트 설명 예제

그림 2.8을 통해 콘콜릭 테스트의 장점을 설명할 수 있다. 함수 f 의 라인 5에 존재하는 Assert 함수에 도달하기 위해서는 라인 2, 3, 4의 조건을 모두 만족하는 입력값인 $(a=2, b=3, c=6)$ 을 찾아야 한다. 만약 이를 블랙박스 퍼징처럼 랜덤하게 입력값을 생성해서 수행할 경우, 정수형 변수인 a 의 값이 2가 되는 가능성이 $1/2^{32}$ 이므로, 3개 변수가 동시에 만족할 확률은 $1/2^{96} = 1/2^{32} \times 1/2^{32} \times 1/2^{32}$ 이다. 사실상 오류를 발견할 확률이 매우 낮다고 볼 수 있다.

반면 콘콜릭 테스트는 소스 코드의 모든 분기를 수행하게 함으로써 4번째 테스트 케이스 생성만에 오류를 찾아낼 수 있다. 첫 입력값 (혹은 시드)이 $(0,0,0)$ 으로 가정을 했을 때, f 을 실행하면 라인 2에서 $a \neq 2$ 의 경로가 수행한다. 이때, 수행했던 경로의 논리식 $a \neq 2$ 를 저장한다. 두 번째 입력값을 생성할 때는, 기존에 저장했던 논리식을 부정한 (negation) 논리식 $a == 2$ 를 구성하고, SMT 해법기가 해당 논리식을 만족하는 해를 풀어내면 $(2,0,0)$ 인 입력값이 생성되어 이번엔 $a == 2$ 이면서 $b \neq 3$ 인 경로가 수행된다. 이때는 가장 마지막에 수행되었던 경로만 부정한 논리식 $(a == 2 \ \&\& \ ! (b \neq 3))$ 을 풀어내어, 입력값 $(2,3,0)$ 을 생성할 수 있다. 이와 같은 방법으로 가장 마지막으로 수행되었던 경로의 논리식을 부정하여 새로운 경로에 도달할 수 있는 입력값을 찾아내고, 이 과정을 모든 경로에 도달할 때까지, 오류를 찾아낼 때까지 반복하게 된다.

2.3.3 소프트웨어 모델 체크

소프트웨어 모델 체크이란, 소스 코드 수준의 정형 검증 기법 중 하나로 검증 대상 소프트웨어의 모든 가능한 상태를 탐색하여 주어진 명세를 만족하는지 자동으로 증명하는 검증 기법이다 [3]. 그러나 소프트웨어 모델 체크는 대상 소프트웨어가 도달 가능한 모든 실행 경로와, 가능한 모든 상태 (각 변수들이 현재 할당된 값의 모든 조합)을 전부 확인해야 하기 때문에 검증 대상 소프트웨어의 규모가 커짐에 따라 탐색해야 하는 상태가 기하급수적으로 커지는, 이른바 상태 폭발 문제가 존재한다. 상태 폭발 문제를 해결하기 위해, 주로 반례 기반 추상화 및 정제 기법 (Counterexample-Guided Abstraction Refinement, CEGAR) [34]의 연구가 많이 진행된 상태이다. CGEAR 기법 중 특히 Predicate abstraction 방식 [35,36]이 대표적으로, 검증 대상 프로그램의 제어 구조 (control-flow)는 동일하나, 기존 데이터를 저장하고 있던 변수들을 제거하고 오직 Boolean 변수로만 구성된 프로그램으로 추상화하여 검증하는 기법이다.

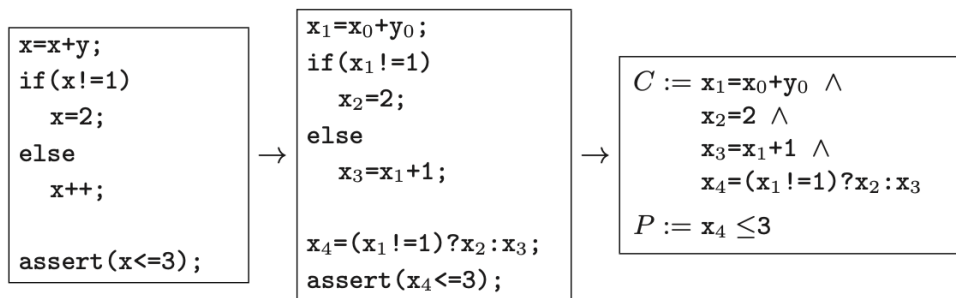


그림 2.9: CBMC가 대상 프로그램을 논리식(모델)으로 변환시키는 과정

C 프로그램을 대상으로 하는 소프트웨어 모델 체크 도구는 CBMC (C Bounded Model Checker) [37]가 대표적이다. 그림 2.9는 CBMC가 C 프로그램을 (1) SSA (Single Static Assignment) 프로그램으로 변환하고, (2) SSA 프로그램을 SAT 해법기 (SATisfiability solver) [38]가 해석할 수 있는 논리식으로 변환되는 과정을 보여준다. SSA는 각 변수가 딱 한 번만 할당되도록 하는 조건을 가진 형태로, 변수 x 가 여러

변 할당되더라도 서로 다른 변수 (x_1, x_2, x_3 등)으로 각각 표현하는 것을 확인할 수 있다. 이러한 방법으로 전체 프로그램에 대한 논리식 모델인 C , 검증 속성 (명제)에 대한 논리식인 P 을 구성하고, SMT 해법기와 유사한 SAT 해법기를 통해 $(C \wedge \neg P)$ 인 논리식을 풀어낸다. 만약 논리식이 참이 되도록 하는 입력값 (x, y)이 존재한다면 해당 입력 값은 검증 속성을 위반하게 되는 것이므로, 이에 대한 반례 트레이스를 반환한다. 이러한 입력값이 존재하지 않는다면 해당 검증 속성을 만족하는 것으로 결론지을 수 있다.

2.4 관련 연구

스텝/목을 자동 생성하는 연구가 많이 진행되었다 [12, 13, 39]. Tilmann 외의 연구 [12]에서는 capture-replay 방식 (테스트 수행 중 실제로 실행된 입출력 값을 저장하고 있다가 이를 그대로 스텝 코드로 사용하는 방식)을 사용하는 것이 아닌, 단위 테스트 코드를 심볼릭 실행을 통해 분석하여 스텝 코드가 어떤 값이 반환되는지에 따라 단위 테스트 코드가 어떤 실행 경로를 가지게 되는지를 계산한다. 이러한 분석 결과를 적용하여, 다양한 조건에 따라 다양한 출력값을 내는 스텝을 생성함으로써 단위 테스트 코드의 코드 커버리지를 높이는 방법을 제시하였다. 그러나, 해당 방법으로 생성된 스텝은 여전히 특정 입력에 대해서만 처리 가능하기 때문에, 미탐 (false positive)이 자주 발생하는 등 재사용 가능한 스텝이 아니다.

Alshahwan 외의 연구 [13]에서는 [12]와 비슷하게 심볼릭 실행을 이용하여 스텝을 자동 생성함으로써 테스트 코드의 커버리지를 높이는 목적을 가지고 있다. 구체적으로는 심볼릭 실행을 통해 스텝으로 대체된 모듈의 post-condition을 추적하여 새로운 반환 값을 생성하는 방식을 제안한다. 그러나, 이 방식 역시 재사용 가능한 스텝을 생성하지 못한다.

Taneja 외의 연구 [39]에서는 대상 프로그램과 데이터베이스 스키마가 주어졌을 때, 심볼릭 실행을 통해 데이터베이스의 행동을 포착하고 이를 토대로 데이터베이

스의 스텝 코드를 자동 생성하는 방법을 제안한다. 해당 방식은 오직 데이터베이스만을 다루기 때문에, 프로그램 합성 도구로 스텝 코드를 생성할 수 있다면 모두 본 연구의 대상이 되기 때문에 목적이 서로 다르다.

Fazzini 외의 연구 [16]에서는 모바일 앱을 대상으로 프로그램 합성 기법을 이용해 재사용 가능한 테스트 목을 자동 생성하는 프레임워크를 고안하였다. 해당 연구에서 최초로 PBE 기반 프로그램 합성 기법을 이용하는 방법을 제안하였으나, 해당 프레임워크가 어떤 프로젝트에 적용될 수 있는지에만 사전 탐색을 진행했을 뿐, 구체적인 스텝 생성 결과에 대해서는 아직 발표되지 않았다.

Seshia 외의 연구 [40]: 시스템을 모델링하고 이를 검증하는 프레임워크로, 그중에서 특히 연역적 프로그램 합성 기법 [41]을 사용하여 모델의 검증을 자동화하는 방법을 제시한다. 작성된 모델이 만족해야 하는 명세를 정의하고 이를 만족하는지 모델 체킹을 통해 증명한다고 할 때, 증명에 실패할 경우 기존 명세보다 더 강력한 조건을 가진 새로운 명세를 수작업으로 작성하고 재증명해야 한다. 이때, 해당 연구에서 제시하는 프레임워크인 UCLID5는 연역적 프로그램 합성 기법을 이용하여 이러한 명세를 자동으로 합성함으로써 사용자의 노력을 상당량 줄일 수 있고, 자동으로 검증 프로세스가 진행된다. 해당 연구는 프로그램 합성 기법을 이용하여 검증에 도움을 준다는 점에서 유사하나, 연역적 프로그램 합성 기법을 사용한다는 점에서 본 연구에서 사용하는 PBE 기반 합성 기법과는 다르다.

Christakis 외의 연구 [42]에서는 스텝 생성 규칙 문법을 제시하며, 이로부터 사용자는 실제 스텝 코드가 아닌 스텝의 명세를 작성함으로써 스텝을 자동 생성하고, 생성된 스텝을 다이내믹하게 주입하여 테스트하는 방법을 제시한다. 그러나 해당 연구는 생성되는 스텝이 일부러 오류를 유발하게 함으로써 대상 프로그램의 오류 처리 코드의 버그를 찾아내는데 주안을 두고 있어 최대한 원본 함수와 비슷한 행동을 하도록 PBE 기반 프로그램 합성 기법을 적용하여 재사용 가능한 스텝을 생성하는 본 연구의 목적과는 다르다.

제 3 장 스텝 자동 생성 방식

3.1 전체 프로세스 개요

본 섹션에서는 검증 대상 프로그램과 상호작용하는 외부 라이브러리 함수의 스텝을 자동 생성하는, 전반적인 프로세스에 대해 설명한다. 먼저 대상 프로그램에서 호출하고 있는 외부 라이브러리 함수들을 파악하고, 다양한 테스트 케이스 생성 기법을 통해 외부 함수들의 입출력 예제들을 수집하며, 수집한 입출력 예제로부터 프로그램 합성 기법을 적용하여 스텝 코드를 자동 생성한다. 합성된 스텝 코드는 소프트웨어 테스트를 적용할 때 검증 비용을 줄이기 위해 기존 외부 라이브러리를 단순히 시뮬레이션할 수 있으며, 소프트웨어 모델 체킹에 적용할 때는 기존에 공개되지 않았던 라이브러리 코드를 대체하는 스텝 코드를 추가함으로써 더 엄밀한 검증을 가능케한다.

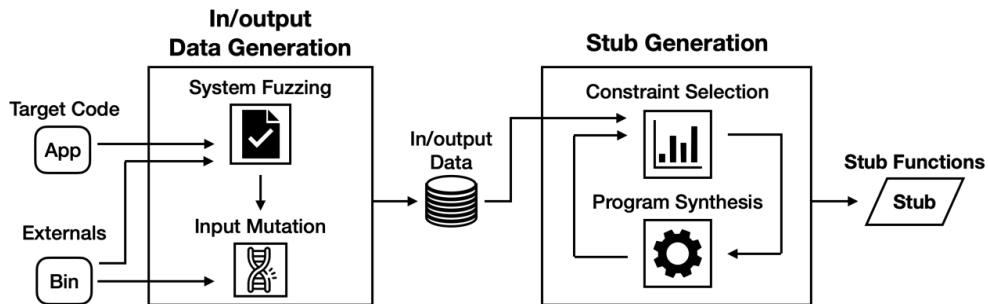


그림 3.1: 입출력 예제 생성 및 검증용 스텝 합성 과정 개요

그림 3.1에서는 본 연구에서 제시하는 입출력 예제 수집 및 검증용 스텝 함수를 자동으로 생성하는 합성 방식에 대해 대략적으로 보여준다. 제안하는 방법은 외부 모듈의 실행 파일 (Externals)이 존재하고, 실행 파일로부터 입/출력 로그를 수집할 수 있다는 가정하에 고안되었다. 혹은, 외부 모듈의 소스 코드나 실행 파일을 접근할

수 없더라도 외부 모듈 함수에 대한 입/출력 로그는 따로 요청하여 사용할 수 있다고 가정하였다.

먼저 검증 대상 프로그램 코드 (Target Code)와 소스 코드가 공개되지 않은 외부 라이브러리의 실행 파일이 입력으로 주어질 때, 바이너리 코드에서도 적용 가능한 시스템 수준의 퍼징을 수행하여 외부 라이브러리 함수들이 실제로 수행된 입출력 예제들을 수집한다. 또한, 시스템 퍼징에서 수집된 입력값들을 적절히 변이 (mutation) 하여 입출력 예제를 충분히 수집한다. 생성된 입출력 예제로부터 점증적으로 하나씩 제약 조건에 추가해가며 합성을 진행하며, 매 합성마다 생성된 스텝 코드의 정확도, 복잡도, 출력 범위 측면에서 스텝 코드를 평가하며, 평가 결과에 따라 새로운 제약 조건이 추가/제거된다. 최종적으로는 가장 높은 출력 커버리지를 가지는, 혹은 출력 커버리지가 같다면 그중에서 정확도가 가장 높은 스텝 코드가 반환된다. 자세한 방법은 후술할 알고리즘 2에서 설명한다.

3.2 입출력 예제 수집 방법

본 섹션에서는 PBE 기반 프로그램 합성 기법에서 제약 조건으로 사용될 입출력 예제를 생성하는 방법에 대해 구체적으로 설명한다. 어떠한 제약 조건을 선택하는가에 따라 합성되는 코드의 정확도 및 효율성이 좌우된다는 것을 이전 장에서 설명하였다. 그렇다면, 제약 조건을 선택하기 이전에 어떠한 제약 조건들을 선택할 수 있는지를 고려하는 것도 중요하다. 입출력 예제 생성 방식을 단순히 랜덤하게 생성하는 것으로 끝내면 안 되는 이유는, 단순히 랜덤하게 생성된 예제 집합은 특정 분기의 조건을 만족하지 못해 특정 실행 경로만 수행되는 등, 편향성이 존재할 수 있기 때문이다. 이처럼 편향성이 존재하는 예제 집합으로 합성하게 될 경우, 필연적으로 편향된 코드를 합성하게 되어 다른 실행 경로에 대한 정확도가 떨어지고 가능한 행동 (출력) 범위가 감소하는 등, 재사용이 어려운 스텝이 생성될 것이다. 따라서, 소스 코드 정보가 없는 상황에서 최대한 다양한 실행 경로 및 출력 범위를 가진 입출력

예제들을 수집하는 것을 목표로 이에 대해 논의하도록 한다.

각 대상 함수의 입출력 예제를 수집하는 데 있어서, 크게 전체 시스템을 실행시켜서 수집하는 것과, 단위 함수를 실행시켜 수집하는 것의 장단점을 설명하고 이를 비교하고자 한다. 우선, 시스템 수준 그레이박스 퍼징을 적용했을 때의 장점은 다음과 같다.

- 한 번의 수행으로 다양한 함수들의 입출력 예제를 동시에 수집할 수 있다.
- 실제 시스템에서 호출되는 입출력 예제를 수집할 수 있다. 즉, 유효한 입력 범위에서만 수행된다.
- AFL++의 경우, 기본 옵션 기준으로 서로 다른 실행 경로를 가진 실행 트레이스만을 제공한다. 즉, 수집되는 입출력 예제가 특정 경로에 편향되지 않는다.
- AFL++의 경우, 소스 코드가 공개되지 않았더라도 실행 파일만 가지고 있다면, 바이너리 코드 수준에서 그레이박스 퍼징을 수행할 수 있다.

시스템 수준 그레이박스 퍼징을 적용했을 때의 단점은 다음과 같다.

- 실행 경로 제약 조건 (path constraint)을 계산하지 않고 단순히 뮤테이션하여 입력값을 생성하기 때문에, 분기 조건의 복잡함에 따라 도달하지 못한 함수가 존재할 수 있다.
- AFL++의 경우, 서로 다른 실행 경로를 가진 실행 트레이스만을 제공하는 점이 오히려 단점이 될 수 있다. 즉, 충분한 수의 예제를 수집하지 못할 수 있다.

정리하자면, 소스 코드가 공개되지 않았더라도 실행 파일만 가지고 있다면 다양한 실행 경로를 수행한 입출력 예제들을 수집할 수 있다는 장점이 있으나, 경우에 따라 수집되는 입출력 예제 수가 적거나, 특정 함수까지 도달하지 못할 수 있는 등, 다양한 실행 경로를 가지도록 확실하게 보장하지 못한다는 단점이 존재한다. 다음은 단위 수준 입력값 뮤테이션을 적용했을 때의 장점을 나열한다.

- 시스템 수준 그레이박스 퍼징을 적용했을 때 호출하지 못했던 함수들을 대상으로 각 함수를 개별적으로 수행하여 입출력 예제를 보완할 수 있다.
- AFL++의 경우, 기본 옵션 기준으로 실행 경로 수가 적은 함수는 상대적으로 적은 수의 예제만을 수집하기 때문에 단위 수준 뮤테이션을 통해 충분한 수의 입출력 예제를 보완할 수 있다.

마지막으로, 단위 수준 입력값 뮤테이션을 적용했을 때의 단점이다.

- 단위 수준 테스트 공통의 문제로, 생성된 함수의 입력값이 전체 시스템에서 유효한 입력값이 아닐 수도 있다. 즉, 실제로는 수행되지 않을 입출력 예제로 합성하게 될 수도 있다. 이는 단위 테스트보다는 모델 체크 등의 전체 시스템에서 엄밀 검증을 적용할 때 특히 문제가 된다.
- 분기 커버리지 정보를 사용하는 그레이박스 퍼징과 달리, 오직 입력 및 출력값만 확인하기 때문에 다양한 실행 경로 및 출력값을 가진 입출력 예제를 생성할 수 있다고 보장할 수 없다.

시스템 수준 그레이박스 퍼징과 단위 수준 입력값 뮤테이션의 장단점을 살펴보면, 시스템 수준 퍼징을 통해 유효한 입력값이면서 다양한 실행 경로를 가진 입출력 예제들을 수집할 수 있으며, 시스템 수준 퍼징에서 도달하지 못했거나 수집된 예제 수가 적은 함수가 존재하더라도 이를 단위 수준 뮤테이션을 통해 개별적으로 실행 및 입출력 예제를 수집하여 보완할 수 있기 때문에 서로의 장점과 단점을 상호보완하는 형태임을 알 수 있다. 이러한 이유로 해당 두 기법을 모두 사용하는 것이 적절하다. 그러나, 근본적으로 두 방법 모두 모든 실행 경로에 도달한 예제 집합을 수집할 수 있다는 보장이 없다.

구체적인 입출력 예제 수집 방법은 다음과 같다. 먼저, 시스템 수준 퍼징을 수행하여 다양한 함수에 대한 입출력 예제들을 수집한다. 그다음, 현재까지 수집한 입출력 예제 집합에서 랜덤하게 일부를 선택하여 각 입력값마다 최소 1개, 최대 3

개의 파라미터에 대해, 자료형에 따라 기존 값에서 최소 ± 8 에서 최대 ± 64 까지 뒤흔들어 새로운 입력값을 생성한다. 또한, 최대 반복 횟수를 지정하여 이를 기준으로 종료한다. 만약 대상 함수의 입출력 예제가 하나도 존재하지 않을 경우, 모든 파라미터가 0인 입력값을 하나 생성하고, 이를 기준으로 뒤흔들는다.

단순히 입력값의 모든 파라미터 값을 랜덤하게 생성하지 않고 일부를 뒤흔드는 이유는, 특정 출력값을 반환하거나 특정 실행 경로를 거친 입출력 예제의 충분한 수를 보장하는 데 있어서 강점이 존재하기 때문이다. 예를 들면, 출력값에 크게 영향을 주지 않는 파라미터 값만 약간 변이 함으로써 동일한 출력값을 가지는 예제를 다양하게 생성할 수 있는 가능성이 더 증가한다. 이렇게 수집된 예제는 제약 조건에 사용되지 않더라도 정확도를 평가할 때 사용된다. 물론 다양한 출력값을 가지도록 하는 것도 중요하지만, PBE 합성 기법의 과적합 문제를 방지하기 위해서는 해당 출력값을 반환하는 또 다른 입력값들이 더 필요하다고 생각하였다.

입출력 수집 방법에 대한 실험은 표 4.1과 표 4.2에서 확인할 수 있다. 만약 라이브러리 함수의 소스 코드가 존재한다고 가정할 경우 적용할 수 있는 방법인 콘콜릭 테스트를 입출력 예제 생성 시 추가적으로 적용하여 최대한 다양한 실행 경로를 가지는 입출력 예제 집합을 생성하고, 후술할 입출력 예제 집합이 얼마나 다양한 실행 경로를 가지는가에 대한 평가 지표를 정의하여 소스 코드가 공개 및 미공개 시 각각 수집할 수 있는 입출력 예제 집합을 비교하고자 하였다.

3.3 점증적 합성 프로세스

본 섹션에서는 재사용 가능한 검증용 스텝을 생성하기 위해, 합성 스펙을 생성하고 스펙으로부터 합성된 스텝 코드를 평가하며, 평가된 정보를 기반으로 새로운 제약 조건을 합성 스펙에 추가해 다시 합성하는, 점증적 합성 프로세스를 제안한다. 합성 대상 함수의 소스 코드가 미공개된 경우 합성되는 스텝 코드의 넓은 출력 범위를 달성하기 위해 매 합성마다 현재 합성된 스텝이 커버하지 못하는 출력값을 가진 입

출력 예제 하나를 새롭게 제약 조건에 추가하는 방식을 사용한다.

본 합성 프로세스를 제안하는 이유는 합성되는 스텝의 성능 (정확도, 복잡도, 출력 범위)을 일정 수준 이상으로 보장하기 위함이다. 제약 조건들의 조합을 임의로 선택하는 경우, 매 합성마다 기존 합성된 스텝의 정보를 사용하지 않고 제약 조건을 선택하기 때문에 합성되는 스텝이 편향된 출력 범위를 가지거나, 너무 많은 제약 조건을 추가하여 합성되는 스텝의 복잡도가 필요 이상으로 증가하거나 제한 시간 안에 합성이 되지 않는 등의 문제가 발생할 수 있다. 반면, 점증적으로 기존 합성 정보를 이용하여 합성하는 방법은 (1) 기존 스텝에서 틀린 출력값을 반환했던 입출력 예제를 제약 조건에 추가함으로써 정확도를 높일 수 있고, (2) 기존 스텝에서 커버하지 못하는 출력값을 가진 예제를 제약 조건에 추가하여 출력 범위를 높일 수 있으며, (3) 제약 조건의 수를 1개부터 조금씩 늘려가며 합성함으로써 스텝의 복잡도가 너무 많이 증가하는 것을 방지할 수 있다. 또한, 제약 조건이 계속 추가되기만 하면 지역 최적화 문제에 노출될 수 있으므로, 기존 제약 조건 집합에서 임의의 제약 조건 하나를 제외함으로써 최적의 제약 조건 집합을 찾도록 계속 시도한다.

다음은 본 장에서 제안하는 점증적 합성 프로세스에서 핵심적으로 사용되는 모듈을 정리한 내용이다.

- **합성 스펙 생성자:** SyGuS PBE 합성 스펙은 크게 사용 가능한 입력 변수, 연산자, 상수를 제한할 수 있는 문법 제약과, 주어진 입출력 예제를 모두 만족해야 하는 의미 제약으로 구성된다. 문법 제약은 합성 대상 함수 코드 내에서 사용되는 연산자, 상수 등을 수집하여 추가할 수 있으며, 입력 변수는 함수의 파라미터 변수들과 함수 내에서 사용되는 전역 변수들을 입력 변수로 수집하였다. 의미 제약은 (1) 현재 의미 제약이 하나도 없는 경우, 주어진 입/출력 예제에서 랜덤하게 하나 선택하거나, (2) 현재 합성된 스텝에서 틀렸던 입출력 예제 중 하나 선택하거나, (3) 현재 합성된 스텝이 달성하지 못한 출력 범위나 분기를 커버하는 입출력 예제를 선택하거나, (4) 지역 최적화 문제를 방지하기 위해 현재 제약 조건 중 하나를 제외하는 등의 연산을 수행한다.

- **스텝 코드 생성자:** 합성 스펙 생성자에 의해 생성된 합성 스펙을 입력으로 받아 주어진 제한 시간까지 스텝 코드를 PBE 합성 도구를 통해 합성한다. 합성된 스텝 코드는 SyGuS 형식으로 반환되므로, 이를 C 언어에서 컴파일 가능한 코드로 변환하고 이 과정에서 상수 폴딩 (Constant Folding)을 수행하여 최적화한다. 특정 상수 값을 표현하기 위해 상수 간의 연산을 자주 사용하는 합성 도구의 특성상 상수 폴딩이 유효하다.
- **스텝 코드 평가자:** 스텝 코드 생성자가 반환한 스텝 코드의 전체 문장 개수를 통해 복잡도를 계산한다. 또한, 주어진 입출력 예제 집합의 입력값을 스텝 코드에 실행시켜 정확도를 평가한다. 이를 통해 스텝 코드의 틀린 입출력 예제 집합 및 유효한 출력 범위를 계산할 수 있다. 이렇게 계산된 정보는 새로운 합성 스펙을 생성하는 데 사용된다.

이렇게 합성 스펙 생성자, 스텝 코드 생성자 및 평가자를 통해 합성 프로세스를 반복하며, 목표 정확도 (100%)을 달성하거나, 주어진 제한 시간 내에 합성된 모든 스텝 코드 중 정확도, 출력 범위, 복잡도를 모두 고려한 스텝 코드를 선택함으로써 최종적으로 원본 함수를 대체할 수 있는, 재사용 가능한 스텝으로서 사용된다.

알고리즘 2는 소스 코드가 공개되지 않았을 때의 합성 프로세스를 보여준다. 주요 프로시저는 *IterativeSynthesis*이며, 원본 함수 f , f 의 입/출력 예제 집합 X , 해당 프로시저의 전체 제한 시간인 t_{entire} , 그리고 각 합성 도구 실행의 제한 시간인 t_{unit} 이 입력으로 주어지고, 그중에서 가장 정확도가 높고 출력 범위가 넓은 스텝인 f_{best} 을 반환한다.

- 가장 먼저 *RandomSelection* 함수를 호출하여, 입출력 예제 집합 X 에서 임의로 하나의 입/출력 예제를 선택하고 이를 합성 스펙의 제약 조건 집합인 C 에 추가한다. (라인 8).
- f_{best} , acc_{best} , cov_{best} , O 을 초기화한다. acc_{best} 와 cov_{best} 는 f_{best} 의 정확도 및

Algorithm 2 소스 코드 미공개 시 합성 프로세스

```
1: procedure GETNEWCONSTRAINT( $W$ : set of incorrect examples,  $O_{not}$ : set of not covered output values)
2:    $T \leftarrow \emptyset$ 
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  $RandomSelection(T)$ 

7: procedure ITERATIVESYNTHESIS( $f$ : original function,  $X$ : set of in/output examples,  $t_{entire}$ : entire timeout,
    $t_{unit}$ : timeout of each synthesis)
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  $f_{best}$ 
```

유효한 출력 범위를 의미하며, O 는 X 에 존재하는 모든 출력값들의 집합이다 (라인 9).

- 소스 코드가 미공개이므로, 함수 f 에서 사용되는 연산자 및 상수값을 알 수 없어 고정된 문법을 사용한다. 사용 가능한 연산자 집합 op_{basic} 은 사칙연산 및 `ite` (if-then-else) 연산, 그 외 비교 및 논리 연산자가 포함되어 있다. 사용 가능한 상수값 집합 $const_{basic}$ 은 -1, 0, 1, 2, 3이다. 그 외의 상수값을 사용하고 싶은 경우, 합성 도구가 상수 간의 연산을 통해 사용해야 한다 (라인 10).
- 최대 t_{entire} 이내에서 함수가 점진적으로 합성되며 (라인 11), 고정된 문법 제약과, 의미적 제약 조건 집합인 C 을 통해 작성된 SyGuS 합성 스펙을 작성하고 (라인 12), 최대 t_{unit} 시간까지 PBE 합성 도구를 실행하여 스텝 코드 f_{stub} 을 합성한다 (라인 13).
- 합성된 스텝 f_{stub} 이 제한 시간 안에 합성된 경우 (라인 14), f_{stub} 을 X 에 대해 실행하여 f_{stub} 의 정확도 acc , X 에서 틀린 예제 집합인 $W (\subset X)$, 그리고 정확한 예제 집합 $(X - W)$ 의 출력 범위인 $O_{covered}$ 을 계산한다 (라인 15). 이때, f_{stub} 의 유효한 출력 커버리지인 cov 역시 계산한다 (라인 16).
- 만약 f_{stub} 의 정확도 acc 가 100%인 경우 (즉, $W = \emptyset$ 인 경우), X 에 존재하는 모든 출력값 집합인 O 역시 모두 커버했다는 의미이므로 (cov 역시 100%), f_{stub} 을 반환하고 프로세스를 종료한다 (라인 17-18).
- 만약 새롭게 합성된 f_{stub} 의 출력 커버리지 cov 가 현재까지 가장 높은 출력 커버리지인 cov_{best} 보다 더 높거나, 적어도 같은 출력 커버리지를 유지하면서 f_{stub} 의 정확도 acc 가 acc_{best} 보다 더 높을 경우, 이를 업데이트한다 (라인 19-21). 즉, 새롭게 합성되는 스텝이 기존보다 정확도가 낮더라도 새로운 출력값을 커버하거나, 혹은 출력 범위가 같으면서 더 높은 정확도를 달성하는 것이 목표이다.

- 라인 19의 조건을 만족하지 못하면서, f_{stub} 의 출력 커버리지 cov 가 100%인 경우, 출력 커버리지는 충분하나 정확도가 더 낮아졌다는 의미이다. 따라서, 제약 조건 집합 C 에서 임의로 하나의 제약 조건을 제외하고, 다시 새로운 제약 조건을 하나 추가하여 정확도가 더 높아질 수 있도록 유도한다 (라인 22-24).
- 제약 조건을 추가할 때는, 틀린 입출력 예제 집합 W 중에서 아직까지 커버하지 못한 출력값을 가진 예제를 하나 골라 C 에 추가함으로써 다음에 합성할 때 더 높은 정확도와 출력 커버리지를 달성할 수 있도록 유도한다 (라인 1-6, 24).
- 만약 라인 14의 조건을 충족하지 못하는 경우, C 의 수가 너무 많아 제한 시간 내에 합성하지 못하는 것으로 판단할 수 있다. 이 경우 임의로 C 에서 하나를 제외함으로써 합성의 난이도를 낮춘다 (라인 26). 이 과정에서 남아있는 제약 조건이 존재하지 않는 경우, 그때는 X 에서 다시 하나를 골라 보충함으로써 해결한다 (라인 27-28).
- 마지막으로, t_{entire} 동안 정확도 100%의 스텝을 생성하지 못했을 경우, 지금까지 생성된 스텝 중 가장 정확도 및 출력 커버리지가 높았던 스텝인 f_{best} 을 반환하고 프로세스를 종료한다 (라인 29).

정리하자면, 첫 번째 합성 시에는 랜덤하게 입출력 예제를 하나 골라 제약 조건에 추가하여 합성하며, 그다음 합성부터는 점증적으로 입출력 예제를 하나씩 제약 조건에 추가해가며 합성한다. 이때 선택되는 제약 조건은 (1) 현재 합성된 스텝이 틀린 출력값을 반환한 입출력 예제이면서, (2) 현재 스텝이 출력하지 못하는 출력값을 반환하는 입출력 예제 중 랜덤하게 선택한다. 이를 통해, 스텝의 출력 범위를 최우선 사항으로 하여 다양한 입력에 대해 다양한 처리를 수행할 수 있는 재사용 가능한 스텝을 생성한다. 스텝의 정확도는 출력 범위가 모두 충족되었을 때만 고려한다.

제 4 장 실험

내가 제안한 합성 방식의 효과를 평가하기 위해, 다음과 같은 3개의 테스트 셋을 사용하였다:

TS1. Lego Mind Storms NXT [43]에서 수행되는 임베디드 로봇 자동차인 Object Follower [44],

TS2. 임베디드 시스템인 엘리베이터 컨트롤러의 시뮬레이션 프로그램 [45],

TS3. SV-COMP 2022 [18]의 벤치마크 일부.

TS1은 카메라에서 측정된 물체와의 거리를 계산하는 태스크와, 자동차의 속도 및 방향을 제어하는 태스크로 구성된다. 합성 대상 함수로는 측정된 물체와의 거리를 가지고 여러 전처리 과정을 수행하는 함수들을 선정하였다. TS2는 엘리베이터에 여러 입력 (버튼 누르기 등)이 들어왔을 때 엘리베이터를 시뮬레이션하는 프로그램이다. 합성 대상 함수로는 엘리베이터의 버튼이 눌렸는지 확인하는 함수와 몇 층으로 가는 버튼이 눌렸는지 확인하는 함수들을 선정하였다. TS3는 다양한 종류의 프로그램 및 함수가 존재하여 제안된 합성 방식을 일반화할 수 있어 선택하였다.

제안된 점증적 스텝 합성 방식을 평가하기 위해, 다음 두 가지 질문을 제시하고 이를 기반으로 실험을 진행하였다.

RQ1. 제안된 점증적 합성 방식은 정확도 측면에서 수용 가능한 수준인가?

RQ2. 오직 입출력 예제만을 사용하는 합성 방식은 효율 측면에서 타당한가?

RQ1의 경우, 제안된 방식을 통해 생성된 스텝의 정확도를 테스트 예제 집합의 기대되는 균형 정확도 (balanced accuracy)보다 높을 경우 정확도 측면에서 수용 가능하다고 판단하였다. 왜냐하면, 현존하는 SyGuS 기반 프로그램 합성 기법들의 성능 측정을 위한 벤치마크는 SyGuS-Comp [24]가 존재하나, 평가 기준은 합성의 성공

유무와 합성 시간 정도이다. 대부분의 벤치마크는 본 실험 예제와 비교했을 때 합성 난이도가 낮으며, 현존하는 합성 기법 자체의 성능 문제로 PBE 트랙의 경우 정수형을 반환하는 벤치마크가 존재하지 않는다. 반면 본 실험 예제는 18개 함수 중 10개가 정수를 반환하며, 나머지 8개가 참/거짓을 구분하는 부울형이다. SyGuS 기반 프로그램 합성 기법을 이용해 합성된 프로그램의 정확도를 측정하는 기존 연구는 존재하지 않으며, 이러한 점을 감안하여 정확도를 평가해야 한다고 판단하였다.

RQ2의 경우, 합성 대상 외부 라이브러리 함수의 소스 코드가 존재한다고 가정된 상태에서 추가적으로 사용할 수 있는 정보 (소스 코드 내의 분기, 연산자, 상수값 등)를 이용한 합성 방식과의 비교를 통해 제안된 합성 방식의 효율을 파악하고자 하였다. 다음에 설명할 내용은 소스 코드가 공개되었을 때 입출력 예제를 수집하는 방법과 스텝을 합성하는 방식에 대해 설명하고, 이를 제안된 방식과 실험 결과를 비교하여 소스 코드가 존재하지 않을 때도 뒤지지 않는 성능임을 보이고, 혹여 성능이 뒤떨어진다고 해도 그 원인을 파악하고자 하였다.

4.1 소스 코드 공개 시 입출력 예제 수집 및 합성 방법

본 섹션에서는 소스 코드가 존재할 때 콘콜릭 테스트 기법을 추가하여 높은 분기 커버리지를 가지는 입출력 예제 집합을 생성하는 방식을 설명하고, 그다음 소스 코드가 존재할 때 얻을 수 있는 정보인 (1) 합성 대상 함수 내에서 사용된 연산자 및 상수값, (2) 각 입력값으로 대상 함수를 실행했을 때 분기의 수행 유무 정보, (3) 대상 함수의 예상 복잡도 정보를 반영한 합성 프로세스를 설명한다.

4.1.1 유닛 콘콜릭 테스트를 통한 입출력 예제 생성

분기 커버리지 (Branch coverage)는 소프트웨어 테스트에서 검증 대상 프로그램 코드가 얼마나 테스트 되었는지를 나타내는 지표 중 하나로, 프로그램 코드 내에 존재하는 조건문의 참/거짓 조건을 얼마나 달성했느냐를 기준으로 평가하는 지표이

다 [46]. 따라서, 입출력 예제 집합의 분기 커버리지가 충분히 높다면 다양한 분기를 수행했기 때문에 어느 정도 다양한 실행 경로를 수행했다고 평가할 수 있다. 그러나, 분기 커버리지는 단순히 테스트의 평가 목적이기 때문에, 합성을 위한 입출력 예제 생성이라는 목적과는 다르다. 예를 들면, 분기 커버리지는 단순히 하나의 입력값만 해당 분기를 수행한다면 분기 커버리지가 증가한다. 그러나, 합성을 위해서는 특정한 하나의 입력값으로는 불충분할 수 있다. PBE 합성 기법의 과적합 문제를 방지하기 위해, 해당 분기를 수행하는 또 다른 입력값들이 더 필요하다고 생각하였다.

이러한 고찰을 통해, 입출력 예제 집합이 얼마나 다양한 분기 수행 정보를 가지고 있는지를 반영하는 평가 기준을 마련하였다. 이를 확장된 분기 커버리지라 정의하였으며, 함수 f 의 입출력 예제 집합 $(x, f(x)) \in X$ 에 대해, $B_{full}(X), B_{scarse}(X), B_{not}(X)$ 은 다음과 같이 정의된다:

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

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

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

$$C(b) = \sum_{(x, f(x)) \in X} \llbracket b \in B(x) \rrbracket$$

여기서 B 는 함수 f 의 모든 분기 집합을, b 는 각 분기를 나타내고, 카운팅 함수 $C(b)$ 는 분기 b 를 수행한 입출력 예제의 수를 의미한다. $\llbracket P \rrbracket$ 는 아이버슨(Inversion) 괄호로, P 가 참이라면 1을, 거짓이라면 0을 반환한다는 의미이다. $B(x)$ 는 입력값 x 을 함수 f 에 실행시켰을 때 수행된 분기 b 의 집합을 의미한다. 즉, $B_{full}(X)$ 는 각 분기 b 을 수행한 입출력 예제의 수가 임의의 목표인 τ 이상 존재하는, 충분히 (fully) 수행된 분기의 수를 의미한다. $B_{scarse}(X)$ 는 $B_{full}(X)$ 에 포함되지 않으면서 $C(b) > 0$ 인 분기의 수를 의미한다. 즉, 해당 분기가 수행된 적은 있으나 그 수가 부족하다는 (scarce) 의미이다. 마지막으로, $B_{not}(X)$ 는 아예 해당 분기가 수행된 경우가 없었다는 뜻으로 (not), $C(b) = 0$ 인 분기의 수를 의미한다. 이를 통해, 확장된 분기 커버리지는 $(|B_{full}(X)| \cdot 2 + |B_{scarse}(X)| \cdot 1 + |B_{not}(X)| \cdot 0) / |B| \cdot 2$ 로 정의할 수 있다.

이는 $|B_{full}(X)|$ 이 크면 클수록, $|B_{not}(X)|$ 이 작으면 작을수록 더 다양한 분기를 만족하면서 합성하기에 충분한 예제 수가 존재한다고 할 수 있다.

이를 통해, 소스 코드가 존재할 때는 다음과 같은 프로세스를 가진다: (1) 시스템 수준 퍼징과 각 단위 함수별로 유닛 콘콜릭 테스트를 통해 수집한 예제를 가지고 있다. (2) 소스 코드 내부의 분기 정보를 이용하여, 현재 가지고 있는 입출력 예제 집합에 대해 $B_{full}(X)$, $B_{scarse}(X)$, $B_{not}(X)$ 로 분류한다. (3) 만약 모든 분기가 $B_{full}(X)$ 에 포함되어 있다면, 해당 프로세스를 종료한다. (4) 매 반복마다, $B_{scarse}(X)$ 에 포함된 분기를 수행하는 입력값 중 일부를 선택하고 뮤테이션하여 새로운 입력값을 생성한다. 뮤테이션된 입력값을 실행하여 입출력 예제를 수집하고 (2)로 돌아간다.

4.1.2 소스 코드가 존재할 때의 합성 프로세스

먼저 사용 가능한 연산자의 경우, 소스 코드 미공개 시에도 나머지 연산을 제외한 대부분의 연산이 포함되었기 때문에 큰 영향이 없다. 그러나, 상수값의 경우 큰 영향을 줄 수도 있는데, 예를 들면 고정된 상수값들로 얻어내기 힘든 큰 상수값들이 함수 내부의 주요 조건식에서 사용되는 경우, 해당 상수값을 사용할 수 있는 유무에 따라 합성되는 스텝의 성능에 큰 영향을 미칠 수 있다. 그다음, 확장된 분기 커버리지 정보를 이용하여 소스 코드가 공개되지 않았을 때 입출력 예제를 선택하는 방식보다 더 효율적으로 제약 조건을 선택할 수 있도록 하였다. 또한, 스텝은 원본 코드보다 더 단순하게 시뮬레이션돼야 하므로, 대상 함수의 예상 복잡도를 정의하고 미리 파악하여 합성되는 스텝의 복잡도를 원본 함수의 복잡도보다 더 낮게 합성하도록 하는 제약을 추가하였다.

우선, 함수 f 의 복잡도 $cmplx(f)$ 는 다음과 같이 정의할 수 있다:

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

여기서 $unit(f)$ 는 함수 f 의 조건 (branch), 할당 (assignment), 함수 호출 (call), 반환 (return) 문장의 총합을 의미하고, $down(f)$ 는 f 가 호출하는 함수 호출의 집합이다.

C_k^g 의 의미는 f 에서 호출되는 함수 g 의 k 번째 호출을 의미한다. 따라서 프로그램 P 의 복잡도 $cmplx(P)$ 는 P 의 *main* 함수의 복잡도 $cmplx(main)$ 으로 표현할 수 있다. 또한, 더 정확한 복잡도를 반영하기 위해, 루프 (loop) 및 재귀 호출 (recursive call)을 주어진 최대 반복 횟수까지만 수행되도록 가정한 상태에서 복잡도를 측정하였다. 예를 들어 최대 반복 횟수가 N 번이라면, 해당 루프 내부에서 수행되는 문장 개수에 N 을 곱하여 복잡도를 측정함으로써 반복문에 의한 복잡도 역시 반영하였다.

다음은 소스 코드가 존재할 때의 합성 방식을 설명한다. 알고리즘 2와 매우 유사한 형태이나, 몇 가지 다른 점이 존재한다. 알고리즘 2의 각 라인에서 추가된 내용을 설명한다.

- 라인 12: SyGuS 합성 스펙을 생성할 때 고정된 문법이 아닌, 함수 f 내부에서 실제로 사용된 연산자 및 상수값을 문법 스펙에 추가할 수 있다.
- 라인 14: 스텝이 시간제한 안에 합성되더라도, 원본 함수 f 의 복잡도 ($cmplx(f)$)보다 합성된 스텝 함수의 복잡도 $cmplx(f_{stub})$ 이 더 낮도록 제약을 추가하였다. 스텝의 복잡도가 더 높을 경우, 기존 제약 조건 집합 C 에서 임의로 하나의 제약 조건을 제외 후 다시 합성하여 복잡도를 낮춘다.
- 라인 16: 알고리즘 2은 합성된 스텝의 출력 커버리지만을 고려했다면, 소스 코드가 존재할 때는 확장된 분기 커버리지 정보까지 고려한다. 제약 조건 집합 C 와 임의의 목표 예제 수인 τ 에 따라 각 분기를 $B_{full}(C)$, $B_{scarse}(C)$ 로 분류하고, 이를 토대로 확장된 분기 커버리지 cov_B 을 계산한다. 알고리즘 2과 동일한 방법으로 계산된 출력 커버리지 cov_O 를 통해 최종적인 커버리지 $cov = cov_O \cdot 0.5 + cov_B \cdot 0.5$ 을 계산한다.
- 라인 4, 24: 라인 24에서 *GetNewConstraint* 함수의 파라미터가 하나 추가된다. 기존 합성된 스텝 f_{stub} 이 커버하지 못한 출력값의 집합 O_{not} 에 더해, (커버하는) 예제 수가 부족한 분기의 집합인 $B_{scarse}(C)$ 을 제약 조건을 추가하는 또 하나의 조건으로써 사용한다. 라인 4에서 틀린 예제 집합 중, 아직까지

커버하지 못한 출력값을 가지면서, 예제 수가 부족한 분기를 수행하는 입출력 값을 새로운 제약 조건으로서 추가한다.

4.2 실험 방법

4.2.1 합성 대상 함수 선정

먼저, 모든 함수가 합성의 대상이 될 수는 없다. 당연한 얘기지만, 입력 변수가 존재하지 않는 경우 (파라미터도 존재하지 않고, 사용되는 (referenced) 전역 변수도 없는 경우)와 출력 변수가 존재하지 않는 경우 (반환하는 값이 없거나, 정의/할당되는 (defined/assigned) 전역 변수도 없는 경우)는 PBE 합성 도구에 전달할 수 있는 마땅한 합성 스펙을 작성할 수 없으므로 합성 대상이 될 수 없다. 이와 같은 이유로, 합성 대상이 될 수 없는 함수를 사전에 분류할 필요가 있다.

- 입력 변수와 출력 변수에서 포인터 등 동적 할당된 변수가 존재하는 경우: SyGuS 합성 스펙 자체는 다양한 자료구조를 지원하나, 본 연구에서 사용하는 합성 도구는 정수 및 부울 (Boolean) 타입의 변수만 지원하기 때문에 배열 및 구조체 변수를 사용하더라도 고정된 사이즈의 경우에만 합성이 가능하다.
- 입력 변수와 출력 변수에서 부동 소수점 (floating point) 자료형을 사용하는 경우: 소수점 자리를 반올림하여 정수형 값으로 변환하는 등의 방법을 사용하면 합성 자체가 불가능한 것은 아니나, 정확도가 매우 떨어지기 때문에, 본 실험에서는 대상 함수에서 제외하였다.
- 후보 함수 내부에서 여러 개의 전역 변수가 정의/할당되는 경우가 존재하는 경우: SyGuS 합성 스펙 특성상, 하나의 스펙에는 하나의 출력만을 정의할 수 있기 때문에, 출력 변수가 여러 개인 경우 합성 스펙을 각각 작성해야 한다. 특히, 여러 개의 전역 변수가 동시에 사용되면서 할당되는 경우에는 입출력

변수들 간의 관계를 파악하기 매우 힘들기 때문에, 본 실험에서는 대상 함수에서 제외하였다.

- 후보 함수 내부에서 외부 입력 함수 (scanf() 등) 호출이 존재하는 경우: 이는 같은 입력에 서로 다른 출력값이 반환될 수 있기 때문에, 제약 조건을 모두 만족해야 하는 PBE 합성 기법의 특성상 합성 대상이 될 수 없다.

위의 전제 조건들을 모두 통과한 함수들 중에서, 각 실험 예제별로 앞서 정의한 함수의 복잡도 ($cmplx(f)$) 상위 5개까지 합성 대상 함수로 선정하였다. 대상 함수들은 소스 코드가 모두 존재하여 알고리즘 2에 대한 실험을 위해 소스 코드가 공개되지 않았다고 가정한 상태에서 실험을 진행하였다.

4.2.2 실험 옵션

모든 실험은 3.3-GHz Intel Xeon Gold 6234 CPU와 187 GB 메모리 하에서 수행되었다. 합성에 필요한 입출력 예제들은 각각 AFL++ [17]을 통한 전체 시스템에 대한 퍼징을 1시간 동안 실행시켜 시스템 테스트 케이스를 수집하였고, 이를 실행하여 얻은 실행 트레이스에서 각 함수의 입출력 예제를 수집하였다. 이때의 시드 (seed)는 콘콜릭 테스트 도구인 CROWN [31]을 전체 시스템을 대상으로 실행하여 수집한 5개의 입력 시퀀스를 사용하였다.

함수의 소스 코드가 존재한다고 가정한 경우, CROWN을 통해 각 단위 함수 별로 최대 600초씩 실행하여 입출력 예제를 추가적으로 수집하였다. 이때의 최대 반복 횟수 (iteration)은 함수 내에 존재하는 조건문 (branch statement)의 두 배로 설정하여 모든 조건문의 참/거짓을 수행하도록 했다 (CROWN의 실행 옵션은 -dfs). 이러한 방법을 사용하여 입력값 뮤테이션을 통해 최대 10,000개의 입출력 예제를 생성하도록 하였다.

PBE 합성 도구는 DUET [15]을 사용하였고 옵션은 "-max-size 256 -max-height 10"을 사용했다. 전체 합성 시간의 timeout (t_{entire})은 1시간으로 설정하였고, 매

DUET의 실행 timeout (t_{unit})은 300초로 설정하였다. 합성된 스텝의 정확도 측정은 소스 코드 공개/비공개에서 각각 수집했던 입출력 예제를 하나로 합쳐 이를 테스트 집합으로 두고 합성된 스텝을 실행했을 때 기대 출력값과 일치하는 비율을 계산하여 측정하였다. 소스 코드가 존재한다고 가정한 경우, 확장된 분기 커버리지를 계산하기 위해 필요한 τ (각 분기를 커버하는 필요한 입출력 예제 수)는 (1) 입출력 예제 수집 단계에서는 $\tau = 10000/|B|$ 로 설정하여 대상 함수 내의 분기 수에 따라 달라지도록 정의하였고, (2) 스텝 합성 단계에서는 $\tau = 2$ 로 설정하여 각 분기를 커버하는 제약 조건 (입출력 예제) 수가 최소 2개씩 가질 수 있도록 하였다.

4.3 실험 결과

표 4.1: 실험 대상 프로그램의 시스템 퍼징 실행 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

표 4.1은 각 실험 예제의 시스템 퍼징을 수행한 결과를 나타낸 것이다. 맨 왼쪽 열부터 대상 프로그램 이름, 수집한 실행 트레이스 수 (시드 포함), 이때의 분기 커버리지, 합성 가능한 후보 함수의 수, 후보 함수들의 함수 커버리지, 마지막으로 총 수집된 입출력 예제 수를 나타낸다.

총 7개의 실험 예제 중, 2개 프로그램 (Elevator Simulator (이하 elev), pals_lcr.7.1.ufo.UNBOUNDED.pals (이하 pals))이 함수 커버리지 100%을 달성하지 못했다. AFL++ 및 CROWN 모두 대상 프로그램이 언젠가 종료되는 것을 전제로 하는데, elev 예제의 경우 시뮬레이터이기 때문에 일반적인 상황에서는 종료되지 않아

시스템 퍼징 및 콘콜릭 테스트를 적용할 수 없는 문제가 발생하였다. 이 경우 임의의 길이의 임의의 값이 할당된 시드 5개를 자동 생성하여 AFL++을 수행하고자 하였으나, 해당 시드 역시 프로그램이 종료되지 않아 그대로 종료되었다. 결국 elev 예제에 대해서는 단순히 시드 5개를 수행했을 때의 결과를 나타낸 것이고, 합성 가능한 후보 함수에 도달하지 못해 입출력 예제를 하나도 수집하지 못했다.

pals 예제의 경우, 후술할 init 함수 (50개의 조건문 모두 참으로 통과해야 1을 반환하고, 통과하지 못할 경우 0을 반환하는 함수)의 출력값이 1이 되어야 나머지 함수들에 도달할 수 있는 구조이다. 즉, 해당 결과는 우선 init 함수에 도달하여 48개의 입출력 예제를 수집하였으나, 해당 예제 모두 조건을 통과하지 못하고 0을 반환한 채 종료되어 다른 함수에 도달하지 못한 것이다. 이는 시스템 퍼징의 단점이자, 단위 수준에서도 입출력 예제를 수집해야 하는 근거라 볼 수 있다.

그 외에도, AFL++에서 찾아낸 모든 실행 트레이스는 서로 다른 실행 경로를 가져야 하기 때문에, 상대적으로 프로그램 규모가 작은 경우 (서로 다른 실행 경로 수가 적은 경우) 수집되는 실행 트레이스 수 자체가 적기 때문에 입출력 예제를 많이 수집할 수 없었다. gcd_3+newton_1_4 (이하 gcd) 및 Fibonacci05 (이하 fibo) 예제가 이에 해당한다고 볼 수 있다. 또한, 실행 경로가 다양하다고 해도 항상 후보 함수들을 호출하는 것이 목적이 아니기 때문에, 실행 트레이스 수가 많다고 해도 상대적으로 적은 수의 예제만 수집되었다.

표 4.2는 소스 코드가 공개/미공개로 가정한 상태에서, 표 4.1의 결과를 기반으로 단위 수준의 콘콜릭 테스트 및 입력값 뮤테이션을 적용하여 각 합성 대상 함수의 입출력 예제를 수집한 결과를 보여준다. 맨 왼쪽 열부터 대상 프로그램의 이름, 각 대상 프로그램의 합성 대상 함수 이름, 각 함수의 파라미터 수, 각 함수의 분기 수, 그리고 소스 코드가 공개 및 미공개인 상황에서 수집된 입출력 예제의 확장된 분기 커버리지 (Ex. Cov) 및 그때의 예제 수 (# of I/O ex.)을 나타낸다. 또한, 대상 함수 내에서 분기가 존재하지 않을 경우, 확장된 분기 커버리지를 계산할 수 없기 때문에 N/A로 표기하였다 (getWidth, getHeight, getArea 함수).

표 4.2: 소스 코드 공개된/공개되지 않은 상황에서 각 합성 대상 함수의 입출력 예제 수집 결과

Program	Function	# Params	B	With Source Code		Without Source Code	
				Ex. Cov (%)	# of I/O ex.	Ex. Cov (%)	# of I/O ex.
Object Follower	getbiggestrect	43	6	100	9,399	91.7	9,362
	getWidth	42	0	N/A	9,308	N/A	9,331
	getHeight	42	0	N/A	9,305	N/A	9,287
	getArea	42	0	N/A	9,325	N/A	9,369
	fisqrt	1	4	50	568	50	530
Elevator Simulator	isButtonPressed	12	6	100	9,888	91.7	9,867
	floorIsOrdered	14	32	100	8,688	56.3	8,638
gcd_3+newton_1_4	gcd_test	2	6	100	5,674	91.7	5,367
email_spec27_productSimulator.cil	getEmailFrom	3	4	75	9,527	75	9,756
	isReadable	4	8	100	9,890	87.5	9,890
	isReadable_role_Encrypt	3	6	100	9,576	75	9,709
	findPublicKey	14	18	100	9,934	83.3	9,923
	isEncrypted	3	4	75	9,496	75	9,787
Fibonacci05	fibonacci	1	4	50	548	50	569
pals_lcr.7.1.ufo.UNBOUNDED.pals	check	8	10	85	9,280	85	9,401
	init	29	100	57	9,094	64.5	9,119
token_ring.05.cil-2	is_master_triggered	2	4	62.5	6,614	62.5	7,466
	is_transmit1_triggered	2	4	62.5	6,564	62.5	7,400

전체적으로, 단위 콘콜릭 테스트를 수행하여 다양한 분기를 수행하는 입력값을 수집한 뒤, 이를 기반으로 뮤테이션하여 입출력 예제를 수집하는 방법이 예상대로 더 높은 확장된 분기 커버리지 비율을 보여주었다. 분기가 존재하는 15개의 대상 함수 중, 확장된 분기 커버리지가 더 높은 경우는 7개이었고, 최소한 커버리지가 같은 경우도 7개이었다.

init 함수의 경우 소스 코드가 공개되지 않았다고 가정한 상태에서 오히려 더 높은 커버리지를 기록했다. 해당 함수는 50개의 조건을 모두 참으로 통과해야 1이 반환되는 함수인데, 이때의 콘콜릭 테스트 실행 결과를 확인해 보니 시간제한 (600 초) 안에 1을 반환하는 입력값을 찾아내지 못하고 종료되었다. 이 때문에 콘콜릭 테스트 수행 유무가 큰 차이를 가져다주지 못했다. 결국 수집된 입출력 예제는 모두 0만 반환하는 경우만 존재하며, 이러한 예제로는 어떻게 합성하더라도 재사용이 불가능한 스텝밖에 생성하지 못하게 된다.

표 4.3: 소스 코드 공개된/공개되지 않은 상황에서 각 합성 대상 함수의 스텝 생성 결과

Function	O	Ex. Acc (%)	Original Cmplx	With Source Code			Without Source Code		
				Acc (%)	Cmplx	C	Acc (%)	Cmplx	C
getbiggestrect	9	34.99	405	47.69	22	34	48.85	18	33
getWidth	251	37.3	3	55.85	2	2	78.19	66	90
getHeight	247	42.66	3	56.76	2	2	81.86	48	78
getArea	700	72.4	11	76.6	10	6	71.32	52	36
fisqrt	23	6.04	67	62.87	44	26	16.64	22	11
isButtonPressed	2	69.88	1,683	93.5	16	11	89.87	14	14
floorIsOrdered	2	74.79	2,587	56.13	94	63	86.22	6	10
gcd_test	102	40.2	14	6.41	12	10	8.58	24	13
getEmailFrom	199	64.49	8	100	6	6	100	6	4
isReadable	2	67.66	27	99.85	12	17	99.3	16	10
isReadable__role__Encrypt	2	67.36	18	99.86	14	13	99.76	10	9
findPublicKey	227	70.19	29	72.19	16	25	44.96	88	73
isEncrypted	113	64.51	8	100	6	5	100	6	4
fibonacci	18	93.82	225	97.56	10	5	100	32	19
check	2	92.23	11	76.5	10	10	95.38	6	12
init	1	100	102	100	2	1	100	2	1
is_master_triggered	2	99.98	5	99.39	4	5	100	6	6
is_transmit1_triggered	2	99.98	5	99.18	4	4	100	6	5

그 외에는, 상대적으로 수집된 로그 수가 적은 경우가 여럿 존재한다 (fisqrt, fibonacci 함수). 이는 해당 함수들의 파라미터 수가 너무 적어서 (1개), 해당 파라미터 값에 대해 뮤테이션을 적용하는 특성상 중복된 입력값이 많이 생성되었다. 그중에서 fibonacci 함수의 경우 입력값이 커지면 실행 시간이 너무 오래 걸려 수집할 수 없었다. 이 경우, 애초에 수집된 예제 수가 적기 때문에 확장된 분기 커버리지가 상대적으로 낮게 측정되었다.

표 4.3은 소스 코드가 공개/미공개로 가정한 상황에서 각 합성 대상 함수의 스텝을 합성 기법을 통해 생성한 결과를 보여준다. 맨 왼쪽 열부터 합성 대상 함수의 이름, 수집된 입출력 예제들의 서로 다른 출력값 개수 (출력 범위), 테스트 예제 집

합의 기대되는 균형 정확도, 원본 함수의 예상 복잡도, 최종적으로 합성된 스텝의 정확도 (Acc), 복잡도 (Cmplx), 이때 사용된 제약 조건의 수 ($|C|$)를 각각 나타낸다. 우선, token_ring.05.cil-2 예제의 경우 (이하 token) 나머지 3개의 합성된 함수들이 모두 거의 비슷한 코드이고, 합성 결과 역시 거의 동일하기 때문에 실험 결과 표에서 제외하였다.

테스트 집합의 기대되는 균형 정확도는 일반적으로 랜덤값을 반환했을 때 기대할 수 있는 정확도인 $1/|O|$ 보다 더 공평하게 평가하기 위해 사용하였다. 이는 테스트 집합에서 서로 다른 출력값의 비율이 불균형할 때 효과가 있다. 예를 들어, 참/거짓을 분류하는 문제에서 테스트 예제 10개 중 9개가 참, 1개가 거짓이라고 하면 기대되는 균형 정확도는 참을 반환할 확률을 90%, 거짓을 반환할 확률을 10%로 가정하고 정확도를 계산한다 (이 경우 기대 균형 정확도는 82%이다).

합성된 스텝의 정확도의 경우, 표 4.2에서 소스 코드 공개/미공개에서 각각 수집했던 두 입출력 예제 집합의 합집합을 통해 측정하였다. RQ1의 경우, 기대되는 균형 정확도보다 소스 코드가 존재하지 않을 때의 생성된 스텝의 정확도가 더 낮은 경우가 전체 18개 함수 중 3가지 (getArea, gcd_test, findPublicKey 함수) 이었다. 또한, RQ2을 기준으로 판단했을 때 소스 코드가 존재할 때 정확도에 확실한 우위가 있는 경우 (최소 5% 이상의 차이)는 getArea, fisqrt, findPublicKey 함수였고, 오차 범위 내에 포함된 함수는 총 18개 합성 대상 함수 중 11개였다. 나머지 getWidth, getArea, floorIsOrdered, check 함수는 소스 코드가 미공개일 때 오히려 더 높은 정확도를 가진 스텝을 합성하였다. 이를 통해 소스 코드가 존재하지 않을 때도 정확도 면에서 큰 차이가 존재하지 않았다고 평가할 수 있다.

일반적으로, 합성 대상 함수의 파라미터 수가 적거나 입력 범위가 좁을수록, 출력 범위가 좁을수록, 원본 함수의 복잡도가 작을수록, 주어진 제약 조건의 수가 많을수록 합성되는 스텝의 정확도가 증가할 것이라고 기대된다. 그러나, fisqrt 함수처럼 파라미터 수가 너무 없으면 오히려 정보가 부족해 정확도가 떨어지는 경우도 존재하고, getEmailFrom 함수처럼 출력 범위가 넓어도 정확도가 매우 높은 경우가

존재하는 등, 다양하고 복잡한 요인이 서로 결합되어 있기 때문에 선불리 경향성을 파악할 수 없었다. 특히 본 실험에서는 표본이 매우 적기 때문에 더욱 어려운 문제이다. 그 대신, 제안된 합성 방식의 성능이 확실하게 낮았던 경우를 분석하여 나열하고자 한다.

- `getArea` 함수의 경우, DUET이 지원하지 못하는 배열의 인덱스를 접근하는 연산을 사용하고 있기 때문에 정확도가 낮았다. 특히 합성된 스텝의 복잡도가 원본 함수의 복잡도보다 더 높았는데, 원본 함수는 쉽게 배열 인덱스에 접근할 수 있어 필요한 문장 (예: `array[index] = 0`)이 적은 반면, 인덱스 접근식 없이 배열에 접근해야 하는 경우 `if (index == 0) array[0] = 0;`의 형식으로 조건식을 통해 풀어서 사용해야 한다. `getArea` 함수의 입력 배열 사이즈는 41이므로, 최대 41개의 조건식을 사용해야 해결할 수 있다는 의미이다. `getbiggestrect`, `getWidth`, `getHeight` 함수 모두 같은 입력 배열을 사용하고 똑같이 인덱스를 접근해야 한다.
- `fisqrt` 함수의 경우, 두 합성 프로세스의 정확도 차이가 제일 큰 경우이다 (약 46% 차이). 이러한 결과에 대한 가장 큰 차이점은, 함수 내부에서 사용되는 상수값의 유무가 가장 크다. `fisqrt` 함수는 입력으로 받은 값의 제곱근 (square root)을 반환하는 함수로, 가능한 출력값에 대한 근사치를 사전에 배열에 저장해두고 사용한다. 이렇게 저장된 근사치는 약 650개이며, 그 값도 0부터 45977까지 다양하다. 이 모든 값이 사용 가능한 상수값으로 취급되기 때문에, 이러한 상수값들을 사용하지 못하는 경우에는 높은 정확도를 달성하기 어려웠다.
- `findPublicKey` 함수의 경우, 소스 코드가 존재할 때 정확도에 확실한 우위가 존재하는 경우이다. 해당 함수는 분기가 18개로 비교적 많으면서, 출력 범위 역시 상대적으로 넓다. 소스 코드 미공개 시의 합성 방식은 항상 더 많은 출력값을 커버하도록 설계되어 있기 때문에, 대상 함수의 출력 범위가 넓은 경우에는 정확도 면에서 단점으로 작용할 수 있다. 반면, 소스 코드가 공개된 경우

확장된 분기 커버리지 정보를 통해 적은 제약 조건 수로도 대상 함수의 의도를 파악하기 용이했을 것이라고 분석하였다. 반면, 출력 범위가 적으면서 (참/거짓 구분 문제) 분기 수도 비교적 많은 경우에는 오히려 분기 정보가 단점으로 작용하여 소스 코드가 미공개인 상황에서 정확도가 더 높은 경우가 있었다. `floorIsOrdered`, `check` 함수가 이에 해당된다.

- `gcd.test` 함수의 경우, 소스 코드 공개/미공개 시의 합성 방식에서 모두 상대적으로 낮은 정확도를 가진 스텝을 생성하였다. 분석 결과, 합성된 스텝의 출력 커버리지를 최우선 사항으로 두었기 때문이다. 알고리즘 2에서 f_{best} 가 업데이트될 때마다 그때의 정확도와 커버리지를 추적한 결과, 커버리지가 85%에서 94%로 증가할 때 정확도가 44%에서 8%로 매우 급격하게 감소하는 경우를 확인하였다. 하지만 그렇다고 해서 정확도를 일정 부분 반영하게 될 경우, 이 역시 상충 관계 (trade-off) 문제를 겪을 수밖에 없다. 본 연구에서는 다양한 행동을 수행할 수 있는, 재사용 가능한 스텝을 생성하는 것이 목표이기 때문에 이러한 합성 알고리즘을 구성하였다.

표 4.3의 결과를 살펴봤을 때, (1) 대상 함수에서 배열 인덱스 접근식을 사용하거나, (2) 대상 함수 내에서 사용되는 상수값들이 주요하게 사용될 경우, (3) 출력 범위가 넓으면서 분기 수도 많은 경우에는 여러 한계로 인해 높은 정확도를 보장할 수 없다. 또한, 본 논문에서 제시한 합성 알고리즘은 합성될 스텝의 출력 범위를 최우선으로 하기 때문에 경우에 따라 낮은 정확도를 가진 스텝이 생성될 수도 있다는 단점도 존재한다. 이러한 단점에도 불구하고, 대상 함수 18개 중 14개 함수에서 소스 코드가 없는 상황에서도 더 높거나 최소한 비슷한 수준의 정확도를 가진 스텝을 생성했기 때문에, 비교적 높은 정확도를 가진 스텝 코드를 PBE 기반 프로그램 합성 기법을 통해 자동 생성할 수 있었다고 결론지을 수 있다.

제 5 장 결론 및 향후 계획

본 논문에서는 PBE 기반 프로그램 합성 도구를 이용해 비교적 정확하면서 다양한 행동을 수행할 수 있는, 재사용 가능한 스텝을 자동 생성하는 합성 프로세스를 제안하였다. 시스템 퍼징 및 입력값 뮤테이션으로부터 입출력 예제를 생성하고, 점증적으로 스텝을 합성하고 평가하여 이를 기반으로 새로운 제약 조건을 선택하는 방식을 제안하였다. 제안된 방식은 합성되는 스텝의 출력 범위를 최우선으로 고려하여 다양한 입력에 대해 다양한 행동을 수행할 수 있도록 하였고, 이를 위해 스텝이 출력하지 못하는 출력값을 가진 예제를 제약 조건에 추가하는 방식을 제안하였다.

실험에서는 2개의 임베디드 소프트웨어 예제와, SV-COMP 2022의 벤치마크 중 일부를 선택하여 해당 프로그램의 여러 함수들을 합성 대상으로 삼아 소스 코드가 공개/미공개 되었을 때를 가정하고, 각각 다른 입출력 예제 생성 방식과 합성 방식을 적용하여 스텝을 생성하였다. 소스 코드가 공개되었을 때는 유닛 콘콜릭 테스트를 추가적으로 적용하고, 코드 내의 상수값, 연산자, 분기 정보 등을 합성 프로세스에서 이용함으로써 두 합성 방식의 차이를 두어도 성능 면에서 뒤지지 않음을 확인하고자 했다. 실험 결과, 소스 코드가 공개되지 않은 상태라도 크게 정확도가 감소되는 일이 없이 일정 수준 이상의 성능을 보일 수 있었다. 다만, 합성 기법 자체의 한계나 부족한 소스 코드 정보로 인해 단점이 두드러지는 경우가 존재하기 때문에 이러한 점을 극복할 필요가 있다.

또한, 본 논문에서는 원본 함수의 예상 검증 복잡도를 정의하고, 소스 코드가 존재할 때 합성되는 스텝의 복잡도가 원본 함수의 복잡도를 넘지 않도록 하는 제약을 추가하여 더 간단한 형태의 스텝을 생성하는 것 까지는 시도하였다. 따라서 향후 연구로는, 정형 검증 기법의 검증 시간을 단축하기 위해 원본 함수보다 더 간단한 형태의 스텝을 합성하여 대체 후 검증하는, 추상화 기법을 연구할 예정이다.

참고 문헌

- [1] D. Beyer and T. Lemberger, “Software verification: Testing vs. model checking,” in *Haifa Verification Conference*. Springer, 2017, pp. 99–114.
- [2] B. Meyer, “On formalism in specifications,” in *Program Verification*. Springer, 1993, pp. 155–189.
- [3] R. Jhala and R. Majumdar, “Software model checking,” *ACM Computing Surveys (CSUR)*, vol. 41, no. 4, pp. 1–54, 2009.
- [4] D. Rzońca, J. Sadolewski, A. Stec, Z. Świder, B. Trybus, and L. Trybus, “Developing a multiplatform control environment,” *Journal of Automation Mobile Robotics and Intelligent Systems*, vol. 13, 2019.
- [5] R. Osherove, “The art of unit testing: with examples in c,” pp. 50–78, 2013.
- [6] T. Mackinnon, S. Freeman, and P. Craig, “Endo-testing: unit testing with mock objects,” *Extreme programming examined*, pp. 287–301, 2000.
- [7] “Mockito,” <https://github.com/mockito/mockito/> (accessed Dec. 20, 2022).
- [8] “EasyMock,” <https://easymock.org/> (accessed Dec. 20, 2022).
- [9] “Mimicc,” <https://mimicc.dev/> (accessed Dec. 20, 2022).
- [10] “Cgreen,” <https://github.com/cgreen-devs/cgreen/> (accessed Dec. 20, 2022).
- [11] “CMocka,” <https://cmocka.org/> (accessed Dec. 20, 2022).

- [12] N. Tillmann and W. Schulte, “Mock-object generation with behavior,” in *21st IEEE/ACM International Conference on Automated Software Engineering (ASE’06)*. IEEE, 2006, pp. 365–368.
- [13] N. Alshahwan, Y. Jia, K. Lakhotia, G. Fraser, D. Shuler, and P. Tonella, “Auto-mock: Automated synthesis of a mock environment for test case generation,” in *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2010.
- [14] Y. Kim, Y. Choi, and M. Kim, “Precise concolic unit testing of c programs using extended units and symbolic alarm filtering,” in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 315–326.
- [15] W. Lee, “Combining the top-down propagation and bottom-up enumeration for inductive program synthesis,” *Proceedings of the ACM on Programming Languages*, vol. 5, no. POPL, pp. 1–28, 2021.
- [16] M. Fazzini, A. Gorla, and A. Orso, “A framework for automated test mocking of mobile apps,” in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2020, pp. 1204–1208.
- [17] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, “{AFL++}: Combining incremental steps of fuzzing research,” in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*, 2020.
- [18] D. Beyer, ““progress on software verification: SV-COMP 2022”,” in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2022, pp. 375–402.

- [19] S. Gulwani, “Dimensions in program synthesis,” in *Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming*, 2010, pp. 13–24.
- [20] O. Polozov and S. Gulwani, “Flashmeta: A framework for inductive program synthesis,” in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2015, pp. 107–126.
- [21] K. Huang, X. Qiu, P. Shen, and Y. Wang, “Reconciling enumerative and deductive program synthesis,” in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 1159–1174.
- [22] A. Solar-Lezama, “The sketching approach to program synthesis,” in *Asian Symposium on Programming Languages and Systems*. Springer, 2009, pp. 4–13.
- [23] R. Alur, R. Bodik, G. Juniwal, M. M. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa, *Syntax-guided synthesis*. IEEE, 2013.
- [24] “Sygus competition,” <https://sygus.org/comp/> (accessed Dec. 20, 2022).
- [25] R. Alur, A. Radhakrishna, and A. Udupa, “Scaling enumerative program synthesis via divide and conquer,” in *International conference on tools and algorithms for the construction and analysis of systems*. Springer, 2017, pp. 319–336.
- [26] Y. Kim and Y. Choi, “A case study on the performance of program synthesis in embedded software domain,” in *KCSE*, 2021.

- [27] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, “Directed grey-box fuzzing,” in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2329–2344.
- [28] M. Woo, S. K. Cha, S. Gottlieb, and D. Brumley, “Scheduling black-box mutational fuzzing,” in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, 2013, pp. 511–522.
- [29] C. Cadar, D. Dunbar, D. R. Engler *et al.*, “Klee: unassisted and automatic generation of high-coverage tests for complex systems programs.” in *OSDI*, vol. 8, 2008, pp. 209–224.
- [30] J. Burnim and K. Sen, “Heuristics for scalable dynamic test generation,” in *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2008, pp. 443–446.
- [31] “CROWN: Concolic testing for Real-wOrld softWare aNalysis,” <https://github.com/swtv-kaist/CROWN/> (accessed Dec. 20, 2022).
- [32] V. J. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, “The art, science, and engineering of fuzzing: A survey,” *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2312–2331, 2019.
- [33] C. Barrett and C. Tinelli, “Satisfiability modulo theories,” in *Handbook of model checking*. Springer, 2018, pp. 305–343.
- [34] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, “Counterexample-guided abstraction refinement,” in *International Conference on Computer Aided Verification*. Springer, 2000, pp. 154–169.

- [35] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani, “Automatic predicate abstraction of c programs,” in *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, 2001, pp. 203–213.
- [36] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav, “Satabs: Sat-based predicate abstraction for ansi-c,” in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2005, pp. 570–574.
- [37] E. Clarke, D. Kroening, and F. Lerda, “A tool for checking ansi-c programs,” in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2004, pp. 168–176.
- [38] A. Biere, M. Heule, and H. van Maaren, *Handbook of satisfiability*. IOS press, 2009, vol. 185.
- [39] K. Taneja, Y. Zhang, and T. Xie, “Moda: Automated test generation for database applications via mock objects,” in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, 2010, pp. 289–292.
- [40] S. A. Seshia and P. Subramanyan, “Uclid5: Integrating modeling, verification, synthesis and learning,” in *2018 16th ACM/IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*. IEEE, 2018, pp. 1–10.
- [41] F. Mora, K. Cheang, E. Polgreen, and S. A. Seshia, “Synthesis in uclid5,” *arXiv preprint arXiv:2007.06760*, 2020.
- [42] M. Christakis, P. Emmisberger, P. Godefroid, and P. Müller, “A general framework for dynamic stub injection,” in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 586–596.

- [43] “nxtOSEK: ANSI C/C++ with OSEK/ μ ITRON RTOS for LEGO MIND-STORMS NXT,” <http://lejos-osek.sourceforge.net/> (accessed Dec. 20, 2022).
- [44] “Object follower,” <https://github.com/addud/object-follower/> (accessed Dec. 20, 2022).
- [45] “Elevator controller simulator,” <https://github.com/Sandbergo/elevator-controller/> (accessed Dec. 20, 2022).
- [46] H. Zhu, P. A. Hall, and J. H. May, “Software unit test coverage and adequacy,” *Acm computing surveys (csur)*, vol. 29, no. 4, pp. 366–427, 1997.

An Automated Stub Generation Approach Using Program Synthesis for Software Verification*

Yoel Kim

*School of Computer Science and Engineering
Graduate School, Kyungpook National University
Daegu, Korea
(Supervised by Professor Yunja Choi)*

(Abstract) The difficulty of software verification is the construction of the verification environment. There is a high-cost problem while operating external library modules that interact with the target software. In this situation, we usually use stubs that replace external libraries and simulate them more simply. However, users typically must write stub's behavior manually. Moreover, the generated stubs can only be used in a few specific tests. Given the circumstances, this paper proposes an automated generation approach of reusable stubs with high accuracy and high output coverage using PBE-based program synthesis techniques. This paper first explains collecting input/output examples of each module through system fuzzing and unit mutation and proposes a synthesis and evaluation process when the source code is closed. The process incrementally selects constraints that have outputs currently not covered by the synthesized stub. For the experiment, this work selected two embedded software and several benchmark programs on the SV-Comp 2022. As a result, the proposed approach can show high accuracy and high output coverage of synthesized stubs.

* A thesis submitted to the Council of the Graduate School of Kyungpook National University in partial fulfillment of the requirements for the degree of Master of Science in December 2022