임베디드 소프트웨어 도메인에서의 프로그램 합성 기법의 성능 확인을 위한 사례연구

김요엘○, 최윤자*
경북대학교 컴퓨터학부
kimyoel2305@gmail.com, yuchoi76@knu.ac.kr

A Case Study on the Performance of Program Synthesis in Embedded Software Domain

Yoel Kim○, Yunja Choi*
School of Computer Science and Engineering, Kyungpook National University

요 약

프로그램 합성 기법은 다양한 형태의 제약조건을 받아들여 프로그램 탐색 공간에서 제약조건을 만족하는 실행 가능한 프로그램을 찾아내는 기법이다. 임베디드 소프트웨어의 검증 시간과 복잡도를 단축하기 위해 높은 정확도를 가지고 더 단순한 형태로 합성할 수 있는 합성 기법을 찾아볼 가치가 높다고 판단하여 본 연구에서는 Duet과 DryadSynth 두 합성기를 통해 Object Follower의 함수들을 합성하고 합성 결과를 확인했다. Duet은 입/출력 예제 생성이 쉽고 자동화가 가능하며 프로그램 탐색 공간이 좁은 경우 성능이 매우 뛰어났지만 반대의 경우 합성 결과가 좋지 못했고, DryadSynth는 입/출력 사이의 관계를 완벽하게 표현하는 제약조건을 작성했을 때 성능이 매우 뛰어났지만 사용자가 직접 작성해야 하며 그 과정이 어렵고 시간이 오래 걸린다는 단점이 있음을 확인할 수 있었다. 이러한 장단점을 토대로 함수 유형별로 어떤 합성 기법을 선택해야 하는지에 대한 기준도 제시했다.

1. 서 론

프로그램 합성 기법은 다양한 형태의 제약조건(입/출력 예제, 입/출력 사이의 논리적인 관계, 자연어로 기술된 명세서, 프로그램의 일부분, 비효율적인 프로그램 등)을 받아들여 프로그램 탐색 공간에서 제약조건을 만족하는 실행 가능한 프로그램을 찾아내는 기법이다 [1].

[2]에서는 모바일 앱을 테스트하기 위해 안드로이드 프레임워크나 서드파티 라이브러리의 모의 객체(test mock)를 프로그램 합성 기법을 통해 생성하려는 시도가 있었다. 이와 비슷하게 프로그램 합성 기법을 통해 기존의 함수를 더 단순한 형태로 합성할 수 있을 경우 합성 결과를 소프트웨어 동적 검증에 사용하여 검증 시간과 복잡도를 단축시킬 수 있다. 특히 이러한 시도를 임베디드 소프트웨어 도메인에서 적용한다면 특정 하드웨어를 호출하는 함수를 플랫폼 독립적인 함수로 변환하여 검증에 사용될 수 있다. 따라서 높은 정확도를 가지면서 더 단순한 형태로 합성할 수 있는 프로그램 합성 기법을 연구해볼 가치가 높다고 판단한다.

플랫폼 독립적인 함수를 검증하기 앞서 본 연구에서는 임베디드 소프트웨어 도메인에서 자주 사용되는 함수별 합성 기법의 성능을 확인하기 위해 입/출력 예제를 제약조건으로 받아들이는 Duet [3]과, 입/출력 사이의 논리적인 관계에 대한 표현식을 제약조건으로 받아들이는 DryadSynth [4]를 소개하고, 간단한 예제를 통해 두 합성기를 비교한다. Object Follower [5]의 함수들을 대상으로 두 합성 기법을 적용하여 얼마나 잘 합성하는지를 비교하고 분석한다. 마지막으로 두 합성 기법의 장단점을 파악하고 함수 유형별로 어떤 합성 기법을 선택해야 하는지에 대한 기준을 제시한다.

2. 프로그램 합성 기법 소개

본 연구에서는 프로그램 합성 기법 중 SyGuS(Syntax-Guided Synthesis) [6] 방식을 채택하는 합성 기법을 사용한다. SyGuS는 사용자가 합성하고 싶은 프로그램의 스펙(Specification)을 작성할 때 기존의 제약조건에다 사용 가능한 자료형과 연산자, 상수 값들을 추가로 제한하여 프로그램 탐색 공간을 최적화하고 이를 표준화해서 작성하는 방법이다. 합성 결과 역시 동일한 방식으로 표현되기 때문에 여러 가지 합성 기법을 사용해도 동일한 방법으로 실행할 수 있고 서로의 결과를 비교하기 쉽다.

Duet은 가장 작은 표현식부터 점차 큰 표현식을 나열하는 방법을 사용하여 작은 프로그램 부품(컴포넌트)을 만들어내고, 큰 문제를 작은 문제들로 나눠 컴포넌트들이 작은 문제를 해결하고 서로 결합되어 최종적으로 입/출력 예제 제약조건을 만족하는 프로그램을 찾아내는 합성 기법이다. SyGuS-Comp 2019 [7]에서 우승을 차지했던 CVC4보다 SyGuS 벤치마크 문제들을 더 빠르고 더 많이 풀어내어 다른 유사 합성 기법들보다 우위에 있음이 입증되었다.

DryadSynth는 제약조건 표현식들을 연역적으로 풀어내어 합성하는 기법을 사용하고, 연역적으로 풀지 못할 때 해당 문제를 작은 문제들로 나눌 수 있으면 다시 연역적 기법을, 그렇지 않으면 표현식을 나열하는 기법을 사용하면서 서로 협력하여 제약조건을 만족하는 프로그램을 찾아내는 합성 기법이다. SyGuS-Comp 2019의 CLIA(Conditional Linear Integer Arithmetic) 트랙에서 1위를 차지하였다. CLIA는 문법엔 제약이 없지만 산술 연산과 조건 연산, 해당 연산으로 구성된 함수만 사용 가능하다는 의미이다.

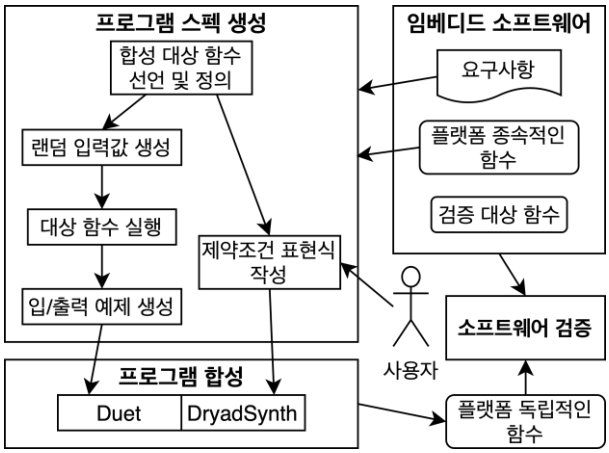

그림 1 프로그램 합성 기법의 활용 방안

3. 프로그램 합성 기법 비교

3.1 합성 과정 및 활용 방안

그림 1은 Duet과 DryadSynth의 구체적인 합성 과정을 비교해서 보여주고, 임베디드 소프트웨어 도메인에서의 프로그램 합성 기법 활용 방안을 보여준다. 합성 대상 함수와 요구사항이 입력으로 주어지면 프로그램 스펙 생성 단계에서 먼저 대상 함수의 정의 부분을 이용하여 함수의 이름, 매개변수, 반환형을 스펙에 작성한다. Duet의 경우 추가로 대상 함수의 소스 코드를 읽어서 사용할 연산자 및 상수 값을 스펙에 작성해야 한다. 반면 DryadSynth의 경우 CLIA 기반이므로 그 부분은 작성할 필요가 없다.

그다음, (1) 요구사항에 부합하는 입력값 범위 안의 랜덤 값을 생성하고, 생성된 입력값으로 대상 함수를 실행한 뒤에 제약조건에 해당하는 입/출력 예제를 생성하여 스펙에 추가하여 Duet을 통해 합성한다. (2) 사용자가 대상 함수의 소스 코드와 요구사항을 읽고, 이를 기반으로 어떠한 함수인지를 파악한 뒤 제약조건에 해당하는 표현식을 작성하고 스펙에 추가하여 DryadSynth를 통해 합성한다.

임베디드 소프트웨어 도메인에서 소프트웨어 검증 대상에 포함되기 힘들었던 특정 플랫폼에서만 호출 가능한 플랫폼 종속적인 함수를, 앞서 설명했던 프로그램 합성 기법을 통해 플랫폼 독립적인 함수로 변환함으로써 검증에 포함시킬 수 있다. 단, 검증에서 사용될 정도라면 합성 결과의 정확도(합성 결과의 출력값이 기존 함수의 출력값과 최대한 비슷한 값을 반환해야 함)가 매우 높으면서도 기존 함수보다 덜 복잡해야 하기 때문에 합성 기법의 성능이 매우 중요하다.

3.2 Duet 스펙 예제

Duet은 대상 함수의 실제 입/출력 예제를 기반으로 합성하기 때문에 대상 함수에 랜덤 입력값을 주고 실행하여 쉽게 입/출력 예제를 생성하여 스펙 생성을 자동화할 수 있다. 그렇게 생성된 스펙의 합성 결과는 표 1을 보면 알 수 있다. 맨 왼쪽 열부터 함수의 이름, 입/출력 예제의 수, 합성 시간, 합성 결과의 사이즈, 랜덤 입/출력 예제 1000개를 기준으로 한 정확도, 합성 시에 사용 가능한 연산자와 상수 값의 수, 그리고 출력값의 범위를 나타낸다. ‘int’는 정수형의 모든 값을, ‘0~’은 0을 포함한 자연수이다.

모든 함수는 매개변수와 반환형이 정수형이고 요구사항에 부합하는 입력값 범위 내에서 랜덤 입/출력값을 100개씩 추가하여 합성했으며, 정확도가 더 이상 증가하지 않을 때까지 합성했다. 대체적으로 대상 함수와 거의 정확하게 합성하면서도 합성 결과의 복잡도(합성 사이즈)가 낮았지만, 운행 거리와 운행 시간, 탑승 시간이 주어졌을 때 택시 요금을 반환하는 taxi 함수는 다른 함수에 비해 정확도가 매우 낮았고 합성 사이즈도 거대했다.

표 1 Duet의 합성 결과

함수 명	입/출력 예제 수	합성 시간	합성 사이즈	정확도	연산/상수 합계	출력 범위
stock	100개	0.3초	24	100%	10개	0~4
area	100개	0.1초	40	100%	5개	0~
median	600개	4.5초	461	100%	4개	int
bool	500개	9.1초	282	99.5%	7개	0~1
max	500개	6.1초	1821	52.2%	10개	0~7
taxi	200개	284.9초	3262	6.5%	15개	3300~

Duet이 taxi 함수를 제대로 합성하지 못했던 이유는 사용 가능한 연산자의 수와 상수 값이 15개로 표 1중에서 제일 많아서 원하는 합성 결과와 전혀 상관없는 작은 컴포넌트들을 많이 만들어내어 컴포넌트 사이즈가 더 이상 증가(복잡하고 큰 표현식을 생성)하지 않아도 쓸모없는 컴포넌트들이 서로 결합하여 합성 자체는 되었지만, 그 결과가 스펙에 작성된 입/출력 예제는 만족하나 사용자의 의도는 만족하지 못하는 과적합한 합성 결과를 도출하기 때문이었다.

3.3 DryadSynth 스펙 예제

DryadSynth는 대상 함수의 입/출력 사이의 논리적인 관계를 표현식으로 작성하여 프로그램을 합성한다. 실제 입/출력 예제를 받아들이지 않기 때문에 사용자가 대상 함수의 소스 코드나 대상 함수와 관련된 정보(요구사항, 입/출력 예제, 주석으로 된 설명 등)를 가지고 함수의 특징을 파악한 뒤에 직접 제약조건에 해당하는 표현식을 작성해야 한다. 표현식을 작성하는 과정에서 입/출력 사이의 관계를 완벽하게 표현할 수 있다면 대상 함수와 서로 같은 함수를 합성할 수 있다.

그림 2는 taxi 함수에 대한 DryadSynth 스펙의 제약조건 부분을 보여준다. 그림 2처럼 taxi 함수의 소스 코드를 보고 같은 연산을 수행하도록 스펙을 작성해서 Duet이 제대로 합성하지 못했던 taxi 함수를 0.2초 만에 합성 사이즈가 156, 정확도가 100%인 함수를 합성할 수 있었다. 또한, 대상 함수의 실제 구현과는 전혀 달라도 대상 함수의 요구사항을 만족하는 새로운 스펙을 작성해서 기존 함수와 비슷한 수준의, 혹은 더 낮은 복잡도를 가진 함수를 합성할 수 있다면 더 좋은 합성 결과를 얻을 수 있다. 하지만, DryadSynth는 스펙 생성을 자동화할 수 없고 직접 작성해야 하기 때문에 스펙 작성 시간이 길어질 수 있다.

```
(define-fun macro((meter Int) (second Int)) Int (
    + 3300 (+ (* 100 (/ meter 131)) (* 100 (/ second 31)))
))

(constraint (= (taxi meter second time)
    (ite (or (and (<= 0 time) (<= time 4)) (= time 24))
        (+ (macro meter second) (* 2 (/ (macro meter second) 10)))
        (macro meter second`)
    )
))
```

그림 2 taxi 함수의 제약조건이 담긴 DryadSynth의 스펙

4. 실험

4.1 실험 대상 함수

Object Follower [5]는 근처의 표적 물체를 따라 이동하면서도 항상 표적으로부터 일정 거리를 유지하는 로봇이다. Object Follower는 크게 카메라에서 측정된 센서 값을 해석하는 함수와 표적과의 거리를 계산하는 함수, 그리고 로봇의 움직임을 제어하는 함수로 이루어져 있다. 이 중에서 센서 값을 해석하는 함수를 두 합성 기법이 얼마나 잘 합성하는지를 실험해보고자 한다.

그림 3은 실험 대상 함수를 보여준다. 대상 함수들은 임베디드 소프트웨어의 전형적인 특징인 배열, 전역변수, 정수형의 사용과 넓은 입/출력의 범위 등이 두드러진다. 대상 함수들은 모두 nxtcamdata라는 전역변수를 사용하며, nxtcamdata은 최대 8개의 rectindex를 가지고 각 rectindex마다 nxtcamdata에 두 점의 x, y 좌표(총 4개의 값)를 저장하고 있어 직사각형의 넓이를 구할 수 있다. 각 좌표 값의 범위는 U8(unsigned char)로 정의했으므로 0~255 사이의 값이다.

```
typedef unsigned char U8;
static U8 nxtcamdata[41];
int getWidth(U8 rectindex);
int getHeight(U8 rectindex);
int getArea(U8 rectindex);
int getbiggestrect(U8 pcolorid, int minarea);

#define MYABS(x) (((x) >= 0)? (x):-(x));
int getWidth(U8 rectindex) {
    int xul = (int) nxtcamdata[5 * rectindex + 1 + 1];
    int xlr = (int) nxtcamdata[5 * rectindex + 1 + 3];
    return MYABS(xlr - xul);
}

int getHeight(U8 rectindex) {
    int yul = (int) nxtcamdata[5 * rectindex + 1 + 2];
    int ylr = (int) nxtcamdata[5 * rectindex + 1 + 4];
    return MYABS(ylr - yul);
}

int getArea(U8 rectindex) {
    return getWidth(rectindex) * getHeight(rectindex);
}

int getbiggestrect(U8 pcolorid, int pminarea) {
    int rectindex = -1;
    int maxarea = pminarea;
    for (int i = 0; i < nxtcamdata[0]; i++) {
        int colorid = (int) nxtcamdata[1 + 5 * i + 0];
        if (colorid == pcolorid) {
            int area = getArea(i);
            if (area >= maxarea) {
                maxarea = area;
                rectindex = i;
            }
        }
    }
    return rectindex;
}
```

그림 3 Object Follower의 실험 대상 함수

getWidth와 getHeight 함수는 rectindex에 해당하는 x/y 좌표 값 2개를 읽은 뒤에 두 수의 차의 절댓값을 반환한다. 출력값의 범위는 0~255 사이의 값이다. getArea 함수는 getWidth와 getHeight 함수의 결과를 서로 곱한 값을 반환한다. 출력값의 범위는 0~65025 사이의 값이다. getbiggestrect 함수는 getArea 함수를 통해 각 rectindex에 해당하는 직사각형의 넓이를 계산해 가장 큰 넓이를 가진 rectindex를 반환한다. 만약 rectindex에 있는 값이 모두 pcolorid와 일치하지 않거나, 가장 큰 넓이가 pminarea보다 작거나, nxtcamdata에 데이터가 없다면 -1을 반환한다. 그러므로 해당 함수의 출력값 범위는 -1~7 사이의 값이다.

4.2 실험 방법

- Duet과 DryadSynth는 실수형을 지원하지 않기 때문에 본 실험에서는 합성 기법이 정수, 논리 자료형 및 해당 자료형의 연산자만 사용 가능하도록 감안했다.
- nxtcamdata과 같이 전역변수를 사용할 경우 스펙에서 전역변수를 매개변수로 추가했다.
- Duet과 DryadSynth는 배열을 지원하지 않기 때문에 nxtcamdata과 같이 배열을 사용할 경우 각 요소들을 정수형 일반 변수로 변환하여 스펙에 추가했다.
- Duet을 통해 합성할 때는 랜덤 입/출력 예제를 100개씩 생성해 스펙에 추가하고 합성했을 때 가장 정확도가 높은 합성 결과를 사용했다. 시간 초과는 1시간으로 설정했으며, 실행 옵션은 lbu, max_size, init_comp_size, all을 사용했고, 적절히 값을 조절해서 사용했다.
- DryadSynth를 통해 합성할 때는 대상 함수의 소스 코드를 직접 확인하면서 스펙을 작성하였으며, getbiggestrect는 반복문이 포함되어 있어 소스 코드 그대로 스펙을 작성할 수 없었기 때문에 기존과는 다른 방식으로 구현되는 스펙을 만들어서 합성했다.
- 합성에 사용된 입/출력 예제를 제외한 1000개의 랜덤 입력값을 생성하여 실제 대상 함수의 출력값과 합성 결과의 출력값을 서로 비교하여 합성 결과의 정확도를 측정했다.

4.3 실험 결과

표 2는 실험 대상 함수에 대한 Duet과 DryadSynth의 합성 결과를 보여준다. 맨 왼쪽 열부터 함수의 이름을 나타내고, Duet에서는 입/출력 예제 수, 합성 시간, 합성 결과의 사이즈, 정확도, 컴포넌트 사이즈(표현식의 복잡도), 합성 시 사용 가능한 연산자와 상수 값의 수를 나타내고, DryadSynth에서는 합성 시간, 합성 결과의 사이즈, 정확도를 나타낸다.

먼저 Duet의 getHeight, getWidth, getArea 함수 합성 결과는 정확도가 전부 5% 미만이었다. 이렇게 합성 결과가 좋지 않았던 이유를 생각해 보면 매개변수가 42개라 입력값의 경우의 수가 매우 많았고, 매개변수 중에서 실제로 사용되는 값은 5개밖에 없어 사용되지 않는 나머지 37개의 매개변수가 합성을 방해했다.

표 2 Duet과 DryadSynth의 실험 결과

함수 명	Duet						DryadSynth		
	예제 수	합성 시간	합성 사이즈	정확도	컴포넌트	연산/상수 합계	합성 시간	합성 사이즈	정확도
getWidth	200개	177.7초	1341	2.7%	3	6개	0.3초	687	100%
getHeight	300개	458.1초	2006	2.2%	3	6개	0.3초	687	100%
getArea	300개	325.9초	3129	4.8%	3	6개	0.4초	2292	100%
getbiggestrect	100개	5.2초	416	37.9%	1	15개	3.0초	56411	100%

특히 출력값의 경우의 수가 각각 256, 256, 65026개로 매우 넓었다. 이러한 과적합 문제를 해결하기 위해 init_comp_size 옵션을 사용했지만 컴포넌트 사이즈를 5 이상으로 증가시키면 컴포넌트 생성 시간이 기하급수적으로 증가해 시간 초과였고, 시작 컴포넌트 사이즈를 4로 해도 3과 동일한 결과이면서 합성 시간만 증가했다.

Duet의 getbiggestrect 함수 합성 결과 약 38% 정도의 정확도를 보여주었다. 이전 함수들보다 정확도가 높은 이유는 출력값의 경우의 수가 8가지 밖에 되지 않았기 때문에 매개변수 간의 복잡한 계산 없이 조건 연산만으로도 그럴듯한 결과를 낼 수 있었기 때문이다. 그러나 랜덤 입/출력 예제 수가 증가한다고 해서 정확도의 유의미한 증가는 없었으며, 오히려 합성 시간과 사이즈만 증가할 뿐이었다.

마지막으로 DryadSynth의 합성 결과를 살펴보면 실험 대상 함수와 동일한 함수를 합성할 수 있었고, 수작업이 필요한 스펙 작성 시간을 제외한 합성 시간은 매우 빨랐다. 하지만 합성 사이즈가 기존 함수보다 너무 길어서 검증에 쓰이기엔 어려울 것으로 예상된다.'

표 3 Duet과 DryadSynth의 장단점 요약

합성 기법	장단점
Duet	'간단한' 함수의 경우 성능이 뛰어남
	예제로 스펙을 생성하기 때문에 자동화가 쉬움
	'복잡한' 함수의 경우 성능이 낮음
DryadSynth	'복잡한' 함수를 합성할 수 있고, 성능이 뛰어남
	스펙 작성을 직접 해야 하기 때문에 자동화가 힘듦
	스펙 작성이 어렵고 시간이 오래 걸림

표 3은 3, 4장의 실험 결과를 토대로 Duet과 DryadSynth의 장단점을 요약한 것으로, '간단한' 함수의 기준은 프로그램 탐색 공간이 좁은 경우(컴포넌트 수가 적고 필요한 컴포넌트 사이즈가 5 미만으로 작은 경우), 출력값의 범위가 몇 가지 경우로 고정되는 경우를 말한다. '복잡한' 함수의 기준은 상수 값이 아닌 여러 매개변수 간의 복잡한 표현식을 찾아야 하는 경우(필요한 컴포넌트 사이즈 5 이상으로 큰 경우), 출력값의 범위가 넓은 경우(입력 값에 따라 출력값이 변동하는 경우), 사용 가능한 연산자 및 상수 값이 많은 경우(10개 이상), 사용되지 않는 입력값이 합성에 방해를 하는 경우를 말한다. 물론 절대적인 기준이 아니며, 정확도가 함수 유형마다 다른 것처럼 이러한 기준도 상대적으로 적용된다.

실험 결과로 보았을 때 '복잡한' 함수를 합성해야 할 경우 DryadSynth를 사용하여 사용자가 직접 스펙을 작성해서 합성하는 것이 더 나은 성능을 보인다. 다만 DryadSynth는 스펙 작성을 자동화하기 어렵고 시간이 오래 걸린다는 단점이 존재한다. 그에 비해 표 3에서 언급했듯이 스펙 생성 자동화가 가능하다는 점에서 Duet의 합성 가능성과 정확도를 높이는 방법을 조금 더 연구해볼 필요가 있다.

5. 결론 및 향후 연구

본 연구에서는 Duet과 DryadSynth를 사용하여 Object Follower의 함수들을 합성해 보았다. Duet은 스펙 생성이 쉽고 자동화가 가능하며 프로그램 탐색 공간이 좁은 경우 성능이 매우 뛰어났지만, 매개변수 간의 복잡한 표현식을 통해 출력값을 계산해야 하는 경우 합성 결과가 좋지 못했다. DryadSynth는 입/출력 사이의 논리적 관계를 완벽하게 표현했을 경우 성능이 매우 뛰어났지만, 사용자가 스펙을 작성하기 어렵고 스펙 작성 시간이 오래 걸린다는 단점이 있었다. 향후 연구에서는 본 연구에서 제시한 합성 기법이 동적 검증에 사용될 수 있는 수준까지 정확도를 끌어 올릴 수 있는 방법을 연구해볼 예정이다.

Acknowledgement

이 논문은 2020년도 정부(과학기술정보통신부, 교육부)의 재원으로 한국연구재단-차세대정보컴퓨팅기술개발사업의 지원을 받아 수행된 연구임 (NRF-2017M3C4A7068175, NRF-2016R1D1A3B01011685).

참고 문헌

[1]. S. Gulwani, "Dimensions in Program Synthesis," *Principles and Practice of Declarative Programming*, 2010

[2]. M. Fazzini, A. Gorla, A. Orso, "A Framework for Automated Test Mocking of Mobile Apps," *Automated Software Engineering*, 2020

[3]. W. Lee, "Combining the Top-down Propagation and Bottom-up Enumeration for Inductive Program Synthesis," *Principles of Programming Languages,* 2021 (accepted)

[4]. K. Huang, X. Qiu, P. Shen, et al., "Reconciling Enumerative and Deductive Synthesis," *Programming Language Design and Implementation*, 2020

[5]. Object Following Robot, https://devpost.com/software/object-follower

[6]. R. Alur, R. Bodik, G. Juniwal, et al., "Syntax-Guided Synthesis," *Formal Methods in Computer-Aided Design*, 2013

[7]. SyGuS-Competition 2019, https://sygus.org/comp/2019