기호실행을 이용한 C 언어 단위테스트 케이스 자동 생성기의 구현

서윤주, 김택수, 이춘우, 김기문, 박복남*, 신철오*, 우치수
서울대학교 컴퓨터공학부
*LG 전자 DTV 연구소
e-mail : {iamonly, dolicoli, oniguni, gildream, wuchisu}@selab.snu.ac.kr
{bnpark, sinco}* @lge.com

Implementation of Automated C Unit Test Case Generator using Symbolic Execution

Yunju Seo, Taeksu Kim, Chunwoo Lee, Kimun Kim, Boknam Park*, Chuloh Shin*, Chisu Wu
Dept. of Computer Science Engineering, Seoul National University
*Dept. of DTV R&D Center, LG Electronics

요 약

본 연구에서는 소프트웨어의 구현 코드로부터 테스트 케이스 자동에 관해 연구하며 도구를 구현한다. 이를 통해 개발자가 직접 테스트 케이스를 작성하는 데 소요되는 비용을 절감하고, 소프트웨어의 요구사항 명세가 잘 작성되어 있지 않거나 실제 구현과 차이가 있는 경우에도 영향을 받지 않고 테스트 케이스를 생성 가능하도록 한다.

1. 서론

단위 테스트 케이스 생성은 단위 함수들 각각에 대해 지속적이고 반복적으로 수행하는 과정이기 때문에 비용이 많이 든다. 이러한 비용을 줄이고자 하는 시도의 일환으로 테스트 케이스 생성 자동화가 있다. 이는 개발자의 업무량 즉 비용을 감소시킬 수 있어서 그 유용성이 크다. 테스트 케이스는 일반적으로 기능별 요구사항 명세(functional requirement specification)로부터 생성된다. 그러나 자연어 기반의 명세로부터 생성하고자 하는 경우 자연어의 특성으로 인해 자동화가 어렵다. 또한 비용의 제약으로 인해 제대로 된 기능별 요구사항 명세가 존재하지 않거나, 실제 구현된 코드의 수정이나 변경으로 인해 코드와 일치하지 않는 경우가 많다.

본 연구에서는 소프트웨어의 산출물 중 구현 소스 코드에 기반하여 테스트 케이스를 자동으로 생성하는 방법에 대해 연구하고 자동생성도구인 CUTIG(C Unit Test Input Generator)를 구현하였다. CUTIG 는 전처리된 C 소스 코드로부터 수행 가능한 경로를 추출한다. 각 경로를 기호실행(symbolic execution)을 이용해 경로조건(path condition)을 생성하고 이를 만족하는 자유변수의 범위를 구함으로써, 테스트 케이스의 입력값을 자동 생성한다. CUTIG 는 가능한 많은 경로에 대한 입력값을 생성함으로써 높은 테스트 커버리지를 확보하고자 한다.

본 논문은 다음과 같이 구성된다. 2 장에서는 테스트 케이스 생성에 관련된 기존 연구를 기술하고, 3 장에서는 CUTIG 에 대해 설명한다. 그리고 실제 코드를 CUTIG 에 적용한 사례를 4 장에서 소개하고, 마지막으로 결론 및 향후 개선 방향을 제시한다.

2. 관련연구

테스트 케이스 생성에 관한 연구는 테스트 케이스를 무작위로(random) 생성하는 접근법으로부터 시작되어 프로그램의 모델을 기반으로 생성하는 방식으로 발전되어 왔다. 이 방법들은 서로 영향을 끼치며 동시에 적용되기도 한다.

랜덤 테스트 케이스 생성은 테스트 입력을 무작위로 생성하기 때문에 만들기는 쉬우나 결과값의 수준이 대체로 낮은 편이다. 하지만 최근의 연구들을 보면 기존의 모델 기반 테스트 생성과 결합하여 그 가능성을 높이고 있다.

Sen 은 “concolic testing”이라는 DART 접근법을 개발하였다[1]. 이는 기호실행과 랜덤 테스트를 결합한 방식으로서 랜덤 테스트 생성과 기호실행이 동시에 수행되면서 서로 도움을 준다. 이런 방법은 어느 정도 실행 가능한 테스트 입력을 생성하지만 조건을 풀어내는 과정이 임의의 수를 생성하는데 의지하기 때문에 부정확할 수 있다.

임의로 생성한 기존의 테스트 결과를 피드백(feedback)하는 방식도 존재한다[2]. 이는 피드백한 결과를 조건 또는 필터로써 임의의 테스트 생성에 도움을 주는 방식이다. 이런 방식 또한 Sen 의 연구처럼 부정확하거나 너무 많은 수의 테스트 입력이 생성될

수 있다.

모델 기반 테스트 생성은 프로그램의 모델로부터 테스트를 생성하는 방법으로 그 역사가 매우 깊다. Moore 는 이미 1956 년도에 유한상태기계(finite-state machine)기반의 테스트 생성을 연구하기 시작했다[3]. 모델 기반 테스트에는 다양한 모델을 사용할 수 있고 이를 통해서 나온 조건을 만족하는 입력값을 찾는다.

기호실행은 오래 전부터 모델 기반 테스트 생성에 사용되어 왔다[4]. 이 기법은 테스트 생성의 조건을 찾는데 유리하지만, 조건에 맞는 테스트 입력값을 찾는 능력에 따라 전체적인 성능이 결정된다. 따라서 최근에는 모델체킹(model checking)과 같은 기술과 결합하여 연구가 진행되고 있다[5]. 그러나 모델체킹은 상태 폭발(state explosion)의 가능성이 있기 때문에 최상의 성능을 내는 조건을 찾는 것이 필요하다

3. CUTIG

CUTIG 는 5 개의 모듈로 구성되며 각 모듈은 다음과 같은 역할을 한다.

- 제어 흐름도 생성기(control-flow graph generator)

전처리기(preprocessor)를 거친 소스코드를 분석하여 제어흐름도를 생성한다.

- 경로 생성기(path generator)

제어흐름도의 노드를 방문하면서 수행 가능한 경로를 생성한다.

- 경로 조건 생성기(evaluator)

경로를 만족하는 자유변수의 부등식을 생성한다.

- 부등식 풀이기(inequality solver)

생성된 부등식을 푼다.

- 결과 생성기(output writer)

계산된 결과를 알맞은 형태로 출력한다.

그림 1 은 CUTIG 의 구조를 보여준다. 이 중 CIL[6]을 이용하여 구현한 제어 흐름도 생성기와 비교적 구조가 단순한 결과 생성기를 제외한 세 개의 핵심 모듈에 대하여 아래에 상세히 설명한다.

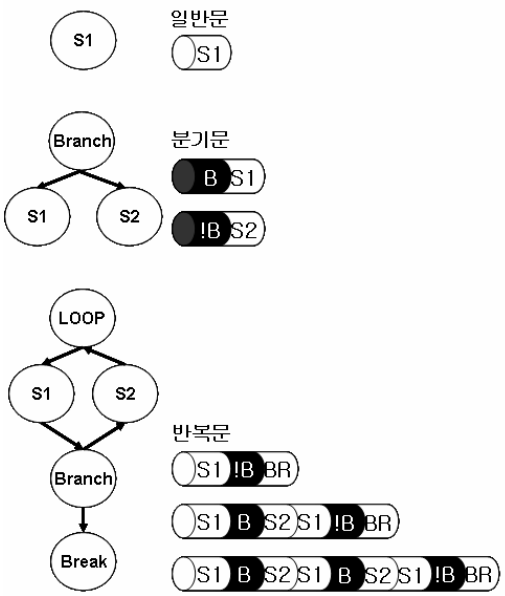
(그림 1) CUTIG 구조

3.1 경로 생성기

경로생성기는 제어흐름도로부터 다음의 기준을 만족하는 경로를 생성한다.

- 모든 문장(statement)은 한번 이상 수행하여야 한다.
- 모든 분기문(branch)은 한번씩 수행하여야 한다.
- 모든 반복문(loop)은 0 번, 1 번, 2 번 반복하여 수행하여야 한다.

그림 2 는 문장의 종류에 따른 경로 생성 방법을 보여주고 있다. 분기문은 조건에 따라 두 개의 분기가 수행 가능하다. 그러므로 생성중인 경로에 참-분기(B)와 거짓-분기(B!)를 추가한 두 개의 경로를 생성한다. 참-분기문 경로에는 참인 경우에 오는 노드(S1)가 뒤에 추가되고 거짓-분기문 경로에는 거짓인 경우에 오는 노드(S2)가 뒤에 추가된다. 반복문은 무한히 수행될 경우까지 고려하여 경로를 생성하면 경로가 무한으로 많아지기 때문에 모든 경로를 찾는 것이 불가능하다. 그러므로 경로생성기에서는 반복문이 0 번(S1-!B-BR), 1 번(S1-B-S2-S1-!B-BR) 그리고 적어도 1 번 이상(S1-B-S2-S1-B-S2-S1-!B-BR) 수행하는 세 가지 경로를 생성한다. 그 외 일반문은 프로그램에서 한 번 수행되며 생성중인 경로(S1)에 추가된다.

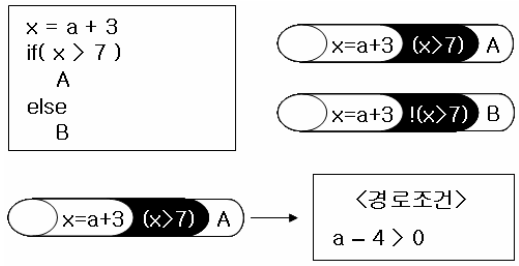


(그림 2) 경로 생성 종류

3.2 경로 조건 생성기

경로 조건 생성기는 경로 분석을 통해 생성된 각각의 경로를 받아서 기호실행 과정을 통하여 해당 경로로 제어를 흐르게 하기 위해 가능한 경로 조건 리스트를 추출하는 모듈이다. 기호실행을 이용해 자유 변수(함수의 인자, 호출되는 함수, 전역 변수)들이 만족해야 할 조건이 다원 다차 부등식으로 변형되며 그 결과인 부등식이 경로 조건이 된다. 경로는 메모리에 값을 저장하는 대입문(assignment), 조건문(condition)과 그 외의 일반문(statement)의 리스트로 구성된다. 대입문의 경우 변수의 값을 메모리에 저장하고 조건문은 메모리에 저장된 변수의 값을 사용하여 계산된 뒤 변환된 부등식은 경로 조건에 저장된다. 그러므로 경로 조건에는 조건문의 개수만큼 부등식이 저장된다.

그림 3 은 함수의 일부분으로 경로 조건 생성기 모듈이 경로 조건을 추출하는 예이다. 그림 왼쪽에 있는 코드로부터 추출된 2 개의 경로가 있으며 그 중 조건문에서 참-조건인 경로로부터 경로조건을 추출하고자 한다. 해당 경로는 하나의 대입문과 하나의 조건문으로 구성되어있다. 첫 번째 ‘x = a + 3’은 대입문으로 해당 문장을 수행한 후 메모리에 x 가 ‘a + 3’의 값을 가진다는 정보가 저장된다. 두 번째 ‘x > 7’은 조건문으로 메모리에 저장되어 있는 x 의 값이 대입되어 계산된다. 계산의 결과는 ‘a – 4 > 0’이고 이 부등식은 경로 조건으로 저장된다.



(그림 3) 경로조건 추출 예시

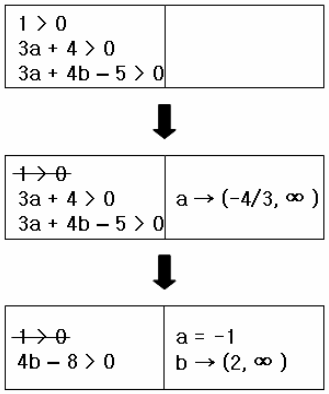
3.3 부등식 풀이기

부등식 풀이기는 경로 조건을 부등식 풀이를 통해 자유변수가 가질 수 있는 가능한 값을 선택한다. 다원 다차 연립 부등식의 일반적인 해집합을 구하는 것은 주어진 시간 내에 해결하기 힘들다. 즉, 주어진 연립 부등식으로부터 미지수가 가질 수 있는 충분한 전 영역을 구해내는 것은 어렵다. 그러나 테스트 케이스의 경우 각 미지수의 영역에서 하나의 수를 선택하는 것으로 충분하므로 다음과 같이 세 단계의 단순화된 부등식 풀이 알고리즘을 사용하여 자유변수 값을 구할 수 있다.

첫 번째 단계로 간단한 연산으로 구할 수 있는 미지수의 영역부터 구한다. 연산 결과는 변수와 그 변수의 범위의 형태로 저장되며 연산의 결과가 불능일 경우는 수행 불가능한 경로임을 의미하므로 해당 경로에 대한 테스트 케이스 입력값을 추출하지 못하기 때문에 현재 계산되고 있는 경로에 대한 연산이 종료된다. 연립 부등식 가운데 상수항만 존재하는 부등식을 검색한 다음, 이 부등식이 참인지 거짓인지를 판별한다. 판별 값이 거짓일 경우 해당 경로에 대한 계산이 종료된다. 존재하는 모든 상수항 부등식을 계산한 뒤 일원 일차 부등식을 계산하여 그 영역을 구한다. 도출된 영역이 이전의 결과 값의 영역 내에 포함이 되는 것인지 파악한 뒤 그렇다면 남은 일원 일차 부등식을 계속 해서 계산하며 그렇지 않다면 해당 경로에 대한 계산을 종료한다.

두 번째 단계는 부등식의 전체 차수를 낮춤으로써 문제의 복잡도를 단순화 하는 단계이다. 첫 번째 단계에서 구한 미지수에 대해 영역 안의 하나의 수를 대표 값으로 정한다. 그 다음 전체 연립 부등식에 대해 그 대표 값을 대입하여 전체 부등식의 차수를 낮춘다. 이렇게 차수가 낮아진 연립 부등식을 이용해 첫 번째 단계를 반복하여 다른 미지수에 대한 영역 값을 구한다.

마지막 단계는 연산의 종료 단계이다. 위의 과정을 반복하는 도중 모든 미지수에 대해 영역 값을 구할 수 있게 되었다면 해당 경로를 수행하기 위한 테스트 케이스의 입력값을 추출한 것이므로 결과를 반환하고 종료한다. 그렇지 않고 더 이상 미지수를 구할 수 없는 경우 해당 경로에 대한 연산이 종료된다.



(그림 4) 부등식 풀이 예시

그림 4 는 세 개의 경로 조건의 부등식 풀이 예이다. 이 경로를 수행하기 위한 첫 번째 단계로 상수항 부등식을 검색한다. '1 > 0'이 검색이 되며 이 상수 부등식은 항상 참이므로 미지수와 관계없이 항상 성립한다.

그 다음 차수가 낮은 1 원 1 차 부등식인 부등식 '3a + 4 > 0'이 검색되며 계산 결과 a 는 -4/3 보다 큰 값을 가져야 하며 a 의 범위는 (-4/3, ∞) 가 된다는 것을 알 수 있다. 그리고 더 이상 계산 가능한 부등식이 검색되지 않으므로 1 단계를 종료한다.

두 번째 단계에서 첫 번째 단계에서 구해진 미지수 a 의 범위 가운데 대표 값으로 -1 을 선택한다. 그리고 선택된 변수의 값을 경로 조건 집합에 대입하여 전체 차수를 감소시킨다. '3a + 4 > 0'이던 부등식이 '1 > 0'로 간소화 되며 값은 항상 참이다. '3a + 4b – 5 > 0'은 '4b - 8 > 0'로 변경되며 미지수 b 의 범위는 (2, ∞)이다.

그리고 모든 미지수에 대해 범위를 구했으므로 전체 단계를 종료한다. 위의 과정을 통해 이 경로에 대한 테스트 케이스의 입력값은 a=-1, b=3 이라는 것을 알 수 있다.

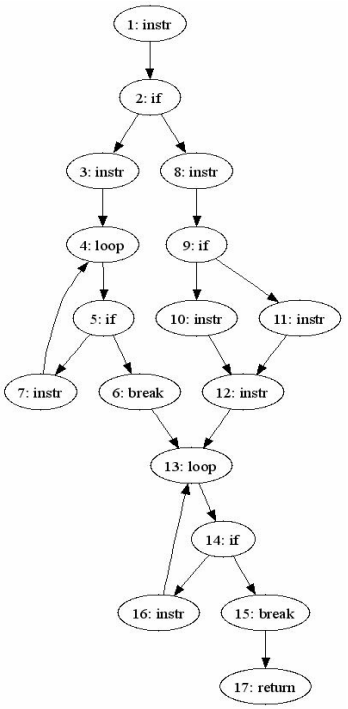
4. 적용사례

그림 5 는 스캔 모드와 현재 채널을 입력으로 받아 채널 검색을 하는 프로그램 코드의 일부이다. 입력 모드에 따라 전체 스캔을 하거나 일부에 대해서만 스캔을 하도록 분기가 주어지며 각 분기 내에서는 해당 채널을 반복적으로 검색하게 된다. 본 코드를 이용해 제어흐름도를 그리면 그림 6 과 같다.

```
1   #define FULL_SCAN          0
2   #define UBOUND             25
3   #define TV_CHANNEL         13
4   #define MAX_CHANNEL        150
5
6   void scan(int mode, int c) {
7   int c_ch;
8   int v, offset, length;
9
10  c_ch = c;
11  if (mode > FULL_SCAN) {
12      v = c_ch + 10;
13      while (c_ch < UBOUND) {
14          storech(c_ch);
15          c_ch *= 2;
16      }
17  }else{
```

```
18        v = c_ch - TV_CHANNEL;
19           if (c_ch <= MAX_CHANNEL) {
20           c_ch = 2 * c_ch - 5;
21        }else{
22           c_ch = 3 * c_ch;
23        }
24        storech(c_ch);
25     }
26     while (v < c_ch + 10) {
27        put_to_mem(v);
28        v += 25;
29        }
30     }
```

(그림 5) 채널 검색 프로그램 코드



(그림 6) 채널 검색 프로그램의 제어흐름도

제어흐름도에 의하면 본 프로그램의 순환복잡도(cyclomatic complexity)는 5 가 되어 크게 복잡하지 않은 단일 함수임을 알 수 있다. 하지만 기준을 만족하는 테스트 케이스의 개수는 총 15 가지가 된다. 이처럼 분기와 반복이 조합되어 있는 프로그램의 경우 가능한 테스트 케이스의 수가 급격히 증가되기 때문에 테스트 케이스를 작성하는데 드는 비용 역시 그에 따라서 커지게 됨을 알 수 있다.

CUTIG 를 이용해 추출한 테스트 케이스는 표 1 과 같다. 표에서 나타난 테스트 케이스의 입력값은 총 8 개인데 이는 가능한 총 15 개의 경로 중 실제로 수행이 불가능한 경우가 존재하기 때문이다. 예를 들어 26 번 라인의 반복 조건의 경우 18 번 라인을 거칠 경우 실제의 v 값은 언제나 c_ch + 10 보다 적은 값이 되기 때문에 반복 수행을 0 번 하는 것은 불가능하므로 해당 경우에 대한 2 종류의 경로는 수행이 불가능하다. CUTIG 를 활용하면 이와 같이 불가능한 경로에 대해서 테스트 케이스를 작성하기 위한 시도에 드는 노력을 줄일 수 있는 부가적인 효과가 있다.

<표 1> CUTIG 에 의해 추출된 테스트 케이스

순번	변수	값의 범위	입력값
1	mode	(-∞, 0]	-1
	c	[7, 7]	7
2	mode	(-∞, 0]	-1
	c	[-18, 18]	-18
3	mode	(-∞, 0]	-1
	c	(-∞, 0]	-1
4	mode	(0, ∞)	1
	c	[7, 7]	7
5	mode	(0, ∞)	1
	c	[13, 13]	13
6	mode	(0, ∞)	1
	c	(-∞, ∞)	0
7	mode	(0, ∞)	1
	c	(25, ∞)	26
8	mode	(0, ∞)	1
	c	(-∞, ∞)	0

5. 결론

본 연구에서는 신뢰성 있는 단위 테스트 케이스 생성의 자동화를 위한 방법을 연구하였으며 자동화 도구인 CUTIG 를 구현하였다. CUTIG 는 테스트하고자 하는 코드를 기반으로 기호실행을 사용하여 변수들간의 관계를 추출한 뒤 그 관계를 계산하여 입력 가능한 변수의 값을 선택한다.

테스트 케이스 입력값이 코드로부터 자동 생성되어 테스트 비용이 줄어들었으며 코드 기반으로 제약사항 내의 가능한 경로를 추출하여 테스트 커버리지를 높였다. CUTIG 를 사용함으로써 신뢰성 있는 테스트 입력값을 얻을 수 있으며 테스트 입력값을 추출하는데 소요되는 시간이 감소됨으로 인해 테스트 비용을 절감할 수 있다.

향후에는 코드 내에 다른 함수를 호출하는 경우에 대해 개발자로부터 함수 내에 호출되는 함수의 결과값을 고정된 값으로 지정하도록 하여 테스트 입력값의 정확성을 높이고자 한다. 또한 부등식 풀이 과정에서 고차 부등식의 풀이를 위해 고차 부등식에서 근사한 낮은 차수의 부등식을 구하는 방법을 고안하고자 한다.

참고문헌

[1] K. Sen, D. Marinov, and G. Agha. “CUTE A concolic unit testing engine for C”. In joint meeting of the European Soft. Eng. Conf. and ACM/SIGSOFT Symp. On Foundations of Soft. Eng. (ESEC/FSE’05) pages 263-272, ACM, 2005

[2] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. “Feedback-directed random test generation”. Technical Report MSR-TR-2006-125, Microsoft Research, Redmond, WA.

[3] E. F. Moore. “Gedanken-experiments on sequential machines”. Automata Studies, pages 129-153, 1956.

[4] J. C. King. “Symbolic execution and program testing.” Communications of the ACM, 19:385-394, 1976.

[5] W. Visser, C. S. pasareanu, and S. Huurshid. “Test Input Generation with Java” PathFinder, Int. Symposium on Software Testing and Analysis (ISSTA’04), 2004.

[6] CIL, www.cs.berkeley.edu/~necula/cil/