COMUT: 사용자의 의도대로 효과적인 변이를 생성할 수 있는 C 프로그램 변이 도구

Phan Duy Loc, 고봉석 ○, 김윤호, 김문주

한국과학기술원 전산학부

duyloc_1503@kaist.ac.kr, bsko5006@gmail.com, yunho.kim03@gmail.com, moonzoo@cs.kaist.ac.kr

COMUT: A Configurable Mutant Generation Tool for C programs for effective and efficient mutation analysis

Phan Duy Loc, Bongeok Ko ○, Yunho Kim, Moonzoo Kim

School of Computing, KAIST

요 약

COMUT(Configurable MUtation Tool)은 C 프로그램 변이 분석에 사용할 수 있는 변이 도구로써 사용자에게 변이 생성을 통제할 수 있는 옵션을 제공하여 변이 분석에 효과적인 변이를 많이 만들 수 있도록 한다. 본 논문에서는 SIR 벤치마크의 grep 프로그램을 이용하여 COMUT의 장점을 보였다. 첫 번째로 일반적인 상황에서 COMUT과 Milu가 grep 프로그램에 대해 효과적인 변이를 얼마나 생성하는지를 비교하였다. 실험 결과 COMUT은 효과적인 변이를 7.9% 만들었으며 Milu는 2.0% 만들었다. 또한 COMUT의 변이 생성 통제 옵션의 유용성을 보이기 위해 grep 프로그램에서 복잡성 지표가 가장 높은 *regex_compile* 함수에 대해 유저가 변이 생성을 통제하는 조건을 사용했을 때와 사용하지 않았을 때 각각 효과적인 변이를 생성하는 비율을 측정하였다. 그 결과 아무런 조건을 사용하지 않았을 때 전체 변이 219개 중 68.90%의 효과적인 변이를 생성했고 유저가 제약 조건을 사용했을 때 전체 변이 96개 중 효과적인 변이의 비율은 75.00%였다. 제약 조건을 사용함으로써 전체 생성된 변이의 개수가 줄어든데다가 효과적인 변이의 비율도 높아졌기에 COMUT으로 유저가 변이 생성을 통제할 경우 변이 분석의 비용을 줄일 수 있고 변이 분석의 효과가 상승할 수 있다.

1. 서 론

변이 분석(mutation analysis)[1]은 분석을 하고자 하는 프로그램을 구성하는 요소 예를 들어, 프로그램의 변수 값, 표현식, 연산자 등을 변이 연산자를 이용하여 변이를 만들고 주어진 테스트에 대하여 변이와 원본 프로그램의 실행 결과 차이를 분석하는 기법이다. 변이 분석을 위하여 변이를 만들 때는 변이 생성 도구를 사용하여 이미 정의된 변이 연산자를 이용하여 변이를 자동으로 생성한다.

본 논문에서는 우리가 개발한 C 프로그램 변이 도구인 COMUT (**CO**nfigurable **MU**tation **T**ool)을 소개한다. COMUT은 Agrawal 외 8인이 수행한 기존 연구[2]에 정의된 73개의 변이 연산자를 지원한다. 현재 공개되어 사용할 수 있는 C 프로그램 변이 도구는 Proteum[3]과 Milu[4]가 있다. 이들과 차별화되는 COMUT의 특징은 변이를 생성할 때 여러 설정 가능한(configurable) 옵션을 제공하여 유저가 변이 생성을 통제함으로써 효과적인 변이를 더 많이 만들 수 있게 한다.

본 논문에서는 COMUT을 평가하기 위해 SIR 벤치마크[5]의 grep 프로그램에서 복잡성 지표가 가장 높은 함수인 *regex_compile*을 대상으로 유저가 변이 생성을 통제할 경우 얼마나 많은 효과적인 변이를 만드는지를 측정했고 또한 grep 프로그램에 대해 변이를 생성했을 때 다른 변이 도구인 Milu에 비해 얼마나 많은 효과적인 변이를 생성했는지 비교하였다.

논문 구성은 다음과 같다. 2장에서는 변이 분석 기법과 변이의 종류에 대하여 설명한다. 3장에서는 COMUT에 대하여 설명하고 4장에서는 실험 환경과 실험 방법을, 5장에서는 실험 결과를 설명한다. 마지막으로 6장에서는 본 논문의 결론 및 향후 연구 방향을 제시한다.

2. 변이 분석 기법

2.1 변이 분석 기법

변이 분석 기법은 원래 프로그램의 구문, 연산자, 변수 등 프로그램을 구성하고 있는 요소를 다른 형태로 변경하여 변이(mutant)를 만들고 이를 실행하여 원본 프로그램과 변이들의 실행 결과 차이를 분석하는 기법이다. 변이를 만들 때 원본 프로그램의 모든 구성 요소를 탐색한 뒤 원본 프로그램의 패턴과 이 패턴을 어떻게 변이할 것인지 정의된 규칙을 이용하여 만들게 되는데 이러한 정의된 규칙을 변이 연산자 (mutation operator)라고 한다. 원본 프로그램과 변이 프로그램의 실행 결과 차이를 분석하기 위해 주어진 테스트를 이용한다. 테스트를 원본 프로그램과 변이 프로그램에서 실행했을 때 원본 프로그램과 변이 프로그램의 결과가 서로 다른 경우 죽은 변이(killed mutant)라고 하며 두 개의 프로그램 테스트 결과가 동일한 경우 살아있는 변이(live mutant)라고 한다.

2.2 변이 도구를 평가하는 요소

변이 도구의 질은 변이 도구가 생성한 변이가 얼마나 프로그램 분석에 효과적인지에 따라 결정된다. 변이 분석의 비용은 생성한 변이의 개수에 비례하므로 서로 다른 변이 도구가 같은 개수의 변이를 생성했을 때 효과적인 변이가 많을 수록 같은 테스트 비용을 투자하더라도 더 효과적인 분석을 할 수 있다. 변이 도구가 생성하는 불필요한 변이들에는 동일 변이(Equivalent mutant), 중복

"이 논문은 정부(과학기술정보통신부)의 재원으로 한국연구재단의 지원(NRF-2016R1A2B4008113), 정부(과학기술정보통신부)의 재원으로 한국연구재단-차세대 정보 컴퓨팅기술개발사업의 지원(NRF-2017M3C4A7068177), 과학기술정보통신부 및 정보통신기술진흥센터의 SW 중심대학지원사업의 지원(2016-0-00018), 정부(교육부)의 재원으로 한국연구재단의 지원(NRF-2017R1D1A1B03035851), KAIST High Risk High Return Project(HRHRP) 사업의 지원을 받아 수행한 연구임."

변이(Redundant mutant), 쉽게 죽일 수 있는 변이(Easy-to-kill mutant), 컴파일 할 수 없는 변이(Uncompilable mutant)들이 있고 효과적인 변이로는 죽이기 어려운 변이 (Hard-to-kill mutant)가 있다.

3. COMUT

COMUT은 C++와 Clang/LLVM 3.4를 이용하여 구현한 C 프로그램의 변이를 생성하는 변이 도구다. COMUT은 Agrawal 외 8인이 수행한 연구[2]에서 제안한 변이 연산자 중 73개를 지원한다. 현재 공개된 C 프로그램 변이 도구인 Proteum은 75개의 변이 연산자를 지원하며 Milu는 23개의 변이 연산자를 지원한다. COMUT이 다른 C 프로그램 변이 도구와 차별화 되는 점은 COMUT은 사용자가 COMUT이 제공하는 옵션을 이용하여 변이 생성을 통제할 수 있다는 점이다. COMUT이 제공하는 옵션은 다음과 같다.

- 유저가 원하는 코드의 범위를 지정하여 그 범위에 있는 코드를 변형하여 변이를 만들도록 하는 옵션
- 변이가 일어나는 코드 라인에서 생성되는 변이의 최대 개수를 제한하는 옵션
- 유저가 원하는 프로그램 요소를 선택하여 그 요소만 변형하여 변이를 만들도록 하는 옵션.

그림 1은 배열에 있는 두 값을 바꾸는 프로그램이다. 이 프로그램은 3개의 부분으로 나눌 수 있다. 첫 번째 부분은 배열 2개를 초기화 하는 부분이고 (3번째 줄부터 4번째 줄) 두 번째 부분은 배열 arr의 값을 서로 바꾸는 부분 (7번째 줄부터 9번째 줄) 마지막 부분은 배열의 값을 바꾼 후의 결과를 출력하는 부분이다. 이러한 프로그램이 있을 때 다음과 같은 시나리오를 생각할 수 있다.

- 시나리오 1. arr 의 값을 서로 바꾸는 부분을 테스팅
  이 프로그램의 가장 중요한 부분은 arr에 저장된 값을 서로 바꾸는 부분이다. 그렇기에 유저는 가장 먼저 arr의 값을 바꾸는 부분을 검증하기를 원할 수 있다. Proteum의 경우 모든 코드 라인에 대해 변이를 만들고 Milu의 경우 변이를 만들 함수를 지정해서 그 함수에 대한 변이를 만들 수 있지만 그림 1의 경우처럼 하나의 함수가 여러 부분으로 나뉘어져 있을 때 특정 부분에 대한 변이를 만들 수 없다. 이와 반대로 COMUT은 하나의 함수 내부여도 자신이 원하는 구역을 지정하여 변이를 만들 수 있다
- 시나리오 2. 특정 코드 라인에 대한 변이 과다 생성
  그림 1 의 8 번째 줄의 경우 arr[1]의 값에 정수인 temp 를 저장하는 내용이다. 이 경우 temp 대신 임의의 상수 값을 대입하여 변이를 생성할 수 있고 '=' 연산자를 바꿀 수 있는 등 많은 변이가 만들어질 수 있다. 변이가 많이 생성되는 것은 변이 분석에 유용하다고 할 수 있지만 만들어진 변이가 중복 변이나 컴파일 할 수 없는 변이 같은 효과적이지 않는 변이일 경우 변이 분석 시간만 늘어나게 된다. COMUT 은 각 코드 라인에서 생성되는 변이의 최대 개수를 제한하는 옵션과 유저가 변이 연산자가 지원하는 한도 내에서 특정 연산자를 자신이 원하는 연산자로 바꿀 수 있는 옵션 (하나의 예로, = 연산자를 +=으로만 바꾸게 하는 제약 조건을 걸 수 있다.) 을 제공하여 사용자가 변이 개수를 통제할 수 있다.

```
1.  int main(void){
2.    /*Set up array arr */
3.      int arr[2];
4.      arr[0] =2; arr[1]=1;
5.  /* Swap 2 elements of arr */
6.      int temp = arr[0];
7.      arr[0] = arr[1]
8.      arr[1] = temp;
9.      /*Print result*/
10.     print_array(arr);
11. }
```

그림 1 배열 값 반전 프로그램

4. 실험 환경 및 실험 방법

표 1. 대상 함수 정보

대상 프로그램	테스트 개수	전체 LoC	대상 함수	복잡성 지표
grep-2.0	809	5956	*regex_compile*	455
			re_match_2	253
			lex	122

먼저 COMUT이 일반적인 경우에도 변이를 잘 생성한다는 것을 보이기 위해 3장에서 설명한 COMUT의 장점을 보이기 위해 SIR 벤치마크에서 제공하는 grep 2.0버전에서 복잡성 지표(cyclomatic complexity)[7]가 높은 3개의 함수 *regex_compile*, *re_match_2*, *lex*에 대하여 COMUT과 Milu가 얼마나 많은 효과적인 변이를 생성하는지 비교하였다. 표 1은 각 함수의 복잡성 지표를 나타낸다. Proteum은 모든 코드 라인에 대해 변이를 생성하므로 적합하지 않아 제외하였다. 이 때 중복 변이의 개수는 서로 다른 변이를 컴파일 했을 때 생성된 바이너리 파일의 md5checksum 결과가 동일할 경우 동일한 바이너리 중 하나를 제외한 다른 변이들을 중복 변이라고 정의 하였고 동일 변이의 경우 원본 프로그램의 구조를 분석하여 비교하여야 하기에 측정하지 못했다. 마지막으로 죽이기 어려운 변이의 경우 죽은 변이 중에서 grep-2.0이 제공하는 809개의 테스트 중 1/10이하의 테스트들만 죽일 수 있는 변이를 죽이기 어려운 변이라고 정의하였고 그 외의 것을 죽이기 쉬운 변이라고 정의하였다.

또한 3 장에서 설명한 COMUT 의 변이 생성 통제 옵션의 유용성을 증명하기 위하여 grep 에서 가장 복잡성 지표가 높은 *regex_compile* 함수를 선택하여 제약 조건의 유무에 따라 죽이기 어려운 변이를 생성하는 비율을 측정했다. 실험에 사용한 변이 연산자는 OAAN 으로 '+, -, *, /, %' 연산자를 서로 바꾸는 변이 연산자이다. 본 논문의 실험에서는 COMUT 을 이용하여 OAAN 연산자에 % 연산자에 관해서는 변이를 만들지 않고 +와 -, *과 / 끼리의 치환만 가능하도록 제약 조건을 설정했다.

표 2. COMUT과 Milu가 생성한 불필요한 변이 개수

대상 함수	COMUT 생성 변이						Milu 생성 변이					
	컴파일 가능한 변이				컴파일 하지 못한 변이	전체 변이	컴파일 가능한 변이				컴파일 하지 못한 변이	전체 변이
	중복이 아닌 변이			중복 변이			중복이 아닌 변이			중복 변이		
	죽은 변이		살아있는 변이				죽은 변이		살아있는 변이			
	죽이기 어려운 변이	죽이기 쉬운 변이					죽이기 어려운 변이	죽이기 쉬운 변이				
regex_compile	8,413 (17.3%)	6,268 (12.9%)	16,588 (34.1%)	16,175 (33.3%)	1,138 (2.3%)	48,582 (100%)	–	–	–	–	3,067 (100%)	3,067 (100%)
re_match_2	0 (0.0%)	70 (0.2%)	31,061 (78.4%)	8,480 (21.4%)	0 (0.0%)	39,611 (100%)	–	–	–	–	1,827 (100%)	1,827 (100%)
lex	0 (0.0%)	0 (0.0%)	14,584 (77.9%)	4,148 (22.1%)	0 (0.0%)	18,732 (100%)	103 (29.7%)	55 (15.9%)	140 (40.3%)	45 (13.0%)	4 (1.2%)	347 (100%)
총합	8,413 (7.9%)	6,338 (5.9%)	62,233 (58.2%)	28,803 (26.9%)	1,138 (1.1%)	106,925 (100%)	103 (2.0%)	55 (1.0%)	140 (2.7%)	45 (0.9%)	4,898 (93.5%)	5,241 (100%)

표 3 *regex_compile*에 대한 변이 생성 결과

대상 함수	조건	죽이기 어려운 변이	전체 변이
regex_compile	제약조건 없음	151 (68.90%)	213 (100%)
	제약 조건 있음	72 (75.00%)	96 (100%)

5. 실험 결과

표 2는 변이 분석 결과를 나타낸다. COMUT과 Milu는 grep의 3개 함수에 대하여 평균 35,641.7개와 1747.0개의 변이를 만들었다. COMUT이 Milu보다 더 많은 변이 연산자를 지원하기 때문에 변이를 20.4배 만들 수 있었다. COMUT의 경우 re_match_2 함수와 lex 함수는 죽이기 어려운 변이를 만들지 못하는 것을 볼 수 있는데 중복이 아닌 변이들이 대부분 살아있는 변이였고 죽은 변이는 만들지 못했다.

COMUT에 비해 Milu는 컴파일 하지 못 하는 변이를 만드는 비율이 높다. Milu를 분석한 결과 COMUT과 달리 Milu는 변이를 생성할 때 변이를 생성하는 코드 라인 전체를 고려하지 않기 때문이다. 예를 들어 p+2와 같은 포인터와 정수의 덧셈을 할 경우 '+' 연산자는 '-'연산자 외에 다른 연산자로 바뀔 수 없지만 Milu는 이를 고려하지 않고 모든 연산자를 사용하여 변이를 생성하여 컴파일 할 수 없는 변이가 생성된다. 세 함수를 모두 통틀어 COMUT은 7.9%의 죽이기 어려운 변이를 만들었고 Milu는 2.0%의 죽이기 어려운 변이를 만들었다.

표 3은 *regex_compile*에 대한 변이 생성 결과를 나타낸다. 제한을 두지 않고 변이를 만들었을 경우 변이는 총 219개가 생성되며 아무런 제한 없이 OAAN을 사용하여 변이를 만들 경우 죽이기 어려운 변이의 비율은 68.90% (151/219)였고 제한을 두어 변이를 만들 경우 죽이기 변이의 비율은 75.00%(72/96)로 조건을 사용하지 않았을 때보다 상대적으로 8.9% 높았다. 전체 변이의 개수는 감소했지만 죽이기 어려운 변이의 비율을 증가했기에 변이 분석의 비용을 줄일 수 있고 효과적이다.

6. 결론 및 향후 연구 방향

본 논문에서는 C 프로그램 변이 도구인 COMUT을 소개했다. 먼저 COMUT이 일반적인 상황에서도 변이를 잘 만든다는 것을 보이기 위해 COMUT을 Milu와 비교했다. grep에서 복잡성 지수가 높은 3개의 함수에 대해 변이를 생성한 결과 COMUT에서는 7.9%의 죽이기 어려운 변이를, Milu는 2.0%의 죽이기 어려운 변이를 만들었다. 또한 COMUT의 변이 생성 통제 옵션이 유용하다는 것을 보이기 위하여 grep에서 복잡성 지표가 가장 높은 *regex_compile* 함수를 대상으로 변이를 생성했으며 변이를 생성하는데 제약을 설정할 경우 전체 생성되는 효과적인 변이인 죽이기 어려운 변이의 비율은 상대적으로 8.9% 상승하는 것을 확인했다. 향후 연구로는 grep을 제외한 SIR 벤치마크의 다른 프로그램과 CoREBench[7]의 프로그램들을 대상으로 COMUT이 얼마나 효과적인 변이를 생성하는 측정하고자 한다.

참고문헌

[1] Jia and Harman, An Analysis and Survey of the Development of Mutation Testing. TSE 37(5), 2011

[2] Agrawal et al., Design of mutant operators for the C programming language, SERC-TR-120-P, Purdue University, 1989

[3] Maldonado et al., Proteum: A family of tools to support specification and program testing based on mutation, Mutation testing for the new century, 2001

[4] Jia and Harman, A Customizable, Runtime-Optimized Higher Order Mutation Testing Tool for the Full C Language, TAIC PART, 2008

[5] Do et al., Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact, ESE, 10(4), 2005

[6] Mccabe, A Complexity Measure, TSE 2(4), 1976

[7] Bohme and Roychoudhury, CoREBench: Studying Complexity of Regression Errors, ISSTA,2014