심볼릭 라이브러리를 이용한 효과적인 Concolic 테스팅

박용배* 김윤호* 조준희+ 김문주*

*KAIST 전산학과
대전광역시 유성구 구성동373-1

+LG 전자 소프트웨어 역량강화 센터
서울특별시 서초구 양재대로11길 19

yongbae.park@kaist.ac.kr kimyunho@kaist.ac.kr junhee.cho@lge.com moonzoo@cs.kaist.ac.kr

요약: 본 논문에서는 concolic 테스팅의 테스팅 커버리지를 높이기 위해서, 외부 환경을 모델링한 심볼릭 라이브러리 구현에 관련된 연구내용을 다룬다. 심볼릭 실행 기법과 동적 실행 기법을 결합한 concolic 테스팅은 높은 커버리지를 달성하는 테스트 케이스를 자동으로 생성하는 장점이 있다. 그러나 대상 프로그램의 외부 환경(예. 파일 시스템, 네트웍)과 상호작용하는 라이브러리 함수(예. 파일시스템 관련 system call)가 호출된 결과가 심볼릭 변수가 아니기 때문에 심볼릭 실행의 범위가 제한되고, 그로 인해 테스팅 커버리지가 저하된다. 이러한 약점을 극복하기 위해 본 연구에서는 concolic 테스팅 도구인 CREST를 기반으로 외부 환경을 모델링한 코드(예. 파일시스템의 `fread()`)들을 심볼릭 라이브러리로 구현한다. 또한 구현한 심볼릭 라이브러리를 활용하여 `Gzip`에 CREST를 적용하는 사례연구를 통해서 심볼릭 라이브러리를 사용하면 더 높은 분기 커버리지를 달성하는 것을 실험적으로 입증하였다.

핵심어: Concolic 테스팅, 심볼릭 라이브러리, CREST

1. 서 론

테스팅은 소프트웨어의 신뢰성을 높일 수 있는 중요한 기술이지만 사람이 직접 테스트 케이스를 만들고 테스트하는 것은 많은 비용과 시간이 필요하다. 랜덤 테스팅 기법 [1]은 테스트 케이스를 자동으로 생성하지만 동일한 테스트 케이스를 많이 생성하기 때문에 프로그램의 높은 테스트 커버리지를 달성하는 것이 어렵다. 따라서 랜덤 테스팅 대신에 소프트웨어를 체계적으로 테스트하여 높은 커버리지를 달성하는 concolic 테스팅이 각광을 받고 있다.

본 연구는 미래창조과학부 및 정보통신산업진흥원의 대학 IT연구센터 지원사업(NIPA-2013-H0301-13-5004), 한국연구재단의 중견연구자지원사업-핵심연구(NRF-2012R1A2A2A01046172), 미래창조과학부의 2013년 정보통신·방송(ICT) 연구개발사업, LG전자 SW역량강화센터의 지원으로 수행되었음

Concolic(CONCreate + symbOLIC) 테스팅 [2][3][4]은 동적 실행 기법과 심볼릭 실행 기법 [5][6][7]을 함께 사용하여 프로그램의 모든 실행 경로를 탐색하고 찾은 실행 경로를 지나가는 테스트 케이스를 생성하는 기법이다. Concolic 테스팅은 체계적인 실행 경로 탐색으로 동일한 경로를 지나는 테스트 케이스를 줄이며, 프로그램의 오

류가 발생하는 특별한 조건을 만족시키는 것이 가능하다.

그러나 대상 프로그램이 외부 환경과 상호작용하는 함수를 호출하는 경우, 함수의 결과 값이 항상 동일하여 테스팅 커버리지가 낮아지는 현상이 발생한다. 이는 concolic 테스팅은 지금까지 지난 경로가 아닌 다른 경로를 탐색하기 위해서 새로운 심볼릭 변수 값을 찾지만, 외부 라이브러리 함수의 결과 값이 심볼릭 변수가 아니기 때문에 심볼릭 실행의 범위가 제한되기 때문이다.

본 연구에서는 파일 입출력을 하는 프로그램에 concolic 테스팅을 적용할 때 낮은 커버리지를 달성하는 약점을 극복하기 위해서 파일 입출력을 모델링한 심볼릭 라이브러리를 구현하였다. 그리고 구현한 심볼릭 라이브러리를 사용시 더 높은 분기 커버리지를 달성함을 실험으로 입증하였다. `Gzip` [8]에 concolic 테스팅 도구인 CREST [9]를 적용하는 사례연구를 수행한 결과, 심볼릭 라이브러리를 사용할 경우 실제 파일을 사용한 concolic 테스팅 보다 상대적으로 8.7% 더 높은 분기 커버리지를 달성하였다.

2. 연구 배경

이 장에서는 concolic 테스팅 기법과 concolic 테스팅 기법을 사용한 도구인 CREST에 대해서 설명하며, 외부 라이브러리 함수를 호출하는 프로그램에서 concolic 테스팅 기법을 적용할 경우 나타나는 문제점을 설명한다.

Concolic 테스팅 기법은 검증 대상 프로그램의 테스트 케이스 실행 과정에서 심볼릭 수행을 동시에 시행하여 테스트 실행 결과로 발생한 심볼릭 경로 조건을 분석하여 테스팅 과정에서 검사하지 못한 실

```
 1 #include <crest.h>
 2 #include <stdio.h>
 3 void main() {
 4   int x;
 5   CREST_int(x);
 6   if (x == 0)
 7     printf("x == 0");
 8   else {
 9     char buf[10];
10     FILE * f = fopen("f.txt","r");
11     fread(buf, sizeof(char),10,f);
12     if (buf[0] == 1 && buf[1] == 2)
13       printf("a file contains 12");
14   }
15 }
```

그림 1 심볼릭 변수를 설정한 예제 프로그램

행 경로를 탐색하도록 테스트 케이스를 생성하는 기법이다. Concolic 테스팅 알고리즘은 입력이 심볼릭 변수로 표현된 대상 프로그램과 종료 조건을 입력으로 받으며, 알고리즘의 수행 결과로 심볼릭 테스팅으로 생성한 테스트 입력 값을 출력 한다. 종료 조건으로는 알고리즘이 계속 진행되더라도 새로운 입력 값을 찾을 수 없는 경우 등을 사용자가 정할 수 있다.

Concolic 테스팅 기법의 알고리즘에는 다음 6 단계가 있다. 1) 프로그램에서 심볼릭 변수로 표현할 변수를 결정한다. 2) 프로그램을 수정(instrument)하여 심볼릭 경로 조건을 분석할 수 있도록 한다. 3) 수정된 프로그램의 심볼릭 변수의 입력 값을 정하고 실행하여 심볼릭 경로 조건을 구한다. 4) 심볼릭 경로 조건의 일부분을 부정하여 새로운 심볼릭 경로 조건을 찾는다. 5) 경로 조건을 만족하는 입력 값을 constraint solver를 사용하여 구한다. 6) 종료 조건을 만족하면 알고리즘을 종료하며, 만족하지 못한다면 단계 3으로 이동한다. 단계 3에서 심볼릭 변수의 입력 값은 단계 5에서 구한 값을 사용한다.

본 논문에서는 CREST가 모든 분기를 만족하는 테스트 케이스를 생성하는 과정과 파일 입출력을 사용하는 프로그램에서 발생되는 문제점을 예제로 설명한다. CREST로 C 프로그램을 concolic 테스팅 할 수 있으며, `char`, `short`, `int` 정수형 변수를 `CREST_char()`, `CREST_short()`, `CREST_int()` 함수를 이용하여 심볼릭 변수로 선언한다. 그림 1은 CREST를 사용하여 입력을 심볼릭 변수로 선언한 예제 프로그램이며 `x`(줄 4)가 위의 함수를 이용하여 심볼릭 변수로 선언되었다. CREST를 이용하여 예제 프로그램을 수정하고 실행하면, CREST는 심볼릭 변수의 값을 결정한다. 초기 실행에서는 심볼릭 변수 `x`의 값을 0으로 설정하며, 실행하면 줄 6의 조건은 참이므로 심볼릭 경로 조건 `(x==0)`이 생성된다. 이를 부정한 경로 조건은 `!(x==0)`이며, constraint solver로 조건을 만족하는 `x`의 입력 값인 1을 찾을 수 있다. 새로운 경로 조건과 경로 조건을 만족하는 입력 값을 찾았으므로 알고리즘을 반복한다. `x=1`이라는 입력 값으로 프로그램을 실행하면 줄 6의 조건은 거짓이며 심볼릭 경로 조건 `(x==1)`이 생성된다. 여기서 파일 내용이 저장되는 변수인 `buf`는 심볼릭 변수가 아니기 때문에 줄 12에 있는 조건은 심볼릭 경로 조건에 포함되지 않으며, CREST는 줄 12에 있는 조건을 만족시키는 입력 값을 만들 수 없다. 이 조건이 참이 되려면 "12"로 시작하는 `f.txt`라는 파일이 있어야 한다. 이후 CREST는 새로운 입력 값을 찾지 못하고 모든 분기에 도달하지 못한 채 종료한다.

이와 같이 파일 입출력을 사용하는 프로그램에 concolic 테스팅을 적용 시, 파일이 존재하지 않거나 파일의 내용이 고정되어 있으면 높은 커버리지를 달성하기 어렵다는 문제점이 발생한다. 따라서 높은 커버리지를 달성하기 위해서 파일의 내용을 심볼릭 변수로 변환하는 과정이 필요하며 파일을 열고 읽는 부분과 파일의 내용이 저장되는 변수를 찾아서 심볼릭 변수로 변환해야 한다.

위 과정은 프로그램의 크기가 크고 파일 입출력 관련 함수를 호출하는 부분이 많을수록 더 많은 시간과 비용이 들며, 대상 프로그램마다 이 과정을 새로 적용해야 하는 문제점이 있다. 따라서 대상 프로그램을 수정하지 않고도 높은 커버리지를 달성할 수 있는 방법이 필요하며, 본 논문에서는 파일 입력을 심볼릭 변수로 처리하는 심볼릭 라이브러리로 위 문제점을 해결하고자 한다.

3. 심볼릭 라이브러리

이 장에서는 외부 라이브러리 함수를 심볼릭 변수를 이용하여 구현한 심볼릭 라이브러리를 설명한다. 본 논문에서는 외부 라이브러리 함수 중에 많이 쓰이는 파일 입출력에 중점을 두었다. 심볼릭 라이브러리에는 파일 열기 함수 2개와 파일 읽기 함수 8개가 구현되어 있다. 파일을 쓰는 함수를 구현하지 않은 것은 파일을 쓰고 동일한 파일을 다시 읽는 경우가 드물기 때문이다.

3.1 심볼릭 파일 열기 함수

심볼릭 파일 열기 함수는 파일의 크기와 내용을 심볼릭 변수로 선언된 심볼릭 파일을 생성한다. 심볼릭 라이브러리는 GNU C 라이브러리에서 제공하는 8개의 파일 열기 함수 중 자주 쓰이는 `open()`과 `fopen()` 함수를 심볼릭 변수를 이용하여 구현하였다. 구현한 함수의 이름은 GNU C 라이브러리와 겹치지 않도록 `sOpen()`과 `sFopen()` 함수로 선언하였다. 본 논문에서는 `sFopen()`의 코드를 통해서 심볼릭 파일 열기 함수를 설명한다. 그림 2는 심볼릭 파일에 관련된 정보를 저장하기 위한 데이터 구조와 심볼릭 파일 열기 함수 `sFopen()`의 코드이다.

심볼릭 파일의 정보를 저장하기 위한 구조체

```
 1 #define FILE_SIZE 30
 2 typedef struct {
 3   int pos;
 4   int size;
 5   char buf[FILE_SIZE];
 6   FILE * stream;
 7 } SYMBOLIC_FILE;
 8 SYMBOLIC_FILE sFile;
 9 FILE* sFopen(char *name, char *type){
10   if( !checkName(name) )
11     return fopen(name, type);
12   sFile.pos = 0;
13   CREST_int(sFile.size);
14   if (sFile.size < 0)
15     sFile.size = 0;
16   else if (sFile.size > FILE_SIZE)
17     sFile.size = FILE_SIZE;
18   for (int i=0; i<sFile.size; i++)
19     CREST_char(sFile.buf[i]);
20   sFile.stream = fopen(name, "w");
21   fwrite(sFile.buf, sizeof(char),
   sFile.size, sFile.stream);
22   fseek(sFile.stream, 0, SEEK_SET);
23   return sFile.stream;
24 }
```

그림 2 sFopen() 함수

SYMBOLIC_FILE에는(줄 2~7) 파일의 읽기 위치(pos), 크기(size), 내용(buf), 실제 파일 스트림을 저장하기 위한 변수(stream)가 선언되어 있다.

sFopen()은 대상 프로그램이 실제 파일을 읽어야 할 경우에도 심볼릭 파일을 읽게 되는 현상을 막기 위해서 심볼릭 파일은 파일명에 symbolic#(#은 숫자)이 포함되는 경우만 생성된다. 파일명에 symbolic#이 포함되지 않은 경우는 외부 라이브러리의 함수를 이용하여 실제 파일을 읽는다. 줄 10에서 checkName() 함수는 파일명에 symbolic#이 포함되면 참을 리턴하는 함수이다.

심볼릭 파일 생성 과정은 심볼릭 변수로 심볼릭 파일의 크기 선언(줄 13~17), 내용을 심볼릭 변수로 선언(줄 18~19), 심볼릭 파일과 내용이 동일한 실제 파일을 생성(줄 20~22)의 3 단계로 나눌 수 있다. 심볼릭 파일의 최대 크기는 FILE_SIZE라는 매크로 변수로 제한되며(줄 16~17), 심볼릭 파일의 내용을 담는 배열 buf에 내용이 심볼릭 변수 결정된다(줄 18~19). 내용이 같은 실제 파일을 생성하는 이유는 심볼릭 라이브러리가 지원하지 않는 파일 입출력 함수(예. fwrite())를 호출하는 프로그램에 심볼릭 라이브러리를 적용하여도 정상적으로 실행되도록 하기 위함이다. 마지막으로 sFopen()은 생성한 실제 파일의 스트림 포인터를 리턴한다(줄 23).

3.2 심볼릭 파일 읽기 함수

심볼릭 파일 읽기 함수는 매개변수의 파일 스트림 포인터가 sFopen()의 리턴값과 동일하다면 심볼릭

```
 1 size_t sFread(void *d, size_t size,
   size_t c, FILE *stream) {
 2   if (stream != sFile.stream)
 3     return fread(d, size, c, stream);
 4   int i;
 5   for (i = 0; sFile.pos+size <=
   sFile.size && i < c; i++)
 6     for (int j = 0; j < size; j++)
 7       ((char*)d)[size*i+j] =
   sFile.buf[sFile.pos++];
 8   return i;
 9 }
```

그림 3 sFread() 함수

파일의 내용을 읽으며, 아닌 경우 실제 파일을 읽는다. GNU C 라이브러리에서 제공하는 파일 읽기 함수 22개 중에서 multibyte 문자 관련 함수 9개와 파일 잠금을 사용하지 않는 함수(unlocked 함수) 5개는 자주 사용되지 않기 때문에 제외하였으며, 구현한 함수는 fread(), read(), pread(), fgetc(), getc(), fgets(), getline(), getdelim()의 8개이다. 심볼릭 라이브러리에서는 각 함수명을 이름 첫 글자를 대문자로 바꾸고 s를 붙여 변경하였다.

그림 3은 sFread() 함수의 코드이다. sFread()는 파일 스트림 포인터 변수(stream)가 심볼릭 파일의 파일 스트림과 동일하다면(줄 2) 심볼릭 파일의 내용을 복사한다(줄 5~7). 복사하는 과정에서 심볼릭 파일의 읽기 위치가 읽은 만큼 전진한다. sFread()는 파일을 읽은 크기를 리턴하며, 파일의 끝에 도달하여 더 이상 읽을 내용이 없을 경우 0을 리턴한다(줄 8).

sRead()와 sPread()는 sFread()와 비슷한 방식으로 구현하였으며, sFgetc()와 sGetc()는 심볼릭 파일에서 한 글자만을 읽으며, 파일의 끝에 도달한 경우 EOF를 리턴한다. sFgets(), sGetLine(), sGetDelim()은 특별한 문자의 위치 또는 심볼릭 파일의 끝까지 심볼릭 파일을 읽는 함수이다.

4. 사례연구

본 논문에서는 구현한 심볼릭 라이브러리가 concolic 테스트에서 분기 커버리지를 향상할 수 있는지 실험적으로 증명하기 위해서, 심볼릭 라이브러리를 적용하고 concolic 테스팅하는 사례연구를 수행하였다. 실험에 사용한 프로그램은 저자에게 익숙한 Gzip 1.5를 선택하였으며, Gzip은 데이터를 압축하고 해제하는 프로그램이다. Gzip은 파일 입출력 함수인 open(), read(), write(), fopen(), fdopen(), getc(), ungetc(), fscanf()를 12회 사용한다. fdopen(), fscanf(), ungetc()는 각 1회만 사용되었다. Concolic 테스팅 도구는 bitwise 연산이 많은 Gzip을 테스트하기 위해서 bitwise 연산을 처리할 수 있는 CREST-BV [10]를 사용하였다.

표 1 실험에서 달성한 분기 커버리지

	Concolic+ 실제파일	Concolic+심볼릭 라이브러리
깊이 우선 탐색	799 (21.3%)	494 (13.2%)
제어 흐름 탐색	801 (21.4%)	846 (22.6%)
랜덤 탐색	802 (21.4%)	777 (20.7%)
랜덤 입력	748 (21.4%)	500 (13.3%)
혼성 탐색	802 (21.4%)	706 (18.8%)
모든 방식 결합	829 (22.1%)	901 (24.1%)

4.1 실험 방법

실험은 `Gzip`의 옵션을 심볼릭 변수로 선언하고 파일은 실제 파일을 이용한 concolic 테스트와 `Gzip`의 옵션을 심볼릭 변수로 선언하고 심볼릭 라이브러리를 적용한 concolic 테스트의 분기 커버리지를 비교하여 심볼릭 라이브러리가 분기 커버리지를 향상시키는지 검증한다. `Gzip`이 제공하는 옵션의 종류는 30개이며, 각 실행에서 30개 중에서 5개 이하의 옵션을 선택한다. 심볼릭 라이브러리를 이용한 concolic 테스팅에서 실제 파일을 사용한 실험에서 사용한 `Gzip` 코드에 심볼릭 라이브러리를 추가하였다.

실험에 사용한 실제 파일은 `Gzip` 내의 가장 큰 코드 파일인 `vasnprintf.c`(218 kbyte)를 사용하였다. 심볼릭 파일의 크기가 크면 프로그램의 모든 행동을 확인하기 어렵기 때문에 가능한 최소 크기인 30[1]을 `FILE_SIZE` 매크로 값으로 설정하였다.

실험에 사용한 탐색방식은 CREST-BV가 지원하는 깊이 우선 탐색(dfs), 제어 흐름 탐색(cfg), 랜덤 탐색(random), 랜덤 입력(random_input), 랜덤과 깊이 우선 탐색을 결합한 혼성 탐색(hybrid)을 모두 사용하였으며, 각 탐색 방식마다 5000회를 실행하였다. 이후 생성된 모든 테스트 케이스를 이용하여 `Gzip`을 실행하고 `gcov`로 분기 커버리지를 측정하였다.

실험에 Intel Core2 Duo 3.6 GHz 프로세서와 16GB 램이 장착되어 있고 Debian 64 bit OS가 설치된 PC에서 수행하였다.

4.2 실험 결과

실험 결과는 표 1과 같으며, 심볼릭 라이브러리를 사용한 concolic 테스팅으로 8.7%(=(901-829)/829) 더 높은 분기 커버리지를 달성하였다. 모든 방식 결합은 실험에 사용한 5개의 탐색 방식에서 생성된 모든 테스트 케이스를 사용하였을 때 달성한 분기 커버리지이다. 실제 파일을 사용한 concolic 테스팅은 322초 동안 829개의 분기에 도달하였으며, 심볼릭 라이브러리를 사용한 concolic 테스팅은 664초 동안 901개의 분기에 도달하였다. `Gzip`에 포함된 테스트 케이스를 실행하여 달성한 분기 커버리지 857 (22.9%)와 비교하였을 때, 심볼릭 라이브러리를 사용한 concolic 테스팅이 `Gzip` 개발자가 확인하지 못한 부분을 테스트하였다.

[1] 30 byte의 기준은 크기가 0 byte이고 이름이 `symbolic0`인 파일을 `Gzip`으로 압축했을 때 압축 파일의 크기이다.

5. 결 론

본 논문에서는 concolic 테스팅에서 더 높은 분기 커버리지 달성을 돕는 심볼릭 라이브러리를 구현하였다. 심볼릭 라이브러리는 파일 입력 함수를 심볼릭 변수를 사용하여 구현하였으며, 파일 입출력을 사용하는 프로그램에서 다양한 파일 입력을 생성하고 높은 분기 커버리지를 달성한다.

향후 연구 방향으로는 본 논문에서 구현하지 않은 네트워크 등의 외부와 상호작용하는 함수를 추가적으로 구현하는 것과 `Gzip`외의 다른 프로그램에 적용하는 것이 있다.

참고문헌

[1] J. Duran, “An evaluation of random testing”, IEEE Transactions on Software Engineering, 10(4):438-444, 1984.

[2] K. Sen, D. Marinov, and G. Agha, “CUTE: A Concolic Unit Testing Engine for C”, ESEC/FSE, 2005.

[3] R. Majumda, and K. Sen, “Hybrid Concolic Testing”, International Conference on Software Engineering, 2007.

[4] M. Kim, Y. Kim, and Y. Jang, “Industrial Application of Concolic Testing on Embedded Software: Case Studies”, IEEE International Conference on Software Testing, Verification and Validation Industrial Track, 2012.

[5] Y. Kim, M. Kim, and Y. Jang, “Concolic Testing on Embedded Software - Case Studies on Mobile Platform Programs”, ACM SIGSOFT Foundation of Software Engineering (FSE) Industrial track, 2011.

[6] J. King, “Symbolic execution and program testing” Communications of the ACM, 19(7):385-394, 1976.

[7] L. A. Clarke, “A system to generate test data and symbolically execute programs”, IEEE Transactions on Software Engineering, 2(3):215-222, 1976.

[8] Gzip, http://www.gnu.org/software/gzip/.

[9] CREST, http://code.google.com/p/crest/.

[10] Y. Kim, M. Kim, and Y. Jang, “CREST-BV: An Improved Concolic Testing Technique Supporting Bitwise Operations for Embedded Software”, Journal of KIISE: Software and Applications, 40(2):90-98, 2013.