ISSN 2383-630X(Print) / ISSN 2383-6296(Online)
Journal of KIISE, Vol. 45, No. 10, pp. 1071-1079, 2018. 10
https://doi.org/10.5626/JOK.2018.45.10.1071

Concolic 테스팅 도구 CROWN의 적용 가능성 및 사용성 개선 연구

(Improving Applicability and Usability of a Concolic Testing Tool CROWN)

김 현 우 † 김 윤 호 †† 김 문 주 †††
(Hyunwoo Kim) (Yunho Kim) (Moonzoo Kim)

요 약 본 연구에서는 Concolic 테스팅 기법의 적용 가능성 및 사용성을 개선하기 위한 CROWN(Concolic testing for Real-wOrld software aNalysis) 도구 확장에 대해 소개한다. 기존 CROWN 도구는 적용 대상 개발 환경이 Linux 환경으로 제한되어 있고 Concolic 테스팅 실행 결과를 쉽게 확인하기 어려워 사용자가 테스트 결과를 분석하기 어려웠다. 본 연구에서는 먼저 Concolic 테스팅의 적용 플랫폼을 넓히기 위해 CROWN 도구를 Windows 개발 환경에서 사용할 수 있게 확장하고 Concolic 테스팅 결과를 사용자가 쉽게 분석할 수 있도록 사용자 인터페이스를 개선하였다.

키워드: Concolic 테스팅, 소프트웨어 테스트 자동 생성, CROWN 도구 개선

Abstract The paper presents an extension of the Concolic testing tool CROWN(Concolic testing for Real-wOrld softWare aNalysis) for improving the applicability and usability of Concolic testing. The existing CROWN tool is limited to Linux platforms and the Concolic testing results generated from CROWN are hard for users to understand. We extend CROWN to run on Windows OS to increase the running platforms and improve user interface with CROWN and its applicability to help users analyze Concolic testing results in an easier way.

Keywords: Concolic testing, automated software test generation, CROWN tool improvement

· 이 논문은 과학기술정보통신부의 재원으로 한국연구재단의 지원(NRF-2016R1A2B4008113), 과학기술정보통신부의 재원으로 한국연구재단-차세대 정보컴퓨팅기술개발사업의 지원(NRF-2017M3C4A7068177), 교육부의 재원으로 한국연구재단의 지원(NRF-2017R1D1A1B03035851)을 받아 수행한 연구임
· 이 논문은 2018 한국 소프트웨어공학 학술대회에서 'Concolic 테스팅 도구 CREST의 사용자 친화성 향상 연구: Windows OS 로의 포팅과 개선된 CREST UI을 통한 CREST 활용 및 분기 커버리지 분석 작업의 효율 증가'의 제목으로 발표된 논문을 확장한 것임

† 비 회 원 : 한국과학기술원 전산학부
hyunooudy@gmail.com
†† 정 회 원 : 한국과학기술원 전산학부 연구 교수
yunho.kim03@gmail.com
††† 종신회원 : 한국과학기술원 전산학부 교수(KAIST)
moonzoo@cs.kaist.ac.kr
(Corresponding author임)

논문접수 : 2018년 3월 26일
(Received 26 March 2018)
논문수정 : 2018년 7월 11일
(Revised 11 July 2018)
심사완료 : 2018년 7월 13일
(Accepted 13 July 2018)

정보과학회논문지 제45권 제10호(2018. 10)

1. 서 론

소프트웨어의 늘어나는 필요성과 요구사항으로 인해 그 복잡성이 높아짐에 따라 자동화 테스팅에 대한 중요성 역시 높아지고 있다. 테스팅의 목적은 소프트웨어 내부에 존재하는 에러를 검출하는 것으로, 이를 달성하기 위해서 소프트웨어의 가능한 많은 실행 경로의 탐색을 필요로 한다.

일반적으로 널리 알려진 테스팅 기법으로는 퍼징 기법이 있다. 이 기법은 랜덤 테스트들을 만들어 실행하기 때문에 타겟 소프트웨어에 대한 이해나 배경이 없어도 손쉬운 사용이 가능하다. 하지만 이 기법은 랜덤에 기반한 특성 때문에 꼼꼼하게 소프트웨어의 실행 경로를 탐색하는 것에 한계가 있으며, 소프트웨어의 복잡성이 약간만 높아져도 탐색하는 실행 경로의 범위는 기하급수적으로 떨어진다. Concolic 기법[1]은 이러한 퍼징 기법의 한계를 극복한 테스팅 기법으로, 활발히 연구가 진행되고 있는 동시에 많은 관심을 받고 있는 기법이다. Concolic 기법은 초기 테스트 케이스로 소프트웨어를 실행한 뒤, 실행한 경로를 기반으로 새로운 경로를 탐색하는 테스트를 생성하고, 이 테스트로 타겟 소프트웨어를 다시 실행하는 일련의 과정을 반복한다. 이론적으로 Concolic 기법을 사용하면 타겟 소프트웨어의 모든 실행 경로를 탐색하는 것이 가능하다.

CROWN(Concolic testing for Real-wOrld software aNalysis)[2]은 Concolic 기법을 사용한 C 프로그램 대상 테스팅 도구이다. CROWN을 사용하기 위한 사전작업으로 사용자는 타겟 소스 코드에 추가 코드를 심어주어야 한다. 이 작업은 타겟 소스 코드에 대한 이해와 배경을 필요로 하며, 이 작업에 따라 탐색할 수 있는 경로의 범위나 탐색의 우선순위가 달라진다. 따라서 CROWN을 사용한 결과가 사용자의 의도와 다를 경우, 원인을 분석하여 타겟 소스 코드에 개선한 추가 코드를 심어주고, 다시 결과를 확인하는 작업을 반복해나가야 한다. 즉, CROWN사용 결과가 사용자의 의도와 다른 원인을 빠르고 정확하게 분석하는 것은 제한 된 시간 내에 성공적인 테스팅을 위해서 반드시 필요한 사항 중 하나라고 볼 수 있다.

CROWN은 결과를 분석하는데 필요한 노력과 시간을 줄여서 효율적인 사용이 가능케 하고 생산성을 높이고자 하였다. 본 논문에서 설명하는CROWN의 개선 작업은 구체적으로 다음과 같다. 첫 번째로, 국내 산업체에서 가장 많이 사용되는 개발 환경 OS 중 하나인 Windows OS에서 사용이 가능하도록 포팅을 수행하였다. 두 번째로, 테스팅 결과 분석에 필요한 정보들을 사용자가 쉽게 이해할 수 있는 형태로 추가 제공하여 결과 분석에 필요한 추가 비용을 크게 줄이고자 했다.

2. 관련 연구

소프트웨어의 규모가 증가함에 자동화 테스팅에 대한 중요성 역시 높아지고 있다. 간단하지만 효과가 낮은 자동화 테스팅 방법으로 임의의 입력 값을 생성하는 퍼징 기법[3]이 있다. 이 기법은 생성 된 입력들이 대부분 프로그램의 동일한 경로를 비효율적으로 반복하는 문제를 갖고 있다. Symbolic execution[4,5]은 이러한 문제점을 해결한 기법이다. 이 기법은 각각 유일한 경로를 실행하는 입력 값의 생성이 가능하지만 프로그램 규모가 커질 경우, 경로 폭발(path explosion) 문제[6]가 존재한다. Concolic 기법은 이 한계를 해결하고자 한 방법으로 실제 산업의 다양한 분야의 소프트웨어를 대상으로 활용[7-9]되고 있는 상태이다. Concolic 기법 도구 중 하나인 CREST[10]는 Concolic 기법을 사용한 연구에 활발히 사용[11-13]되고 있는 대표적인 도구이다.

Concolic 기법을 사용한 도구는 타겟 프로그램에 따라 다양하며, 그 중 C 프로그램을 대상으로 삼는 도구는 CREST 외에도 CUTE[1], DART[14], KLEE[15], EGT[16], EXE[17]가 존재한다. 이 중 CREST와 KLEE 도구만 현재 배포되고 있으며, 사용 가능한 도구이다. 더불어 두 도구는 모두 Windows OS를 지원 하지 않기 때문에 현재 Windows OS에서 실행 가능한 C 프로그램 용 Concolic 기법 도구는 존재하지 않는다고 볼 수 있다.

3. Concolic 테스팅 소개

3.1 Concolic 테스팅 기법 및 예제

Concolic 테스팅은 실제 실행(concrete execution)된 경로를 따라 기호 실행(symbolic execution)을 수행한다. 실제 실행에서 입력 변수에 값을 넣고, 타겟 프로그램의 한 경로를 실행시킨다. 기호 실행은 입력 변수를 심볼로 설정하고, 실제 실행한 경로에 도달하기 위해 입력 변수가 만족해야하는 심볼릭 경로 수식(symbolic path formula)을 수집한다. 수집한 심볼릭 경로 수식을 수정하여 아직 실행하지 못한 경로에 도달하기 위한 새로운 심볼릭 경로 수식(next formula)을 만들어낸다. 심볼릭 경로 수식을 수정하는 방법은 여러가지가 존재하며 기본적인 깊이 우선 탐색 방법의 경우 마지막 조건식에 부정을 취해서 새로운 심볼릭 경로 수식을 획득한다. Z3 solver[18]에 새롭게 구한 심볼릭 경로 수식을 전달하면 심볼(입력 변수)의 해(값)를 구할 수 있다. 새롭게 구한 입력 변수 값으로 다시 타겟 프로그램을 실행시키고, 또 다른 입력 변수의 값을 추출하는 일련의 과정을 반복하면서 프로그램의 경로들을 탐색하게 된다.

그림 1은 Concolic 테스팅을 적용할 타겟 소스 코드

```
1    #include <crown.h>
2    void main () {
3         int a, b, res;
4         SYM_int(a);
5         SYM_int(b);
6         if(a == 1) {
7              if(b == 1) res = 1;
8              else res = 2;
9         }
10        else {
11             if(b == 2) res = 3;
12             else res = 4;
13        }
14   }
```

그림 1 예제 코드 1

Fig. 1 Example code 1

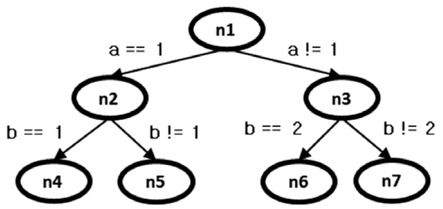

그림 2 그림 1의 분기 그래프

Fig. 2 Branch graph of Fig. 1

이고, 그림 2는 소스 코드의 분기 지점들을 그래프로 표현한 것이다. 그림 2의 노드(n1,n2,…,n7)들은 각 분기 지점을 나타낸다. Concolic 테스팅은 모든 가능한 실행 경로의 탐색을 목표로 한다. 그림 1에서 4, 5라인의 SYM_int 함수는 인자로 받은 변수를 심볼릭 변수로 선언하는 기능을 한다. 심볼릭 변수로 선언되면 기호 실행에서 해당 변수는 심볼로 간주된다.

표 1은 concolic 테스팅 과정에서 생성한 테스트와 해당 테스트의 심볼릭 경로 수식, 새로운 심볼릭 경로 수식을 나타낸 것이다. 처음에 심볼릭 변수에 임의의 초기 값(여기선 a=0,b=0)을 부여하고 타겟 프로그램을 실행한다. 이 과정에서 심볼릭 경로 수식을 수집한다. 프로그램은 분기 지점 n1, n3, n7을 차례대로 수행하고 심볼릭 경로 수식을 는 a!=1 && b!=2가 된다. 이후, 심볼릭 경로 수식을 의 마지막 식(b!=2)에 부정하여 심볼릭 경로 수식을 와 겹치지 않는 새로운 경로 수식(a!=1 && b==2)를 생성한다. 다음으로 새로 생성된 경로 수식을 Z3 solver에 전달하여 경로 수식을 만족하는 테스트(여기선 a=0,b=2)를 획득한다. 이제 획득한 테스트로 타겟 프로그램을 실행하는 과정을 다시 반복한다. 이러한 과정을 통해 표 1처럼 총 4개의 테스트를 생성할 수 있고 모든 분기 지점(n1,n2,…,n7)을 성공적으로 탐색할 수 있다.

3.2 CROWN을 사용한 Concolic 테스팅 적용 및 실행 절차

CROWN을 사용하여 Concolic 테스팅을 적용 및 실행하기 위해서는 크게 다음 3단계 작업을 수행해야 한다.

첫 번째 단계는 심볼릭 변수 선언이다. 소스 코드에서 심볼릭 변수로 사용할 입력 변수를 SYM_<type> 함수를 사용하여 선언하는 작업이다. 이 심볼은 심볼릭 경로 수식에서 사용된다. 이 때, 인자로 심볼릭 선언할 변수를 넘겨준다.

두 번째 단계에서는 기호 실행을 위한 탐침 함수를 삽입한다. 심볼릭 경로 수식 등을 기록하기 위한 probe들을 삽입해주는 작업으로, CROWN 내부 도구(cilly)에 의해 정적으로 자동 삽입하여 확장자가 .cil.c 인 파일을 생성한다.

마지막으로 탐침 함수가 삽입된 소스 코드 파일을 컴파일하여 Concolic 테스팅을 수행한다. .cil.c 파일을 컴파일하여 실행파일을 만들고, CROWN을 실행하면 CROWN에서 실행파일에 대해 실제 실행과 기호 실행을 반복적으로 수행하면서 경로를 탐색하게 된다.

.cil.c 파일은 탐침 단계를 마치고 생성되는 파일이다. 타겟 소스에 존재했던 복합 조건문들은 .cil.c 파일에서 단일 조건문들로 변환된다. 더불어 각 단일 조건문, 즉 분기 지점마다 _CrownBranch 탐침 함수가 삽입된다. 해당 probe는 여러 개의 인자를 가지며, 이 중 2번째 인자를 분기 id라고 정의한다. 분기 id는 분기를 식별하기 위해 사용되는 id이다. CROWN은 실행 중에 실행된 분기 id를 “coverage” 파일에 기록하는데, 사용자는 기록된 분기 id와 동일한 분기 id를 .cil.c 파일에서 찾아 어느 위치의 분기가 달성/미달성 되었는지 여부를 확인할 수 있다.

그림 3은 타겟 소스 코드이고, 그림 4는 탐침이 삽입된 코드이다. 편의상 필요한 부분만 간략화하여 탐침이 삽입된 코드를 표현하였다. 그림 3의 6라인에 해당하는 복합 조건문이 그림 4에서 단일 조건문으로 변환되었고,

표 1 Concolic 테스팅에서 생성한 그림 1의 테스트 케이스

Table 1 Generated test cases of Fig 1 in Concolic testing

No	Input value	Symbolic path formula	Next formula
1	a=0, b=0	a !=1 && b!=2	a!=1 && b==2
2	a=0, b=2	a !=1 && b==2	a==1
3	a=1, b=0	a==1 && b!=1	a==1 && b==1
4	a=1, b=1	a==1 && b==1	none

```
1    #include <crown.h>
2    void main () {
3        int a, b, res;
4        SYM_int(a);
5        SYM_int(b);
6        if( a==1 && b==1 ) res = 1;
7        else res = 2;
8    }
```

그림 3 예제 코드 2

Fig. 3 Example code 2

```
1    …
2        if ( a == 1) {
3            __CrownBranch(6, 2, …);
4            if(b == 1) {
5                __CrownBranch(11, 3, …);
6                res = 1;
7            }
8            else
9                __CrownBranch(12, 4, …);
10       }
11       else {
12           __CrownBranch(7, 5, …);
13           res = 2;
14       }
15   …
```

그림 4 그림 3의 탐침이 삽입된 코드

Fig. 4 Instrumented code of Fig. 3

각 분기 지점마다 총 4개의 _CrownBranch 탐침 함수가 삽입되었다. 각각 분기 id 2, 3, 4, 5를 갖는 것을 확인할 수 있다.

4. CROWN 의 향상 기능 소개

CROWN[2]은 CREST[10]을 확장하여 개발되었으며, CROWN이 CREST와 비교하여 개선한 기능을 설명한다. 차례대로 국내 산업체의 개발 환경에 맞추어 Windows OS로 포팅한 작업과 산업체에서의 사용을 고려하여 생산성을 높이고자 개선한 UI를 소개한다.

4.1 Windows 포팅

Windows OS는 개발 환경에서 가장 많이 사용되는 OS 중 하나이다. 하지만 현재 사용 가능한 C대상 Concolic 기법 도구 중 Windows OS를 지원하는 도구는 전무하기 때문에 주 개발 환경이 Windows OS인 많은 기업에서는 C 대상 Concolic 기법도구의 보급과 적용이 어려운 실정이다.

따라서 현재 Windows OS에서 동작하는 소스 코드에 C대상 Concolic 기법 도구를 적용하기 위해서는 소스 코드를 해당 도구가 지원하는 환경에서 동작하도록 포팅하는 작업을 거쳐야 한다. 소스 코드와 빌드 관계가 복잡할수록 포팅 비용은 비례하기 때문에 기업 단위에서 다루는 소프트웨어의 경우 포팅 작업은 큰 부담이 될 수 있다. 더불어 Windows OS에서만 지원하는 코드 문법과 헤더 파일들이 존재하기 때문에 Windows OS에서 동작하는 것과 완벽히 동일하도록 포팅을 하는 것은 매우 어려운 작업이다. 또한 프로그램을 실행할 때에도 OS간 서로 다른 정책(policy)을 따르므로, 확인한 테스팅 결과가 Windows OS에서는 다른 결과로 이어질 수 있다.

본 논문에서는 C대상 Concolic 기법 도구가 개발 환경에서 널리 사용되는 Windows OS를 지원하지 못하여 초래되는 문제와 비효율성을 없애고자, CROWN이 Windows OS를 지원하도록 포팅하는 작업을 수행했다.

그림 5는 Windows OS에서 사용 가능한 CROWN의 실행 로직이다. 표 2는 Front-end, Middle-end, Back-end 별로 분류한 포팅을 위해 변경한 주요 구문이다.

다음과 같은 Visual Studio 컴파일러 확장 코드는 CIL이 파싱하지 못하는 구문이다. 이 구문들은 실제 프

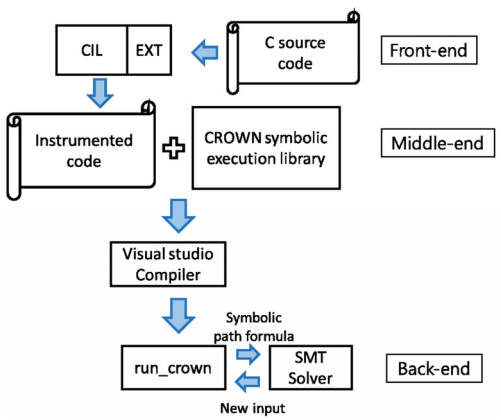

그림 5 Windows 버전 CROWN의 실행 로직

Fig. 5 Execution logic of Windows version CROWN

표 2 포팅 시 변경된 주요 구문

Table 2 Major changed statements for porting

Component	Changed history
Front-end	1 Adding statements to remove codes CIL can't parse. 2 Removing generating GCC extensions Visual Studio compiler doesn't support.
Middle-end	1 Removing GCC extensions in CROWN code Visual Studio compiler doesn't support. 2 Replacing POSIX libraries Visual Studio doesn't support.
Back-end	1 Replacing POSIX libraries Visual Studio doesn't support. 2 Replacing output formats Visual Studio doesn't support.

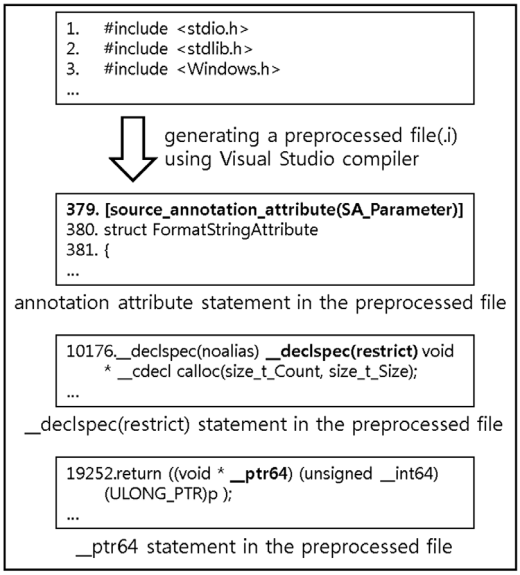

그림 6 CIL에 의해 파싱 불가능한 구문 예제
Fig. 6 Code example CIL can't parse

로그램의 의미에 영향을 주지 않고, 컴파일러 최적화 힌트로만 사용되는 구문이기 때문에 안전하게 제거가 가능하다. 그림 6은 이러한 구문들의 예제를 나타낸다.

1 Standard Annotation Language (SAL)
2 __declspec(restrict) 구문
3 __ptr64 구문

Visual studio 컴파일러가 지원하지 않는 CROWN 코드 안의 GCC 확장 구문을 제거한다. CROWN 소스 코드 중 visual studio 컴파일러가 지원하지 않는 키워드 __attribute__를 생성하는 매크로와 __SKIP 구문을 제거한다.

Visual Studio가 지원하지 않는 POSIX 라이브러리를 동일한 의미를 갖는 Windows 라이브러리로 대체한다. CROWN 소스 코드 중 visual studio 컴파일러가 지원하지 않는 헤더파일을 제거하고, 해당 헤더파일에 정의된 자료구조를 Visual studio 컴파일러가 지원 가능한 자료구조로 변경한다. 표 3은 포팅 전에 사용했던 자료구조와 포팅 후, 대체한 자료구조를 나타낸다. 표 4는 포팅 전에 사용했던 함수와 포팅 후, 대체한 함수를 나타낸다.

표 3 포팅 전 후로 사용된 자료구조
Table 3 Used data structure before and after porting

POSIX data structure	Replaced data structure for visual studio compiler
hash_map	unordered_map

표 4 포팅 전 후로 사용된 함수
Table 4 Used function before and after porting

POSIX function	Replaced function for visual studio compiler
Mkdir	CreateDirectory
Gettimeofday	User defined

4.2 달성/미달성 분기정보 UI 향상

CROWN에서는 print_execution 명령어의 결과를 사용자 친화적으로 변경하였고, 커버리지 결과 분석에 필요한 추가 정보가 담긴 파일 "new_coverage"를 생성하는 새로운 명령어 gen_new_cov를 추가하였다.

CROWN은 타겟 코드에 바로 적용하는 것이 아니라, 사용자에 의해 세팅된 타겟 코드에 적용하며, 당연히 이 세팅에 따라 생성되는 테스트들이 달라진다. 이 때문에 CROWN을 적용해서 높은 분기 커버리지를 달성하는 테스트들을 생성하려면 타겟 코드에 적절한 세팅을 해주어야 한다. CROWN 실행 후 커버리지가 낮은 원인을 분석하고, 분석한 결과를 피드백으로 타겟 코드를 새로 세팅하면서 점차적으로 분기 커버리지를 높여갈 수 있다. 그림 7은 이 과정을 도식화하여 표현한 것이다.

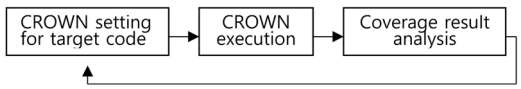

그림 7 달성 커버리지 향상을 위한 CROWN 적용 사이클
Fig. 7 Cycle of applying CROWN for coverage improvement

따라서 CROWN에서는 커버리지 결과 분석에 도움이 되는 정보들을 사용자에게 알기 쉬운 형태로 추가 제공함으로써 커버리지 결과에 대한 정확한 분석 수행과 분석에 소요되는 시간을 줄이고자 하였다.

4.2.1 Concolic 테스팅 기호 실행 결과 출력 개선

Concolic 테스팅의 기호 실행 결과 출력을 개선하였다. print_execution은 CROWN으로 타겟 프로그램을 실행시킨 후 사용 가능한 명령어로, 타겟 프로그램을 실행했을 때의 심볼릭 정보들을 출력해준다. 심볼릭 정보에는 심볼릭 변수가 갖는 값, 심볼릭 경로 수식, 실행된 분기 id가 출력된다. 이러한 정보들을 바탕으로, 각각의 심볼릭 변수가 갖는 값과 그에 대응되는 심볼릭 경로 수식을 확인할 수 있다. 본 절에서는 개선 전의 print_execution 출력과 그로 인한 문제점 및 개선한 사항들을 차례대로 소개한다.

그림 8은 심볼릭 변수 선언을 마친 예제 소스 코드이다. 이 코드에 대해 CROWN 실행 후, 개선 전의 print_execution 출력 결과는 그림 9와 같다. 빈 줄을 기준으로 세 부분으로 분리되어 있으며 차례대로 심볼

```
1   #include <stdio.h>
2   #include <crown.h>
3
4   void main(){
5   int var_a, var_b;
6
7   SYM_int(var_a);
8   SYM_int(var_b);
9
10  if((double)var_a > 10.0)
11      printf("reach 1");
12  if(var_b < -10)
13      printf("reach 2");
14  }
```

그림 8 예제 코드 3

Fig. 8 Example code 3

```
1   (= x0 32770)
2   (= x1 0)
3
4   (> (s_cast[64] x0) 10)
5   (! (< x1 -10))
6
7   -1
8   3
9   7
10  -2
```

그림 9 그림 8의 개선 전print_execution 출력

Fig. 9 Outputs of print_execution before refinement for Fig. 8

릭 변수가 갖는 값, 심볼릭 경로 수식, 실행 된 분기 id 정보를 포함한다. 즉, 1-2 라인은 심볼릭 변수가 갖는 값 정보, 4-5 라인은 심볼릭 경로 수식 정보, 7-10 라인은 실행 된 분기 id 정보를 나타낸다. 1, 2라인에서 x0, x1은 각각 심볼릭 변수 var_a, var_b를 의미하며 차례대로 32770, 0의 값을 입력으로 갖는 것을 의미한다. 4-5 라인은 심볼릭 경로 수식이며 전위 표기법으로 출력된다. 특히, 4라인의 s_cast[64] 구문은 double형으로 형 변환이 발생했음을 나타낸다. 7-10 라인은 실행 된 분기 id들이며 -1은 프로그램 실행 중 함수(main)의 진입을 뜻하고, -2는 실행 중 함수(main)에서 빠져나감을 의미한다. .cil.c 소스 코드에서 분기 id를 찾아 비교해본 결과, 분기 id 3은 분기 (var_a > 10.0)을 의미하고, 분기 id 7은 분기 !(var_b < -10)을 의미함을 확인했다.

다음 항목들은 개선 전의 print_execution 출력에서 확인할 수 있는 문제점들이다.

1. 출력되는 심볼릭 변수명과 선언 된 심볼릭 변수명의 불일치.
   print_execution의 출력에서 심볼릭 변수명은 x0, x1, x2, …과 같이 가명이 사용되기 때문에 어떤 심볼릭 변수가 무슨 값을 가지는지 확인하기 위해서는 실제 변수명과 출력에 사용된 변수명을 매칭시키는 추가 작업이 필요하다. 이를 위해 print_execution 출력 외에 소스 코드를 추가로 확인해야 하는데 이는 분석 작업의 효율을 떨어뜨린다. 소스 코드가 복잡하고 사용 된 심볼릭 변수 개수가 많을수록 매칭 작업은 더 어려워진다.
2. 전위(pre-order) 형태의 심볼릭 경로 수식 출력으로 인한 해석의 어려움.
   일반 사용자에게 익숙한 중위 표현(infix notation) 형태가 아닌 전위 표현(prefix notation) 형태로 심볼릭 경로 수식이 출력되기 때문에 이를 해석하는 작업이 추가로 필요하다. 이는 분석 작업의 효율을 떨어뜨리게 된다. 심볼릭 경로 수식의 길이가 길거나 복잡할수록 해석에 더 많은 어려움이 존재한다.
3. 형 변환 표기 시, 변환된 자료형이 아닌 자료형의 부호(sign)와 크기가 출력.
   심볼릭 경로 수식에 형 변환(casting)이 포함된 경우, 형 변환된 자료형(int, char, short, …)이 아닌 u_cast[16], u_cast[32], s_cast[8], …와 같이 자료형의 부호와 크기가 출력된다. 표 5는 자료형과 해당 자료형으로 형 변환 시 출력되는 구문을 나타낸다.

표 5 print_execution에서의 형 변환 출력구문

Table 5 Output format of casting code in print_execution

Data type to cast	Output format in print_execution
char	s_cast[8]
unsigned char	u_cast[8]
int	s_cast[32]
unsigned int	u_cast[32]
float	s_cast[32]
double	s_cast[64]

   이는 사용자가 출력된 형 변환 출력 형식을 확인했을 때, 형 변환 된 자료형 식별이 어렵게 한다. 특히, 부호와 크기가 같은 자료형(int와 float)은 구분이 불가능하다.
4. 위치 정보(line, file) 부재
   선언된 심볼릭 변수의 위치, 심볼릭 경로 수식의 위치, 실행된 분기 id의 위치 정보가 출력되지 않기 때문에 사용자가 대응되는 부분을 소스 코드에서 확인하고자 할 때, 어려움이 존재한다.

개선된 사항은 다음 1 - 4항목과 같다.

1. 출력되는 심볼릭 변수명을 가명이 아닌 실제 변수명 사용.
2. 출력되는 경로 조건을 기존 전위(prefix notation)방식에서 사용자에게 친숙한 중위(infix notation) 방식으로 변경.
3. 형 변환 표기 시, 실제 형 변환된 자료형으로 출력.
4. 각 정보 출력 시, 위치 정보(line, file)를 함께 표기하는 방식으로 변경.

```
1    Symbolic variables & input values
2    (var_a = 32770)   [ Line: 7, File: main.c ]
3    (var_b = 0)       [ Line: 8, File: main.c ]
4
5    Symbolic path for the input
6    ((double) var_a > 10) [ Line: 10, File: main.c ]
7    ! (var_b < -10)       [ Line: 12, File: main.c ]
8
9    Sequence of reached branch ids
10   -1        [Function enters]
11   3         [ Line: 10, File: main.c ]
12   7         [ Line: 12, File: main.c ]
13   -2        [Function exits]
```

그림 10 그림 8에 대한 개선 후 print_execution 출력
Fig. 10 Refined print_execution output of Fig. 8

그림 10은 CROWN에서 실행한 그림 8에 대한 개선 후 print_execution 출력 결과이다. 그림 10에서 1, 5, 9 라인에 각각 심볼릭 변수가 갖는 값 정보, 심볼릭 경로 수식, 실행 된 분기 id 정보를 구분 짓는 메시지가 추가되었다. 심볼릭 변수가 갖는 값 정보(2-3 라인)에서 심볼릭 변수명 표기 시 가명(x0, x1)이 아닌 실제 변수명(var_a, var_b)이 사용되었으며 중위 표기법으로 출력 형식이 변경됐다. 입력 값 정보와 더불어 심볼릭 변수가 선언 된 위치 정보 (라인, 파일)가 함께 표기된다. 심볼릭 경로 수식 정보(6-7 라인)에서는 수식이 중위 표기법으로 표기되며, 형 변환 구문 출력 시 형 변환된 자료형(double)이 표기된다. 수식이 위치한 정보도 함께 표기된다.

4.2.2 Concolic 테스팅 커버리지 출력 개선

Concolic 테스팅 커버리지의 출력을 보기 좋게 개선하였다. 기존에도 분기 정보가 담긴 파일 “coverage”이 생성되었지만, 이 파일에는 달성한 분기 id만 나열되어 있어서 사용자가 .cil.c 소스 코드와 분기 id를 비교하면서 달성한 분기와 달성하지 못한 분기를 가려내는 추가 작업을 수행해야 했다. 또한 coverage 파일에는 소스 코드, 함수 별로 분기 id가 나누어진 것이 아니라서 어느 소스 코드, 함수에서 얼마 정도의 분기를 달성했는지에 대한 동향을 한 눈에 파악하는데 한계가 존재했다. 분석 작업에 착수할 때, 달성한 분기의 동향을 파악한 뒤, 미달성 분기가 밀집되어 있는 부분부터 분석을 시작하기 때문에 달성 분기에 대한 동향은 반드시 파악해야 하는 정보이다.

이를 위해서 new_coverage 파일에 분기 상세 정보를 추가하고, 분기의 동향을 파악할 수 있는 분기 요약 정보를 추가했다. 따라서, 기존 coverage 파일의 정보 부족으로 인한 추가 수행 작업을 줄일 수 있었다. 즉, 사용자는 더 이상 .cil.c 소스 코드에서 분기 id를 탐색하는 작업을 할 필요가 없으며, 분기 요약 정보로 인해 분기 동향도 쉽게 파악이 가능하다. 즉, 분석에 대한 부담을 크게 줄였다고 볼 수 있다.

gen_new_cov 구현은 크게 두 부분으로 분류된다. 첫 번째 부분은 입/출력 부분이고, 두번째 부분은 파싱 부분이다. 입/출력 부분은 실제 사용자가 실행하는 부분으로, gen_new_cov 스크립트 자체가 입/출력 부분에 해당한다. gen_new_cov는 명령어를 실행하는 디렉토리에 존재하는 .cil.c 파일들을 파싱을 담당하는 프로그램에 넘겨주고, new_coverage 파일 작성에 필요한 정보들을 받는다. 이 정보는 (함수명, 함수가 포함된 파일 이름) 쌍들, (분기 id, 분기가 존재하는 라인, 분기의 expression) 쌍들을 포함한다. 이후, 받은 정보를 출력 형식에 맞게 가공하여 new_coverage 파일에 출력하는 기능을 하고 있다.

파싱 부분은 clang/llvm[19]을 사용하여 구현되었으며, .cil.c 파일에서 분기에 관련 된 정보와 파일이름, 함수 이름 쌍들을 파싱하는 역할을 한다. 파일이름이나 함수 이름의 경우, clang에서 AST(abstract syntax tree)를 탐색할 때, 함수 선언 구문에 도달했을 때 clang API를 사용하여 파일 이름과 함수 이름의 획득이 가능하므로, 이를 쌍으로 묶어 출력을 한다. .cil.c 파일에는 분기마다 __CrownBranch 함수 호출 구문이 존재하므로 __CrownBranch 함수 호출 구문의 위치로 분기를 파악할 수가 있다. 따라서 AST를 탐색할 때, __CrownBranch 함수 호출 구문에 해당하는 노드에 도달했을 경우, 다음과 작업을 진행한다. 먼저 출력해야 할 정보는 (분기 id, 분기가 존재하는 라인, 분기의 expression) 쌍이다. 이 중 분기 id는 __CrownBranch 함수 호출 구문의 2번째 인자에 해당하므로, clang API를 사용하여 __CrownBranch 함수 호출 구문의 2번째 인자, 즉 분기 id를 파싱하는 것이 가능하다. 코드 상에서 분기가 포함된 조건문 바로 아래에 __CrownBranch 함수 호출구문이 삽입되므로, 해당 노드의 조상 노드를 탐색하여 분기에 해당하는 조건문을 찾을 수 있고, 조건문 내부의 조건식을 파싱해서 분기 expression의 획득이 가능하다. 추가로 조건문의 라인정보를 clang API를 사용하여 획득할 수 있다.

new_coverage 파일 구성은 다음과 같다. 먼저 상세 분기 정보와 분기 요약 정보 두 부분으로 구성된다. 상세 분기 정보는 분기 id, 분기(달성에 필요한 조건), 라인 정보가 파일 별, 함수 별, 분기의 달성/미달성 별로 분류되어 나타난다. 분기 요약 정보는 파일별, 함수별로 달성한 분기 개수, 미달성한 분기 개수, 전체 분기 개수, 분기 커버리지가 출력된다. 그리고 달성한 전체 분기 개수, 미달성한 전체 분기 개수, 전체 분기 개수, 전체 분기 커버리지가 뒤따른다.

그림 11과 그림 12는 심볼릭 변수가 선언 된 예제 소스 코드이다. 그림 12에서는 6, 7라인에서 두 변수 var_a와

```
1    #include <stdio.h>
2    #include <crown.h>
3    void func() {
4        int var_c;
5
6        SYM_int(var_c);
7        if(var_c == 123) printf("reach");
8    }
```

그림 11 예제 코드(target1.c)
Fig. 11 Example code (target1.c)

```
1    #include <stdio.h>
2    #include <crown.h>
3    void main() {
4        int var_a, var_b;
5
6        SYM_int(var_a);
7        SYM_int(var_b);
8        if(var_a > 10 && var_b > 10) {
9            if(var_a < -10)
10               printf("infeasible");
11           func();
12       }
13   }
```

그림 12 예제 코드(target2.c)
Fig. 12 Example code (target2.c)

```
1    Source file: target1.c
2        Function: func
3        Covered Branches
4        | Branch ID | Line No.   | Condition to cover the BR |
5        |          14 |          7 | var_c == 123              |
6        |          15 |          7 | !(var_c == 123)           |
7        No uncovered Branch
8
9    Source file: target2.c
10       Function: main
11       Covered Branches
12       | Branch ID | Line No. | Condition to cover the BR   |
13       |          3 |          8 | var_a > 10                  |
14       |          4 |          8 | var_b > 10 && var_a > 10    |
15       |          8 |          8 | !(var_b > 10) && var_a > 10 |
16       |          9 |          8 | !(var_a > 10)               |
17       |          6 |          9 | !(var_a < -10)              |
18       Uncovered Branches
19       | Branch ID | Line No . | Condition to cover the BR   |
20       |          5 |          9 | var_a < -10                 |
21
22
23   Summary
24   Source file: target1.c
25   | Function name | cov BR# | uncov BR# | total BR# | cov rate(%) |
26   | func          |       2 |         0 |         2 |       100.0 |
27   Source file: target2.c
28   | Function name | cov BR# | uncov BR# | total BR# | cov rate(%) |
29   | main          |       5 |         1 |         6 |        83.3 |
30
31   Total Coverage
32   | cov BR# | uncov BR# | total BR# | cov rate(%) |
33   |       7 |         1 |         8 |        87.5 |
```

그림 13 new_coverage 파일 내용
Fig. 13 Contents of the new_coverage file

var_b를 심볼릭 선언하고 있으며, 11라인에서 그림 11에 포함된 함수 func를 호출하고 있다. 특히, 8라인의 조건 var_a > 10으로 인해 9라인의 조건 var_a < -10은 달성할 수 없는 분기이다. 그림 11에서는 6라인에 변수 var_c를 심볼릭 선언하고 있다.

그림 13은 “new_coverage” 파일을 내용이다. 1-20라인은 분기 상세 정보를 나타내며, 23-33 라인은 분기 요약 정보를 나타낸다. 분기 상세 정보에서 1-7라인은 그림 11에 포함된 분기 정보를 나타내며 9-20라인은 그림 12에 포함된 분기 정보를 나타낸다. 각 파일마다 포함 된 함수별로 출력되며, 달성한 분기와 미달성한 분기로 분류되어 출력된다. 각 분기 정보는 분기 id, 라인 정보, 분기 달성을 위한 조건 정보가 차례대로 표기된다. 1개의 분기를 제외하고 모든 분기를 달성한 것을 확인할 수 있으며, 미달성한 1개의 분기는 그림 12의 main 함수 내부에 포함 된 9라인 분기(var_a < -10)임을 확인할 수 있다.

5. 결론 및 향후 연구

본 논문에서는concolic 테스팅 도구인CROWN에 대해 소개했다. 먼저, 기존 Linux 환경에서만 동작한 것과 달리 Windows OS에서도 사용이 가능하도록 CROWN에서 확장하는 포팅 작업을 수행했다. 이전까지는 Windows OS에서만 실행 가능한 C 소스 코드에는 concolic 테스팅 적용을 포기하거나, Linux 환경에서 동작 가능하도록 변환하는 작업을 거치고 난 뒤, OS간의 동작 차이를 감안하고 concolic 테`스팅을 적용해야 했다. 하지만 본 연구에서 수행한 포팅 작업으로 인해 Windows OS에서 동작하는 C 소스 코드에 대해서도 쉽게 CROWN의 적용이 가능해졌다. 저자가 아는 한 Windows OS에서 현재 사용 가능한 C 프로그램 용 Concolic 기법 도구가 없기 때문에 본 작업은 C 프로그램 용 Concolic 기법 도구 전체의 확장성을 높이는데 큰 기여를 했다고 보인다.

두번째로 print_execution의 기존 출력 메시지를 개선하고, 커버리지 분석에 필요한 추가 정보가 담긴 new_coverage를 사용자에게 제공하여 테스팅에 필요한 시간을 크게 줄이고자 하였다.

CROWN은 이미 [20]에서 사용 도구로 활용된 바 있으며, industry의 개발 환경과 특성을 고려한 만큼 실제 개발 환경에서의 활용성이 높을 것으로 보인다.

Windows OS 버전의 CROWN은 현재 프롬프트 창에서 명령어로 동작하고 있기 때문에 향후 연구로써 Windows OS의 이점인 GUI 특성을 살리도록 GUI 인터페이스를 추가할 계획이다.

References

[1] K. Sen, D. Marinov, G. Agha, "CUTE: a concolic unit testing engine for C," *Proc. of European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pp. 263-272, 2005.

[2] "Concolic testing for Real-wOrld software aNalysis

(CROWN)," available at http://github.com/swtv-kaist/CROWN

[3] D. L. Bird and C. U. Munoz, "Automatic Generation of Random Self-Checking Test Cases," *Journal of IBM Systems*, Vol. 22, No. 3, pp. 229-245, 1983.

[4] J. C. King, "Symbolic execution and program testing," *Journal of Communications of the ACM*, Vol. 19, No. 7, pp. 385-394, Jul. 1976.

[5] T. Xie, D. Marinov, W. Schulte, D. Notkin, "Symstra: A Framework for Generating Object-Oriented Unit Tests Using Symbolic Execution," *Proc. of the 11th international conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 365-381, 2005.

[6] P. Boonstoppel, C. Cadar, D. Engler, "RWset: Attacking path explosion in constraint-based test generation," *Proc. of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, pp. 351-366, 2008.

[7] Y. Kim, M. Kim, Y. J. Kim, Y. Jang, "Industrial application of concolic testing approach: A case study on libexif by using CREST-BV and KLEE," *Proc. of the 34th International Conference on Software Engineering*, pp. 1143-1152, 2012.

[8] Y. Kim, M. Kim, N. Dang, "Scalable distributed concolic testing: a case study on a flash storage platform," *Proc. of the 7th International colloquium conference on Theoretical aspects of computing*, pp. 199-213, 2010.

[9] Y. Park, S. Hong, M. Kim, D. Lee, J. Cho, "Systematic testing of reactive software with non-deterministic events: A case study on LG electric oven," *Proc. of the 37th International Conference on Software Engineering*, pp. 29-38, 2015.

[10] J. Burnim and K. Sen, "Heuristics for scalable dynamic test generation," *Proc. of the 23rd IEEE/ACM International Conference on Automated Software Engineering*, 443-446, 2008.

[11] T. F. Chen, C. Hsieh, O. Lengál, T .J. Lii, M. H. Tsai, B. Y. Wang, F. Wang, "PAC Learning-Based Verification and Model Synthesis," *Proc. of the 38th International Conference on Software Engineering*, pp. 714-724, 2016.

[12] S. Godboley, D. P. Mohapatra, A. D, R. M, "An improved distributed concolic testing approach," *Journal of Software: Practice and Experience*, Vol. 47, No. 2, pp. 311-342, Feb. 2017.

[13] Y. Koroglu and A. Sen, "Design of a Modified Concolic Testing Algorithm with Smaller Constraints," *Proc. of International Workshop on Constraints in Software Testing, Verification and Analysis @ISSTA*, pp. 3-14, 2016.

[14] P. Godefroid, N. Klarlund, K. Sen, "DART: directed automated random testing," *Proc. of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pp. 213-223, 2005.

[15] C. Cadar, D, Dunbar, D. Engler, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs," *Proc. of the 8th USENIX conference on Operating systems design and implementation*, pp. 209-224, 2008.

[16] C. Cadar, D. Engler, "Execution generated test cases: How to make systems code crash itself," *Proc. of the 12th international conference on Model Checking Software*, pp. 2-23, 2005.

[17] C. Cadar, V. Ganesh, P.M. Pawlowski, D.L. Dill, D.R. Engler, "EXE: automatically generating inputs of death," *Journal of ACM Transactions on Information and System Security*, Vol. 12, No. 2, Article No. 10, Dec. 2008.

[18] L. de Moura and N. Bjørner, "Z3: An efficient SMT solver," *Proc. of the 14th international conference on Tools and algorithms for the construction and analysis of systems*, pp. 337-340, 2008.

[19] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," *Proc. of the international symposium on Code generation and optimization*, pp. 75, 2004.

[20] Y. Kim, Y. Choi, M. Kim, "Precise Concolic Unit Testing of C Programs with Extended Units and Symbolic Alarm Filtering," *Proc. of the International Conference on Software Engineering*, 2018.

김 현 우
정보과학회논문지
제 45 권 제 4 호 참조

김 윤 호
정보과학회논문지
제 45 권 제 4 호 참조

김 문 주
정보과학회논문지
제 45 권 제 4 호 참조