석 사 학 위 논 문
Master's Thesis

Concolic 테스팅을 위한 테스트 커버리지 향상 가이드라인 및 산업체 사례 연구

Concolic Testing Guidelines for Test Coverage Improvement: an Industrial Case Study

2018

김 현 우 (金 鉉 雨 Kim, Hyunwoo)

한 국 과 학 기 술 원

Korea Advanced Institute of Science and Technology

석 사 학 위 논 문

Concolic 테스팅을 위한 테스트 커버리지 향상 가이드라인 및 산업체 사례 연구

2018

김 현 우

한 국 과 학 기 술 원

전산학부

Concolic 테스팅을 위한 테스트 커버리지 향상 가이드라인 및 산업체 사례 연구

김 현 우

위 논문은 한국과학기술원 석사학위논문으로
학위논문 심사위원회의 심사를 통과하였음

2018년 6월 18일

심사위원장 김 문 주 (인)

심 사 위 원 고 인 영 (인)

심 사 위 원 백 종 문 (인)

Concolic Testing Guidelines for Test Coverage Improvement: an Industrial Case Study

Hyunwoo Kim

Advisor: Moonzoo Kim

A dissertation submitted to the faculty of
Korea Advanced Institute of Science and Technology in
partial fulfillment of the requirements for the degree of
Master of Science in Computer Science

Daejeon, Korea
June 18, 2018

Approved by

Moonzoo Kim
Professor of School of Computing

The study was conducted in accordance with Code of Research Ethics[1].

[1] Declaration of Ethical Conduct in Research: I, as a graduate student of Korea Advanced Institute of Science and Technology, hereby declare that I have not committed any act that may damage the credibility of my research. This includes, but is not limited to, falsification, thesis written by someone else, distortion of research findings, and plagiarism. I confirm that my thesis contains honest conclusions based on my own careful research under the guidance of my advisor.

MCS
20163180

김현우. Concolic 테스팅을 위한 테스트 커버리지 향상 가이드라인 및 산업체 사례 연구. 전산학부 . 2018년. 54+v 쪽. 지도교수: 김문주. (한글 논문)

Hyunwoo Kim. Concolic Testing Guidelines for Test Coverage Improvement: an Industrial Case Study. School of Computing . 2018. 54+v pages. Advisor: Moonzoo Kim. (Text in Korean)

초 록

Concolic 테스팅에서 커버리지 향상을 위해서는 미달성 영역을 탐색하고 테스팅 환경을 재구축하는 과정이 필요하다. 하지만 이를 위한 가이드라인이나 정형화된 기준이 없어서 재구축한 환경이 사용자에 따라 다르고, 타당한지도 검증할 수 없는 문제가 존재한다. 이를 해결하고자 본 연구에서는 미달성 분기들을 유형별로 나누고, C 대상 Concolic 테스팅을 통해 유형별 분기를 달성하기 위한 가이드라인을 소개한다. 가이드라인의 효과를 알아보기 위해 상용 자동차 SW를 대상으로 테스팅을 수행한 뒤 가이드라인을 적용하여 효과적으로 테스트 커버리지를 향상할 수 있었다. 결과적으로 90%미만의 커버리지를 기록한 상용 SW 모듈의 일부 파일들을 대상으로 가이드라인을 적용하여 기존 43%에서 97%으로 커버리지를 향상했다. 가이드라인 적용 시 테스팅 결과를 빨리 분석하기 위해서 자체 개발한 C 대상 Concolic 테스팅 도구 CROWN을 사용했으며 이로인해 사례 연구를 성공적으로 마칠 수 있었다. 사용자가 구축한 테스팅 환경에 따라 영향을 크게 받는 Concolic 테스팅 특성상 본 논문에서 제안한 구체적 가이드라인은 중요한 지표로서 사용될 수 있으며 상용 자동차 SW에 대해 성공적인 결과를 보인 만큼 실제 환경에서도 그 활용성이 높을 것이라 예상한다.

핵 심 낱 말 커버리지 향상 기술, Concolic 테스팅, 상용 자동차 SW 사례 연구, 소프트웨어 자동화 테스팅, CROWN

Abstract

The process of exploring uncovered area and reconstructing testing environment is required to improve coverage in concolic testing. However, reconstructed environment is different from user to user and its validity is also unreliable because there isn't proper guideline nor formal standard to refer to. To solve this problem, we define uncovered branch types and introduce guideline per type to improve coverage in concolic testing for C program. In order to check effectiveness of the guidelines, we showed coverage improvement by testing industrial automobile SW and applying guidelines. As a result, applying guidelines on target files which achieved coverage under 90% increased the test coverage from 43% to 97%. To quickly analyze testing result in experiment, we developed and utilized CROWN, the concolic testing tool for C program, and then we were able to successfully finish the case study for large industrial automobile SW. Considering concolic testing depends on testing environment set by user, we expect our guidelines to be valuable indicator and to be highly applicable even to real world in light of its effective coverage improvement in the industrial automobile SW.

Keywords coverage improvement, concolic testing, industrial study for automobile SW, automated software testing, CROWN

차 례

표 차례

그림 차례

제 1 장 머릿말

소프트웨어의 늘어나는 필요성과 요구사항으로 인해 그 복잡성이 높아짐에 따라 소프트웨어 테스팅에 대한 중요성 역시 증가하고 있다. 테스팅의 목적은 소프트웨어 내부에 존재하는 에러를 검출하는 것으로, 이를 달성하기 위해서는 소프트웨어의 가능한 많은 경로의 탐색이 이루어져야 한다. 과거에는 개발자가 직접 프로그램에 대한 테스팅을 수행하기도 했지만, 현재에 이르러 개발자가 수작업으로 직접 수십 ~ 수백만 라인 규모의 소프트웨어를 분석하면서 테스팅을 수행한다는 것은 사실상 불가능에 가까워졌다. 따라서 최근에는 자동으로 테스팅을 수행하는 기법들이 활용되고 있으며, 이 기법들을 사용하는 것이 필수적인 단계에 접어들었다.

일반적으로 널리 알려진 자동화 테스팅 기법으로 퍼징 기법[18]이 있다. 대부분의 상용 테스팅 도구에 탑재된 이 기법은 랜덤 테스트들을 만들어 실행하는 기법으로, 대상 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 조금만 높아져도 탐색하는 실행 경로의 범위는 기하급수적으로 떨어지게 된다. Concolic 기법은 이러한 퍼징 기법의 한계를 극복한 효과적인 테스팅 기법으로, 학계에서 지난 10년부터 지금까지 활발히 연구가 진행되고 있는 기법이다. Concolic 기법의 실행 과정은 초기 테스트로 대상 프로그램을 실행한 뒤, 실행한 경로를 기반으로 새로운 경로를 탐색하는 테스트를 생성하고 이 테스트로 대상 프로그램을 다시 실행하는 일련의 과정을 반복하면서 프로그램의 실행 경로를 탐색해나가는 기법이다. 이론적으로 Concolic 기법을 사용하면 대상 프로그램의 모든 실행 경로를 탐색하는 것이 가능하다. Concolic 기법을 사용하기 위한 사전작업으로 사용자는 대상 소스 코드에 Concolic 테스팅을 위한 추가 코드를 삽입하여 테스팅 환경을 구축해야 한다. 이 작업은 대상 소스 코드에 대한 이해와 배경이 필요하며, 이 작업에 따라 탐색할 수 있는 프로그램 경로의 범위나 탐색의 우선순위가 크게 달라진다. 따라서 Concolic 테스팅의 효과를 높이기 위해선 최적화된 테스팅 환경 구축이 필수적이라 볼 수 있다.

1.1 기존 접근의 문제점

앞서 소개한 것처럼 Concolic 테스팅의 결과는 구축한 테스팅 환경에 따라 크게 달라진다. 이처럼 테스팅 환경은 Concolic 테스팅에서 중요한 비중을 차지하기 때문에 효과적인 Concolic 테스팅을 위해 테스트 드라이버/스텁을 자동으로 생성하여 테스팅 환경을 구축하고 범용적으로 사용이 가능한 프레임워크를 제안한 연구[20]도 존재한다. 문제는 탐색하지 못한 영역에 초점을 두고, 해당 영역을 탐색하기 위한 구체적 가이드라인을 제안하는 연구는 찾아볼 수 없다는 점이다. 앞서 연구된 Concolic 테스팅 환경 구축 프레임워크 역시 범용적으로 적용은 할 수 있으나 최적의 테스팅 결과를 보장하지는 않는다. 즉, 무수히 많은 코드 스타일이 존재하는 만큼 프레임워크에서 구축한 환경만으로는 달성하기 어렵거나 달성이 불가능한 예외적인 경우도 필연적으로 많이 존재하기 때문이다.

따라서 일반적으로는 테스팅 한 번으로 끝나는 것이 아니라 만족할만한 커버리지를 달성하기 위해 탐색이 안 된 영역을 분석하고 테스팅 환경을 재구축하는 일련의 과정을 반복하는 작업이 필요하다. 이 과정에서 앞에 언급한 것처럼 탐색하지 않은 영역을 달성하기 위한 정형화 된 가이드라인이나 연구를 찾아볼 수 없기 때문에 사용자는 어떤 식으로 테스팅 환경을 구축하고 새로 테스팅 환경을 구축했을 때 얼만큼의 효과가 발생할지 등을 전적으로 경험적인 측면에 의존할 수밖에 없다.

1.2 제안하는 접근 방식

본 논문에서는 실제 환경에서 많이 사용하는 C 프로그램을 대상으로 Concolic 테스팅 수행 시 발생할 수 있는 미달성 분기 유형들을 정의한 뒤, 유형별로 미달성 분기를 달성할 수 있는 가이드라인을 제공하고 있다. 더불어 각 미달성 분기 유형을 구체적인 예제와 함께 소개하고 있기 때문에 Concolic 테스팅에 대한 경험이 적은 사용자들이 커버리지를 높이기 위한 테스팅 환경을 구축하는데 참고할 수 있는 좋은 지침서가 될 수 있다. 제안한 가이드라인의 효과를 확인하기 위해서 실제 상용 자동차 SW을 대상으로 Concolic 테스팅을 수행한 뒤, 본 논문에서 정의한 미달성 유형에 따라 미달성 분기들을 분류하고 가이드라인을 적용하였으며 성공적으로 커버리지를 높일 수 있었다. 상용 자동차 SW을 대상으로 효과적인 커버리지 상승을 거둔 것을 기반으로 실제 환경에서도 많은 활용과 효과적인 커버리지 상승이 가능할 것이라 보고 있다.

1.3 논제 및 기여

본 논문의 논제는 다음과 같다.

논문에서 제안한 가이드라인을 적용하여 테스트 커버리지를 향상시킬 수 있다.

다음으로 본 논문은 다음과 같은 기여를 한다.

- C 대상 Concolic 테스팅 수행 시 달성에 실패할 수 있는 6개의 분기 유형을 정의하고 각 유형의 미달성 분기를 달성할 수 있는 가이드라인 제안
- 상용 자동차 SW를 대상으로 가이드라인을 적용하여 기존 43%의 테스트 커버리지를 97%로 향상시킨 실험 결과를 통해 본 논문에서 제안한 가이드라인의 효과가 있음을 보임
- 효율적인 실험 진행을 위해 기존 C 대상 Concolic 테스팅 도구의 UI 측면을 개선한 도구 CROWN을 자체 개발 및 사용

1.4 논문의 전체 구성

본 논문은 다음과 같이 구성된다. 제 2장 연구 배경에서는 Concolic 테스팅에 대한 소개와 [20]에 기반을 둔 동적 기호 실행 기반 자동화된 테스크 테스팅 기법을 소개한다. 해당 기법은 테스트 드라이버/스텁을 자동으로 생성하여 Concolic 테스팅 수행에 필요한 테스팅 환경을 구축한다. 본 논문의 실험에서는 해당 기법을 사용해서 테스팅 환경을 구축한다. 제 3장 관련 연구 장에서는 기존 Concolic 기법 도구들을 소개하고, Concolic 기법 도구를 활용한 기존 사례 연구와 본 논문에서 수행한 사례 연구와의 차이를 보인다. 제 4장 CROWN의 향상 기능 소개에서는 본 논문에서 미달성 분기 분석 작업을 쉽게 하려고 자체 개발한 Concolic 테스팅 도구 CROWN을 소개한다. 제 5장 테스트 커버리지 향상 가이드라인에서는 6개의 미달성 분기 유형들을 소개하고 유형별 미달성 분기를 달성할 수 있는 가이드라인을 소개한다. 제 6장 사례 연구에서는 상용 자동차 SW를 대상으로 본 논문에서 제안하는 가이드라인을 적용한 효과를 보이고자 한다. 제 7장 결론 및 향후 연구에서는 본 논문을 요약하고, 앞으로 추가 연구할 사항을 소개하며 논문을 마친다.

제 2 장 연구 배경

2.1 Concolic 테스팅 소개

본 절에서는 Concolic 테스팅 기법을 소개하고 Concolic 테스팅이 구체적으로 어떻게 동작하는지를 예제와 함께 보인다. 이후, 현존하는 코드 기반의 C 대상 Concolic 기법 도구에 존재하는 한계점을 소개하고 마지막으로 이 한계점을 완화하여 실험을 진행하기 적합하도록 기존 Concolic 도구 CREST를 개선하여 만든 CROWN(Concolic testing for Real-wOrld software aNalysis)에 대해 개괄적으로 소개한다.

2.1.1 Concolic 테스팅 기법

동적 분석에 속하는 Concolic 테스팅 기법(Concrete + Symbolic)은 대상 프로그램을 실행하면서 이론적으로는 프로그램의 모든 경로를 실행하는 기법으로 현재까지 많은 연구가 진행되고 있는 동시에 실제 산업의 다양한 분야의 소프트웨어를 대상으로 활용[23, 24]되고 있는 기법이다. Concolic 테스팅은 Concrete execution을 통해 Symbolic execution을 유도한다. Concrete execution은 입력 변수에 실제 값을 넣고, 대상 프로그램의 한 경로를 실행시킨다. Symbolic execution은 입력 변수를 심볼로 가정하고, Concrete execution에서 실행한 경로에 도달하기 위해 입력 변수(심볼)가 만족해야 하는 심볼릭 경로 수식(symbolic path formula)을 수집한다[28]. 이후 수집한 심볼릭 경로 수식에 변경을 가해서 아직 실행하지 못한 경로에 도달하기 위한 새로운 심볼릭 경로 수식(next formula)을 만들어낸다. 심볼릭 경로 수식을 수정하는 방법은 여러 가지가 존재하며 기본적인 "DFS(Depth-First Search)" 방법의 경우에는 심볼릭 경로 수식의 마지막 조건에 negation을 취해서 새로운 심볼릭 경로 수식을 획득한다. 이후, Z3 solver[15], Yices 또는 lp_solver[41]와 같은 SMT solver에 새롭게 구한 심볼릭 경로 수식을 전달하여 심볼(입력 변수)의 해(값)를 구할 수 있다. 새롭게 구한 입력 변숫값을 사용해서 다시 대상 프로그램을 실행시키고, 새로운 입력 변수의 값을 추출하는 일련의 과정을 반복하면서 프로그램의 경로들을 탐색하게 된다. 이 과정을 모든 가능한 실행경로가 수행되거나 사용자가 지정한 특정 조건(시간 또는 횟수)을 만족할 때까지 반복한다.

Concolic 테스팅 실행 절차

1. 심볼릭 변수 선언

Concolic 테스팅 기법 도구에서 지원하는 API를 사용하여 소스 코드의 입력 변수를 심볼릭 선언하는 작업이다. 변수를 심볼릭 선언 시, 해당 변수는 심볼로 간주되며 이 심볼은 심볼릭 경로 수식에서 사용된다. 심볼릭 선언에 사용하는 API는 도구마다 다르며, 예시로 본 논문에서 개발한 도구 CROWN 같은 경우 표 2.1와 같이 입력 변수 타입에 따라 다른 API를 정의하여 사용하고 있으며 인자로 심볼릭 선언할 변수를 넘겨받는다. 포인터에 대한 심볼릭 선언 API는 지원하지 않기 때문에 포인터가 아닌, 포인터가 가리키는 공간 또는 변수에 대해 심볼릭 선언을 해줄 수 있다. 배열과 구조체도 마찬가지로 각각 배열의 원소와 구조체의 필드들을 하나하나씩 심볼릭 선언해줄 수 있다. KLEE 같은 경우에는 심볼릭 선언을 위해 하나의 API, klee_make_symbolic를 사용한다. 이 API는 심볼릭 선언할 변수 외에도 해당 변수의 크기와 변수명(string)을 인자로 받는다.

표 2.1: CROWN에서 사용하는 심볼릭 선언 API 목록

입력 변수 x의 타입	심볼력 선언 API
signed char	SYM_char(x)
unsigned char	SYM_unsigned_char(x)
signed short	SYM_short(x)
unsigned short	SYM_unsigned_short(x)
signed int	SYM_int(x)
unsigned int	SYM_unsigned_int(x)
signed long	SYM_long(x)
unsigned long	SYM_unsigned_long(x)
signed long long	SYM_longlong(x)
unsigned long long	SYM_unsigned_longlong(x)
float	SYM_float(x)
double	SYM_double(x)

2. Instrumentation 작업

대상 프로그램과 동일한 기능(semantic)을 수행하지만 Concolic 테스팅 수행에 필요한 프로브들을 추가로 삽입한 새로운 코드(instrumentation code)를 생성하는 작업이다. 이러한 프로브들은 Concolic 테스팅에 필요한 심볼릭 수식을 수집하고, 조건(if, for, while, switch)문에서 어떤 분기(then, else)가 실행되는지 등을 기록하는 작업을 수행한다. 또한 원본 코드의 복잡한 조건문들은 단순한 형태로 변형된다. 단, 원본 코드의 의미는 변경되지 않는다. 예를 들면, 원본 코드의 복합 조건문은 단일(atomic) 조건문들로 변형되면서 분기가 세분화되며, for 문도 단순한 while 문 형태로 변형된다. Instrumentation 작업을 마쳤으면 새로 생성된 코드(Instrumentation code)를 컴파일하여 실행 파일을 생성하고, 3 $\sim$ 5 과정을 반복하면서 Concolic 테스팅을 수행한다.

3. 프로그램 실행

Instrumentation 코드를 컴파일하여 생성한 실행 파일을 테스트 케이스를 사용해서 프로그램 경로 e를 실행한다. 테스트 케이스는 5단계에서 생성되는데 실행 파일을 처음 실행할 경우, 획득한 테스트 케이스가 없기 때문에 임의의 값이나 0 값을 초기 테스트 케이스로 사용한다.

프로그램이 실행되면 Instrumentation 작업에서 삽입한 프로브들에 의해 심볼릭 경로 수식 p가 수집된다. 심볼릭 경로 수식 p는 경로 e를 실행하기 위해 심볼릭 변수들이 만족해야 하는 조건이라고 볼 수 있다.

4. 다음 실행 경로 설정

심볼릭 경로 수식 p를 적절히 변형하여 다음 실행할 경로가 만족해야 하는 심볼릭 경로 수식 p'를 생성한다. 이때, 이미 실행한 경로를 중복해서 탐색하는 것을 막기 위해 p'는 이전과 중복되지 않는 새로운 심볼릭 경로 수식이어야 한다. 다음 단계에서 p'를 만족하는 테스트 케이스를 만들어서 다음 실행 경로를 수행하게 된다. 탐색 방법에 따라 p를 변형하는 방법이 다르며, 가장 일반적인 DFS(Depth-First Search) 방법을 사용할 경우 p의 가장 마지막 조건에 부정 연산(negation operator)을 적용해서 p'를 생성한다. 탐색 가능한 프로그램 경로를 이미 모두 실행한 경우, 중복되지 않는 새로운 심볼릭 경로 수식 p'의 생성이 불가하므로 Concolic 테스팅을 종료한다.

```
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 2.1: 예제 코드 1

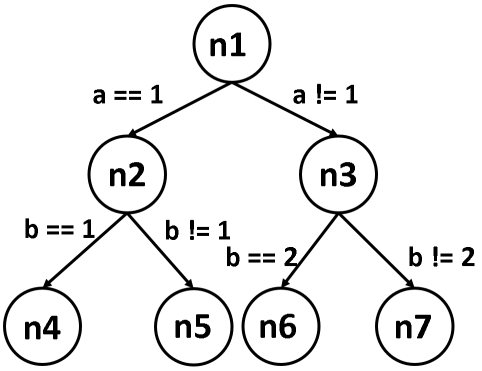

그림 2.2: 예제 코드 1의 분기 그래프

5. 다음 실행 경로를 위한 테스트 케이스 생성

SMT solver를 사용해서 심볼릭 경로 수식 p'를 만족하는 해(테스트 케이스)를 구한다. SMT solver는 특성에 따라 lp_solver, Yices[16], STP, Z3와 같이 다양한 도구가 존재하며, CROWN에서는 Z3를 채택하고 있다. Z3는 MIT License를 사용하는 오픈소스이기 때문에 자유롭게 사용 및 수정을 할 수 있다. 이후, SMT solver를 사용해서 구한 해(테스트 케이스)로 프로그램을 실행하는 과정(3단계)부터 재반복하면서 새로운 실행 경로를 탐색해나간다.

2.1.2 Concolic 테스팅 예제

본 절에서는 Concolic 테스팅이 어떻게 수행되는지를 예제와 함께 구체적으로 소개한다. 그림 2.1은 Concolic 테스팅을 적용할 대상 소스 코드이고, 그림 2.2는 소스 코드의 분기 지점들을 그래프로 표현한 것이다. 그림 2.2의 노드(n1, n2,. . . , n7)들은 각 분기 지점을 나타낸다. 예를 들면, n1에 해당하는 노드는 대상 소스 코드 6라인에 해당하는 분기 지점으로, 조건 "a == 1"을 만족할 경우 n2 노드(then 분기)로 이동하고, 조건을 만족하지 못할 경우 n3 노드(else 분기)로 이동한다. Concolic 테스팅은 모든 가능한 실행 경로의 탐색을 목표로 한다. 그림 2.1에서 4, 5라인의 SYM_int 함수는 인자로 받은 변수를 심볼릭 변수로 선언하는 기능을 한다. 심볼릭 변수로 선언되면 Symbolic execution에서 해당 변수는 심볼로 간주된다.

표 2.2은 Concolic 테스팅 과정에서 생성한 테스트와 해당 테스트의 심볼릭 경로 수식, 새로운 심볼릭 경로 수식을 나타낸다. 처음에 심볼릭 변수에 임의의 초기값(여기선 a=0, b=0)을 부여하고 대상 프로그램을 실행한다. 프로그램이 실행되면 심볼릭 경로 수식이 수집된다. 프로그램은 분기 지점 n1, n3, n7을 차례대로 수행하고 수집한 심볼릭 경로 수식은 a!=1 && b!=2가 된다. 이후, 심볼릭 경로 수식의 마지막 식(b!=2)에 부정 연산(negation operator)을 적용하여 기존 심볼릭 경로 수식과 겹치지 않는 새로운 다음 심볼릭 경로 수식(next formula, a!=1 && b==2)을 생성한다. 다음으로, 새로 생성한 심볼릭 경로 수식(next formula)을 SMT solver에 전달하여 해당 수식을 만족하는 테스트 케이스(여기선 a=0, b=2)를 획득한다. 이후, 획득한 테스트 케이스로 대상 프로그램을 실행하는 과정을 다시 반복한다. 이러한 과정을 통해 표 2.2처럼 총 4개의 테스트 케이스를 생성할 수 있고 모든 분기 지점(n1, n2, . . . , n7)을 성공적으로 탐색할 수 있다.

표 2.2: Concolic 테스팅에서 생성한 테스트 케이스

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

2.1.3 C 대상 Concolic 기법 도구의 한계점

Concolic 테스팅에서 커버리지를 향상하기 위해서는 먼저 테스팅 결과를 분석해야 하는데 문제는 C 대상 Concolic 기법 도구 중에 테스팅 결과 분석에 필요한 정보를 활용하기 쉬운 형태로 제공하는 도구는 찾아보기 어렵다는 점이다.

Concolic 테스팅 결과는 사용자가 구축한 테스트 드라이버/스텁 및 심볼릭 설정과 같은 작업에 따라 크게 달라지기 때문에 사용자는 테스팅 구축 환경을 대상 프로그램에 맞게 수정하고 개선해 나가면서 더 좋은 테스팅 결과를 유도해야 한다. 즉, Concolic 테스팅 수행 후 탐색하지 못한 경로들에 대해서는 그 원인을 분석하여 피드백으로 사용할 수 있어야 한다. 이러한 테스팅 결과 분석을 위해서는 수집한 심볼릭 경로 수식이나 어떤 분기가 달성되었고 미달성되었는지와 같은 상세한 정보가 알기 쉬운 형태로 제공되어야 하는데 현재 C 대상 Concolic 기법 도구 중에 이러한 정보를 제공하는 도구를 찾기는 어렵다. 최종 분기 커버리지만 제공하고 있으며, 심볼릭 수식 정보 또는 어떤 분기가 달성되었고 미달성되었는지와 같이 상세한 정보는 제공하고 있지 않거나 제공하더라도 사용자가 바로 사용하기 어려운 형태로 제공하고 있다. 이로 인해 사용자는 테스팅 결과 분석에 훨씬 많은 시간을 비효율적으로 할애하게 된다.

2.1.4 CROWN 소개

본 논문에서는 실험을 효율적으로 진행할 목적으로 기존 Concolic 기법 도구 문제를 해결한 CROWN(Concolic testing for Real-wOrld softWare aNalysis)을 만들어 사용했다. CROWN은 C 대상 Concolic 테스팅 도구인 CREST를 기반으로 만든 도구이며, CREST는 Concolic 기법을 사용한 연구에 활발히 사용[25, 26, 27]되고 있는 대표적인 도구이다. 본 논문에서는 첫 번째로 기존 CREST에서 제공하는 심볼릭 정보 출력 기능을 개선했다. 기존 심볼릭 정보 출력은 사용자가 사용하고 이해하기 어려운 형식이었기 때문에 활용성이 떨어지고 심볼릭 정보를 사용하기 위해서 사용자는 해당 정보를 해석해야하는 추가 작업이 필요했기 때문이다. CROWN에서는 이러한 문제점을 해결하고자 기존 심볼릭 정보 출력 형식을 변경하고 추가 정보를 제공함으로서 보다 나은 편의성을 제공했다. 더불어 기존 CREST에서는 테스팅 후 사용자가 커버리지 분석에 사용 가능할 수 있는 달성/미달성 분기에 대한 정보가 미흡하여 커버리지 분석에 어려움을 겪었으나, CROWN에서는 사용자 친화적인 분기 정보를 제공하는 기능을 추가하여 사용자가 커버리지 분석에 쉽게 활용할 수 있도록 유도했다.

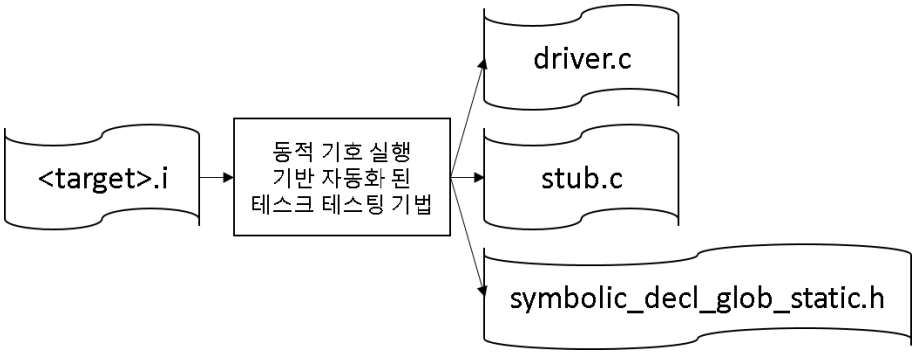

그림 2.3: 동적 기호 실행 기반 자동화된 테스크 테스팅 기법 입출력

2.2 동적 기호 실행 기반 자동화된 테스크 테스팅 기법

동적 기호 실행 기반 자동화된 테스크 테스팅 기법은 테스트 드라이버/스텁을 자동 생성하고 테스트의 입력 값을 동적 기호 실행 입력 설정에 따라 자동으로 심볼릭 선언하는 구문을 자동으로 추가하는 기법으로 테스크 단위의 Concolic 테스팅 환경을 자동으로 구축하는 프레임워크로 볼 수 있다. 본 논문의 사례 연구에서는 해당 기법으로 구축한 테스팅 환경에서 Concolic 테스팅을 수행했다.

여기서 테스크(task)란 대상 파일 내 non-static 함수를 호출할 때 대상 파일 내에서 실행되는 일련의 작업을 의미한다. 따라서 테스크는 non-static 함수 개수만큼 존재하며, non-static 함수를 호출했을 때 실행 가능한 함수들을 묶어 같은 테스크에 속한다고 할 수 있다. 또 테스크를 실행하는 non-static 함수를 해당 테스크의 인터페이스(interface)라고 부른다. 테스크에 대한 상세 설명은 6.1절을 참고한다.

동적 기호 실행 기반 자동화된 테스크 테스팅 기법은 동적 기호 실행 기반 자동화된 유닛 테스팅 기법[20]에서 제안하는 프레임워크를 사용한다. 동적 기호 실행 기반 자동화된 유닛 테스팅 기법에서는 효과적인 Concolic 테스팅을 위한 자동 테스팅 환경 구축 프레임워크를 제안했으며, 실제 환경에서 동작하는 산업체 프로그램을 대상으로 프레임워크를 적용하여 성공적으로 실제 프로그램 내 오류들을 탐지해냄으로서 그 효과를 확인했다. 단, [20]에서는 함수 유닛 테스팅 목적으로 필요한 테스트 드라이버/스텁이 생성된 반면 본 논문에서는 테스크 단위로 테스팅을 수행했기 때문에 생성한 테스트 드라이버/스텁에 차이가 존재한다. 이 밖에도 [20]와 달리 본 논문에서는 테스트 드라이버에서 대상 함수를 1번이 아닌 반복 호출이 가능하도록 설정을 변경한 것과 static 전역 변수에 대해서도 심볼릭 선언하는 구문을 자동 추가하도록 변경하였다. 본 절에서는 동적 기호 실행 기반 자동화된 테스크 테스팅 기법에 대해 구체적으로 소개하도록 한다.

그림 2.3은 동적 기호 실행 기반 자동화된 테스크 테스팅 기법 입출력을 나타낸다. 입력으로 사용하는 <target>.i는 전처리 된 대상 파일을 의미한다. 원본 소스 코드(.c)가 아닌 전처리 된 코드를 입력으로 받는 이유는 전처리 된 파일에서는 헤더 파일이 코드로 확장되기 때문이다. 즉, 동적 기호 실행 기반 자동화된 테스크 테스팅 기법에서는 clang 라이브러리[17]를 사용해서 대상 소스 코드에서 사용하는 변수 타입 및 함수 정보 등을 알아낸 뒤, 이를 기반으로 테스트 드라이버/스텁 함수 및 심볼릭 선언 구문을 추가하는데, 원본 소스 코드(.c)를 입력으로 사용할 경우, 소스 코드가 헤더 파일에 감추어져 있어서 헤더 파일에서 정의한 변수의 타입이나 함수의 시그니처 등을 식별하기 어렵기 때문이다.

driver.c, stub.c, symbolic_decl_glob_static.h는 동적 기호 실행 기반 자동화된 테스크 테스팅 기법의 출력을 의미한다. driver.c, stub.c, symbolic_decl_glob_static.h는 차례대로 테스트 드라이버, 스텁 함수, 전역 static 변수에 대한 심볼릭 선언 구문을 포함한다.

```
target.c
1. int g_var1, g_var2;
2. static int g_static_var = 0;
3. extern int external_func();
4. static void func (int param) {
5.   if (external_func()==g_var1)
6.     g_var1++;
7. }
8.
9. int interface1 (int param) {
10.  if (g_var1==1) return 1;
11.  else return 2;
12.}
13.void interface2 (char param) {
14.  if (param=='a') func();
15.  if (g_static_var < 10)
16.    g_static_var++;
17.}
```

그림 2.4: 예제 코드 2

```
driver.c
1. extern int g_var1;
2. void interface1_driver (int iter) {
3.   for (int i=0;i<iter;i++) {
4.     int sym_var;
5.     SYM_int(sym_var);
6.     SYM_int(g_var1);
7.     symbolic_decl_glob_static();
8.     interface1 (sym_var);
9. } }
10.void interface2_driver (int iter) {
11.  for (int i=0;i<iter;i++) {
12.    char sym_var;
13.    SYM_char(sym_var);
14.    SYM_int(g_var1);
15.    symbolic_decl_glob_static();
16.    interface2 (sym_var);
17.} }
18.int main(int argc, char *argv[]) {
19.  int n = atoi(argv[1]);
20.  int iter = atoi(argv[2]);
21.  switch ( n ) {
22.    case 1:interface1_driver(iter);
23.           break;
23.    case 2:interface2_driver(iter);
24.           break;
25.  }
26.}
```

그림 2.5: 예제 코드 2의 테스트 드라이버

테스트 드라이버 소개

driver.c에는 대상 파일에 포함된 각 인터페이스 함수(non-static function)를 호출하는 테스트 드라이버가 존재한다. 예를 들면, 대상 파일에 인터페이스 f(), g()와 static 함수 h()가 존재할 때, 동적 기호 실행 기반 자동화된 테스크 테스팅 기법으로 생성한 driver.c에는 f()와 g()에 대한 테스트 드라이버(f_driver(), g_driver())가 각각 존재한다. 테스트 드라이버는 심볼릭 선언 구문과 인터페이스 함수 호출 구문을 포함한다. 심볼릭 선언은 대상 인터페이스 또는 대상 인터페이스의 하위함수(대상 인터페이스를 실행했을 때 호출될 수 있는 함수)가 사용하는 모든 전역 변수에 대해 적용된다. 추가로 대상 인터페이스가 매개변수를 갖는 경우, 인자로 넘길 변수 선언 후 심볼릭 선언을 적용한다. 마지막에는 인터페이스를 호출하는 구문을 테스트 드라이버에 추가한다.

그림 2.4는 대상 코드(target.c)이고, 그림 2.5은 동적 기호 실행 기반 자동화된 테스크 테스팅 기법으로 자동 생성한 테스트 드라이버 파일(driver.c) 이다. 대상 코드 내부에는 2개의 인터페이스 함수(interface1, interface2)가 존재한다. interface1()은 int형 매개 변수를 가지며, 10라인에서 전역 변수 g_var1을 사용하고 있다. Interface2()는 char형 매개 변수를 가지며, 14라인에서 함수 func를 호출하고 있다.

생성된 테스트 드라이버 파일(driver.c)를 살펴보면 대상 파일의 인터페이스마다 테스트 드라이버(interface1_driver, interface2_driver)가 생성된 것을 확인할 수 있다. main 함수에서는 각각 어떤 인터페이스를 실행할 것인지를 가리키는 식별값(n)과 테스트 드라이버에서 인터페이스를 몇 번 호출할 것인지를 나타내는 값(iter)을 19∼20라인에서 사용자로부터 입력받고 있다. 인터페이스를 1번 호출하는 것만으로는 실행할 수 없는 분기가 존재하기 때문에 대상 함수를 호출하는 횟수를 설정할 수 있도록하여 다양한 경로 탐색이 가능하도록 유도했다. 테스트 드라이버 내부를 살펴보면 심볼릭 선언 후 대상 함수를 호출하는 일련의 과정을

```
stub.c
1. int externanl_func () {
2.    int var_dummy;
3.    SYM_int(var_dummy);
4.    return var_dummy;
5. }
```

그림 2.6: 예제 코드 2의 스텁

```
symbolic_decl_glob_static.h
1. void symbolic_decl_glob_static () {
2.    SYM_int(g_static_var);
3. }
```

그림 2.7: 예제 코드 2의 static 전역 변수 심볼릭 선언 구문

iter만큼 수행하는 반복문이 존재하는 것을 볼 수 있다. 또한 실제 환경에서는 대상 인터페이스만 연속적으로 호출되는 것이 아니라 다른 인터페이스도 중간에 호출되어 전역 변수의 값을 변경할 수도 있으므로, 다양한 환경을 만들어주기 위해 테스트 드라이버의 반복문 내부에 심볼릭 선언 구문을 추가하여 반복마다 새롭게 심볼릭 선언이 되도록 유도했다. Interface2_driver를 살펴보면 13, 14라인에서 변수 sym_var, g_var1를 심볼릭 선언하고 있다. 변수 sym_var는 interface2 함수 호출 시 인자로 사용하고, g_var1은 interface2 함수를 호출했을 때 (func에서) 사용되는 전역변수이므로, 이 두 변수에 대해 심볼릭 선언하는 구문이 추가된다. 15라인의 함수 symbolic_decl_glob_static()는 파일에 선언된 static 전역 변수를 모두 심볼릭 선언하는 함수로, symbolic_decl_glob_static.h에 정의되어 있으며 후에 설명하도록 한다. 심볼릭 선언을 마쳤으면 16라인과 같이 대상 인터페이스 함수를 호출한다.

스텁 소개

본 논문의 실험에서는 테스크 단위로 테스팅을 수행한다. 따라서 대상이 아닌 파일에 있는 함수는 모두 테스크 범위에서 제외되기 때문에 스텁으로 대체한다. 즉, 대상 파일에 없는 함수 호출의 경우 실제 함수가 아닌 스텁을 호출하도록 설정했다.

그림 2.4(예제 코드 2)를 대상 파일이라 할 때, 그림 2.6은 동적 기호 실행 기반 자동화된 테스크 테스팅 기법에서 자동 생성한 스텁(stub.c)이다. 그림 2.4(예제 코드 2) 5라인에서 대상이 아닌 파일에 있는 함수 external_func를 호출하므로, stub.c에 해당 함수의 스텁이 생성된다. stub.c 파일의 1라인과 같이 함수 시그니처를 대체하는 함수(int external_func())와 동일하게 만들고, 반환형(return type)과 일치하는 변수를 선언 및 심볼릭 선언한 뒤 반환한다. 여기서는 external_func의 반환형이 int형이므로, 2~3라인에서 int형 변수 var_dummy를 선언 및 심볼릭 선언 후, 4라인에서 변수를 반환한다.

static 전역 변수 심볼릭 선언 구문 소개

static 전역 변수는 변수가 선언된 파일 내에서만 값을 참조/변경할 수 있기 때문에 이 변수에 대한 심볼릭 선언 구문 역시 static 전역 변수가 선언된 대상 파일 내에 추가해야 한다. 이를 위해 헤더 파일(symbolic_decl_glob_static.h)에 static 전역 변수에 대해 심볼릭 선언하는 함수(symbolic_decl_glob_static)를 추가하고, 대상 소스 코드에서 symbolic_decl_glob_static.h을 포함(include)시켰다. 이후 테스트 드라이버에서 symbolic_decl_glob_static 함수를 호출함으로서 대상 소스 코드 내에서 static 전역 변수를 심볼릭 선언하는 것과 같은 기능을 하도록 유도했다. 대상 소스 코드에서 사용하는 모든 static 전역 변수에 대해 심볼릭 선언했으며, symbolic_decl_glob_static 함수를 호출하는 구문은 테스트 드라이버마다 존재한다.

그림 2.7는 생성된 static 전역 변수 심볼릭 구문(symbolic_decl_glob_static.h)이다. 그림 2.6(예제 코드 2) 15라인에서 static 전역 변수 g_static_var를 사용하고 있으므로, symbolic_decl_glob_static.h 헤더의 symbolic_decl_glob_static 함수에서 해당 변수를 심볼릭 선언한다.

표 2.3: 데이터 타입에 따른 심볼릭 선언 형식

입력 값 자료형	기호 실행 입력
기본 자료형 bit field 자료형	해당 자료형에 맞는 기호 실행 입력 값 설정
배열	모든 배열 원소에 대해 배열 원소의 자료형에 맞는 기호 실행 입력 값 설정
구조체	구조체의 모든 필드에 대해 해당 자료형에 맞는 기호 실행 입력 값 설정
포인터	1. 구조체 포인터 P: 포인터 자료형 P가 가리키는 자료형인 *P의 메모리 크기 만큼 메모리를 할당하고 *T에 대해 동적 기호 실행 입력을 설정하여 포인터 입력 값으로 할당 2. 구조체가 아닌 포인터 Q: 포인터 자료형 Q가 가리키는 자료형인 *Q의 메모리 크기 10개(10개 원소)에 해당하는 만큼 메모리를 할당하고 각 원소 *Q에 대해 동적 기호 실행 입력을 설정하여 포인터 입력 값으로 할당

```
1. struct _linked_list{
2.   int data;
3.   struct _linked_list *next;
4. } var;
5.
6. SYM_int(var.data);
7. var.next = (_linked_list *)malloc(sizeof(_linked_list));
8. SYM_int((var.next_->data);
9. (var.next)->next = (_linked_list *)malloc(sizeof(_linked_list));
10.SYM_int(((var.next)->next)->data);
11.((var.next)->next)->next = (_linked_list *)malloc(sizeof(_linked_list));
12.SYM_int((((var.next)->next)->next)->data);
13.(((var.next)->next)->next)->next = NULL;
```

그림 2.8: 메모리 할당이 필요한 심볼릭 선언 예제

앞서 소개한 것과 같이 심볼릭 선언 구문은 테스트 드라이버(driver.c), 스텁(stub.c), static 전역 변수 심볼릭 선언(symbolic_decl_glob_static.h)에 추가된다. 테스트 드라이버(driver.c)에서는 인터페이스가 실행하는 테스크에서 사용하는 모든 전역 변수와 인터페이스 호출 시 넘겨줄 인자들에 대해 심볼릭 선언 구문을 추가한다. 스텁(stub.c)에서는 대상 파일에서 호출하는 함수 중 대상 파일에 속하지 않은 함수에 대해 스텁을 생성하며, 반환형과 일치하는 변수를 심볼릭 선언하고 반환한다. static 전역 변수 심볼릭 선언(symbolic_decl_glob_static.h)에서는 대상 파일에서 사용하는 모든 static 전역 변수를 심볼릭 선언한다. 변수 타입에 따라 추가되는 심볼릭 구문 형식은 표 2.3를 따른다. 함수 포인터 변수는 심볼릭 선언 대상에서 제외한다.

단, 구조체 T가 자신을 가리키는 포인터 *T를 필드로 가지고 T 타입의 변수를 심볼릭 선언한다고 할 때, 메모리를 연달아서 끊임없이 할당하여 메모리 부족 현상을 일으킬 수 있다. 이를 방지하기 위해 포인터 변수 심볼릭 변수 선언 시, 연속 3번까지만 메모리 할당을 허용하고, 그 이후부터 포인터 변수에 NULL을 대입하여 메모리 할당을 중지한다.

그림 2.8은 심볼릭 선언 시 메모리 할당이 연달아 필요한 예제이다. 1∼4라인에서는 심볼릭 선언할 구조체 변수 var를 정의한다. 6∼13라인은 변수 var에 대한 심볼릭 선언 구문이다. 구조체 필드 data는 기본

자료형이기 때문에 SYM_int() API를 사용해서 해당 자료형에 맞는 기호 실행 입력 값을 설정한다. 구조체 필드 next는 구조체 포인터이기 때문에 메모리를 할당한 뒤, 할당한 메모리에 대해 심볼릭 선언 작업을 진행한다. 13번 라인에서는 이미 3번의 메모리를 연속해서 할당(7, 9, 11라인)했기 때문에 심볼릭 선언할 포인터 필드에 NULL을 대입하고 심볼릭 선언을 마친다.

제 3 장 관련 연구

3.1 Concolic 테스팅 도구들의 대상 언어 및 사용 가능 여부

Concolic 기법 도구들은 대상 언어, 대상 플랫폼, 입력 형식(ex. 소스 기반 형식, 바이너리)등에 따라 다양한 도구가 존재한다. 본 장에서는 기존의 Concolic 도구를 살펴보고 다양한 Concolic 도구 중에 CREST를 기반으로 CROWN(본 논문에서 효율적인 실험을 위해 자체 개발한 도구)을 확장한 이유를 살펴본다. 표 3.1는 Concolic 테스팅 도구 목록이다. 각 도구가 대상으로 삼는 언어, 입력 파일 형식 및 도구 사용 여부 정보가 표시되어 있다. 입력 파일 형식이 바이너리(binary)인 도구는 대상 언어가 정해져있지 않은데, 이는 대상 언어에 구분 없이 컴파일하여 바이너리를 생성하면 해당 바이너리를 입력으로 사용할 수 있기 때문이다.

본 논문에서는 실제 환경에서 많이 사용하는 C 프로그램을 대상으로 가이드라인을 제공하고자 했기 때문에 CROWN의 기본이 되는 도구로서, C 프로그램을 지원하지 않는 Concolic 도구는 후보에서 제외했다. 표에서 보면 jCute, jFuzz, Pex, ... 등이 C가 아닌 언어를 대상으로 삼고있다. 다음으로 입력 파일 형식으로 바이너리를 가지는 도구 또한 후보에서 제외했다. 본 논문에서는 가이드라인을 적용하기 위해 먼저 각 분기의 미달성 원인을 소스 코드 단위에서 분석을 해야하므로 테스팅 결과를 코드와 직접 매칭하는 것이 가능하고 분석이 쉬운 코드 기반의 도구를 선호하고 있다. 표에서 바이너리를 대상으로 하는 도구는 SAGE, Bap, Triton, Angr등이 있다. 다음으로 현재 사용이 불가능한 도구를 제외하면 CREST와 KLEE, 2가지 도구만 후보로 남는다. 두 도구 모두 많이 사용되는 도구지만 임베디드 시스템을 많이 사용하는 국내 환경의 특성을 고려했을 때 가상 머신을 활용한 KLEE보다 가볍게 대상 프로그램에 대한 instrumentation 작업만 수행하여 실행이 가능한 CREST가 실제 환경에 더 적합하다고 보았고, 추가로 CREST에서 직접 제공하는 정보 중 테스팅 결과 분석 시에 유용하게 사용할 수 있는 심볼릭 정보와 달성한 분기 식별 ID 정보 등은 KLEE에서는 제공하지 않는 정보이므로 CREST를 기반으로 CROWN 개발에 착수했다.

3.2 Concolic 테스팅을 활용한 사례 연구 비교

본 절에서는 Concolic 기법을 사용한 기존 사례 연구와 본 논문에서 진행한 사례 연구를 비교함으로서 기존 사례와의 차별성을 보이고자 한다. 본 논문의 사례 연구에서는 C 대상 Concolic 테스팅을 위해 구축한 환경을 구체적으로 언급하고 테스팅을 수행했을 때 달성하지 못한 각각의 분기 유형들을 예시와 함께 소개하여, 사용자로 하여금 실제 미달성 분기들에 대한 구체적인 정보를 제공했다. 이후, 각 미달성 유형을 달성할 수 있는 가이드라인을 제공함으로서 사용자가 테스팅 환경을 재구축할 때 활용할 수 있는 기준을 마련했다.

사례 연구를 비교했을 때, 첫 번째로 테스팅을 위해 구축한 환경, 사용한 대상 프로그램 종류 또는 사용한 Concolic 테스팅 도구와 같이 실험 환경 측면에서 본 논문에서 수행한 사례 연구와 차이를 갖는 연구들을 발견할 수 있었다.

[21]은 바이너리 기반 Concolic 테스팅 도구들에 대한 사례 연구를 진행했다. 바이너리 기반 Concolic 테스팅 도구는 대상 프로그램을 바이너리 형식으로 입력받는다. 해당 논문에서는 바이너리 기반 Concolic 테스팅 도구로 테스팅 수행 시 발생 가능한 문제점들을 항목별로 정리하였고 사례 연구에서 작은 규모의 바이너리 프로그램을 대상으로 Concolic 테스팅을 수행했을 때, 항목별로 정리한 문제점들이 실제로 발생하는 것을 보였다. 이로 인해, 현재 존재하는 바이너리 기반 Concolic 도구를 사용해서 완벽한 Concolic 테스팅을

표 3.1: Concolic 기법 도구 별 지원 플랫폼 목록

도구 이름	대상 언어	입력 파일 형식	publicly available
CREST[1]	C	code based	O
DART[2]	C	code based	X
SMART[3]	C	code based	X
CUTE[4]	C	code based	X
EXE[5]	C	code based	X
KLEE[6]	C	code based	O
Rwset[7]	C	code based	X
PathCrawler[8]	C	code based	X
jCute[4]	Java	code based	O
JFuzz[9]	Java	code based	O
Pex[10]	.Net	code based	O
SAGE[11]	-	binary	X
NA[12]	PHP	code based	X
Apollo[13]	PHP	code based	X
Jalangi[30]	Java script	code based	O
Bap[31]	-	binary	O
Triton[32]	-	binary	O
Angr[33]	-	binary	O

수행하기는 어려운 점과 한계가 있음을 보였다. 해당 논문에서 정리한 문제점 중에는 바이너리로부터 명령어(instruction) 추출에 실패하여 중간 코드로 해석할수 없는 경우와 같이 바이너리 단계에서 테스팅이 진행될 때에 한정하여 발생가능한 문제도 존재했다. 저자의 논문에서는 소스 코드 기반의 Concolic 테스팅 도구를 사용했으며, 미달성 분기 유형 및 가이드라인을 바이너리가 아닌 소스 코드 단계에서 다루고 있기 때문에 해당 논문에서 다루는 문제와는 성격이 다르고 발생할 수 없는 문제 영역이라고 볼 수 있다.

[29]에서는 현존하는 Concolic 기법 도구들에 대해 객관적인 실험을 수행하고 결과를 보여주고자 했다. Concolic 기법 도구를 평가하는 단일 기준 등이 없기 때문에 Concolic 기법 도구를 소개하는 논문에서 제각각 유리한 방식의 기준이나 테스팅 환경을 구축하고 실험하여 사실상 객관성이 떨어진다는 점을 해당 논문에서는 문제로 삼고 있다. 따라서 해당 논문에서는 Concolic 도구 실행 시 한계나 에러가 발생하기 쉬운 패턴별로 총 300개의 단편 코드를 만들어 각 도구에서 실행시킨 뒤, 각 도구의 특징들을 분석했다. 각 단편 코드를 실행한 결과는 테스트 생성 실패, timeout 발생, ...등과 같이 총 5개 유형으로 분류하여 나타냈다. 해당 논문에서는 Concolic 기법 도구 중 플랫폼이 Java, .NET인 도구 중 일부 도구(CATG, EvoSuite, PET, Pex, SPF)를 선별하여 실험을 진행했다. 저자의 사례 연구에서는 C 프로그램을 대상으로 삼고있지만 [29]에서는 C 대상 Concolic 기법 도구는 대상에 포함되지 않았으며, 초점을 미달성된 영역보다는 각 Concolic 테스팅 도구의 객관적인 평가에 맞추고 사례연구를 진행했다.

[23]에서는 플래시 메모리를 대상으로 효과적인 C 대상 Concolic 테스팅을 위한 프레임워크를 제시했으며 실험에서는 C 대상 Concolic 도구인 CREST를 사용했다. 해당 논문에서는 여러 개의 섹터로 구성된 플래시 메모리를 테스팅하기 위해 모델 및 필요한 환경 등을 디자인했으며, 연산량이 많아 충분한 경로를 탐색(heavy computational cost) 못 하는 상황을 막기 위해 CREST를 수정하여 많은 컴퓨터 노드에서 작업

을 분배하여 담당할 수 있도록 하는 기법(distributed concolic algorithm)을 만들어서 사용했다. 이로 인해, 기존 CREST에서는 여러 대의 컴퓨터가 존재해도 하나의 호스트에서만 테스팅이 가능했지만, 작업량의 분배가 가능해짐에 따라 여러 대의 컴퓨터가 테스팅을 분담하는 것이 가능해졌다. 이는 플래시 메모리 테스팅을 위해 만든 디자인에서 독립적으로 수행하는 루프가 존재했기 때문에, 독립적으로 동작하는 부분을 여러 개의 동작으로 나눔으로서 가능케 했다. 해당 논문은 플래시 메모리에 특화된 대상 프로그램을 대상으로한 프레임워크를 제안하고 테스팅을 수행했기 때문에 비교적 일반적인 프로그램을 대상으로 테스팅을 진행한 저자의 논문과는 성격이 다르다고 볼 수 있다.

두 번째로 실제 환경에서 사용되는 프로그램을 대상으로 사용하고, C 대상 Concolic 테스팅을 진행했다는 점에서 저자의 사례 연구와 동일한 관련 연구들이 존재했지만 달성하지 못한 영역에 초점을 맞추어 예제를 소개하거나 분석을 수행한 연구를 찾아볼 수는 없었다.

[22]에서는 스마트폰을 비롯한 소프트웨어 탑재 기기들의 추가 기능은 늘어나고 개발 주기가 짧아지면서 공개 소프트웨어의 탑재를 가속하고 있기 때문에 공개 소프트웨어에 대한 상세 지식 없이 테스팅을 수행하는 경우가 발생한다고 소개하고 있다. 해당 논문에서는 이러한 세부 배경 지식이 없는 공개 소프트웨어를 대상으로 Concolic 테스팅을 적용하여 효과적인 결과를 보일 수 있음을 보인다. 해당 논문에서는 기업과 협업하여 기업에서 사용하는 오픈 소스 라이브러리 libexif에 대한 테스팅을 수행했다. 서로 다른 2가지 환경 설정에 대해 실험이 진행되었으며, 사용 도구는 CREST-BV(bit vector 연산 지원 버전)과 KLEE가 사용됐다. 설정한 첫 번째 환경 설정은 libexif가 사용하는 파일의 모든 내용을 심볼릭 설정하는 것이고, 두 번째 환경 설정은 탐색 공간을 훨씬 한정하여 파일 내용 중 많은 분기에 영향을 미치는 일부분에 대해서만 심볼릭 설정하고 나머지는 실제 값(Concrete value)을 사용했다. CREST-BV를 사용하여 결과적으로 첫 번째 설정에서 1개의 버그만을 탐지하고, 두 번째 설정에서 5개의 버그를 성공적으로 탐지했다. KLEE를 사용한 경우 첫 번째 설정, 두 번째 설정 모두 1개의 버그만을 탐지하여 CREST-BV가 본 실험에서 효과적이었다는 것을 보였다. 이 실험을 통해 상세 지식이 없는 공개 소프트웨어를 대상으로도 Concolic 테스팅을 적용하여 성공적인 결과를 도출한 것을 보였으며, 첫 번째, 두 번째 설정과 같이 사용자가 정의한 환경 설정에 따라 테스팅의 효과가 크게 다른 것을 보였다. 해당 논문에서 미달성 분기 등은 주요 쟁점이 아니었기 때문에 전체 커버리지 결과만 존재했고, 미달성 분기에 대한 구체적 언급이나 분석은 없었다. 이 점에 대해 저자의 논문과 다르다고 볼 수 있다.

[14]에서는 오픈 소스와 인더스트리(ABB) 소스 총 6개를 대상으로 실험을 진행했다. 이 중, 오픈 소스는 C로 작성된 gcc, linux 프로그램과 Java로 작성된 Jboss, Eclipse를 사용했으며 인더스트리 소스는 C로 작성된 실시간 임베디드 시스템(ABB1)과 Java로 작성된 산업 GUI 시스템(ABB2)를 사용했다. 대상 C 프로그램과 Java 프로그램에 대해 각각 CREST, KLEE 도구를 적용했으며, 각각 함수들을 복잡도(Cyclomatic complexity)에 따라 3개의 카테고리로 분류한 뒤 각 카테고리에서 20개씩 임의로 뽑아 함수 단위 유닛 테스팅을 수행했다. 해당 논문에서는 잘 알려진 Concolic 테스팅 도구들의 한계(실수형 입력 변수 지원여부, 포인터 지원여부, 외부라이브러리 함수 호출, BV(bit vector)연산 등)들을 항목별로 분류한 뒤 각 항목이 6개의 대상 소스에 얼마큼 분포하는지 조사했고, CREST와 KLEE에서 기록한 분기 커버리지를 보였다. 이후, 결과 커버리지를 앞서 분류한 테스팅 도구들의 한계 항목과 연관 지어 설명했다. 해당 논문은 C 대상 인더스트리 소스를 대상으로 사례 연구를 진행했다는 점에서 본 논문과 공통점이 있지만 Concolic 기법 테스팅을 위해 구축한 환경에 대한 언급이 없고, 좋은 결과를 위해 테스팅 환경을 수정하는 작업을 반복했다고 언급하고 있다. 더불어 저자의 사례 연구에서는 달성 못 한 분기를 유형별로 나누어 구체적으로 분석한 반면, 해당 논문에서는 잘 알려져 있는 Concolic 기법 도구의 한계들이 얼마나 분포하는지 통계를 내고 이로인한 분기 커버리지 양상을 개괄적으로 보인다는 점에서 차이가 존재한다.

마지막으로 본 논문과 같이 Concolic 테스팅을 위한 가이드라인 혹은 프레임워크를 제안한 연구들은

존재했으나, 미달성 영역에 초점을 맞추어 해당 영역을 달성하기 위해 진행한 연구는 찾아볼 수 없었다.

저자의 논문은 커버리지 향상에 목적에 두고 있다면 [34]은 질 높은 테스트의 생성(가정에 벗어나는 테스트의 생성을 억제하고 로직 에러를 일으키는 테스트의 생성)에 목적을 두었다는 점이 다르다. 구체적으로 해당 논문은 선행조건(pre condition)과 후행조건(post condition)이 삽입된 테스트 드라이버 프레임워크를 사용해서 Concolic 테스팅을 수행하는 가이드라인을 제공한다. 선행조건은 대상 프로그램이 가정하고 있는 입력값이 맞는지를 검사하는 조건으로, 가정하지 않은 입력값에 의해 발생하는 잘못된 런타임 에러를 제거하기 위해 사용된다. 이와 같이 입력 값의 검증을 위한 방법으로 선행조건을 도입한 프레임워크는 앞서 [35]에서 소개되었다. 후행 조건은 속성 기반의 테스팅[36, 37, 38, 39]의 개념을 사용한 것으로, 프로그램의 결과 값이 만족해야하는 조건(oracle)을 후행 조건에 추가하면 Concolic 기법의 특성상 후행 조건을 만족하지 않는, 즉 로직 에러를 일으키는 테스트도 생성할 수 있다는 점에 착안하여 추가되었다.

[20]에서는 Concolic 유닛 테스팅 적용을 위해 필요한 심볼릭 설정 및 테스트 드라이버/스텁 생성과 같은 환경 구축을 자동화하는 프레임워크를 개발하여 규모가 큰 소프트웨어를 대상으로도 손쉬운 Concolic 유닛 테스팅이 가능하도록 만들었다. 더불어 효과적이고 정확한 버그 탐지를 위해서 가정(assumption) 및 표명(assertion)구문을 삽입하는 휴리스틱도 함께 소개하고 있다. 해당 논문에서는 이 프레임워크를 500만 라인 규모의 산업 임베디드 소프트웨어에 적용하여 버그탐지 효과를 확인하고자 했으며 결과적으로 25425개의 대상 함수에 대해 24개의 crash 오류를 탐지했다. 저자의 논문에서도 해당 프레임워크를 기반으로 테스팅 환경 구축 후 테스팅을 수행했다. 해당 논문은 Concolic 테스팅을 위해 새롭게 테스팅 환경을 자동으로 구축하는 프레임워크를 제공하는 반면, 저자의 논문에서는 테스팅을 수행한 후, 탐색하지 못한 영역에 초점을 맞추어 커버리지를 높이기 위한 테스팅 환경 재구축 가이드라인을 제공하고 있다는 점에서 차이가 존재한다.

제 4 장 CROWN의 향상 기능 소개

CROWN은 기존 C 프로그램 대상 Concolic 테스팅 도구인 CREST를 확장하여 개발한 도구로, 본 논문의 실험에서는 미달성 분기 분석 작업을 효율적으로 진행하기 위해 기존 Concolic 테스팅 도구의 부족한 UI 측면을 향상하여 개발한 도구 CROWN를 사용한다. 본 장에서는 생산성을 높이고 테스팅 결과 분석을 빠르게 할 수 있도록 CROWN에서 개선된 UI를 소개한다.

4.1 사용자 친화적으로 UI 개선

CROWN에서는 사용자가 이해하기 어려웠던 기존 심볼릭 정보 출력을 사용자 친화적으로 변경하였고, 커버리지 결과 분석에 필요한 사용자 친화적인 분기 정보를 출력하는 기능을 새로 추가했다.

Concolic 테스팅 기법 소개 시 언급한 것과 마찬가지로 CROWN 역시 테스팅하기 전에 먼저 사용자가 심볼릭 설정 등과 같이 대상 코드를 세팅해주는 작업이 필요하다. 당연히 이 세팅에 따라 생성되는 테스트 케이스들과 달성 분기 커버리지가 달라지며 높은 분기 커버리지를 달성하기 위해서는 대상 코드에 따라 적절한 설정을 해주는 것이 필요하다. 이를 위해서 CROWN 실행 후 커버리지 결과가 낮을 경우 해당 원인을 분석하고, 분석한 결과를 피드백으로 사용하여 대상 코드를 다시 세팅하고 테스팅할 수 있다. 이러한 과정을 통해 점차 분기 커버리지를 높여갈 수 있다. 그림 4.1은 이 과정을 도식화하여 표현한 것이다.

따라서 CROWN에서는 커버리지 결과 분석에 도움이 되는 정보들을 사용자에게 알기 쉬운 형태로 추가 제공함으로서 커버리지 결과에 대한 정확한 분석이 가능하도록 하고 분석에 걸리는 시간을 크게 줄이고자 했다.

4.1.1 심볼릭 정보 출력

print_execution은 CROWN으로 대상 프로그램을 실행시킨 후 사용 가능한 명령어로, 대상 프로그램을 실행했을 때 프로브(probe)들이 수집한 심볼릭 정보들을 출력하는 기능을 한다. 심볼릭 정보에는 심볼릭 변수가 갖는 값, 심볼릭 경로 수식, 실행된 분기 ID 시퀀스가 출력된다. 이러한 정보들을 바탕으로, 사용자는 각각의 심볼릭 변수가 갖는 값과 그에 대응되는 심볼릭 경로 수식 및 실행된 분기 순서 등을 확인할 수 있다. 본 절에서는 개선 전의 print_execution 출력과 그로 인한 문제점 및 개선한 사항들을 차례대로 소개한다.

개선 전 심볼릭 정보 출력

그림 4.2은 심볼릭 변수 선언을 마친 예제 소스 코드이다. 이 코드에 대해 CROWN 실행 후, 개선 전의 print_execution 출력결과는 그림 4.3와 같다. 빈 줄을 기준으로 세 부분으로 분리되어 있으며 차례대로 심볼릭 변수가 갖는 실제(concrete) 값, 심볼릭 경로 수식, 실행된 분기 ID 시퀀스를 포함한다. 즉, 1~2 라인은 심볼릭 변수가 갖는 값 정보, 4~5 라인은 심볼릭 경로 수식 정보, 7~10 라인은 실행된 분기 ID 시퀀스를 나타낸다. 분기 ID는 대상 프로그램의 분기가 갖는 식별값이며 후에 설명하도록 한다. 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 실행된 분기 ID들이며 -1은 프로그램 실행 중 함수(main)의 진입을 뜻하고, -2는 실행 중 함수(main)에서 빠져나감을 의미한다. 여기서 분기 ID 3은 분기 (var_a >

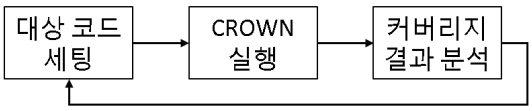

그림 4.1: 달성 커버리지 향상을 위한 CROWN 적용 사이클

```
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 4.2: 예제 코드 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 4.3: print_execution의 개선 전 출력

10.0)을 의미하고, 분기 ID 7은 분기 !(var_b < -10)을 의미한다.

print_execution 출력 개선이 필요한 항목

1. 출력되는 심볼릭 변수명과 소스 코드에 선언된 심볼릭 변수명의 불일치

print_execution 출력에서 심볼릭 변수명은 심볼릭 변수가 선언된 순서대로 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.

예를 들면, 그림 4.3의 심볼릭 변수가 갖는 값 정보(1~2 라인)와 심볼릭 경로 수식(4~5 라인)에서 실제 심볼릭 변수명이 아닌, 가명 x0, x1이 사용되고 있다. 따라서 가명 x0, x1에 매칭되는 실제 심볼릭 변수명을 원본 소스코드에서 찾는 작업이 필요하다. 원본 코드 그림 4.2의 7~8라인에서 변수 var_a와 var_b를 차례대로 심볼릭 선언하고 있으므로, print_execution에서 심볼릭 변수 가명 x0, x1에 매칭되는 실제 심볼릭 변수명은 각각 var_a와 var_b인 것을 확인할 수 있다. 여기서 소개하는 예제는 간단한 프로그램이기 때문에 심볼릭 변수 가명을 쉽게 매칭시킬 수 있었지만, 소스 코드가 더 길어지고 복잡해질 경우 매칭 작업은 훨씬 많은 시간이 걸릴 수 있다. 특히, 심볼릭 변수 선언 구문이 조건문 내부에 위치하거나 여러 함수에 분포할 경우 실행 경로를 따라가면서 매칭 작업을 수행해야 하므로 상당히 까다로운 작업이 될 수 있다.

표 4.1: print_exeuction에서의 형 변환 출력

변환 할 자료형	형 변환 출력 구문
char	s_cast[8]
unsigned char	u_cast[8]
short	s_cast[16]
unsigned short	u_cast[16]
int	s_cast[32]
unsigned int	u_cast[32]
long	s_cast[32]
unsigned long	u_cast[32]
long long	s_cast[64]
unsigned long long	u_cast[64]
float	s_cast[32]
double	s_cast[64]

2. 전위 표기 형식(prefix notation)을 따르는 심볼릭 경로 수식 출력

print_execution 명령어를 사용하여 심볼릭 경로 수식 출력 시, 일반 사용자에게 익숙한 중위 표기 형식(infix notation)이 아닌 전위 표기 형식(prefix notation)으로 출력되기 때문에 심볼릭 경로 수식을 이해하기 위해서 사용자는 출력된 식을 중위 표기 형식으로 재배열하는 추가 작업이 필요하다. 이러한 추가 작업은 분석 작업의 효율을 떨어뜨리며, 심볼릭 경로 수식이 길어지거나 복잡할수록 재배열 과정에서 더 많은 어려움이 존재한다.

3. 심볼릭 경로 수식에 출력되는 형 변환 표기 형식의 차이

심볼릭 경로 수식에 형 변환(casting) 구문이 포함된 경우, 형 변환된 자료형(int, char, short, ...)이 아닌 <sign>_[size] 형식(s_cast[16], u_cast[32], s_cast[8], ...)과 같이 자료형의 부호(s: signed, u: unsigned)와 크기가 출력된다. 표 4.1는 각 자료형으로 형 변환 시 심볼릭 경로 수식에 출력되는 구문을 나타낸다. 이는 사용자가 출력된 형 변환 출력 형식을 확인했을 때, 형 변환된 자료형의 식별을 어렵게 한다. 특히, 부호와 크기가 같은 자료형은 구분이 불가능하다. 예를 들면, int형과 float형은 signed 부호와 32bit 크기를 가지므로 형 변환 출력 구문이 s_cast[32]로 동일하고, 구분이 불가능한 것을 알 수 있다.

4. 위치 정보(line, file) 부재

심볼릭 변수의 선언 위치, 심볼릭 경로 수식에서 각 조건의 위치, 실행된 분기 ID의 위치 정보가 함께 출력되지 않기 때문에 사용자는 대응되는 부분을 소스 코드에서 확인하고자 할 때, 어려움이 존재한다.

```
1.  Symbolic variables & assigned values
2.  (var_a = 32770)    [ Line: 7, File: main.c ]
3.  (var_b = 0)        [ Line: 8, File: main.c ]
4.
5.  Symbolic path formula
6.  ((double) var_a > 10) [ Line: 10, File: main.c ]
7.  ! (var_b < -10)       [ Line: 12, File: main.c ]
8.
9.  Branch ID sequence
10. -1  [ main enters ]
11. 3   [ Line: 10, File: main.c ]
12. 7   [ Line: 12, File: main.c ]
13. -2  [ main exits ]
```

그림 4.4: print_execution의 개선 후 출력

개선 후 심볼릭 정보 출력

그림 4.4은 CROWN에서 실행한 그림 4.2에 대한 개선 후 print_execution 출력 결과이다. 그림 4.4에서 1, 5, 9라인에 각각 심볼릭 변수가 갖는 값 정보, 심볼릭 경로 수식, 실행된 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 선언된 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. 심볼릭 경로 수식의 각 조건이 위치한 정보 나타나는 것을 확인할 수 있다.
개선된 항목은 다음과 같다.

1. 심볼릭 변수명 출력 시 기본적으로 실제 변수명이 사용되며, 사용자가 출력할 변수명 정의 가능
   기존 1개의 인자(심볼릭 선언할 변수)를 갖는 심볼릭 선언 API를 최대 2개의 인자로 받도록 수정.
   만일 사용자가 2번째 인자를 추가할 경우, 심볼릭 변수명 출력 시 2번째 인자가 변수명으로 출력 됨.
2. 출력되는 심볼릭 경로 수식을 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

4.1.2 사용자 친화적인 분기 정보

사용자 친화적인 분기 정보는 CROWN에 새로 추가된 기능이다. gen_new_cov 명령(스크립트) 실행 시 사용자 친화적인 분기 정보가 담긴 파일 “new_coverage”을 생성한다. 본 절에서는 new_coverage 파일 내용을 첫 번째로 소개한다. 이후, CROWN에서 분기 정보를 출력하기 위해 사용하는 Instrumentation 파일과 분기 ID를 설명하고 마지막으로 new_coverage 파일의 필요성과 구현을 소개한다.

```
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)
8.     printf("reach");
9. }
```

그림 4.5: 예제 코드 4-1 (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 4.6: 예제 코드 4-2 (target2.c)

new_coverage 파일

사용자 친화적인 분기 정보가 담긴 new_coverage 파일 구성은 다음과 같다. 먼저 상세 분기 정보와 분기 요약 정보 두 부분으로 구성된다. 상세 분기 정보에서는 분기 ID, 분기 식(달성에 필요한 조건) 및 분기가 위치한 라인 정보들이 파일별, 함수별, 분기의 달성/미달성 별로 분류되어 나타난다. 분기 요약 정보에서는 파일별, 함수별로 달성한 분기 개수, 미달성한 분기 개수, 전체 분기 개수, 분기 커버리지를 보여준다. 그리고 달성한 전체 분기 개수, 미달성한 전체 분기 개수, 전체 분기 개수, 전체 분기 커버리지가 뒤따른다.

그림 4.5과 그림 4.6는 심볼릭 변수가 선언된 예제 소스 코드이다. 그림 4.6에서는 6, 7라인에서 두 변수 var_a와 var_b를 심볼릭 선언하고 있으며, 11라인에서 그림 4.5에 포함된 함수 func를 호출하고 있다. 특히, 8라인의 조건 var_a > 10으로 인해 9라인의 조건 var_a < -10은 달성할 수 없는 분기이다. 그림 4.5에서는 6라인에 변수 var_c를 심볼릭 선언하고 있다.

그림 4.7은 “new_coverage” 파일 내용이다. 1~20 라인은 분기 상세 정보를 나타내며, 22~32 라인은 분기 요약 정보를 나타낸다. 분기 상세 정보에서 1~7라인은 그림 4.5에 속한 분기 정보를 나타내며 9~20 라인은 그림 4.6에 속한 분기 정보를 나타낸다. 파일마다 포함된 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 4.6의 main 함수 내부에 포함된 9라인 분기(var_a < -10)임을 확인할 수 있다.

Instrumentation 적용 파일과 분기 ID

.cil.c 파일은 CROWN에서 instrumentation 단계를 마치고 생성되는 파일이다. 대상 소스 코드에 존재했던 복합 조건문들은 .cil.c 파일에서 단일 조건문들로 변환된다. 더불어 각 단일 조건문, 즉 분기 지점마다 __CrownBranch 프로브가 삽입된다. 해당 프로브는 여러 개의 인자를 가지며, 이 중 2번째 인자를 분기 ID라고 정의한다. 분기 ID는 분기를 식별하기 위해 사용되는 ID이다. CROWN은 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.

예를들면 그림 4.8는 파일(example.c)은 테스팅을 수행할 대상 소스 코드를 나타낸다. 그림 4.9은 대상 소스 코드에 instrumentation을 적용한 코드(example.cil.c)로서 편의상 분기 ID와 관계없는 프로브들을 제거하고 필요한 부분만 간추려 나타냈다. 그림에서 확인할 수 있듯이 example.c 7라인에 해당하는 복합

```
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.Summary
23.Source file: target1.c
24.| Function name | cov BR# | uncov BR# | total BR# | cov rate(%)|
25.| func          |       2 |         0 |         2 |      100.0 |
26.Source file: target2.c
27.| Function name | cov BR# | uncov BR# | total BR# | cov rate(%)|
28.| main          |       5 |         1 |         6 |       83.3 |
29.
30.Total Coverage
31.| cov BR# | uncov BR# | total BR# | cov rate(%)|
32.|       7 |         1 |         8 |       87.5 |
```

그림 4.7: new_coverage 파일 내용

조건문이 그림 example.cil.c에서 각각 1라인, 3라인의 단일 조건문으로 변환되었고, 이로 인해 분기가 더 세분화된 것을 확인할 수 있다. 각 분기 지점마다 총 6개의 __CrownBranch 프로브가 삽입되었으며, 각각 분기 ID 3(2라인), 4(4라인), 5(6라인), 6(10라인), 7(14라인), 8(18라인)을 갖는 것을 확인할 수 있다. ID 3을 갖는 분기는 "var_a > 10"이며, 해당 조건을 만족해야 분기가 실행될 수 있다. 이와 마찬가지로 ID 4를 갖는 분기는 "var_a > 10 && var_b > 10", ID 5를 갖는 분기는 "var_a > 10 && var_b > 10 && var_a < -10", ID 6을 갖는 분기는 "var_a > 10 && var_b > 10 && !(var_a < -10)", ...임을 파악할 수 있다.

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

그림 4.8: 예제 코드 5(example.c)

```
...
1. if(var_a>10) {
2.   __CrownBranch(14,3,1);
3.   if(var_b > 10) {
4.     __CrownBranch(19,4,1);
5.     if(var_a<-10) {
6.       __CrownBranch(24,5,1);
7.       printf("infeasible");
8.     }
9.     else {
10.      __CrownBranch(25,6,0);
11.    }
12.  }
13.  else {
14.    __CrownBranch(20,7,0);
15.  }
16.}
17.else {
18.  __CrownBranch(15,8,0);
19.}
...
```

그림 4.9: Instrumentation 적용 코드(example.cil.c)

new_coverage 파일의 필요성

기존에는 "coverage" 파일이 분기 정보를 확인할 수 있는 유일한 파일이었고, 해당 파일에는 3, 4, 6, 7, 8과 같이 달성한 분기 ID들만 개행(new line)으로 분리되어 나열되어 있기 때문에 사용자가 .cil.c 소스 코드와 분기 ID를 비교하면서 달성한 분기와 달성하지 못한 분기를 가려내는 추가 작업을 수행해야 했다. 또한 coverage 파일에는 소스 코드, 함수별로 분기 ID가 나누어진 것이 아니라서 어느 소스 코드, 함수에서 얼마 정도의 분기를 달성했는지에 대한 동향을 한눈에 파악하는 데 한계가 존재했다. 분석 작업에 착수할 때, 달성한 분기의 동향을 파악한 뒤 우선 미달성 분기가 밀집된 부분 위주로 분석하는 것이 효율적으로 커버리지를 향상할 수 있는 방법이기 때문에 달성 분기에 대한 동향은 분석 작업의 우선순위를 결정하는데 반드시 필요하고 파악해야 하는 정보이다.

테스팅 후 원본 소스 코드에서 어떤 분기를 달성했는지 식별하기 위해 사용자는 다음과 같은 작업을 기존에 수행해야 했다. 먼저, coverage 파일에서 달성한 분기의 ID를 확인하고, 해당 ID를 갖는 분기의 위치를 .cil.c 파일에서 찾는다. 분기 위치는 coverage 파일에서 확인한 분기 ID가 .cil.c 파일에 있는 __CrownBranch 프로브의 두 번째 인자와 일치하는지 비교함으로서 확인이 가능하다. 예를 들면, coverage 파일에서 확인한 달성 분기 ID가 7일 때, example.cil.c 파일 14라인 __CrownBranch 프로브의 두 번째 인자가 7이므로 해당 위치가 분기 ID 7의 위치임을 확인할 수 있다. 이후에는 해당하는 분기를 구성하고 원본 소스 코드에서 해당하는 분기를 매칭한다. 원본 소스 코드의 복합 조건문은 Instrumentation 작업 후 단일 조건문으로 변형되기 때문에 각각의 단일 조건문을 합성하여 분기를 구성하는 작업이 필요하다. 예를 들면, example.cil.c 14라인이 실행되기 위해서는 2개 조건 var_a > 10, !(var_b > 10)을 만족해야 하므로 var_a > 10 && !(var_b > 10)와 같이 단일 조건을 합쳐 분기를 구성한다. 다음으로 구성한 분기를 원본 소스 코드와 매칭시켜서 원본 코드에서 어떤 분기 또는 조건을 달성했는지 확인할 수 있다. 즉, 구성한 분기 var_a > 10 && !(var_b > 10)는 원본 소스 코드(example.c) 7라인 조건을 만족하지 못하기 때문에 12라인에 해당하는 else문을 실

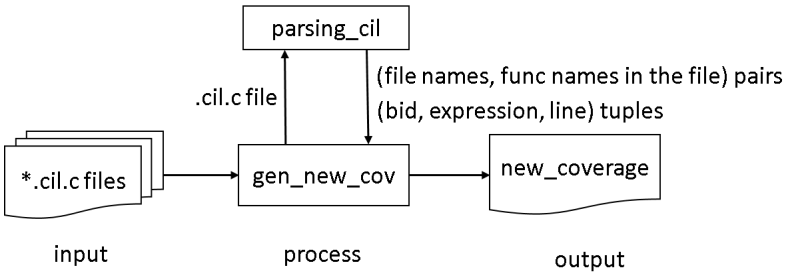

그림 4.10: 사용자 친화적인 분기 정보 출력 과정

행한 것을 확인할 수 있다. 이와 같이 기존에는 달성한 분기 1개를 식별하기 위해서 여러 단계의 추가작업을 수행해야 했고, 분기 각각의 개별 정보가 아닌 분기 달성 동향과 같이 전체적인 정보를 파악하는 것은 무리가 존재했다.

이러한 문제들을 해결하기 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. 이로 인해 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 사용자의 부담을 크게 줄였다고 볼 수 있다.

사용자 친화적인 분기 정보 구현

그림 4.10는 사용자 친화적인 분기 정보를 구하는 과정을 도식화한 것이다. 사용자 친화적인 분기 정보 출력 기능은 .cil.c 파일들을 입력으로 사용하여 사용자 친화적인 분기 정보가 담긴 파일 new_coverage를 출력한다. gen_new_cov는 bash 스크립트로서, 사용자는 해당 스크립트를 실행하여 new_coverage 파일을 생성할 수 있다.

사용자 친화적인 분기 정보 출력 기능 구현은 크게 입/출력 구현은 크게 두 부분으로 분류할 수 있다. 첫 번째 부분은 입/출력 부분이고, 두 번째 부분은 파싱 부분이다. 입/출력 부분은 gen_new_cov 스크립트가 담당하고 있으며, 스크립트 실행 시 디렉토리에 존재하는 *.cil.c를 입력으로 사용한다. 이후 각 .cil.c 파일들을 파싱 담당 프로그램 parsing_cil에 넘겨주고, new_coverage 파일 작성에 필요한 정보 (분기 ID, 분기 식, 분기가 위치한 라인)쌍, (파일명, 해당 파일에 속한 함수명)쌍들을 받는다. 이후, 받은 정보를 출력 형식에 맞게 가공하여 new_coverage 파일에 출력하고 있다.

파싱 부분은 실행 파일 parsing_cil에서 담당하고 있으며 clang(ver 4.0) 라이브러리를 활용하여 구현했다. .cil.c 파일에서 분기에 관련된 정보와 파일 이름, 함수 이름 등을 파싱하여 gen_new_cov에 반환하는 역할을 한다. (파일 이름, 해당 파일에 속한 함수명)쌍 정보는 clang에서 AST(abstract syntax tree)를 탐색할 때, 함수 선언 구문에 해당하는 syntax 노드에 도달했을 때 clang API를 사용하여 파일 이름과 함수 이름의 획득이 가능하므로, 이를 쌍으로 묶어 출력한다. (분기 ID, 분기 식, 분기가 위치한 라인)쌍 정보는 다음과 같은 순서에 따라 획득한다.

```
1. int var;
2.
3. void f() { }
4.
5. void main() {
6.   int res = 0;
7.
8.   if(var == 1)
9.     res = 1;
10.
11.  f();
12.}
```

그림 4.11: 예제 코드

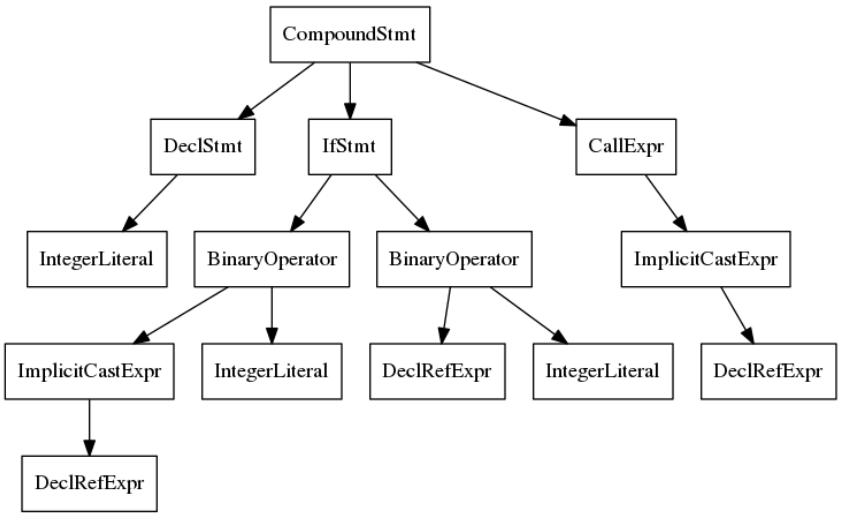

그림 4.12: main 함수 내부 코드에 대한 AST

1. 대상 코드의 AST 탐색

clang 라이브러리에 정의된 RecursiveASTVisitor 클래스를 사용하여 대상 소스 코드의 각 syntax 노드를 순회하는 것이 가능하다. 대상 소스 코드의 syntax 노드는 대상 소스 코드를 AST(abstract syntax tree)로 표현했을 때 각 노드를 의미한다. 이후, RecursiveASTVisitor 클래스 내부의 함수를 재정의하여 If문 또는 For문과 같이 특정 구문을 나타내는 syntax node에 도달했을 때, 수행할 작업을 정의할 수 있다. 여기서는 RecursiveASTVistor 클래스로 .cil.c 파일의 각 syntax 노드를 탐색하면서 필요한 정보 (분기 ID, 분기 식, 분기가 위치한 라인)을 파싱하는 작업을 수행한다.

그림 4.11는 AST로 표현할 예제 소스 코드이고, 그림 4.12는 main 함수 내부에 해당하는 코드를 AST로 표현한 그래프이다. 그래프에서 확인 가능하듯이 각각의 구문에 따른 syntax 노드가 존재하는 것을 확인할 수 있다. 루트 노드에 해당하는 CompoundStmt는 main 함수 내부 6~12라인을 의미한다. CompoundStmt는 자식 노드들을 가지며 이 노드들은 main 함수 내부에 포함된 구문들이다. 예를 들면, 소스 코드 6라인, 8라인, 11라인에 각각 변수 선언, If조건문, 함수 호출 구문이 존재하는 것을 확인할 수 있다. 이는 그래프에서 각각 DeclStmt, IfStmt, CallExpr 노드에 해당한다. 해당 노드들은 추가로 자식 노드들을 가지는 것을 볼 수 있으며, 자식 노드들은 더 세분화한 구문들에 대한 정보들을 가지고 있다. RecursiveASTVistitor 클래스를 활용하여 그래프의 각 노드들을 순회하는 것이 가능하고, 노드에 따라 특정 작업을 수행할 수 있다.

```
1. int a, b, c, d;
2.
3. void main() {
4.   if(a == 1) {
5.     printf("ex1");
6.     if(b == 1) printf("ex2");
7.     else printf("ex3");
8.   }
9.   else printf("ex4");
10.
11.  if(c==1 && d==1) printf("ex5");
12.  else printf("ex6");
13.}
```

그림 4.13: 파일 파싱 예제 원본 코드

```
...
1. #line 4
2.   if(a == 1) {
3.     __CrownBranch(...,1,1);
4.     printf("ex1");
5. #line 6
6.     if(b == 1) {
7.       __CrownBranch(...,2,1);
8.       printf("ex2");
9.     }
10.    else {
11.       __CrownBranch(...,3,0);
12.       printf("ex3");
13.     }
14.  }
15.  else {
16.     __CrownBranch(...,4,0);
17.     printf("ex4");
18.  }
19.#line 11
20.  if(c == 1) {
21.    __CrownBranch(...,5,1);
22.#line 11
23.    if(d == 1) {
24.      __CrownBranch(...,6,1);
25.      printf("ex5");
26.    }
27.    else{
28.      __CrownBranch(...,7,0);
29.      printf("ex6");
30.    }
31.  }
32.  else {
33.    __CrownBranch(...,8,0);
34.    printf("ex6");
35.  }
...
```

그림 4.14: 파일 파싱 예제 .cil.c 코드

2. __CrownBranch 함수 호출 구문 탐색

(분기 ID, 분기 식, 분기가 위치한 라인)쌍을 얻기 위해서는 해당하는 분기를 찾는 과정이 필요하다. 각 분기 지점마다 .cil.c 파일에 __CrownBranch 함수 호출 구문이 존재하기 때문에 탐색한 syntax 노드가 __CrownBranch 함수 호출 구문을 나타내는 노드일 경우, 해당 분기가 갖는 (분기 ID, 분기 식, 분기가 위치한 라인) 정보를 파싱하는 작업을 시작한다. 그림 4.13과 그림 4.14은 이를 설명하기 위한 예제 코드이며, 각각 원본 소스코드와 원본 소스코드에 대한 (간략화한) .cil.c 파일이다. .cil.c 파일을 살펴보면 분기 지점에 해당하는 3, 7, 11, ...라인에 __CrownBranch 함수 호출 구문이 삽입된 것을 확인할 수 있다. 탐색한 syntax 노드가 각각 3, 7, 11, ...라인에 해당하는 __CrownBranch 함수 호출 구문에 해당할 경우 다음 작업을 수행한다.

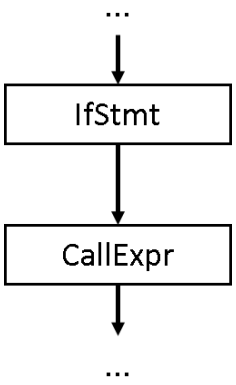

그림 4.15: __CrownBrncch 함수 호출구문의 syntax 노드

3. 분기 ID 파싱

출력해야 하는 정보 중 하나인 분기 ID를 파싱하는 것이 필요하다. 분기 ID는 __CrownBranch 함수 호출 구문의 2번째 인자를 의미하므로, 탐색한 __CrownBranch 함수 호출 구문에서 2번째 인자를 추출할 수가 있다. 함수 호출에 해당하는 syntax 노드일 경우, 내부에 인자를 추출하는 메서드를 제공하고 있기 때문에 해당 메서드를 이용하여 분기 ID를 쉽게 획득할 수 있다. 예를 들어, 그림 4.14 24라인 __CrownBranch 함수 호출 구문에 해당하는 syntax 노드를 탐색한 경우, 2번째 인자 6을 파싱하여 해당하는 분기 ID를 획득한다.

4. 분기가 위치한 라인 파싱

다음으로 탐색한 분기에 해당하는 라인을 파싱한다. 여기서 분기에 해당하는 라인이란 원본 소스 코드에서 해당 분기를 포함하는 조건문의 라인을 의미한다. 예를 들어, 그림 4.14 21라인 __CrownBranch 함수 호출 구문에 해당하는 분기는 "c == 1"이다. 이는 원본 소스코드 11라인 조건문에 포함된 분기이므로 해당 분기의 라인은 11라인으로 간주한다. .cil.c의 각각의 조건문이 원본 소스 코드에서 어느 라인에 매칭되는지 알기 위해 .cil.c 파일 생성 시, line directive(.cil.c 파일의 1, 5, 19, 22라인)를 사용해서 각 if문마다 원본 소스 코드의 라인을 기록한다.

라인 정보를 파싱하기 위해서 탐색한 __CrownBranch 함수 호출 구문 바로 위에 있는 조건문의 syntax 노드로 이동한다. 예를 들면, .cil.c 파일(그림 4.14) 24라인에 해당하는 __CrownBranch 함수 호출 구문을 탐색한 경우, 바로 위에 있는 23라인 조건문에 해당하는 syntax 노드로 이동한다.

그림 4.15는 .cil.c 파일을 실제 AST로 표현했을 때 If조건문에 해당하는 syntax 노드와 __CrownBranch 함수 호출 구문에 해당하는 syntax 노드 관계를 나타낸다. 그림에서 IfStmt 노드는 조건문에 해당하는 syntax 노드이고, CallExpr은 __CrownBranch 함수 호출 구문을 나타내는 syntax 노드이다. __CrownBranch 함수 호출 구문은 조건문 바로 아래에 삽입되기 때문에 그림과 같이 If조건문 노드와 __CrownBranch 함수 호출 구문 노드가 직접 연결된 것을 확인할 수 있다. 따라서 복잡한 과정없이 __CrownBranch 함수 호출 구문 노드에서 If조건문을 나타내는 노드로 이동할 수 있다.

이 후, 이동한 조건문의 라인을 line directiev(원본 소스 코드 라인) 기준으로 획득한다. 즉, .cil.c 파일에서는 해당 조건문의 라인이 24라인이지만 line directive에 기록된 원본 코드 라인을 기준으로 할 경우, 해당 조건문의 라인은 11라인이 된다.

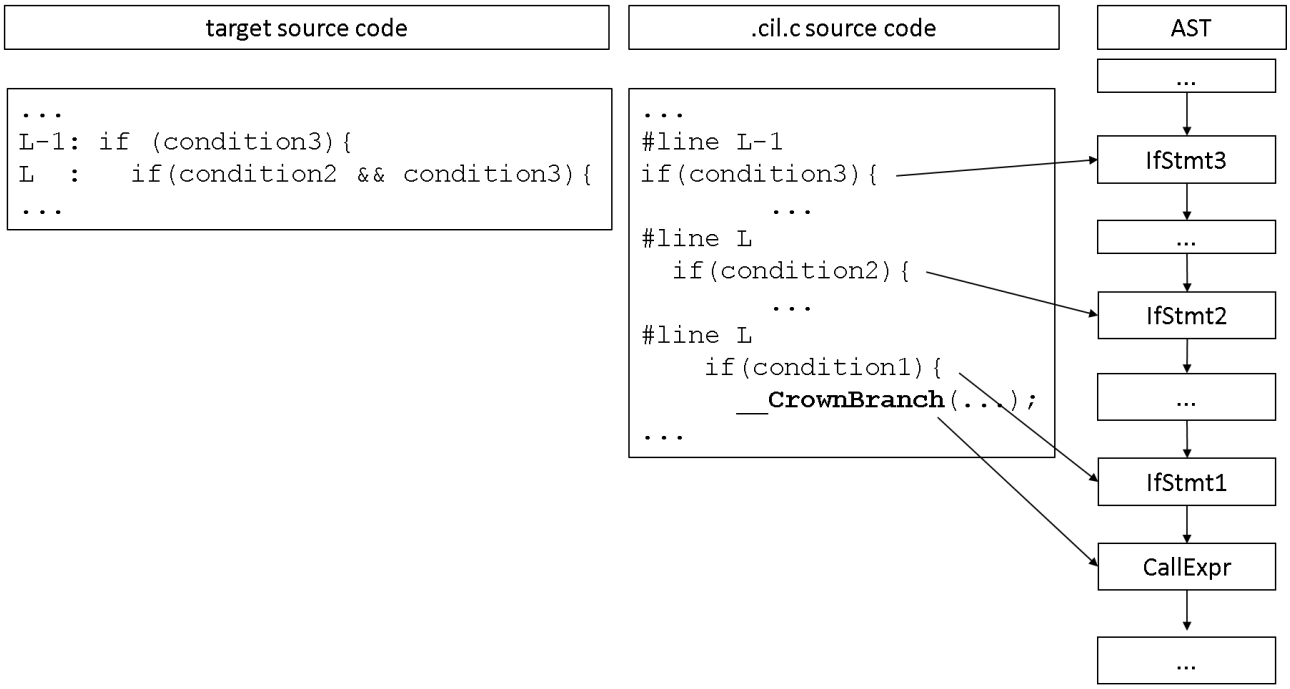

그림 4.16: 분기 식 파싱을 위한 예제

5. 분기식 파싱

다음으로 분기 식을 구한다. 주의할 점은 원본 소스 코드의 복합 조건문이 .cil.c에서는 단일 조건문들로 변형되었다는 점이다. 즉, 그림 4.14의 24라인에 해당하는 분기 식은 23라인의 조건 "d == 1" 뿐만 아니라 20라인의 조건 "c == 1"을 추가로 합성하여 "c == 1 && d == 1"이 되어야 한다. 단, 7라인에 해당하는 분기식은 원본 소스 코드(그림 4.13 6라인)에서도 단일 조건문 형태이기 때문에 "b == 1"이 돼야한다. 이를 위해 탐색한 __CrownBranch 호출 구문 노드의 바로 위에 있는 If조건문의 원본 소스 코드 라인을 L이라 했을 때, 조상 노드에 포함된 If조건문 노드들을 탐색하면서 원본 소스 코드 라인이 L과 같은 If조건문들의 조건들을 합성한다.

그림 4.16는 분기 조건을 파싱하는 예제를 보이기 위한 그림이다. 가장 왼쪽 그림은 원본 소스 코드이고, 가운데 그림은 원본 소스 코드에 대해 간략화한 .cil.c 파일이다. .cil.c파일에 존재하는 line directive는 각 조건문과 매칭되는 원본 소스 코드 라인을 나타낸다. 오른쪽 그림은 .cil.c 파일에 대해 간략화한 AST이다. .cil.c의 if구문과 __CrownBRanch구문에 대해 해당하는 syntax 노드 가리키도록 화살표를 추가했다. 현재 .cil.c 파일에서 굵게 표기한 __CrownBranch 호출 구문을 나타내는 CallExpr 노드를 탐색 중이며, 해당하는 분기 식을 구한다고 가정하자. 처음에는 CallExpr와 직접 연결된 IfStmt1을 탐색하여 해당 IfStmt1이 포함하는 조건(condition1)과 라인 정보(L)을 추출한다. 이후, 부모 노드들을 탐색하면서 If조건문을 나타내는 노드(IfStmt2, IfStmt3)찾고, 라인 정보를 확인한다. IfStmt2 노드의 경우, IfStmt1의 라인과 같은 라인 L을 가지므로, IfStmt1과 IfStmt2의 조건식을 합성하여 분기 "condition1 && condition2"를 구성한다. IfStmt3 노드는 IfStmt1의 라인과 다른 라인 L-1을 가지므로 탐색을 종료하고 현재까지 구성한 분기 식("condition1 && condition2")을 출력한다.

제 5 장 테스트 커버리지 향상 가이드라인

본 절에서는 2.2절에서 소개한 동적 기호 실행 기반 자동화된 테스크 테스팅 기법을 적용한 뒤 C 프로그램 대상 Concolic 테스팅을 수행했을 때 발생 가능한 미달성 분기들을 유형별로 분류하여 테스트 커버리지를 향상하기 위한 가이드라인을 제공한다. 즉, 동적 기호 실행 기반 자동화된 테스크 테스팅 기법은 높은 커버리지 달성을 위한 Concolic 테스팅 환경을 자동으로 구축하지만 각양각색의 소스 코드가 존재하는 만큼, 항상 최적화된 Concolic 테스팅 환경을 구축하는 것은 불가능하며 커버리지 달성이 어려운 예외 상황이 항상 존재한다. 따라서 여기서는 이러한 예외 상황을 유형 별로 소개하고 커버리지를 높일 수 있는 가이드라인을 제시하여 사용자로 하여금 Concolic 테스팅 환경 구축에 대한 이해를 넓히고 더 높은 커버리지를 달성할 수 있도록 유도하고 있다.

대상 코드가 주어졌을 때, 2.2절에서 소개한 프레임워크 보다 최적화된 환경을 사용자가 직접 구축할 수 있지만, 이 경우 최적화된 환경을 구축하기까지 상당히 많은 시간이 걸리게 된다. 더불어 2.2절에서 소개하고 있는 프레임워크에서 달성 못 하는 유형을 달성할 수 있는 다른 프레임워크의 사용도 가능하지만 탐색 공간도 커지게 되어 시간 당 테스팅 효율이 오히려 더 떨어지는 문제가 존재한다. 따라서 본 논문에서는 달성 못 하는 분기 유형이 존재하지만 테스팅 효율이 높은 2.2절의 프레임워크를 적용하여 빠르게 테스팅을 수행한 뒤, 달성하지 못한 분기에 대해서는 분석을 통해서 추가적으로 테스팅하는 방식으로, 최대한 빠르게 커버리지를 높이고자 했다.

차례대로 5.1절에서는 유형별 미달성 분기를 예제와 함께 소개하여 달성에 실패한 원인을 이해하고 5.2절에서는 유형별로 가이드라인을 제안하여 유형 별 미달성 분기를 달성할 수 있는 방안을 소개한다.

5.1 미달성 분기 분석

본 절에서는 2.2절에서 소개한 프레임워크로 환경을 구축했을 때 달성 못하는 분기 유형들을 예제와 함께 소개한다. 표 5.1는 이러한 미달성 분기가 발생하는 경우를 코드 패턴에 따라 나타냈다.

5.1.1 Type 1 - 다양한 함수 호출 시퀀스를 만들지 못해서 실행되지 못한 분기

표 5.1의 타입 1 코드 패턴을 살펴보면 함수 f_def는 fptr에 함수 g의 주소를 대입하는 기능을 수행하고, 함수 f_call은 fptr을 호출하는 것을 확인할 수 있다. 따라서 함수 g를 실행하기 위해서는 테스트 드라이버에서 함수 f_def와 f_call를 차례대로 호출하는 시퀀스가 필요하다. 하지만 2.2절에서 소개한 프레임워크로 구축한 각 테스트 드라이버에서는 한 종류의 테스크만 실행하기 때문에 함수 f_def와 f_call가 서로 다른 테스크에 속할 때 두 함수를 호출하는 시퀀스를 만들지 못한다. 이로 인해 함수 g의 실행이 불가능하고 g에 속한 분기는 달성할 수 없게 된다.

그림 5.1는 이러한 상황을 나타낸 예제 코드이다. 예제 코드 1라인에서는 함수 포인터 전역 변수가 존재하며 초기 값으로 dummy_func 함수 주소를 갖는다. 함수 포인터 인터페이스 함수는 t1 함수와 t2 함수 2개가 존재한다. 인터페이스 함수 t1과 t2를 살펴보면 각각 함수 포인터 변수 fptr을 정의하는 기능과 호출하는 기능을 분리하여 수행한다. 즉, t1 함수에서는 함수 포인터 변수 fptr에 함수 f를 대입하고, t2 함수에서는 함수 포인터 fptr를 호출하는 작업을 수행한다. 테스트 드라이버에서 t1 함수와 t2 함수를 차례대로 호출하는 함수 호출 시퀀스를 만들면 static 함수 f의 호출이 가능하지만, 가능한 함수 호출 시퀀스가 무수히 많고 생성한 함수 호출 시퀀스가 타당한지 확인이 불가능하여 현재 동적 기호 실행 기반 자동화된 테스크 테스팅

표 5.1: 2.2절 프레임워크로 테스팅 환경을 구축했을 때 미달성 분기 유형별 코드 패턴

타입	타입명	코드 패턴
1	다양한 함수 호출 시퀀스를 만들지 못해서 실행되지 못한 분기	• fptr: a function pointer variable • task1: a task including the function f_def • task2: a task including the function f_call File 1. static $ret_type g ($params) { $stmt } 2. $ret_type f_def ($params) { fptr = g; } 3. $ret_type f_call ($params) { fptr($args); } 단, task1과 task2는 다른 테스크
2	경로 폭발(Path explosion) 문제 때문에 주어진 테스팅 시간 동안 실행되지 못한 분기	File 1. $ret_type f($params) { 2. $stmt 3. if ($condition) $stmt; 4. } 단, 3라인 분기는 경로 폭발 문제 때문에 미달성된 분기
3	대상 파일에 함수를 호출하는 구문이 없어서 실행되지 못한 분기	• fptr: a function pointer variable File1 1. static $ret_type g($params) {$stmt} 2. $ret_type f_def($params) {fptr = g;} File2 1. $ret_type f_call($params) {fptr($args);}
4	데드 코드(Dead code)라서 실행되지 못한 분기	Pattern 1: File 1. $ret_type f($params) { 2. $stmt 3. if ($condition) $stmt; 4. } 단, 함수 f는 호출이 가능하며 3라인 분기는 코드 구조상 달성할 수 없는 분기 Pattern 2: File 1. $ret_type f($params) { $stmt }; 단, 함수 f는 호출이 불가능(사용되지 않는 함수)

5	부정확한 스텁 때문에 실행되지 못한 분기	File1
		1. $ret_type f($params) {
		2. g($pointer_arg);
		3. }
		File2
		1. $ret_type g($pointer_param) {
		2. *($pointer_param) = $expression
		3. }

6	테스트 드라이버에서 대상 함수를 호출하는 횟수가 적어서 실행되지 못한 분기	File 1. $ret_type f($params) { 2. static $data_type cnt = $expression; 3. if (cnt >= $constant) $stmt; 4. cnt++; 5. } 단, 3라인 분기는 Type 6 미달성 분기

```
1. void (*fptr) void = dummy_func;
2.
3. static void dummy_func(int par) { }
4. static void f() {
5.   if(par == 1) /* Type 1 유형 분기 */;
6. }
7. void t1(int par) {
8.   fptr = f;
9. }
10.void t2() {
11.  fptr();
12.}
```

그림 5.1: Type 1 미달성 분기 예제

```
1. ...
2. static void f() {
3.   static int flag = 0;
4.   switch (flag) {
5.     case 0: execution 1; flag = 1; break;
6.     case 1: execution 2; break;
7.   }
8. }
```

그림 5.2: Type 2 미달성 분기 예제

기법으로 구축한 테스트 드라이버에서는 t1만 호출하거나 t2만 호출하는 것과 같이 한 종류의 인터페이스만 호출하고 있다. 따라서 현재 실험 환경에서는 함수 f의 호출이 불가능하며 당연히 해당 함수에 속한 분기(5라인) 역시 실행이 불가능한 상황이다.

5.1.2 Type 2 - 경로 폭발(Path explosion) 문제 때문에 주어진 테스팅 시간 동안 실행되지 못한 분기

동적 분석 기법에 속하는 Concolic 테스팅 기법은 한번(iteration)에 하나의 실행 경로를 실행하기 때문에 주어진 시간동안 달성 조건이 까다로운 경로 탐색에 실패할 수 있다. 일반적으로 대상 테스크에 포함된 분기가 많이 존재하고 복잡할 때, 경로 폭발(Path explosion) 문제 때문에 주어진 테스팅 시간 동안 달성에 실패한 분기가 많이 존재했다. 그중에서도 특히 어떤 분기를 달성하기 위한 조건 중 하나로 대상 인터페이스를 반복 호출하는 것이 필요할 때, 해당 분기를 달성하기 위한 어려움이 곱절로 늘어나는 것을 확인했다. 예를들면 switch문까지 도달하기 위한 심볼릭 경로 수식이 복잡하고, swith문 내부에 여러 번 호출 실행해야 달성 가능한 case문이 존재할 때 해당 case문 실행에 실패한 경우가 비교적 상당수 존재했다. 그림 5.2은 이러한 case 조건을 설명하기 위한 예제 코드이다.

그림 5.2은 static 함수 f를 포함하며 함수 f 내부에 switch 문이 존재한다. 함수 f는 인터페이스에서 호출이 가능한 함수라고 할 때 5라인의 case문은 함수 f를 한번 호출하는 것만으로 달성할 수 있지만, 6라인의 case문을 실행하기 위해서는 함수 f를 두 번 호출해야 한다. 함수 f를 호출하기 위한 조건이 까다로울수록 6라인을 실행하기 위한 조건은 곱절로 늘어나게 된다. 즉, static 함수 f를 호출하기 위해 인터페이스의 입력

```
1.extern void (*fptr) (void);
2.static void f_infeasible(int par) {
3.  if(par == 1);/*Type3 유형 미달성 분기*/
4.}
5.
6.void inter() {
7.  fptr = f_infeasible;
8.}
```

그림 5.3: 대상(target) 파일

```
1. void (*fptr) (void);
2. void ext_func() {
3.   fptr()
4. };
```

그림 5.4: 비대상(non-target) 파일

값이 만족해야 하는 조건을 C라고 할 때, 5라인 case문을 실행하기 위해 인터페이스의 입력값이 만족해야 하는 조건은 단순히 C가 되지만, 6라인 case문을 실행하기 위해서는 인터페이스를 두 번 호출해야 하며 각각의 입력값은 모두 조건 C를 만족해야 한다. 따라서 6라인을 만족하기 위한 조건은 5라인을 만족하기 위한 조건에 비해 곱절로 늘어나게 되며 그만큼 달성 어려움이 증가한다고 볼 수 있다.

5.1.3 Type 3 - 대상 파일에 함수를 호출하는 구문이 없어서 실행되지 못한 분기

표 5.1의 타입 3 코드 패턴을 살펴보면 함수 f_def는 fptr에 함수 g의 주소를 대입하는 기능을 수행하고, 함수 f_call은 fptr을 호출하는 것을 확인할 수 있다. 따라서 함수 g를 실행하기 위해서는 테스트 드라이버에서 함수 f_def와 f_call를 차례대로 호출하는 시퀀스가 필요하다. 하지만 File1을 대상 파일이라고 할 때, 2.2절에서 소개한 프레임워크로 구축한 테스트 드라이버는 대상 파일에 포함된 테스크만 실행하기 때문에 File2의 함수 f_call은 실행이 불가능하다. 이로 인해 함수 g의 실행이 불가능하고 g에 속한 분기는 달성할 수 없게 된다. 그림 5.3과 그림 5.4는 이러한 예제를 나타낸다.

대상 파일인 그림 5.3을 살펴보면 static 함수 f_infeasible 함수는 정의되어있지만 호출하는 구문이 존재하지 않는다. 따라서 동적 기호 실행 기반 자동화된 테스크 테스팅 기법에서 구축한 테스트 드라이버를 사용하여 테스팅을 수행할 경우 static 함수 f_infeasible의 호출이 불가능하고 해당 함수에 속한 분기(3라인) 역시 달성이 불가능해진다. 대신 7라인에서 전역 함수 포인터 변수 fptr에 f_infeasible 함수를 대입하고 있으며, 대상이 아닌 코드 그림 5.4의 3라인에서 함수 포인터 변수 fptr을 사용해서 f_infeasible을 호출하고 있다. 따라서 함수 f_infeasible은 실제 환경에서 실행이 가능한 함수이기 때문에 데드 코드와는 다르다고 볼 수 있으며 테스팅 환경을 재구축하여 해당 함수를 실행할 필요가 있다.

5.1.4 Type 4 - 데드 코드(Dead Code)라서 실행되지 못한 분기

데드 코드 종류에 따라 실행되지 못하는 분기는 크게 두 종류로 나눌 수 있다. 첫 번째는 분기가 포함된 함수가 데드 코드인 경우다. 소스 코드가 갱신되면서 기존에 사용되던 함수가 더 이상 사용되지 않을 수 있으며 이에 따라 해당 함수는 정의되어있지만, 해당 함수를 호출하는 구문은 없을 수 있다. 따라서 해당 함수 내부에 존재하는 분기 역시 실행 불가능한 분기가 된다. 이러한 분기들은 분기 자체가 실행이 불가능한 것이 아니기 때문에, 함수만 호출된다면 여전히 실행 가능성이 존재한다.

두 번째는 분기 자체가 실행 불가능한 경우이다. 일부 조건문들에 대해 코드 구조상 조건식을 만족할 수가 없는 경우가 존재하고, 이런 경우 해당 조건문은 데드 코드로서 분기 달성이 불가능하다. 그림 5.5은 이러한 조건식을 포함한 예제 코드이다. 예제 코드 2라인에서 static 변수 flag는 1로 초기화되고, flag의 값은 5, 9라인에 의해서만 변경된다고 가정하자. 이 경우, 변수 flag는 1 또는 2의 값만 가질 수 있으므로, 11

```
1. void f() {
2.   static int flag = 1;
3.   if(flag == 1) {
4.     execution 1;
5.     flag = 2;
6.   }
7.   else if(flag == 2) {
8.     execution 2;
9.     flag = 1;
10.  }
11.  else {
12.    //dead code
13.  }
14.}
```

그림 5.5: 데드 코드 예제

```
target.c
1. extern f_external(int *par1, int par2);
2. void interface() {
3.   int a = 0;
4.   f_external(&a, 1);
5.   if ( a != 0 ); /* Type 5 유형 미달성 분기 */
6. }
```

그림 5.6: 포인터 인자로 인한 분기 달성 실패 예제 대상 코드

라인의 else 분기는 실행할 수 없게 된다.

5.1.5 Type 5 - 부정확한 스텁 때문에 실행되지 못한 분기

표 5.1의 타입 4 코드 패턴을 살펴보면 함수 f는 함수 g를 호출하며 인자로 포인터를 넘기고 있다. 함수 g에서는 매개변수로 받은 포인터가 가리키는 값을 변경하고 있다. File1을 대상 파일이라고 할 때, 2.2절의 프레임워크를 적용하면 함수 g는 대상이 아닌 파일 File2에 존재하기 때문에 스텁으로 대체되고, 대체된 스텁은 매개변수의 값을 변경하지 않는 구조로 되어있다. 따라서 테스팅 수행 시 스텁 g를 호출해도 인자로 넘긴 포인터가 가리키는 값에 변화가 없게 된다. 이로 인해 달성이 불가능한 분기가 생긴다. 그림 5.6는 이러한 분기가 포함된 대상 코드(target.c)이다.

그림 5.6의 4라인에서는 external.c 파일에 있는 함수 f_external를 호출하고 있으며, 변수 a의 주소와 숫자 1을 인자로 넘긴다. 하지만 동적 기호 실행 기반 자동화된 테스크 테스팅 기법으로 구축한 테스팅 환경에서는 외부 파일에 있는 함수를 스텁으로 대체하기 때문에 4라인은 그림 5.8 파일(external.c)에 있는 원본 f_external 함수가 아닌, 그림 5.7에 있는 스텁 f_external 함수를 호출한다. external.c 파일의 원본 f_external 함수에서는 int형 포인터 변수와 int형 변수를 인자로 받은 뒤, 포인터 변수가 가리키는 변수에 인자로 받은 int형 변수 값을 대입하는 기능을 한다. 이 경우, 그림 5.6의 4라인 실행 후 지역 변수 a는 1의 값을 갖게 되고, 5라인의 분기 달성이 가능하다. 하지만 stub.c에 있는 스텁을 호출할 경우, 포인터를 통한 값 변경이 이루어지지 않기 때문에 지역 변수 a는 그대로 값 0을 유지한다. 이로 인해 5라인의 분기 달성에 실패한다.

```
stub.c
1. int f_external(int *par1,int par2){
2.    int var_dmmy;
3.    SYM_int(var_dummy);
4.    return var_dummy;
5. }
```

그림 5.7: 대상 코드에 대한 스텁

```
external.c
1. int f_external(int *par1,int par2){
2.    *par1 = par2;
3.    return 1;
4. }
```

그림 5.8: f_external 함수가 정의된 파일

```
1. void f() {
2.    static int cnt = 1;
3.    if( cnt >= 1000 ); /* Type 6 유형 미달성 분기 */
4.    cnt++;
5. }
```

그림 5.9: Type 6 미달성 분기 예제

5.1.6 Type 6 - 테스트 드라이버에서 대상 함수를 호출하는 횟수가 적어서 실행되지 못한 분기

어떤 함수 f는 내부에 static 지역 변수를 사용해서 함수가 호출될 때마다 상태 값을 기록하고 이 변수를 조건식에 사용하여 값을 비교할 수가 있다. 이 경우, static 지역 변수가 선언된 함수 f의 실행이 끝나도 상태 값이 유지되기 때문에 함수 f를 반복 호출해야 달성이 가능한 분기가 존재할 수 있다.

그림 5.9은 분기 달성을 위해 함수 호출이 여러 번 필요한 예제 소스 코드이다. 변수 cnt는 초기값 1을 갖고, 함수 f가 호출될 때마다 1씩 증가한다. 3라인 분기를 실행하기 위해서는 함수 f를 1000번 호출해야 한다. 따라서 일반적으로 함수 f를 한번 호출하는 것만으로는 3라인 분기를 달성할 수 없다.

5.2 미달성 분기 달성을 위한 가이드라인

본 절에서는 5.1절에서 확인한 각각의 미달성 분기 유형에 대해 분기 달성을 위한 가이드라인을 제시한다.

5.2.1 Type 1 - 다양한 함수 호출 시퀀스를 만들지 못해서 실행되지 못한 분기

본 유형을 달성하기 위해서는 표 5.1의 타입 1 코드 패턴에서 task1(f_def가 속한 테스크)과 task2(f_call가 속한 테스크)를 차례대로 실행시키는 테스트 드라이버를 구축해야하며 이를 위해 다음과 같은 작업을 수행한다. 스텁은 2.2절에서 구축한 스텁을 그대로 사용한다. 이하에서 언급하는 task1, task2는 표 5.1 타입 1 코드 패턴에서의 task1, task2를 의미한다.

(i) task1의 인터페이스와 task2의 인터페이스 호출 시 인자로 사용할 변수 선언

task1의 인터페이스와 task2의 인터페이스가 매개변수를 가질 경우, 인자로 넘길 변수를 테스트 드라이버에 선언 해주어야 한다. task1의 인터페이스 호출 시 인자로 넘길 변수 선언 구문을 $decl_args_interface_of_task1이라고 하고, task2의 인터페이스 호출 시 인자로 넘길 변수 선언 구문을 $decl_args_interface_of_task2이라고 하자.

```
driver.c
1. void test_driver (int iter) {
2.   for(int i=0;i<iter;i++){
3.     $decl_args_interface_of_task1
4.     $decl_symbolic_for_task1
5.     $call_interface_of_task1
6.
7.     $decl_args_interface_of_task2
8.     $decl_symbolic_for_task2
9.     $call_interface_of_task2
10.} }
```

그림 5.10: Type 1 유형 달성을 위한 테스트 드라이버

(ii) task1와 task2에서 사용하는 변수를 심볼릭 선언

Concolic 테스팅을 위해 task1과 task2을 실행하기 전에 입력 변수들을 모두 심볼릭 선언 해주어야 한다. task1의 경우 task1에서 사용하는 모든 전역 변수와 task1의 인터페이스 호출 시 사용할 인자를 심볼릭 선언해준다. 이러한 심볼릭 선언 구문을 $decl_symbolic_for_task1이라고 하자. 마찬가지로 task2에 대한 심볼릭 선언 구문을 $decl_symbolic_for_task2이라고 하자.

(iii) 테스트 드라이버 구축

그림 5.10는 Type 1의 유형을 달성하기 위해 구축한 테스트 드라이버이다. 테스트 드라이버가 인자로 받는 iter는 인터페이스를 반복 호출할 횟수를 의미한다. 2.2 절에서 구축한 환경과 마찬가지로 인터페이스를 반복 호출해야 달성 가능한 분기가 존재하기 때문에 반복 호출할 횟수를 인자로 받는다. 3∼9라인은 필요한 환경 구축을 하고, 차례대로 task1과 task2를 실행하는 구문이다. 3∼5라인은 차례대로 task1 실행에 필요한 변수들을 선언 및 심볼릭 선언한 후 task1를 실행하는 인터페이스를 호출하고 있다. $call_interface_of_task1은 task1의 인터페이스를 호출하는 구문이다. 마찬가지로 7∼9라인에서는 task2 실행을 위한 환경을 구축하고 task2의 인터페이스를 호출하고 있다. $call_interface_of_task2는 task2의 인터페이스를 호출하는 구문이다.

그림 5.11에서 왼쪽 코드는 2.2절 프레임워크으로 구축한 테스트 드라이버로 Type 1 분기가 발생한다. 오른쪽 코드는 Type 1 분기를 달성하기 위해 변경한 테스트 드라이버이다. 왼쪽 코드에서는 1∼7라인과 8∼14라인에 각각 task1과 task2를 실행하는 테스트 드라이버가 분리되어 있으므로 task1만 실행하거나 task2만 실행할 수 있다. Type 1 분기는 task1과 task2를 차례대로 실행해야 달성할 수 있는 분기이기 때문에 오른쪽 그림과 같이 하나의 테스트 드라이버 안에 task1와 task2를 실행하는 코드 구문을 차례대로 삽입하여 task1과 task2가 차례대로 실행되도록 유도한다.

5.2.2 Type 2 - 경로 폭발(Path explosion) 문제 때문에 주어진 테스팅 시간 동안 실행되지 못한 분기

본 유형은 탐색 가능한 실행 경로가 많이 존재하여 주어진 탐색 횟수 동안 달성 가능한 분기를 실행하지 못한 유형이다. 특히 본 유형에서는 조건문이 복잡해짐에 따라 미달성 분기 유형이 본 유형에 속하는지를 결정하는 작업의 어려움도 증가한다. 즉, 미달성 분기의 유형이 경로 폭발 문제인지 데드 코드 때문인지

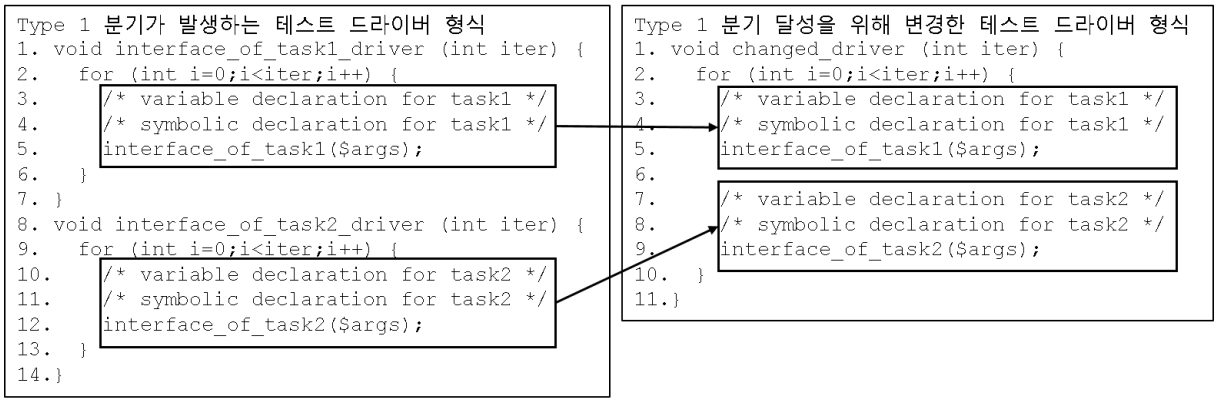

그림 5.11: Type 1 분기 달성을 위한 테스트 드라이버 변경 전/후 형식

식별하기가 어려워지는데 이를 위해 본 논문에서는 복잡한 조건문을 떼어내어 원본 코드보다 복잡도가 낮은 (less constraint) 코드를 구성한 뒤 테스팅을 수행함으로서 데드 코드에 해당하는 분기를 1차적으로 걸러냈다. 즉, 경로 폭발 문제 때문인지, 데드 코드 때문인지 판단이 어려운 미달성 분기 유형이 존재하고, 다른 유형에는 속하지 않는다고 가정하자. 이때, 해당 분기가 속한 조건문을 떼어내서 모든 경로를 탐색해보았을 때 갈 수 없는 분기가 존재한다면 해당 분기는 데드 코드임을 확정할 수가 있다. 이런 식으로 데드 코드인 일부 미달성 분기를 식별하여 분석 작업의 효율을 높일 수 있었다.

본 유형에 속하는 분기들을 달성하기 위해서 큰 탐색 공간을 줄이는 방식을 적용하고 있으며, 기존 테스크 단위 테스팅을 함수 단위로 축소한 뒤 테스팅을 진행한다. 즉, 본 유형의 분기가 존재하는 함수를 f라 할 때, 함수 f가 인터페이스가 아니더라도 함수 f를 직접 호출하는 테스트 드라이버를 구축했으며 함수 f에서 호출하는 모든 함수를 스텁으로 대체함으로서 탐색 공간을 줄이고 함수 f에 속한 분기들을 집중적으로 탐색할 수 있도록 유도했다.

이를 위해 다음과 같은 작업을 수행한다. 아래에서 언급하는 함수 f는 표 5.1의 타입 2 코드 패턴에서의 f를 의미한다.

(i) f 호출 시 인자로 사용할 변수 선언

함수 f가 매개변수를 가질 경우, 인자로 넘길 변수를 테스트 드라이버에 선언 해주어야 한다. f 호출 시 인자로 넘길 변수 선언 구문을 $decl_args_f이라고 하자.

(ii) f에서 사용하는 변수를 심볼릭 선언

Concolic 테스팅을 위해 f를 실행하기 전에 입력 변수들을 모두 심볼릭 선언 해주어야 한다. 따라서 f에서 사용하는 모든 전역 변수와 f 호출 시 사용할 인자를 심볼릭 선언해준다. 이러한 심볼릭 선언 구문을 $decl_symbolic_for_f 이라고 하자.

(iii) 테스트 드라이버 구축

그림 5.12는 Type 2의 유형을 달성하기 위해 구축한 테스트 드라이버이다. 테스트 드라이버가 인자로 받는 iter는 인터페이스를 반복 호출할 횟수를 의미한다. 2.2 절에서 구축한 환경과 마찬가지로 인터페이스를 반복 호출해야 달성 가능한 분기가 존재하기 때문에 반복 호출할 횟수를 인자로 받는다. 3~4라인은 차례대로 함수 f가 사용하는 입력 변수들을 선언 및 심볼릭 선언한 후 f를 호출하고 있다. $call_f는 f를 호출하는 구문이다.

```
driver.c
1. void test_driver_f (int iter) {
2.   for(int i=0;i<iter;i++){
3.     $decl_args_f
4.     $decl_symbolic_for_f
5.     $call_f
6.} }
```

그림 5.12: Type 2 유형 달성을 위한 테스트 드라이버

```
stub.c
1. $ret_type $callee_of_f ($params){
2.   $decl_var_with_ret_type
3.   $decl_symbolic_for_var
4.   $return_var
5. }
```

그림 5.13: Type 2 유형 달성을 위한 스텁

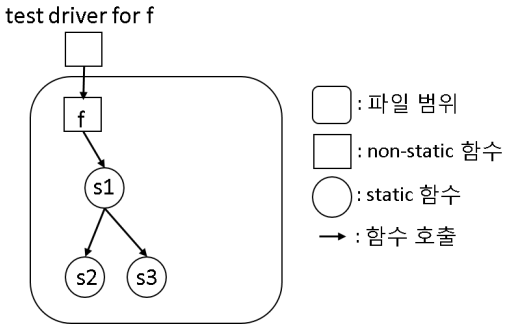

그림 5.14: f에 대한 테스트 드라이버 및 실행 구조

(iv) 테스트 드라이버에서 호출하는 함수 형식을 non-static으로 변경

테스트 드라이버에서 호출하는 함수 f가 static 함수인 경우, 테스트 드라이버에서 호출이 불가능하기 때문에 테스팅을 수행하기 위해 f를 non-static 함수로 변경한다.

(v) 스텁 구축

그림 5.13는 Type 2을 달성하기 위해 구축한 스텁이다. 함수 f에서 호출하는 모든 함수에 대해 스텁을 생성한다. 스텁의 시그니처(반환형, 함수명, 매개변수 형식)는 대체하는 함수와 동일하다. 2라인의 $decl_var_with_ret_type은 반환할 변수를 선언하는 구문으로, 함수의 반환형($ret_type)과 동일한 데이터 타입을 갖는다. 3라인은 해당 변수를 심볼릭 선언하는 구문이고, 4라인은 변수를 반환하는 구문이다.

그림 5.14는 동적 기호 실행 기반 자동화된 테스크 테스팅 기법으로 생성한 테스트 드라이버 및 실행 구조이다. 그림에서 인터페이스 f를 호출하는 테스트 드라이버가 존재하며, f가 호출하는 함수가 같은 파일에 있다면 해당 함수를 모두 실행한다. s1, s2, s3 함수는 static 함수이기 때문에 해당 함수를 호출하는 테스트 드라이버는 존재하지 않는다. 이로 인해 특정 함수에 대해 집중적으로 분기를 탐색하기 어려운 점이 존재한다.

함수 s1에 Type 2의 미달성 분기가 존재한다고 가정할 때, 그림 5.15은 해당 분기를 달성하기 위해 새로 구축한 실행 구조이다. 먼저 s1을 호출하는 테스트 드라이버가 추가됐으며, s1에서 호출하는 모든 함수는 스텁으로 대체된다. 이로 인해, s1에 속하는 분기들을 집중적으로 테스팅할 수 있게 된다.

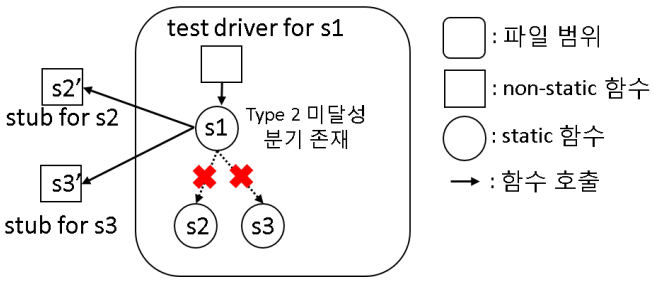

그림 5.15: s1에 대한 테스트 드라이버/스텁 및 실행 구조

5.2.3 Type 3 - 대상 파일에 함수를 호출하는 구문이 없어서 실행되지 못한 분기

본 유형을 달성하기 위해서 표 5.1의 타입 3 코드 패턴에서 함수 g를 실행시키는 테스트 드라이버를 구축할 수 있으며 이를 위해 다음과 같은 작업을 수행한다. 스텁은 2.2절에서 구축한 스텁을 그대로 사용한다. 아래에서 언급하는 함수 g는 표 5.1의 타입 3 코드 패턴에서의 g를 의미한다.

(i) g 호출 시 인자로 사용할 변수 선언

함수 g가 매개변수를 가질 경우, 인자로 넘길 변수를 테스트 드라이버에 선언해주어야 한다. g 호출 시 인자로 넘길 변수 선언 구문을 $decl_args_g이라고 하자.

(ii) g에서 사용하는 변수를 심볼릭 선언

Concolic 테스팅을 위해 g를 실행하기 전에 입력 변수들을 모두 심볼릭 선언해주어야 한다. 따라서 g에서 사용하는 모든 전역 변수와 g 호출 시 사용할 인자를 심볼릭 선언해준다. 이러한 심볼릭 선언 구문을 $decl_symbolic_for_g 이라고 하자.

(iii) 테스트 드라이버 구축

그림 5.16는 Type 3의 유형을 달성하기 위해 구축한 테스트 드라이버이다. 테스트 드라이버가 인자로 받는 iter는 인터페이스를 반복 호출할 횟수를 의미한다. 2.2 절에서 구축한 환경과 마찬가지로 인터페이스를 반복 호출해야 달성 가능한 분기가 존재하기 때문에 반복 호출할 횟수를 인자로 받는다. 3 - 4라인은 차례대로 함수 g가 사용하는 입력 변수들을 선언 및 심볼릭 선언한 후 g를 호출하고 있다. $call_g는 g를 호출하는 구문이다.

(iv) 테스트 드라이버에서 호출하는 함수 형식을 non-static으로 변경

static 함수 g는 테스트 드라이버에서 호출이 불가능하기 때문에 테스팅을 수행하기 위해 g를 non-static 함수로 변경한다.

5.2.4 Type 4 - 데드 코드(Dead code)라서 실행되지 못한 분기

본 유형에 속한 분기들은 데드 코드이기 때문에 해당 분기들을 실행시킬 수 있는 방법은 존재하지 않는다. 대상 코드에 존재하는 데드 코드는 리펙토링(refactoring)하여 제거해주는 작업이 필요하지만 해당 작업은 본 논문의 영역(scope)을 벗어나므로 본 논문에서는 다루지 않도록 한다.

```
driver.c
1. void test_driver (int iter) {
2.   for(int i=0;i<iter;i++){
3.     $decl_args_g
4.     $decl_symbolic_for_g
5.     $call_g
6.} }
```

그림 5.16: Type 3 유형 달성을 위한 테스트 드라이버

```
stub.c
1. $ret_type  g ($pointer_param) {
2.   $decl_symbolic_for_*($pointer_param)
3.   $decl_var_with_ret_type
4.   $decl_symbolic_for_var
5.   $return_var
6. }
```

그림 5.17: Type 5 유형 달성을 위한 스텁

5.2.5 Type 5 - 부정확한 스텁 때문에 실행되지 못한 분기

본 유형을 달성하기 위해서는 2.2절 프레임워크로 구축한 스텁에서 포인터 매개 변수가 가리키는 공간을 심볼릭 선언하여 값 변경이 가능하도록 해주어야 한다. 이를 위해 다음과 같은 작업을 수행한다. 아래에서 언급하는 함수 g는 표 5.1의 타입 5 코드 패턴에서의 g를 의미한다. 테스트 드라이버는 2.2절에서 구축한 테스트 드라이버를 그대로 사용한다.

(i) 스텁 구축

그림 5.17는 Type 5의 유형을 달성하기 위해 구축한 스텁이다. 스텁의 시그니처(반환형, 함수명, 매개변수 형식)는 대체하는 함수와 동일하다. 3~5라인은 반환할 변수를 선언 및 심볼릭 선언까지 마친 후 반환하는 구문으로, 2.2절에의 프레임워크에서 제공하는 스텁과 동일하다. 하지만 Type 5 미달성 분기를 달성하기 위해 2라인에 포인터 매개변수가 가리키는 공간을 심볼릭 선언하는 구문 $decl_symbolic_for_*($pointer_param)이 추가됐다.

그림 5.17의 왼쪽 코드는 시그니처 "int func(int *a, int b)"를 가지는 함수 func에 대해 2.2절 프레임워크로 생성한 스텁이다. 스텁을 살펴보면 포인터 변수 a를 매개변수로 갖지만 변수 a에 대해 아무런 심볼릭 설정을 해주지 않는 것을 확인할 수 있다. 이로인해 포인터 변수 a가 가리키는 변수의 값 변경이 이루어지지 않으며, Type 5의 분기가 발생할 수 있다. 그림 5.18의 오른쪽 코드는 Type 5의 분기를 달성할 수 있도록 변경한 스텁이다. 2라인을 살펴보면 a가 가리키는 변수의 값이 변경될 수 있도록 심볼릭 선언을 하고 있다. 따라서 스텁 호출시 a가 가리키는 변수의 값이 변경될 수 있으므로, Type 5의 분기를 달성할 수 있다.

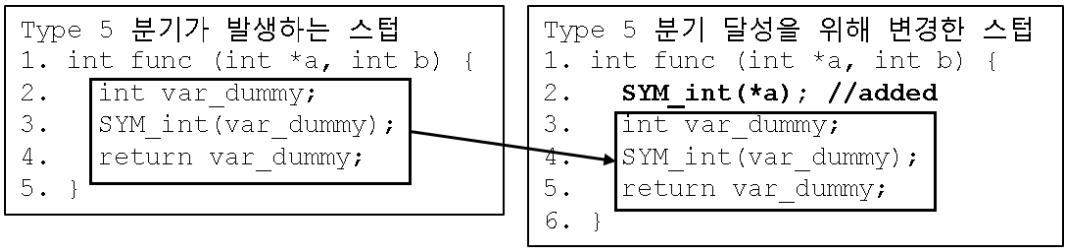

그림 5.18: Type 5 분기 달성을 위한 스텝 변경 전/후 예제

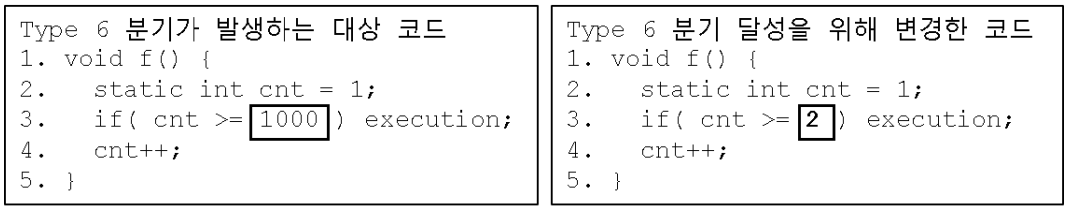

그림 5.19: Type 6 분기 달성을 위한 대상 코드 변경 전/후 예제

5.2.6 Type 6 - 테스트 드라이버에서 대상 함수를 호출하는 횟수가 적어서 실행되지 못한 분기

테스트 드라이버에서 필요한 횟수만큼 해당 함수를 호출할 수 있으나, 테스트 케이스를 한 개 생성하는데 시간이 오래 걸리므로 효과적인 탐색이 불가능하다. 더불어 수집해야 하는 심볼릭 경로 수식의 길이가 기하급수적으로 늘어나기 때문에 메모리 부족으로 인해 테스팅이 빈번히 중단될 가능성이 높다.

(i) 호출 횟수 조건식 변경

본 유형에 속하는 분기를 달성하기 위해서 해당 조건식을 훨씬 적은 횟수만 호출돼도 달성할 수 있도록 조건을 수정할 수 있다. 사례 연구에서는 2번 이상만 호출돼도 달성할 수 있도록 수정했다. 그림 5.19에서 왼쪽 코드는 원본 코드를 의미하며 오른쪽 코드는 분기 달성을 위해 필요한 반복 호출을 2번으로 낮춘 예제 코드이다. 달성을 위해 필요한 함수 호출 횟수가 왼쪽 코드 3라인의 1000번 이상에서 오른쪽 코드 3라인과 같이 2번 이상으로 조건이 변경되었다.

제 6 장 사례 연구

6.1 개요

본 장에서는 상용 자동차 SW을 대상으로 가이드라인을 적용하여 실제로 테스트 커버리지를 향상할 수 있음을 보이기 위한 사례 연구를 진행한다. 가이드라인을 적용할 실험을 먼저 수행한 후, 가이드라인을 적용한 결과를 보인다. 가이드라인을 적용할 실험은 2.2절에서 소개하는 동적 기호 실행 기반 자동화된 테스크 테스팅 기법을 사용해서 구축한 테스팅 환경을 기반으로 수행한다. 동적 기호 실행 기반 자동화된 테스크 테스팅 기법은 동적 기호 실행 기반 자동화된 유닛 테스팅 기법[20] 프레임워크를 사용하며 해당 프레임워크는 실제 산업체 프로그램을 대상으로 버그를 성공적으로 검출하여 그 효과성을 검증하였다. 본 사례 연구에서는 해당 프레임워크를 적용하고도 미달성된 분기에 대해 가이드라인을 적용하여 얼만큼의 커버리지 향상이 있는지를 확인한다.

구성은 다음과 같다. 먼저 사례 연구를 진행할 대상 프로그램과 실험 환경을 차례대로 소개한다. 이후 2.2절 동적 기호 실행 기반 자동화된 테스크 테스팅 기법으로 구축한 테스팅 환경에서 실행한 Concolic 테스팅 실험 결과를 차례대로 소개한다. 마지막으로 5절에서 제안한 가이드라인을 추가 적용하여 테스트 커버리지 향상을 확인한다. 그림 6.1는 대상 파일 구조 예시이다. 모서리가 둥근 네모 박스는 하나의 파일을 나타내며 내부에 있는 각 노드는 함수를 의미한다. 사각형 노드와 원형 노드는 각각 non-static 함수와 static 함수를 의미하며, 노드 간의 화살표는 호출 관계를 의미한다. 그림처럼 각 파일은 non-static 함수와 static 함수들로 구성된다. 각 함수는 같은 파일 내의 non-static 함수, static 함수 또는 다른 파일의 non-static 함수를 호출할 수 있다. 단, 함수 호출 관계에서 사이클은 존재하지 않는다. static 함수는 해당 함수가 선언된 파일 내부에서만 호출할 수 있으므로, 외부에서 파일에 있는 기능을 사용하려면 해당 기능을 실행하는 non-static 함수를 호출해야 한다.

이때, non-static 함수를 인터페이스(interface)라고 부르고 인터페이스를 호출했을 때 실행되는 대상 코드 내의 일련의 작업을 테스크(task)라고 하자. 본 논문에서는 거짓 경보를 최소화하기 위해 각 함수 단위가 아닌, 독립적으로 실행 가능한 최소 단위인 테스크 별로 테스팅을 수행한다. 테스크 별로 테스팅을 수행하기 위해서 대상 파일에 대해 2.2절 동적 기호 실행 기반 자동화된 테스크 테스팅 기법을 기반으로 테스트 드라이버/스텁을 생성하면 대상 파일에 존재하는 인터페이스마다 해당 인터페이스를 호출하여 테스크를 실행하는 테스트 드라이버가 생성된다.

예를 들면, 그림 에서 대상 파일을 a.c라고 할 때, 각각의 인터페이스 f(), g(), h()에 대해 테스트 드라이버 f_driver(), g_driver(), h_driver()가 생성된다. 이 때, 인터페이스 f를 호출했을 때 실행될 수 있는 함수 f, s1, s3, s4를 하나로 묶어 같은 테스크에 속한다고 볼 수 있다. 함수 i는 대상 파일에 속해있지 않으므로 테스크 범위에 포함되지 않는다. 이처럼 대상이 아닌 파일에 있는 함수는 스텁으로 대체한다. 즉, 대상 파일을 a.c라고 할 때, 함수 i는 대상 파일에 속해있지 않으므로, 함수 s4는 실제 함수 i가 아닌 대체된 스텁을 호출한다.

대상 파일을 정하고 해당 파일에 대한 테스트 드라이버/스텁 생성 후, 생성된 테스트 드라이버를 실행시켜서 각 테스크에 대한 테스팅을 수행한다. 이후, 테스팅을 통해 생성한 모든 테스트 케이스들을 대상 파일에서 재실행 시킨 뒤, 누적된 분기 커버리지를 측정한다. 예를 들면 대상 파일 a.c에서 인터페이스 f, g, h에 대한 테스트 드라이버 실행시켜서 생성한 테스트 케이스들을 각각 TCs_f, TCs_g, TCs_f이라고 하자. 이 후, 테스트 케이스 TCs_f, TCs_g, TCs_f를 대상 파일 a.c에 대해 실행하고 측정한 커버리지가 a.c에 대한 커버리지가 된다.

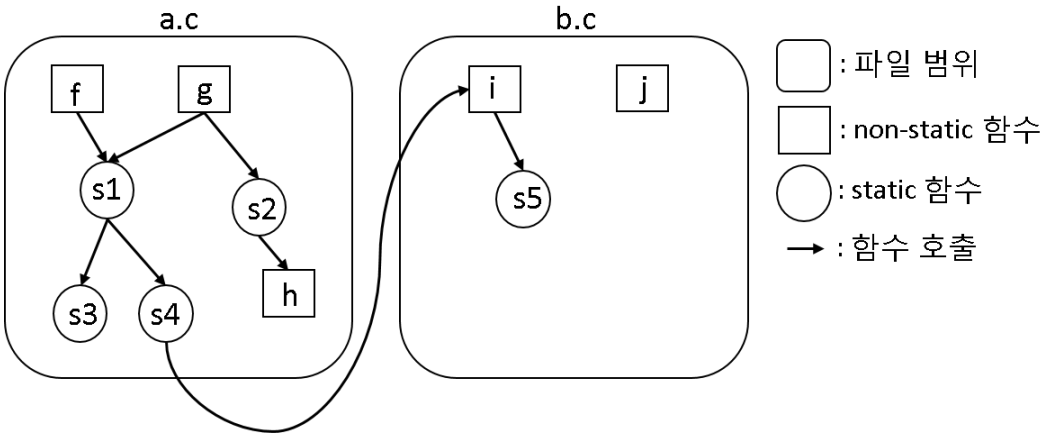

그림 6.1: 대상 파일 구조 예시

표 6.1: 대상 A 모듈 요약 정보

	LoC	# of functions			# of BRs			# of variables		
		non static	static	total	in non static	in static	total	non-static global	static global	local static
Total	44769	341	1183	1524	1185	23538	24723	986	741	0
min	22	2	0	2	0	0	0	0	0	0
max	4928	15	111	113	212	3396	3420	51	61	0
average	347.0	2.6	9.2	11.8	9.2	182.5	191.7	7.6	5.7	0.0

6.2 대상 프로그램 소개

사례 연구 대상 프로그램은 자동차 소프트웨어 회사와 함께 산학과제를 수행한 두 가지 C 프로그램 소프트웨어 모듈(A 모듈, B 모듈)을 사용한다.

6.2.1 A 모듈

저자가 담당한 A 모듈의 총 파일 수는 129개, 파일에 포함된 총 함수 개수는 1524개, 라인 총 라인수는 44769가 된다. 사용되는 변수 자료형 중 실수형 타입은 존재하지 않으며, primitive 타입인 char, unsigned char, unsigned short, unsigned long, long이 존재했다. 그 외, 배열, 구조체, union, enum, bit field 타입 등이 사용되었다. 특히, 대상 모듈 파일 내에 math.h와 같이 외부 라이브러리에 있는 함수 호출 구문은 존재하지 않았다. 외부 라이브러리 함수 정의는 대상이 되는 소스 코드에 포함되어 있지 않으므로 소스 코드 기반의 Concolic 기법에서는 외부 라이브러리 함수에 대한 심볼릭 경로 수식 수집이 불가능해진다. 따라서 외부 라이브러리 함수 호출 구문 여부는 커버리지 결과에 중요한 영향을 미친다. 본 모듈에서는 외부 라이브러리 함수 호출 구문이 없기 때문에 그로 인한 문제점을 배제하고 결과 분석이 가능하다.

표 6.1는 A 모듈의 요약 정보를 나타낸다. 파일 별로 표의 metric값을 계산한 상세 정보는 편의상 생략하였고, 요약 정보만 나타냈다. LoC는 각 파일의 소스 코드 라인을 의미하며 상용 소스 코드 분석 도구 Understand을 사용하여 대상 파일을 분석한 뒤 출력되는 소스 코드 라인 값(Source Lines of Code)[40]을

표 6.2: 대상 B 모듈 요약 정보

	LoC	# of functions			# of BRs			# of variables		
		non static	static	total	in non static	in static	total	non-static global	static global	local static
Total	34262	149	540	689	1491	24159	25650	1327	668	612
min	47	1	0	3	0	0	3	0	0	0
max	4351	36	88	91	832	5899	5899	199	180	77
average	1269.0	5.5	20.0	25.5	55.2	894.8	950.5	49.1	22.7	73.9

사용했다. 따라서 LoC에 대한 요약 정보 Total, min, max, average는 각각 전체 파일 LoC의 합, 가장 적은 LoC를 가지는 파일의 LoC, 가장 많은 LoC를 가지는 파일의 LoC, 파일별 LoC의 평균을 의미한다. # of function은 파일별 포함된 함수 개수를 의미하며, non-static 함수와 static 함수로 분류하여 표기하였다. # of BRs는 파일별 포함된 분기 개수를 의미하며 non-static 함수에 포함된 분기 개수와 static 함수에 포함된 분기 개수로 각각 세분화하여 나타냈다. # of variables는 local 변수를 제외한 파일에서 사용하는 변수 개수를 의미하며, non-static global, static global, local static으로 분류하여 나타냈다. # of functions과 # of variables 값은 대상 소스 코드를 clang 라이브러리(ver.clang-4.0)로 각각의 함수, 함수 타입, 변수, 변수 타입 등을 파싱하여 횟수를 획득했다.

6.2.2 B 모듈

저자가 담당한 B 모듈의 총 파일 수는 27개, 파일에 포함된 총 함수 개수는 689개, 총 라인 수는 34262가 된다. 표 6.2는 B 모듈의 요약 정보를 나타낸다.

사용되는 변수 자료형은 A 모듈과 마찬가지로 실수형 타입은 존재하지 않으며, primitive 타입인 char, unsigned char, unsigned short, unsigned long, long이 존재했다. 그 외, 배열, 구조체, union, enum 데이터 타입 등이 사용되었다. A 모듈과 마찬가지로 B 모듈 파일 중 외부 라이브러리 함수에 대한 호출은 찾아볼 수 없었다.

6.3 실험 환경

실험은 Intel (R) Xeon(R) CPU X5650 @ 2.67GHz, linux 16.04.3 LTS 환경에서 진행하였으며 탐색 기법으로는 DFS와 CFG를 사용했다. A 모듈과 B 모듈의 각 파일에 대해 동적 기호 실행 기반 자동화된 테스크 테스팅 기법을 사용해 필요한 테스트 드라이버/스텁등을 생성한 뒤 각 인터페이스 함수에 대한 테스트 드라이버를 실행하여 실험을 진행했다. 실험은 1차 실험을 수행하고, 아직 탐색하지 못한 경로가 남아있는 테스크에 대해 2차 실험을 수행했다. 1차 실험에서는 DFS 탐색 기법을 사용하여 최대 100000개의 테스트 케이스를 생성한다. 1차 실험에서 메모리 부족 에러(out of memory) 발생 없이 100000개 미만의 테스트 케이스를 생성한 테스크는 저자가 설정한 환경에서 가능한 모든 실행 경로를 탐색했기 때문에 2차 실험을 수행하지 않는다. 1차 실험에서 최대 테스트 케이스(100000개) 개수만큼 테스트 케이스를 생성한 대상에 대해서는 더 많은 프로그램 경로를 탐색할 여지가 남아있기 때문에 다른 탐색기법, CFG를 사용하여 최대 100000개의 테스트 케이스를 추가로 생성한다. 더불어 1차 실험 도중 메모리가 부족(out of memory)하여 100000개 미만의 테스트 케이스를 생성하고 종료된 경우에는 다른 탐색 기법을 사용했을 때 더 많은 프로

```
1. void target_interface(int *arr, int size) {
2.   for(int i=0;i<size;i++) arr[size] = 1;
3. }
```

그림 6.2: 포인터와 포인터가 가리키는 메모리 크기를 인자로 받는 함수 예제

그램 경로를 탐색할 가능성이 존재하기 때문에 해당 대상 테스크에 대해서도 2차 실험을 진행한다. 테스트 생성 중 무한 루프에 빠지는 경우, 현재 테스트 생성 작업을 중지하고 빠르게 다음 테스트 생성 작업을 시작해야 하므로 테스트 케이스 생성 timeout을 설정했다. 즉, 테스트 생성시간이 timeout을 초과하는 경우 현재 테스트 생성 작업을 중단하고 다음 테스트 생성 작업을 시작한다. timeout은 테스트 드라이버에서 호출하는 대상 인터페이스 횟수당 0.5초를 부여했다. 즉, A 모듈의 경우 테스트 드라이버에서 대상 인터페이스를 1번만 호출하기 때문에 timeout을 0.5초로 부여했다. B 모듈의 경우 테스트 드라이버에서 대상 인터페이스를 5번 호출하기 때문에 timeout을 2.5초(0.5초 * 5회 호출)로 부여했다. A 모듈과 B 모듈의 테스트 드라이버에서 대상 인터페이스를 호출하는 횟수가 다른 이유는 본 절의 "대상 인터페이스 호출 횟수 설정"란을 참고한다.

테스트 드라이버 수동 변경 내역

동적 기호 실행 기반 자동화된 테스크 테스팅 기법에서 생성한 테스트 드라이버를 사용했을 때, 일부 대상에 대해 아래와 같이 잘못된 메모리 접근이 발생하거나 불필요한 경로를 탐색하는 경우가 발생하여 문제를 야기하는 테스트 드라이버를 직접 수정했다.

1. 배열을 가리키는 포인터와 해당 배열의 길이 값을 인자로 받는 인터페이스 함수

그림 6.2은 문제가 되는 함수를 간략화하여 나타낸 것이다. target_interface는 인자로 int형 포인터 arr과 배열 사이즈를 의미하는 변수 size를 받아서 배열의 모든 원소를 1로 초기화한다. 테스트 드라이버에서는 배열과 배열 사이즈 간의 관계를 고려하지 않고 모든 인자를 심볼릭 선언하기 때문에 size의 값은 arr의 크기를 초과하거나 0보다 작은 음수 값이 입력으로 주어질 수 있다. 이로 인해 발생하는 잘못된 메모리 접근을 막기 위해 테스트 드라이버에서 target_interface를 호출할 때 인자 size을 심볼릭 선언하는 대신, arr의 크기에 해당하는 고정 상수 값을 넘기도록 수정했다. 해당 유형은 B 모듈 중 한 파일에 존재했다.

2. 미리 정의된 값을 갖는 전역 변수

일부 전역 변수 배열은 모든 가능한 자동차 관련 품명, 옵션과 같은 설정값을 갖도록 초기화되며 프로그램 실행 중에 참조된다. 이러한 전역 변수 배열의 각 원소를 심볼릭 선언하는 것은 심볼릭 경로 수식을 늘려 메모리 사용량을 늘리고, 불필요한 탐색 공간을 늘리게 된다. 이로 인한 문제를 완화하기 위해 테스트 드라이버에서 해당 전역 변수 배열을 심볼릭 선언하지 않고, 원본 소스 코드에서 초기화되는 값과 동일하게 초기화하도록 변경했다. A 모듈 중 2개 파일에서 이러한 유형이 존재했다.

대상 인터페이스 호출 횟수 설정

한 iteration 당 함수 호출 횟수 (동적 기호 실행 기반 자동화된 테스크 테스팅 기법의 iter 값)는 A 모듈과 B 모듈에 따라 다르게 설정했으며, A 모듈은 1번 호출하고 B 모듈은 5번 호출하도록 설정했다. A 모듈의 경우 static 지역 변수가 사용되지 않는 반면, B 모듈에서는 static 지역 변수가 사용되어 1번 호출만으로는 달성 불가능한 분기들이 존재하기 때문이다. 그림 6.3은 static 지역 변수로 인해 1번 호출만으로 달성이 불가능한 분기문 예시이다. 함수 f()를 1번 호출했을 때, 4 라인의 case문은 실행이 가능하지만 5~7라인의

```
1. void f() {
2.   static int flag = 1;
3.   switch (flag) {
4.     case 1: execution1; flag = 2; break;
5.     case 2: execution2; flag = 3; break;
6.     case 3: execution3; flag = 4; break;
7.     case 4: execution4; break;
8.   }
9. }
```

그림 6.3: static 지역 변수를 switch문 입력으로 사용하는 코드 예 예제

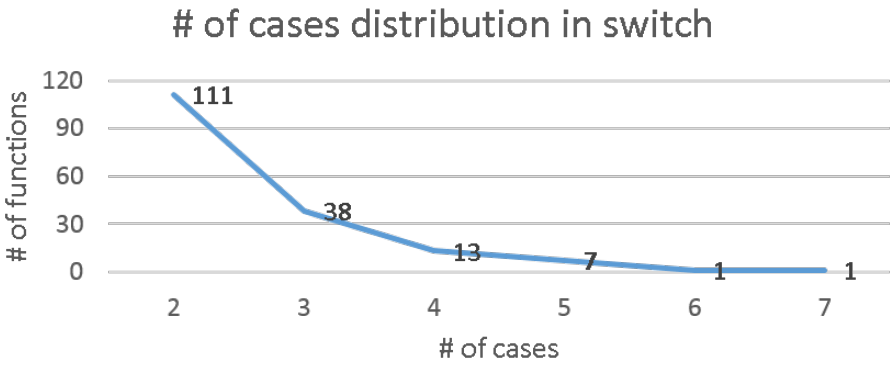

그림 6.4: siwth문이 갖는 case문 개수 분포

case문을 모두 실행하기 위해서는 최소한 case문 개수와 같이 4번 이상 호출하는 것이 필요하다.

그림 6.4는 B 모듈 중 static 지역 변수를 switch 문의 입력으로 사용하는 분포 그래프이다. 가로축은 switch 문 내부에 포함된 case문 개수(함수 내 여러 switch 문이 존재할 경우 가장 많은 case문 개수)을 나타내고, 세로축은 함수 개수를 의미한다. 171개의 함수에서 static 지역 변수를 입력으로 사용하는 switch문이 존재했으며 그래프와 같이 case 개수가 2인 switch문이 가장 많이 존재했고, case 개수가 각각 6과 7인 switch 문은 1개씩 존재했다. 98.8%에 해당하는 대부분의 switch문이 5개 이하의 case문을 가지므로 이 case문들을 실행하는 데 필요한 최소 호출 횟수인 5로 설정했다. 호출 횟수를 더 늘리지 않고 5로 제한한 이유는 첫 번째로 호출 횟수에 비례하여 탐색 공간(search space)이 많이 증가하기 때문이다. 즉, 대상 함수를 한 번만 호출했을 때 탐색 공간을 T라고 했을 때, 두 번 호출했을 때 탐색 공간은 T*T가 된다. 이로 인해 어떤 한 분기를 실행하는 데 필요한 iteration 횟수가 대상 함수 호출 횟수에 따라 지수적으로 증가할 수가 있다. 두 번째로는 호출 횟수에 따라 테스팅 실행시간과 필요한 메모리가 늘어나기 때문이다. 대상 함수 호출 횟수가 증가함에 따라 대상 프로그램의 실행시간뿐만 아니라 수집하는 심볼릭 경로 수식도 늘어나게 되므로 이로 인한 오버헤드 비용(심볼릭 경로 수식의 해를 구하는데 걸리는 시간, 심볼릭 경로 수식 기록을 위한 메모리)을 고려해야 한다.

표 6.3: 대상 A 모듈 실험 결과 요약 정보

	# total brs	# cov brs	BR cov	# inter-faces	DFS search			CFG search		
					# TCs	time(s)	# OOM	# TCs	time(s)	# OOM
Total	24723	23569	95.3%	341	2514045	220554	1	2400000	173565	0
min	0	0	50.0%	2	2	0	0	0	0	0
max	3420	3252	100.0%	15	200032	25521	1	200000	17186	0
average	191.7	182.7	97.9%	2.6	19488.7	1709.7	0.0	18604.7	1345.5	0.0

6.4 실험 결과

각 파일별 상세 실험 결과는 편의상 생략했으며 여기서는 모듈별로 요약한 정보를 보여준다. 실험 결과 표에서 # total brs는 파일별 포함된 분기 개수를 나타낸다. 따라서 # total brs의 Total, min, max, average 값은 차례대로 파일 포함된 분기의 총합, 분기가 가장 적은 파일의 분기 개수, 분기가 가장 많은 파일의 분기 개수, 파일별 평균 분기 개수를 의미한다. # covered brs는 이 중 실행된 분기 개수를 의미한다. BR cov는 각 파일의 분기 커버리지를 의미한다. 단, BR cov의 Total 값은 파일별 분기 커버리지의 총합이 아니라, 전체 분기 커버리지를 의미한다. 즉, 모든 파일에 대해 실행된 분기 개수를 모든 파일에 존재하는 총 분기 개수로 나눈 값이다. # interfaces는 파일에 포함된 인터페이스 개수를 의미한다(테스트 드라이버에서 각 파일에 포함된 인터페이스를 호출). DFS search와 CFG search는 각각 DFS 탐색 기법을 사용한 1차 실험 정보와 CFG 탐색 기법을 사용한 2차 실험 정보를 의미한다. 1차 실험에서 최대 테스트 케이스 개수만큼 생성한 테스크에 대해서만 2차 실험을 추가로 진행하기 때문에, CFG 탐색 기법에서 생성한 테스트 케이스가 0인 대상 파일은 1차 실험에서 실행 가능한 경로를 모두 탐색했으므로 2차 실험을 할 필요가 없다고 볼 수 있다. # TCs는 해당 파일에서 생성한 테스트 케이스 개수를 의미하며, time(s)는 테스트 케이스를 생성하는데 총 소요된 시간을 의미한다. # OOM은 인터페이스 함수를 실행하여 테스트 케이스를 생성하는 과정에서 메모리가 부족(out of memory)하여 도중에 종료된 개수를 의미한다.

6.4.1 A 모듈 결과

표 6.3는 파일별 A 모듈 실험 결과를 요약한 정보이다. 파일별 실험 결과는 편의상 생략한다. 표에서 볼 수 있는 것 처럼 A 모듈의 전체 분기 24723개 중 23569개를 달성하여 전체 95.3% 분기 커버리지를 달성한 것을 확인할 수 있다. 1차 실험에서 생성한 테스트 케이스는 총 2514045개이며 1 코어 실행 기준으로 약 2일 13시간 15분이 소요됐다. 이 중 1개의 인터페이스 함수를 실행하는 과정에서 메모리 부족 에러가 발생했다. 2차 실험에서 생성한 테스트 케이스는 총 2400000개이며 1 코어 실행 기준으로 약 2일 12분이 소요됐다. 2차 실험에서 메모리 부족 에러는 발생하지 않았다.

그림 6.5는 커버리지 별 A 모듈의 파일 개수를 나타낸다. 2개 파일(A29, A126)은 파일 내 분기가 없기 때문에 그래프에서 제외됐다. 그래프에서 분기 커버리지를 90% 보다 높게 기록한 파일 개수는 총 124개로 전체 파일 중 약 96%에 해당하는 수치이다. 이는 B 모듈보다도 높은 수치이며 A 모듈의 코드가 B 모듈보다 간단하기 때문에 더 높은 커버리지를 달성했다고 볼 수 있다. A 모듈과 B 모듈의 분기 수는 각각 24723, 25650개로 비슷한 반면 A 모듈과 B 모듈의 함수 개수는 각각 1524, 689개로 함수당 분기 개수가 B 모듈이 더 많고 코드가 복잡한 것을 알 수 있다.

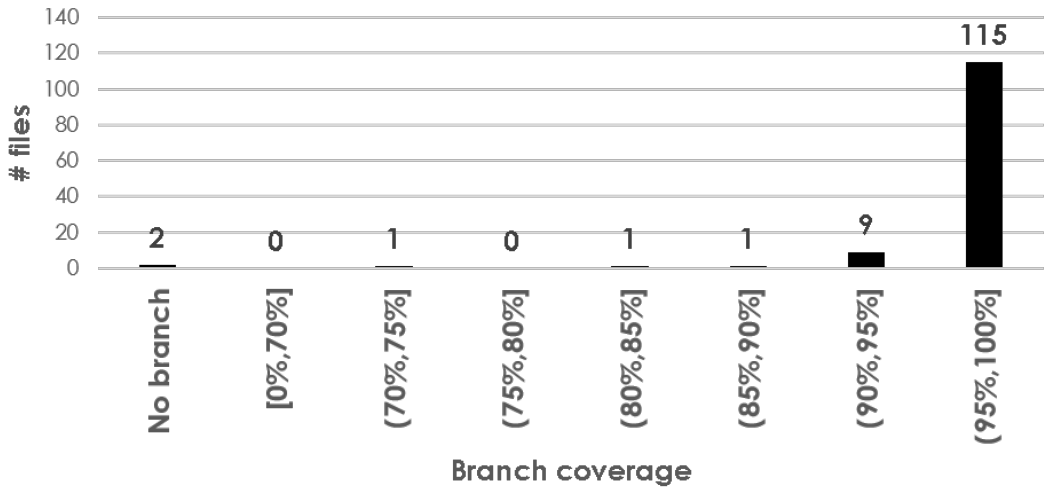

그림 6.5: 커버리지 별 A 파일 개수 분포

표 6.4: 대상 B 모듈 실험 결과 요약 정보

	# total brs	# cov brs	BR cov	# inter-faces	DFS search # TCs	DFS search time(s)	DFS search # OOM	CFG search # TCs	CFG search time(s)	CFG search # OOM
Total	25650	18394	71.7%	149	4417690	451027	12	4004700	279714	12
min	0	0	3.4%	1	8	0	0	0	0	0
max	5899	5336	100.0%	36	908118	121666	7	567091	42345	7
average	950.0	681.3	79.3%	5.5	163618.1	16704.7	0.4	148322.2	10359.8	0.4

6.4.2 B 모듈 결과

표 6.4는 파일별 B 모듈 실험 결과를 요약한 정보이다. 표에서 볼 수 있는 것처럼 B 모듈 전체 분기 25650개 중 18394개를 달성하여 전체 71.7% 분기 커버리지를 달성한 것을 확인할 수 있다. 이는 A 모듈의 결과인 95.3%에 비해 낮은 결과이다. 1차 실험에서 생성한 테스트 케이스는 총 4417690개이며 1 코어 실행 기준으로 약 5일 5시간 17분이 소요됐다. 이 중 12개의 인터페이스 함수를 실행하는 과정에서 메모리 부족 에러가 발생했다. 2차 실험에서 생성한 테스트 케이스는 총 4004700개이며 1 코어 실행 기준으로 약 3일 5시간 41분이 소요됐다. 표에서 파일당 평균 분기 개수는 950.0개로, A 모듈의 평균 분기 개수 191.7개의 약 5배 정도 많은 것을 확인할 수 있다. DFS 탐색 기법에서 생성한 B 모듈의 파일별 평균 테스트 케이스 생성 개수도 16361.1개로 A 모듈의 파일별 평균 테스트 케이스 생성 개수인 261.5개의 약 62배에 해당하는 수치를 보였다. B 모듈의 평균 테스트 케이스 생성 개수가 많은 이유는 B 모듈의 파일당 분기 개수가 A 모듈의 파일당 분기 개수보다 많고, B 모듈은 대상 함수를 5번 호출하므로 이로 인해 탐색 공간이 많이 증가했기 때문이라고 볼 수 있다. OOM (out of memory) 발생 횟수도 대상 함수를 5번 호출하는 과정에서 심볼릭 수식 입력 길이가 증가하여 많은 메모리를 사용한 탓으로 볼 수 있다.

그림 6.6은 커버리지 별 B 파일 개수 분포를 나타낸 그래프이다. 1개 파일(B2)은 파일 내 분기가 없기 때문에 분포 그래프에서 제외됐다. 95% ~ 100% 달성 커버리지 구간에 파일들이 밀집되어있던 A 모듈의 그래프와 달리 파일별 달성 커버리지가 비교적 널리 분포하는 것을 확인 할 수 있다.

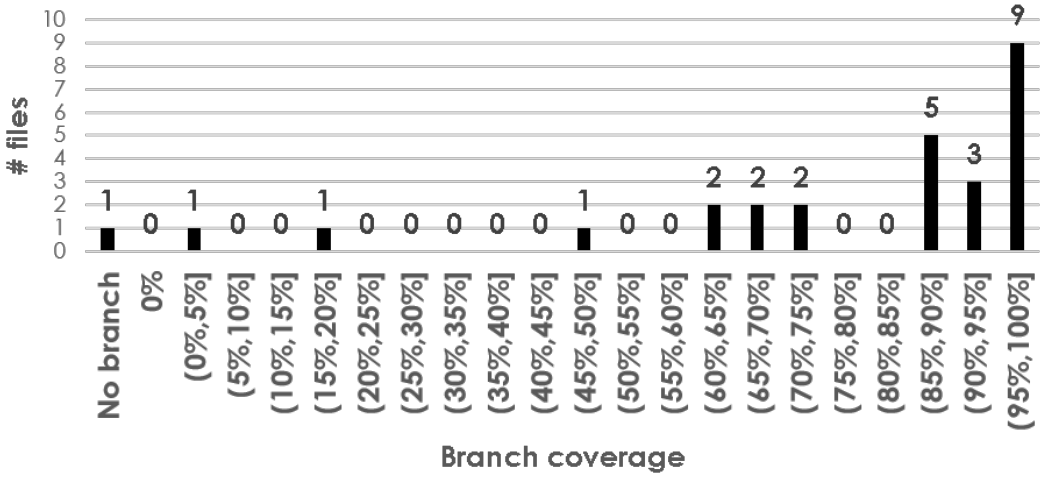

그림 6.6: 커버리지 별 B 모듈 파일 개수 분포

표 6.5: 분기 커버리지가 90% 미만인 B 모듈 파일의 미달성 분기 통계

File	# total brs	# cov brs	# uncov brs	BR cov	# uncovered branches per type					
					Type 1	Type 2	Type 3	Type 4	Type 5	Type 6
B1	8	4	4	50.00%	0	0	0	0	4(0)	0
B6	435	303	132	69.70%	0	0	0	2	130(4)	0
B7	548	470	78	85.80%	0	8(0)	0	15	0	55(24)
B12	4011	679	3332	16.90%	3323(470 (104))	0	0	8	0	1(0)
B19	238	8	230	3.40%	0	0	226(34)	4	0	0
B20	252	154	98	61.10%	0	92(9)	0	6	0	0
B23	442	379	63	85.70%	0	51(6)	0	7	0	5(3)
B25	333	225	108	67.60%	0	0	0	107	0	1(0)
B27	935	806	129	86.20%	0	108(7)	0	19	0	2(0)
Total	7202	3028	4174	42.00%	3323(470(104))	259(22)	226(34)	168	134(4)	64(27)

6.5 가이드라인 적용 결과

본 절에서는 본 장에서 수행한 실험에서 미달성된 분기들을 5장에서 분류한 미달성 분기 유형에 따라 분류한 뒤, 가이드라인을 적용하여 실제로 얼만큼의 미달성 분기를 달성할 수 있는지를 보이고자 한다. B 모듈 파일 중 90%미만의 커버리지를 기록한 14개 파일 중 9개 파일들을 분석하여 식별한 미달성한 분기 원인을 유형 별로 묶어 소개한다. 나머지 5개의 파일(B3, B5, B11, B14, B26)은 분석을 마치지 못하여 대상에서 제외됐다. A 모듈의 경우, B 모듈보다 코드 복잡도가 단순(함수당 분기 개수가 적음)하고 그림 6.5에서 보듯이 대부분의 파일 커버리지가 90%을 넘는 수치를 기록했다. 90%를 넘지 않는 파일의 경우에도 미달성 분기의 절대 개수가 적어서 가이드라인을 적용했을 때의 결과가 편중될 가능성이 있기 때문에 보다 많은 미달성 분기를 기록한 모듈 B를 대상으로 가이드라인을 적용했다. 더불어 유의미한 차이를 확인할 수 있도록 모듈 B 중 90% 미만의 커버리지를 달성한 파일을 대상으로 가이드라인을 적용했다.

표 6.5는 각 파일에 대해 유형별 미달성 분기 개수와 각 유형에 가이드라인을 적용한 후 남아있는 미달성 분기를 보여준다. 이때, 괄호 안에 포함된 숫자가 가이드라인을 적용한 후 남아있는 미달성 분기 개수를

표 6.6: Type 1에 대한 2차 가이드라인 적용 전/후 미달성 분기 개수

Type 2	Type 3	Type 4	Type 5	Type 6	Total
393(66)	0(0)	37	0(0)	40(1)	470(104)

의미한다. Type4의 경우 데드 코드 분기이기 때문에 달성을 위한 가이드라인이 존재하지 않는다. 파일 B12의 Type 1에 해당하는 미달성 분기 개수를 살펴보면 괄호가 중복으로 사용되었는데 이 경우, 가이드라인을 두 번 적용했음을 의미한다. 즉, B12파일 포함된 Type 1의 미달성 분기 3323개에 대해 가이드라인을 첫 번째, 두 번째 적용한 후 남은 미달성 분기 개수가 각각 470, 104개로 줄어든 것을 의미한다. 가이드라인을 적용하기 전 미달성 분기 개수를 살펴보면, Type1(3323개), Type2(259개), ... , Type6(64개) 순서로 미달성 분기가 많이 포함된 것을 확인할 수 있다. Type1과 Type3에 해당하는 미달성 분기 유형은 각각 한 파일(B12, B19)에서만 관찰되지만 미달성 분기 개수는 비교적 많은 개수를 가지고 있는 것을 볼 수 있다. 그 이유로, Type1과 Type3의 미달성 분기 유형은 함수가 호출되지 않는 상황과 관련이 있는데, 함수가 호출되지 않을 경우 해당 함수에 속한 모든 분기가 해당하는 한 분기 유형에 모두 포함되기 때문이다.

Type 1은 첫 번째 가이드라인 적용 후, 미달성 분기 개수가 3323개에서 470개로 약 86%만큼 감소하였다. Type 2 역시 미달성 분기 개수가 259개에서 22개로 약 92%가 줄어들었다. 이와 같이 Type 3, Type 5, Type 6 역시 가이드라인 적용 후, 미달성 분기 개수가 각각 85%, 98%, 58% 줄어들었다. 가이드라인 적용 후에도 Type 1의 미달성 분기 개수는 470개로, 전체 분기 개수를 기준으로 비교적은 높은 수치이기 때문에 미달성 분기 470개에 대해 미달성 유형을 분류하고, 타입 별로 한 번 더 가이드라인을 적용했다. 표 6.6는 Type1에 대해 첫 번째 가이드라인 적용 후 남은 미달성 분기(470개)를 유형에 따라 다시 분류한 것을 보여준다. 괄호 안에 숫자는 두 번째 가이드라인 적용 후 남은 미달성 개수를 의미한다. Type 6의 미달성 분기를 살펴보면 40개에서 2차 가이드라인 적용 후 1개로 효과적으로 줄어든 것을 확인할 수 있다. 결과적으로 Type 1의 미달성 분기 개수는 처음 3323개에서 104개로 약 97%만큼 효과적으로 감소했다.

최종적으로 표 6.5의 각 파일에서 가이드라인 적용 전 달성한 분기 개수는 3028개에서 가이드라인 적용 후 6843개로 두 배가 넘게 증가했으며, 커버리지 역시 기존 43%에서 97%로 많은 상승 폭을 보였다 (단, 데드 코드인 Type 4 분기 개수는 커버리지 대상에서 제외됨).

제 7 장 결론 및 향후 연구

본 논문에서는 미달성 분기들을 6개의 유형으로 분류한 뒤, 유형별 가이드라인을 제공하고 있다. 각 분기를 유형별로 분류하고 테스팅 결과를 확인하는 과정에서 본 논문에서는 더욱 효율적인 작업을 위해서 자체 개발한 concolic 테스팅 도구인 CROWN에 대해 소개했다. CROWN은 기존 심볼릭 정보 출력을 개선하고, 테스팅 결과 분석에 필요한 정보를 사용하기 쉬운 형태로 사용자에게 제공하여 테스팅 분석에 걸리는 시간을 줄이도록 했다. 이로 인해 수만 라인을 대상으로 진행한 사례 연구를 성공적으로 마칠 수 있었다. 사례 연구에서는 CROWN을 활용하여 자동차 SW(A 모듈, B 모듈)를 대상으로 가이드라인을 적용하여 테스트 커버리지 향상을 보이고자 했다. 동적 기호 실행 기반 자동화된 유닛 테스팅 기법[20] 프레임워크를 기반으로 가이드라인을 적용할 테스팅을 수행하였고 모듈별로 각각 95.5%, 71.7%의 전체 분기 커버리지를 달성했다. 이후, B 모듈 중 90% 커버리지 미만의 파일들을 대상으로 가이드라인을 적용하여 기존 43%였던 분기 커버리지를 97%까지 효과적으로 향상된 것을 확인했다. 실제 환경 프로그램인 상용 자동차 SW를 대상으로 효과적인 결과를 보인 것으로부터 본 논문에서 제안한 가이드라인은 실질적으로 활용할 수 있으며, 실제 개발 환경에서도 활용성이 높을 것으로 예상한다.

향후 연구로서 더 다양한 프로그램을 대상으로 본 가이드라인의 효과성을 보이고 현재 가이드라인에서는 분류하고 있지 않은 분기 유형에 대한 조사를 수행하고자 한다. 더불어 본 논문에서 제안한 가이드라인을 C 프로그램 외 다른 언어에 적용할 수 있는지를 확인하고 프로그램 언어별 스타일에 따라 추가로 적용할 수 있는 가이드라인을 고안하고자 한다.

참 고 문 헌

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

[2] 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.

[3] P. Godefroid, "Compositional dynamic test generation," *Proc. of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 47-54, 2007.

[4] K. Sen and G. Gul, "CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools," *Proc. of the Computer Aided Verification*, pp. 419-423, 2006.

[5] 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.

[6] 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.

[7] 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.

[8] C. Marcondes, M.Y. Sanadidi, M. Gerla, R.S. Schwartz, R.O. Santos, M. Martinello, "Pathcrawler: Automatic harvesting web infra-structure," *Proc. of the Network Operations and Management Symposium*, pp. 339-346, 2008.

[9] J. Karthick, H. David, G. Vijay, K. Adam, "jFuzz: A concolic whitebox fuzzer for Java,", *Proc. of the First NASA Formal Methods Symposium*", pp. 121-125, 2009.

[10] N. Tillmann and J. de Halleux, "Pex–White Box Test Generation for .NET," *Proc. of the Second International Conference on Tests and Proofs*, pp. 134-153, 2008.

[11] P. Godefroid, M.Y. Levin, D. Molnar, "Automated whitebox fuzz testing," *Proc. of the Network and Distributed System Security Symposium*, pp. 151-166, 2008.

[12] G. Wassermann, D. Yu, A. Chander, D. Dhurjati, H. Inamura, Z. Su, "Dynamic test input generation for web applications," *Proc. of the 2008 international symposium on Software testing and analysis*, pp. 249-260, 2008.

[13] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, M.D. Ernst, "Finding Bugs in Dynamic Web Applications," *Proc. of the 2008 international symposium on Software testing and analysis*, pp. 261-272, 2008.

[14] X. Qu and B. Robinson, "A case study of concolic testing tools and their limitations," *Proc. of the Empirical Software Engineering and Measurement*, pp. 117-126, 2011.

[15] 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.

[16] Dutertre, B., & De Moura, L. (2006). The yices smt solver. Tool paper at http://yices. csl. sri. com/tool-paper. pdf, 2(2), 1-2.

[17] 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.

[18] 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.

[19] 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.

[20] Y. Kim, Y. Kim, T. Kim, G. Lee, Y. Jang, M. Kim, "Automated Unit Testing of Large Industrial Embedded Software using Concolic Testing," *Proc. of IEEE/ACM Automated Software Engineering*, pp. 519 - 528, 2013.

[21] H. Xu, Y. Zhou, Y. Kang, M.R. Lyu, "Concolic execution on small-size binaries: challenges and empirical study," *Proc. of Dependable Systems and Networks*, pp. 181-188, 2017.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

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

[29] L. Cseppento and Z. Micskei, “Evaluating symbolic execution-based test tools,” *Proc. of the IEEE 8th International Conference on Software Testing, Verification and Validation*, pp. 1-10, 2015.

[30] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs, ”Jalangi: A Selective Record-replay and Dynamic Analysis Framework for JavaScript,” *Proc. of the 2013 9th Joint Meeting on Foundations of Software Engineering, ESEC/FSE*, pp. 488–498, 2013.

[31] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, “Bap: A binary analysis platform,” *Proc. of the International Conference on Computer Aided Verification*, pp. 463-469, 2011.

[32] F. Saudel and J. Salwan, “Triton: a dynamic symbolic execution framework,” *Symposium sur la s´ecurit´e des technologies de linformation et des communications, SSTIC*, France, Rennes, pp. 31-54, 2015.

[33] Y. Shoshitaishvili and et al., “Sok: (state of) the art of war: Offensive techniques in binary analysis,” *Proc. of the IEEE Symposium on Security and Privacy*, pp. 138-157, 2016.

[34] C. D. Manuel, ”A Methodology for Applying Concolic Testing”, URN: urn:nbn:se:uu:diva-303772

[35] A. Giantsios, N. Papaspyrou and K. Sagonas, “Concolic testing for functional languages,” *Proc. of the 17th International Symposium on Principles and Practice of Declarative Programming*, pp. 137-148, 2015.

[36] K. Claessen and J. Hughes, “Quickcheck: a lightweight tool for random testing of haskell programs,” *ACM SIGPLAN notices*, vol. 46, no. 4, pp. 3–64, 2011.

[37] C. Runciman, M. Naylor and F. Lindblad, “Smallcheck and lazy smallcheck: automatic exhaustive testing for small values,” *ACM SIGPLAN notices*, vol. 44, no. 2. ACM, 2008, pp. 37–48.

[38] Erlang QuickCheck, http://www.quviq.com/products/erlang-quickcheck/

[39] M. Papadakis and K. Sagonas, “A proper integration of types and function specifications with property-based testing,” *Proc. of the 10th ACM SIGPLAN workshop on Erlang*, pp. 39–50, 2011.

[40] S. T. Inc., http://www.scitools.com/documents/metrics.php

[41] http://groups.yahoo.com/group/lp_solve/

사 사

석사 과정 동안 제가 앞을 향해서 나아갈 수 있도록 도움을 주시고 응원해 주신 모든 분께 진심을 담아 감사의 말씀을 전합니다. 먼저 제가 지금과 같은 축복을 누릴 수 있도록 무한한 애정으로 저를 키워주신 부모님 감사합니다. 힘들고 지칠 때마다 항상 격려해주시고 위로해주신 덕분에 포기하지 않고 이 자리에 있을 수 있었습니다.

언제나 저를 바른길로 이끌어 주신 김문주 교수님께 감사드립니다. 교수님의 헌신적인 가르침과 지도가 아니었다면 오늘날과 같은 기쁨은 누릴 수 없었을 것입니다. 부족함이 정말 많은 저임에도 불구하고, 교수님께서는 많은 시간을 할애하시며 포기하지 않고 끝까지 지도해주셨습니다. 연구 자체만이 아니라, 연구를 하는데 필요한 기본 소양을 가르쳐주셨고 제 앞길을 위한 소중한 조언을 해주셨습니다. 이를 통해 그동안 깨우치지 못했던 저의 부족한 점을 많이 깨우칠 수 있었고, 전보다 나은 사람으로 거듭날 수 있었습니다. 여전히 부족한 점이 많지만 가르쳐주시고 조언해주신 말씀들 모두 마음속에 깊이 새기면서 살아가도록 하겠습니다. 소프트웨어 공학 분야에서 최고로 손꼽히는 SWTV 연구실의 구성원으로서, 교수님께 가르침을 받으며 연구를 수행한 것은 제게 있어 너무나 큰 축복이고 행운이었습니다. 함께 연구할 기회를 주신 김문주 교수님께 다시 한번 감사드립니다. 본 논문의 심사위원이셨던 고인영 교수님과 백종문 교수님께도 감사 인사를 드리고 싶습니다. 두 교수님께서는 시간을 할애하시어 논문에 대한 피드백과 개선해야 하는 점을 명확히 알려주셨고, 그로 인해 훨씬 나은 방향과 주제를 갖는 논문으로 탈바꿈할 수 있었습니다. 감사드립니다.

제가 연구를 수행하고 과제를 수행하는 데 많은 도움을 주신 김윤호 박사님에게 감사드립니다. 김윤호 박사님은 제가 연구실 생활을 하면서 의지를 많이 했던 분입니다. 연구실에 들어오면서부터 김윤호 박사님에게 연구실 생활에 대한 소개를 받았고 궁금한 사항은 여쭈어보면서 연구실 생활에 잘 적응할 수가 있었습니다. 연구나 과제를 수행할 때도 김윤호 박사님이 수시로 관리를 해준 덕분에 올바른 길로 빠르게 나아갈 수가 있었습니다. 연구나 과제 중 겪는 문제는 김윤호 박사님과 상의를 했는데 그때마다 시간을 할애하여, 어떻게 해야 할지 최적의 방안을 가르쳐준 김윤호 박사님에게 큰 존경심과 고마움을 느낍니다. 석사 1학기 때 연구실 생활에 대한 진심 어린 조언을 해준 김태진 졸업생과 연구실 동기인 양웅규에게도 감사합니다. 덕분에 연구실 생활에 더 빨리 적응할 수가 있었습니다. 매주 좋은 말씀 전해주시는 지준왕 선교사님께도 감사드립니다. 항상 무슨 일이 있는지 걱정해주시고 웃는 얼굴로 격려해주셔서 많은 위로가 되었습니다. 연구실에서 함께 공부한 Loc duy phan, 고봉석, 박건우, 임현수, 김찬수, 이아청 학생에게도 감사드립니다. 모두 뛰어난 친구들이라 배울 점이 많았고 목표한 바를 성공적으로 이룰 것으로 생각합니다.

대학원 생활을 어떻게 해야 할지 방향을 잡는데 진심 어린 조언을 해주신 백승환 형에게도 감사 인사를 드립니다. 많은 도움이 되었습니다. 같은 대학교에서 올라와서 겪은 일들을 즐겁게 이야기 할 수 있었던 강민균, 고창영, 나선진, 최준민, 한승현 학생에게도 감사 인사를 드립니다. 추가로 만나지 못할 때는 통화로 격려해주고, 휴가 때 직접 만나 응원해주던 대학교 친구들과 고등학교 친구들에게도 진심으로 감사를 전하고 싶습니다. 여기에 적지 못한 많은 분께도 감사드립니다.

저의 이 작은 결실이 저를 격려해주시고 응원해주신 분들께 조금이나마 보답이 되기를 바라며 글을 마칩니다. 다시 한번 감사드립니다.

약 력

이 름: 김 현 우

생 년 월 일: 1992년 10월 19일

주 소: 대전광역시 유성구 어은동 52-10번지 희망관 1505호

메 일 주 소: hyunoo@kaist.ac.kr

학 력

2008. 3. – 2011. 2. 한국애니메이션 고등학교

2012. 3. – 2016. 2. 서강대학교 컴퓨터공학과 학사 (B.S.)

2016. 3. – 2018. 8. KAIST 전산학부 석사 (M.S.)

수 상 이 력

2018. 1. 30 한국 소프트웨어공학 학술대회 우수논문상

2016. 12. 22 한국정보과학회 동계학술발표회 우수논문상

학 회 활 동

1. **김현우**, 김윤호, 김문주, Concolic 테스팅 도구 CREST 의 사용자 친화성 향상 연구: Windows OS 로의 포팅과 개선된 CREST UI 을 통한 CREST 활용 및 분기 커버리지 분석 작업의 효율 증가, *Korea Conference on Software Engineering (KCSE)*, Jan 29-31, 2018 (Best paper award)
2. 김윤호, **김현우**, 양웅규, 김문주, 효과적인 변이 분석을 위한 C 프로그램 변이 도구 비교: Proteum과 Milu를 사용한 사례 연구, *Korea Computer Congress (KCC)*, Dec 21-23, 2016 (Best paper award)

연 구 업 적

1. **김현우**, 김윤호, 김문주, Concolic 테스팅 도구 CROWN의 사용자 친화성 측면 강화 연구, *Journal of KIISE: Software and Applications*, 2018 (under revision)