http://dx.doi.org/10.3745/KTSDE.2013.2.7.451

Automated Test Data Generation for Dynamic Branch Coverage

Chung In Sang†

ABSTRACT

In order to achieve high test coverage, it is usual to generate test data using various techniques including symbolic execution, data flow analysis or constraints solving. Recently, a technique for automated test data generation that fulfills high coverage effectively without those sophisticated means has been proposed. However, the technique shows its weakness in the generation of test data that leads to high coverage for programs having branch conditions where different memory locations are binded during execution. For certain programs with flag conditions, in particular, high coverage can not be achieved because specific branches are not executed. To address the problem, this paper presents dynamic branch coverage criteria and a test data generation technique based on the notion of dynamic branch. It is shown that the proposed technique compared to the previous approach is more effective by conducting experiments involving programs with flag conditions.

Keywords : Automated Test Data Generation, Dynamic Branch Coverage, Flag Problem

동적 분기 커버리지를 위한 테스트 데이터 자동 생성

정 인 상†

요 약

높은 테스트 커버리지 달성을 위해 심볼릭 실행, 자료 흐름 분석 및 제약 해결 기법 등을 이용하여 테스트 데이터 생성을 하는 것이 일반적이다. 최근에 그와 같은 정교한 수단 없이도 높은 커버리지를 효과적으로 달성 할 수 있는 방법이 제안되었다. 그러나 이 방법도 실행 중에 다른 메모리 로케이션들이 바인딩되는 분기 조건을 갖는 프로그램에 대해서는 높은 커버리지를 가져오는 테스트 데이터 생성이 어려웠다. 특히 플래그 조건을 가지는 프로그램에 대해서는 특정 분기들이 실행되지 않아 높은 커버리지를 달성하지 못하는 경우가 발생한다. 이 논문에서는 이 문제를 다루기 위하여 기존 커버리지 기준을 개선한 동적 분기 커버리지 기준들과 동적 분기에 기반 한 테스트 데이터 생성 전략을 제안한다. 실험을 통하여 플래그 조건이 있는 프로그램들에 대해서도 제안된 방법이 기존의 방법에 비해 효과적으로 커버리지를 달성함을 보인다.요약은 무슨 연구를 어떻게 수행하였는지, 주된 연구결과와 그 중요성에 관해 간결하게 기술하여야 한다.

키워드 : 테스트 데이터 자동 생성, 동적 분기 커버리지, 플래그 문제

1. 서 론

테스트 데이터를 자동으로 생성하는 것은 소프트웨어 테스팅 비용을 줄이기 위한 매우 효과적인 방법이다. 제안된 테스트 데이터 자동 생성 방법들은 테스트할 경로를 명시적으로 주느냐에 따라 목적 지향적(goal-oriented)과 경로 지향적(path-oriented) 방법으로 구분된다[1]. 목적 지향(goal-oriented) 테스트 데이터 생성 방법은 특정 프로그램 경로를 제공하는 대신에 프로그램 상의 한 블록을 주고 이를 실행할 수 있는 입력 값을 생성하는 방법이다.

많은 목적 지향 테스트 데이터 생성 방법들이 진화 알고리즘(Evolutionary Algorithm, EA)에 바탕을 두고 있다. 진화 알고리즘은 자연세계의 진화과정을 모델링하여 복잡한 실세계의 문제를 해결하고자 하는 계산모델이다. 진화 알고리즘은 구조가 간단하고 방법이 일반적이어서 응용범위가 매우 넓으며, 특히 적응적 탐색과 학습 및 최적화를 통한 공학적인 문제의 해결에 많이 이용되고 있다. 진화 알고리즘을 이용하여 테스트 데이터를 생성하는 방법을 진화 테스팅(Evolutionary Testing, ET)라 한다[2].

ET는 후보 테스트 데이터와 원하는 테스트 데이터간의 차이를 평가하기 위해 적합성 함수(fitness function)를 이용한다[3]. 예를 들면, 어떤 프로그램에서 x==5인 분기 조건이 참이 되게 하는 테스트 데이터를 구하는 문제를 생각해보

※ 본 연구는 한성대학교 교내학술연구비 지원과제임.
† 정 회 원 : 한성대학교 컴퓨터공학과 교수
논문접수 : 2013년 1월 4일
수 정 일 : 1차 2013년 3월 5일
심사완료 : 2013년 3월 27일
* Corresponding Author : Chung In Sang(insang@hansung.ac.kr)

자. 이와 같은 분기 조건은 F(x) = |5-x|와 같은 함수로 간주한다. 이때 함수 F(x)를 최소화하는 입력 데이터 x는 분기 조건 x==5를 참이 되게 하는 입력 값이 원하는 테스트 데이터가 된다. 여기에서 분기 조건 x==5의 분기 거리(branch distance)는 |5-x|로 정의되며 해당 분기가 참이 되기 위해 얼마나 가까이 접근했는지를 나타낸다. 예를 들면 x가 7과 10을 각각 가졌을 경우에 분기 거리는 2와 5가 된다. 이는 x가 7일 때 10인 경우 보다 분기 조건 x==5을 참이 되게 하는 경우에 보다 더 가까이 접근했음을 의미한다. 만약 탐색 알고리즘이 10을 먼저 생성하였다면 이를 감소하는 방향으로 탐색을 진행할 것이다.

그러나 이와 같은 방식은 프로그램이 플래그 변수를 사용하는 경우에는 원하는 테스트 데이터를 찾는데 문제가 발생한다. 여기에서 플래그 변수란 참이나 거짓을 가지는 부울리언 변수를 의미한다. 만약 플래그 변수가 프로그램의 분기 조건에 사용되었다면 분기 거리는 부울리언 함수가 되며 0이나 1 둘 중의 한 값을 갖는다. 즉, 분기 조건을 참이 되게 하는 입력 값들에 대해서는 분기 거리는 0이 되지만 모든 다른 입력 값들에 대해서는 분기 거리가 1이 된다. 분기 조건이 거짓이 되게 하는 입력 값들과 참이 되게 하는 입력 값들 간의 이동에 대한 어떤 단서도 분기 거리가 제공하지 못하기 때문에 탐색 알고리즘의 성능은 랜덤 테스팅을 수행하는 경우와 같게 된다. 이러한 문제를 플래그 문제(flag problem)라고 한다[4-7]. 이러한 플래그 문제를 해결하기 위하여 프로그램 변환 및 자료 흐름 정보를 이용하여 테스트 데이터를 생성하는 방법 등이 제안되었다.

최근에 이르러 콘콜릭 테스팅(concolic testing)이라 불리는 프로그램의 모든 경로들을 탐색하는 방법에 대한 연구가 활발하게 진행되고 있다[8, 9]. 이 방법은 동적 테스트 방법과 심볼릭 실행을 결합하여 높은 테스트 커버리지를 달성하기 위해 개발되었다. 콘콜릭 테스트는 우선 무작위로 생성된 입력으로 프로그램을 수행한다. 이 때 입력에 의해 실행된 경로를 따라 심볼릭 실행(symbolic execution)을 하여 프로그램 경로 제약 조건을 생성한다. 이렇게 생성된 경로 제약 조건을 프로그램의 커버리지를 높이기 위해 이전과는 다른 프로그램 경로를 수행할 수 있는 테스트 데이터를 산출하도록 수정한다. 이러한 과정은 프로그램의 모든 경로가 실행되거나 사용자가 지정한 종료 조건을 만족할 때까지 반복된다.

그러나 콘콜릭 테스팅은 기본적으로 심볼릭 실행, 제약 조건 해결을 위한 도구가 제공되어야 한다. 또한 [10]에서 바운딩 되지 않은 루프 문제 등을 포함하여 콘콜릭 테스팅과 관련된 여러 이슈에 대해 제기되고 있다. 최근에 국내에서 C 프로그램을 테스트하기 위해 분기 커버리지 기준을 만족하는 테스트 데이터를 자동으로 생성하는 BrGen[11]이라 불리는 방법이 소개 되었다. 이 방법은 경로 지향 테스트 데이터 생성을 위해 개발된 Korel의 방법[12]을 프로그램의 모든 분기를 실행하도록 확장하였다. 따라서 심볼릭 실행이나 제약 해결과 같은 복잡하면서도 정교함이 요구되는 과정 및 도구가 필요 없다. 따라서 심볼릭 실행이나 제약 해결에 관련된 도구의 개발이 요구되지 않으며 이는 소프트웨어 테스팅 비용을 줄이는 한 요인으로 작용한다. 또한 특정한 제약 해결 알고리즘에 의존되지 않기 때문에 매우 다양한 테스트 데이터 생성전략을 만들 수 있다[13].

그러나 이 방법도 실행 중에 다른 메모리 로케이션들이 바인딩되는 분기 조건을 갖는 프로그램에서는 효과적으로 커버리지를 달성하기 힘들다. 그 이유는 실제로 실행 중에 다른 메모리 로케이션이 바인딩 되기 때문에 동일한 분기 조건이 반복하여 실행된다 하더라도 각기 다른 분기 조건을 나타낼 수 있기 때문이다. 이와 같은 동적 분기(dynamic branch)들은 보다 정교한 테스트 데이터를 식별하는데 도움을 줄 수 있기 때문에 테스트 커버리지를 높이는데 큰 역할을 담당한다.

그러나 BrGen은 프로그램 코드 상에서 나타난 분기가 이미 실행되었다면 다시는 해당 분기를 실행하는 테스트 데이터를 생성하지 않는다. 따라서 프로그램의 어떤 부분(블록이나 분기)이 특정 동적 분기 조건을 만족되는 경우에만 실행될 수 있는 의존관계를 가지고 있을 때 이 부분들을 실행할 수 있는 테스트 데이터를 식별하는 것이 매우 어렵다. 특히 이 문제는 특히 플래그 변수가 있는 분기 조건이 있는 프로그램에 대해서 커버리지 달성을 어렵게 한다. 그 이유는 대상 플래그 변수를 설정하기 위해서는 특정 블록들이 실행되어야 하며 이 블록의 실행은 동적 분기 조건들이 만족되어야 하는 경우가 많기 때문이다.

이 논문에서는 이 문제를 다루기 위하여 우선 기존 분기 커버리지 기준을 개선한 동적 분기 커버리지 기준들을 제안한다. 또한 동적 분기에 기반 한 DyBrGen이라 불리는 테스트 데이터 생성 방법을 제안한다. 제안된 방법은 기존의 방법 BrGen과 마찬가지로 심볼릭 실행, 제약 해결, 자료 흐름 분석과 같은 수단에 의존하지 않는다. 실험을 통하여 플래그 조건이 있는 프로그램들에 대해서도 제안된 방법이 효과적으로 분기 커버리지를 달성함을 보인다.

이 논문은 다음과 같이 구성된다. 2장에서는 플래그 문제에 대해 간략하게 기술하고 Korel의 방법을 기반으로 분기 커버리지 기준을 만족하는 테스트 데이터를 생성하는 방법 BrGen에 대해 설명한다. 3장에서는 여러 가지 동적 분기 커버리지 기준에 대해 기술하고 동적 분기에 기반한 테스트 데이터 생성 방법 DyBrGen에 대해서 기술한다. 4장에서는 대표적인 플래그 변수를 사용하는 프로그램들을 포함하여 제안된 방법을 랜덤 테스팅 및 BrGen과 비교한 실험 결과를 기술한다. 마지막으로 결론 및 향후 연구에 관해 기술한다.

2. 관련 연구

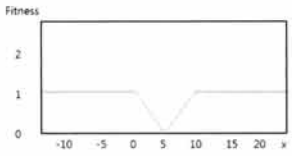
2.1절에서 높은 테스트 커버리지를 달성하게 어렵게 만드는 요인 중의 하나인 플래그 문제에 대해 간단하게 기술하고 이 문제를 해결하기 위한 기존의 연구 방법들에 대해 간략하게 소개한다. 2.2절에서는 이 논문의 선행연구인 BrGen에 대해 간략하게 기술한다.

2.1 플래그 문제

프로그램의 분기 조건에 플래그를 사용하였을 때 플래그 문제가 발생한다. 플래그 문제를 구체적으로 이해하기 위해 Fig. 1A에 주어진 간단한 프로그램을 생각해보자.

```
void flagProc1(int x) {
        int flag;
1:      flag=(x==5);
2:      if (flag)
3:              { // target ... }
}
```

A



B

Fig. 1. A. Example program B. Fitness function

목표 문장을 실행하기 위한 테스트 데이터를 자동으로 생성하기 위해서는 플래그 변수 flag가 참이 되도록 하는 입력 변수 x의 값이 필요하다. 진화 알고리즘(Evolutionary Algorithm)을 이용하여 테스트 데이터를 생성하는 테스트 데이터 방법[4, 5, 14, 15]은 이와 같은 분기 조건을 "F(x) = |flag|"와 같은 함수로 간주한다. 이때 함수 F(x)를 최소화하는 입력 값이 원하는 테스트 데이터가 된다. 그러나 플래그 변수의 값은 참 또는 거짓만을 가질 수 있으므로 |flag|로 주어지는 적합성 함수는 부울리언 함수가 된다. 이 경우에는 Fig. 1B와 같이 입력 변수 x의 값이 5일 때 0을 갖고 나머지 입력 값들에 대해서는 1을 갖는다. 따라서 우연히 입력 값이 5가 되는 경우를 제외하고는 적합성 함수가 어떻게 플래그 변수를 참이 되게 할 수 있는지에 대한 정보를 전혀 제공하지 못한다.

이와 같은 플래그 문제를 처리하기 위해 많은 연구가 진행되었다. Harman은 분기 조건에 있는 플래그 변수를 제거하기 위해 해당 플래그 변수의 정의로 대치하는 프로그램 변환 방법을 제안하였다[5]. 하지만 이러한 방법은 플래그 문제의 유형에 따라 다른 변환 방법들이 요구된다. Bottaci는 플래그 변수의 값을 결정하는 식의 분기 거리를 미리 계산하여 저장하고 실제 플래그가 사용되는 분기 조건에서 적합성을 평가하기 위해 사용하는 방법을 제안하였다[4]. 또 다른 방법으로 Baresel과 Sthamer이 제안한 자료 흐름 분석을 이용한 방법이 있다[14]. 이 모든 방법들은 불행하게도 반복문내에서 플래그 변수가 정의되는 경우에는 적용할 수 없다. Baresel 등은 이 문제를 해결하기 위해 새로운 프로그램 변환 방법을 제안하였다[15]. 그러나 이와 같은 방법들은 기본적으로 프로그램 변환이나 자료 흐름 분석과 같은 정교한 프로그램 분석도구들을 요구하며 이는 프로그램 테스트 비용을 증대 시키는 이유가 된다.

2.2 BrGen: 분기 커버리지 기반 테스트 데이터 생성

BrGen은 C 프로그램에 대해서 Korel의 방법을 분기 커버리지를 달성하기 위하여 확장한 방법이다[11]. Korel의 방법은 함수 최소화 기법에 기반을 두고 있는 대표적인 경로 지향 테스트 생성 방법이다. 함수 최소화 기법은 실제 입력 값을 사용하여 프로그램을 실행하는 대표적인 동적 기반 테스트 데이터 생성 기술이다. 예를 들면 TESTGEN[12]이나 ADTEST[15]와 같은 테스팅 시스템은 특정한 입력 값을 사용하여 주어진 프로그램 경로에 따라 실제로 프로그램을 실행하는 방식을 취한다.

이러한 시스템에서는 프로그램의 경로 상에 있는 분기 조건문을 함수로 간주한다. 예를 들면, 프로그램에 분기 조건 x>=10이 있다고 가정하고 이 조건을 참이 되게 하는 테스트 데이터를 구하는 문제를 생각해보자. 이 분기 조건문은 F(x) = 10-x if x<10, 0 otherwise 함수로 다시 표현할 수 있고 F(x)를 최소로 하는 변수 x 값을 구하는 문제로 변환된다. 즉, 함수 F(x)를 최소화하는 입력 데이터는 분기 조건 x>=10을 참이 되게 하는 입력 데이터 값이 된다.

Korel의 방법은 해당 분기 함수를 최소화하는 입력 값을 찾기 위해 각 입력 변수를 차례대로 선정하여 값을 조정한다. 탐색 이동(exploratory move)이라 불리는 첫 번째 단계에서 선정된 입력 값을 (적은 양) 증가 하거나 감소한다. 만약 이러한 값 조정에 대해 분기 함수의 값이 개선된다면 개선을 가져오는 방향으로 해당 입력 변수의 값을 매우 크게 변화시킨다. 이 단계를 패턴 이동(pattern move)이라 한다.

예를 들어, 현재 선정된 입력 변수의 값이 감소되었을 때 분기 함수의 값이 개선되었다면 해당 입력 변수의 값을 크게 감소시키며 입력 변수의 값이 증가 되었을 때 분기 함수의 값이 개선되었다면 해당 입력 변수의 값을 크게 증가 시킨다. 만약 몇 번의 성공적인 패턴 이동 후에 분기 함수의 값이 개선이 없다면 값의 변화의 크기를 줄여 시도해본다. 또한 현재 조정된 입력 값이 해당 분기를 실행하지 않을 수 있다. 이 경우에 새로운 변수에 대해 탐색 이동을 수행한다. 패턴 이동은 선정된 입력 값에 대해 분기 함수가 최소화될 때 까지 반복한다. 이 과정 후에 다른 입력 변수에 대해 탐색 이동이 다시 시작된다.

BrGen의 목적은 프로그램의 분기를 가능한 많이 실행하게 하는 테스트 데이터를 생성하는 것이다. 이를 위해 각 분기 조건 문 a op b이 실행될 때 아직 실행되지 않은 분기에 대한 분기 함수 F rel 0을 추출하여 이를 최소화 하는 입력 데이터를 생성한다. Table 1은 분기 함수 F와 rel을 보여준다.

Table 1. Branch functions

F	rel
B-A	$<$
B-A	$\leq$
A-B	$<$
A-B	$\leq$
abs(A-B)	$=$
-abs(A-B)	$\neq$

위 논리식에서 F가 분기 함수이다. 분기 함수 F는 해당 논리식이 참일 때 음수 값이나 0을 가지며 거짓인 경우에는 양수 값을 갖는다. 즉 목표 분기를 실행하는 테스트 데이터는 해당 분기에 대한 분기함수를 최소화하는 값이며 BrGen에서는 모든 분기에 대한 분기 함수를 추출하여 이를 최소화 하는 입력 데이터를 식별한다.

예를 들면, if x>5의 분기 조건에 대해 다음과 같은 두 개의 분기 함수를 생성한다: (1) 5-x<0 (2) x-5≤0. 분기 함수 (1)를 최소화하는 테스트 데이터는 참인 분기를 실행하고 분기 함수 (2)를 최소화하는 테스트 데이터는 거짓인 분기를 실행하게 된다.

특정 분기를 실행하는 테스트 데이터를 생성하는 문제는 분기 (bi, bj)에서 분기 함수 F가 주어진 경우에 다음과 같이 형식화 할 수 있다.:

- 최소화 대상 함수: F(x) rel 0 (rel∈{<.≤,=})
- 제약조건: bi가 x에 의해 실행

이는 Korel의 방법과는 다르다. Korel의 방법은 경로 기반 테스팅 방법이기 때문에 생성될 테스트 데이터는 주어진 경로를 반드시 수행해야하는 제약조건이 수반된다. 반면에 BrGen에서는 특정 프로그램 경로를 실행해야 하는 제약조건 대신에 목표 분기의 시작 블록을 실행해야 하는 제약조건만이 있다. 즉, 테스트 데이터가 어떤 경로를 실행해도 상관없다는 의미이다.

3. 동적 분기 기반 테스트 데이터 생성

이 장에서는 분기 커버리지를 만족하는 테스트 데이터를 생성하는 방법에 대해 기술한다. 3.1절에서는 이 논문에서 기반으로 하고 있는 동적 분기 개념 및 관련된 여러 정의들에 대해 소개한다. 3.2절에서는 동적 분기를 기반으로 다양한 동적 분기 커버리지 기준에 대해 기술한다. 3.3절에서는 동적 분기를 식별하는 방법에 대해 기술하고 3.4절에서는 동적 분기 탐색 전략에 대해서 기술한다.

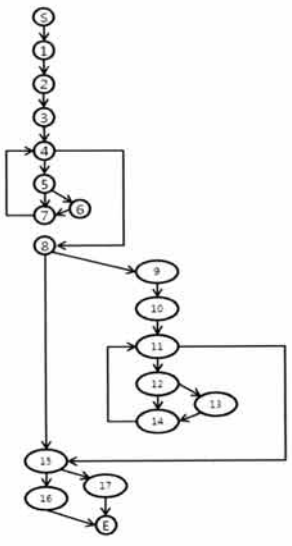
3.1 동적 분기

Fig. 2A는 보여진 프로그램 sample은 Ferguson과 Korel에 의해 개발된 테스트 데이터 생성 방법을 설명하기 위해 사용되었던 프로그램이고 Fig. 2B는 프로그램 sample의 제어 흐름 그래프이다. 입력배열의 크기를 10으로 설정하여 2장에서 기술한 BrGen 방법을 사용하여 프로그램 sample에 적용한 결과 문장 16을 수행할 수 있는 테스트 데이터를 생성할 수 없다는 사실을 몇 번의 반복된 실험으로 확인 할 수 있었다. 그 이유는 BrGen은 분기 <5, 6>, <5, 7>, <12, 13>, <12, 14>를 각각 하나의 분기로 간주하여 일단 이들 분기가 이미 테스트 데이터에 의해 실행되었다면 실행 대상에서 제외되기 때문이다.

예를 들어, k=5일 때 a[3]=10, a[5]=5, b[1]=5, b[9]=10이라면 분기 <5, 6>, <5, 7>, <12, 13>, <12, 14>를 모두 실행할 수 있는 테스트 데이터가 된다. 즉, 이러한 관점은 기존의 분기 커버리지가 요구하는 테스트 요건과 상충되지 않는다. 목표 문장 16을 실행하기 위해서는 분기 <15, 16>가 실행되어야 한다. 이를 위해서는 fb가 참으로 설정되어야 한다. 따라서 배열 a의 최소한 한 원소가 k와 같아야 하고 배열 b의 원소 모두가 k와 같아야 한다. 우선 배열 b의 모든 원소가 k와 같기 위해서는 분기 <12, 14>에서 주어진 분기 함수가 배열 원소 b[0], b[1], ..., b[9]에 대해서 각각 평가되

```
void sample(int a[], int b[], int k)
01:i=0;
02:fa=0;
03 fb=0;
04:while (i<10) {
05:   if (a[i]==k)
06:       fa=1;
07:   i=i+1;
   }
08:if (fa==1){
09:   fb=1;
10:   i=0;
11:   while (i<10) {
12:      if (b[i] !=k)
13:             fb=0
14:      i=i+1;
   }
}
15:if (fb==1)
16:    /* Target */
17: else
```

A



B

Fig. 2. A. An example program B. the corresponding control flow graph

어져야 한다. 따라서 b의 특정 원소가 아닌 b의 모든 원소에 대한 제약 조건을 도출할 수 있으며 훨씬 정교한 테스트 데이터를 추출하은 것을 가능하게 한다. 그러나 현재 BrGen에서는 배열 b의 원소 어느 하나만 분기 함수를 최소화하여도 분기 <12, 14>가 커버되었다고 간주한다. 따라서 변수 fb를 1로 설정할 확률은 거의 없기 때문에 분기 <15, 16>을 실행할 수 있는 테스트 데이터 식별이 거의 불가능하다.

이러한 문제를 해결하기 위해서는 제어 흐름 그래프에서 나타나는 분기를 보다 정제할 필요가 있다. 이를 위하여 분기 br에서 사용된 변수 X에 대해 $BIND_{br}(X)$를 br이 실행될 때 X에 바인딩 될 수 있는 메모리 로케이션들의 집합으로 정의한다. 예를 들면, $BIND_{<15,\ 16>}(fb)$은 {fb}로 변함이 없다 그 이유는 분기 <15, 16> 실행되었을 때 fb에 바인딩 되는 메모리 로케이션이 동일하기 때문이다. 그러나 분기 $BIND_{<12,\ 13>}(a[i])$(또는 $BIND_{<12,\ 14>}(a[i])$)는 실제 분기 <12, 13>(또는 <12, 14>) 실행될 때 a[i]에 a[0], a[1], ..., a[9] 등이 바인딩 될 수 있기 때문에 $BIND(a[i]_{<12,\ 13>})$(또는 $BIND_{<12,\ 14>}(a[i])$) = {a[0], a[1], ..., a[9]}이다. 만약 $K=<K_1, K_2, ..., K_m>$를 분기 br에서 사용되는 변수들의 리스트라고 할 때 $BIND_{br}(X)$ 정의를 바탕으로 DB(br)을 다음과 같이 정의한다:

$$DB(br)=\{br(y_1, y_2, ..., y_m) |\ y_i \in BIND_{br}(K_i) \text{ for } 1 \le i \le m\}$$

이 논문에서는 DB(br)의 원소를 동적 분기라 하며 기존의 분기를 정적 분기로 명확하게 구분한다. Fig. 2A의 프로그램에서는 정적 분기의 개수가 12개이지만 다음과 같이 동적 분기의 개수는 48개가 된다:

DB(<4, 5>) ={i<10},
DB(<4, 8>)= {i>=10},
DB(<5, 6>)={a[0]==k, a[1]==k, ...a[9]==k},
DB(<5, 7>)={a[0]!=k, a[1]!=k, ...a[9]!=k},
DB(<8, 9>) ={fa==1},
DB(<8, 15>)= {fa!=1},
DB(<11, 12>) ={i<10},
DB(<11, 15>)= {i>=10},
DB(<12, 13>)={b[0]!=k, b[1]!=k, ...b[9]!=k},
DB(<12, 14>)={b[0]==k, b[1]==k, ...b[9]==k},
DB(<15, 16>) ={fb==1},
DB(<15, 17>)= {fb!=1}

이 정의를 바탕으로 프로그램의 모든 동적 분기 집합 DB는 다음과 같다:

$$DB = \bigsqcup_{1}^{n} DB(br_i)$$

여기에서 n은 프로그램에 있는 정적 분기의 개수 이다. 이 논문에서는 DB(br)을 정적 분기 br에 의해 생성된 동적 분기들의 집합이라 부른다.

현재까지 얼마만큼 테스트가 되었는지 얼마나 더 테스트를 수행할 필요가 있는지 및 언제 테스트를 종료해야 하는지에 대한 기준을 설정하기 위해서는 테스트 데이터 집합 $T=\{t_i |\ i=1..m\}$에 의해 달성되는 동적 분기 커버리지를 정량화할 필요가 있다. 이를 위하여 실제 테스트 데이터에 의해 실행되는 동적분기를 고려해야 한다. $DB_{ti}(br)$은 br에 의해 생성된 동적 분기들 중에서 테스트 데이터에 t_i의해 실제 실행된 동적 분기들의 집합으로 정의할 때 DB_T는 다음과 정의된다:

$$DB_T = \bigsqcup_{1}^{n} DB_{t_i}(br_i) \text{ for } t_j \in T)$$

따라서 테스트 데이터 T에 의해 달성되는 동적 분기 커버리지 다음과 같다:

$$\frac{DB_T}{DB} X100(\%)$$

3.2 동적 분기 커버리지 기준

기존의 분기 테스트 커버리지 기준은 프로그램의 제어 흐름그래프에 있는 모든 정적 분기들을 최소한 한번은 실행되어야 하는 테스트 요건을 가지고 있다. 즉, $DB_T = DB$. 가장 직관적인 동적 분기 커버리지 기준은 모든 가능한 동적 분기들을 최소한 한 번은 실행될 것을 요구한다. 그러나 경우에 따라서는 테스트해야할 동적 분기가 필요이상으로 매우 많아 질 수 있다. 이 절에서는 동적 분기에 기반 한 여러 테스트 요건에 대해 기술한다.

ACC(All Combinations Coverage): 프로그램의 각 분기에서 사용되는 변수들에 바인딩 될 수 있는 모든 가능한 메모리 로케이션들의 조합으로 생성된 동적 분기들이 테스트 데이터 집합에 의해 최소한 한 번은 실행되어야 한다.

만약 프로그램의 각 분기 $br_i(K_i)=br(<K^i_1, K^i_2, ..., K^i_m>)$에 대해 ACC에 의해 실행되어야 하는 동적 분기의 개수는 $\prod_{j=1}^{m} |BIND_{br_i}(K^i_j)|$이다. 따라서 만약 1이 대상 프로그램의 정적 분기의 개수라면 프로그램의 모든 분기에 대해서 ACC를 만족하는 동적 분기의 개수는 $\sum_{l=1}^{n} (\prod_{j=1}^{m} |BIND_{br_i}(K^i_j)|)_l$이다. 예를 들어 프로그램에 분기 $br(K)=br(<K_1, K_2, ..., K_m>)$가 존재하고 각 K_i에 대해서 $|BIND_{br}(Ki)|=n$이라고 가정하자. 이 때 ACC에 의해 요구되는 동적 분기의 개수는 다음과 같다: $|DB(br(<K_1, K_2, ..., K_m>)|=n^m$. 이는 n의 값에 따라 테스트해야할 동적 분기의 개수가 기하급수적으로 증가될 수 있음을 의미한다.

Fig. 3의 프로그램은 분가기 두 개 존재한다:<1, 2>와 <1, 4>. 이들 분기에 의해 생성된 동적 분기의 개수가 각

```
void cov(int a[10], int b[10], int c[10], int i, int , j, int k) {

1:          if (a[i]>=b[j]+c[k]) {
2:                      ...
3:          }
4: }
```

Fig. 3. An example program

10*10*10이므로 ACC를 만족하기 위한 테스트 케이스의 개수는 2,000이다.

WCC(Weak Combinations Coverage): 프로그램의 각 분기에 대해 생성되는 동적 분기들 중에서 최소한 하나가 테스트 데이터 집합에 의해 실행되어야 한다.

WCC는 기본적으로 정적 분기 커버리지와 동일하다. 예를 들어 i=1, j=2, k=30일 때 a[1]=20, b[2]=0, c[30]=10이라면 분기 <1, 2>에 의해 생성되는 동적 분기 중 하나를 실행한다: a[1]≥b[2]+c[30]. 마찬가지로 i=1, j=2, k=3일 때 a[1]=10, b[2]=20, c[30]=10이라면 분기 <1, 4>에 의해 생성되는 동적 분기 중 하나를 실행한다: a[1]≤b[2]+c[3]. 즉, 이 두 개의 테스트 데이터는 WCC를 만족한다.

ALC(All Locations Coverage): 프로그램의 분기에서 사용되는 각 변수들에 바인딩 될 수 있는 모든 가능한 메모리 로케이션들이 포함되어있는 동적 분기들이 테스트 데이터 집합에 의해 최소한 한 번은 실행되어야 한다.

ALC를 만족할 수 있는 동적 분기들의 집합은 유일하지 않으며 여러 개가 존재 할 수 있다. 예를 들면 동적 분기 집합 {a[i]≥b[i]+c[i] | 0≤i≤9}은 분기 <1, 2>에 의해 생성되는 동적 분기들 중에서 ALC를 만족하는 동적 분기 집합의 한 예이다.

프로그램의 각 분기 $br_i(K_i)$에 대해 ALC에 의해 실행되어야 하는 동적 분기의 개수는 $MAX_{j=1}^{m} |BIND_{br_i}(K_j^i)|$이다. 따라서 l은 대상 프로그램의 정적 분기의 개수일 때 ACC를 만족하기 위해서는 $\sum_{l=1}^{n}(MAX_{j=1}^{m} |BIND_{br_i}(K_j^i)|)_l$개 이 동적 분기가 실행되어야 한다. Fig. 3의 프로그램에서는 2*10개의 동적 분기가 실행될 필요가 있다. 정의에서 명확하게 볼 수 있듯이 ALC를 만족하는 테스트 케이스 집합은 WCC를 만족함을 알 수 있다.

PWC(Pair-Wise Coverage): 프로그램의 분기에서 사용되는 각 변수들에 바인딩 될 수 있는 각 메모리 로케이션이 다른 변수에 바인딩 될 수 있는 각 메모리 로케이션과 최소한 한번은 조합되어 생성되는 동적 분기들이 테스트 데이터 집합에 의해 최소한 한 번은 실행되어야 한다.

PWC는 대표적인 조합 테스팅인 페어와이즈 테스팅과 유사하다. 즉, ACC처럼 모든 바인딩이 가능한 메모리 로케이션의 조합을 고려한 테스트 결과 만큼의 범위를 보장하지는 않지만, 경험적으로 의미있고 결함이 발생할 가능성이 높은 조합을 테스트함으로써 효율성을 높일 수 있다. PWC를 만족하기 위해서는 각 분기 $br_i(K_i)$에 대해 $(MAX_{j=1}^{m} |BIND_{br_i}(K_j^i)|)^2$ 만큼의 동적 분기가 실행되어야 한다.

T-WC(T-Wise Coverage): t개의 변수에 바인딩 될 수 있는 메모리 로케이션들의 모든 가능한 조합으로 만들어진 동적 분기들이 테스트 데이터 집합에 의해 최소한 한 번은 실행되어야 한다.

T-WC는 PWC의 자연스러운 확장이며 만약 t가 대상 분기 조건에 있는 변수들의 개수를 나타낸다면 ACC와 같게 된다. T-WC를 만족하기 위해서는 각 분기 $br_i(K_i)$에 대해 $(MAX_{j=1}^{m} |BIND_{br_i}(K_j^i)|)^t$ 만큼의 동적 분기가 실행되어야 한다. 일반적으로 테스트 데이터를 조합하는 경우에는 페어와이즈 테스팅, 즉 t=2가 많이 사용되며 이를 넘어서는 경우에는 테스트 조합의 경우가 매우 많아지고 실제 오류를 검출할 때에도 도움이 그리 되지 않는다는 연구 결과가 있다[17]. 이 같은 결과가 물론 동적 분기 커버리지 기준에도 적용될지는 미지수이며 앞에서 언급한 기준들 중에서 어떤 기준이 효과적으로 테스트를 가능하게 하는지는 앞으로 연구할 필요가 있다.

3.3 동적 분기 식별

이 절에서는 동적 분기를 식별하는 방법에 대해 기술한다. 동적 분기를 식별하기 위해서는 각 변수에 실제 어떤 메모리 로케이션이 바인딩될 수 있는지를 파악할 필요가 있다. 이를 위해서 포인터 분석과 같은 정적 프로그램 분석 방법이나 실제로 프로그램을 실행하여 바인딩 되는 메모리 로케이션을 식별하는 방법을 사용할 수 있다. 이 논문에서는 실제 프로그램을 실행 했을 때 바인딩 되는 로케이션을 사용하여 동적분기를 식별한다.

기존의 연구에서는 분기 함수의 평가를 위한 정보를 획득하기 위해 HanTestCC 도구[18]를 이용하여 분기 조건 if (x op y) 실행에 앞서 함수 HanTestGen를 호출하도록 탐침 하였다:

```
void HanTestGen(id, op_id, x, y);
```

여기에서 'id'는 대상 분기 조건을 나타내는 고유 식별번호이며 op_id는 op에 해당하는 관계 연산자 식별 번호이다. 이 탐침 함수는 분기 id가 실행될 때 다음 기능들을 수행한다:

- 해당 분기가 처음 방문된 분기라면 이를 저장한다. 만약 깊이 우선 탐색(Depth First Search, DFS)으로 테스트를 진행 한다면 스택에 저장하고 너비 우선 탐색(Breadth

First Search, BFS)으로 탐색한다면 큐에 저장한다. 탐색 전략에 대해서는 3.4절에서 상세하게 기술한다.

- 관계 연산자에 따른 분기 함수를 평가하여 개선된 경우에 한하여 분기 함수 평가 결과를 저장한다.
- 해당 분기가 목표 분기이고 실제 실행되었는지 검사한다.
- 해당 분기가 목표 분기이고 분기 함수가 개선되었는지를 검사한다. 분기 함수의 값이 개선될수록 이와 비례하여 동일 방향으로 더 많은 이동이 이루어지도록 한다. 그러나 분기 함수의 값이 더 이상 개선되지 않는다면 이전 분기 함수의 값을 유지한 상태에서 탐색 이동을 다시 수행하게 된다.

그러나 이 탐침 함수는 기본적으로 정적 분기 커버리지에 바탕을 두고 있기 때문에 해당 정적 분기로부터 생성되는 동적 분기들을 식별할 수 있는 정보를 제공하지 못한다. 이 논문에서는 동적 분기를 식별할 수 있도록 각 분기 문에 추가적으로 DyTestGen 함수를 탐침한다.

```
void DyTestGen(int id1 , int id2, unsigned long loc);
```

여기에서 id1는 고유 식별번호이며 id2는 탐침 대상이 되는 분기문의 식별 번호이다. DyTestGen 함수는 분기문에서 사용된 변수에 바인딩 되는 메모리 로케이션의 주소를 인자 loc를 통하여 전달받는다. 함수 HanTestGen은 이러한 메모리 로케이션 정보들을 바탕으로 동적 분기를 생성하거나 식별하고 동적 분기 관점에서 앞서 언급한 기능들을 수행한다. 예를 들면 Fig. 2A의 분기문 5는 다음과 같이 탐침된다:

```
DyTestGen(1, 3, (unsigned long)(a+i));
DyTestGen(2, 3, (unsigned long)&k);
HanTestGen(3, 12, (int)*(a+i), (int)k);
if (a[i]==k) {
        ...
}
```

예제 프로그램에서 분기문 5가 10번 반복 실행되는데 실행되는데 a[i]에 실행될 때 마다 각기 다른 메모리 로케이션이 바인딩 됨을 알 수 있다: a[0], a[1], ..., a[9]. 즉, 분기 a[i]==k에 의하여 다음과 같은 동적 분기들이 생성된다: {a[0]==k, a[1]==k, ..., a[9]==k}. 기존의 분기 커버리지 관점에서는 이들 동적 분기 커버리지 중의 하나만 실행되어도 해당 (정적) 분기가 커버되는 걸로 간주되었지만 이 논문에서는 실제 실행되어 생성된 모든 동적 분기들이 실행되도록 테스트 데이터를 생성하도록 하였다. 이와 같은 방식은 이이 콘콜릭 테스팅 등에서 사용한 방식이며 3.2절에서 기술한 특정 동적 분기 커버리지를 고려하여 테스트 데이터를 생성하지는 않는다. 그러나 실험에서 사용된 예제 프로그램 등에서는 모두 ALC 기준 등이 만족되었음이 확인 되었다.

3.4 테스트 데이터 생성 전략

이 절에서는 동적 분기 커버리지를 보다 효과적으로 높일 수 있는 테스트 데이터 생성을 위해 동적 분기들을 탐색하는 전략들에 대해 기술한다. 이 논문에서 고려하는 전략은 DFS와 BFS 이다.

BFS 탐색 전략은 실행 중에 방문되는 동적 분기의 순서에 따라 평가한다. 이를 위하여 지금까지 실행 중에 방문되는 동적 분기들 중에서 아직까지 실행이 안된 동적 분기들을 순서대로 큐에 저장한다. 예를 들어, Fig. 2A의 예제 프그램에서 a[i]=b[i]=0(0≤i≤9), k=5 입력에 대해 다음과 같은 동적 분기들이 생성 될 것이다(Fig. 4 참조): {a[0]==k, a[1]==k, ..., a[9]==k, a[0]!=k, a[1]!=k, ..., a[9]!=k, fa==1, fa!=1, fb==1, fb!=1}.

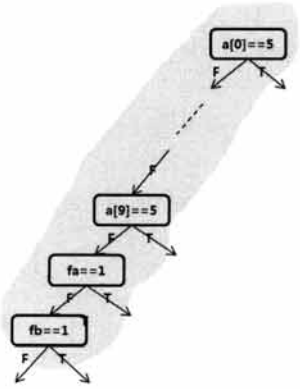


Fig. 4. Dynamic branches generated from the example program in (Fig. 2A).

이 중에서 아직 평가가 안된 동적 분기들을 순서대로 큐에 저장한다. 즉, 큐에 가장 먼저 "a[0]==5"가 저장되고 동적 분기 "a[1]==5" ..., "a[9]==5", "fa==1", "fb==1" 들이 순서대로 저장된다. 따라서 "a[0]==5"에 해당하는 분기 함수 |a[0]-5|를 최소화 하는 입력 값을 가장 먼저 찾는 작업이 수행되며 다음에 "a[1]==5" ..., "a[9]==5", "fa==1", "fb==1" 들이 순서대로 평가된다. 물론 평가 중에 생성된 테스트 데이터에 의해 새로운 동적 분기들이 식별될 수 있으며 이렇게 식별된 동적 분기들은 순서대로 큐에 저장된다.

DFS 탐색 전략은 가장 나중에 방문되는 동적 분기를 가장 먼저 평가한다. 이를 위하여 동적 분기를 저장하기 위한 자료 구조로 스택을 활용한다. 이 예에서는 스택에 "a[0]==5"가 가장 먼저 저장이 되고 가장 나중에 즉, 스택의 탑에 "fb==1"이 저장이 된다. 따라서 분기 "fb==1"에 해당하는 분기 함수가 가장 먼저 평가된다[11].

4. 실험 결과

이 장에서는 이 논문에서 제안한 동적 분기 기반 테스트 데이터 생성방법 DyBrGen의 효용성을 보이기 위해 랜덤 테스팅 및 기존의 정적 분기 기반 테스트 데이터 생성 방법

BrGen과 비교하여 실험을 수행한 결과를 기술한다. 실험에 사용된 프로그램들은 기존의 방법들이 효과적으로 테스트 데이터를 생성하지 못한 대표적인 플래그 조건 등을 가지고 있는 프로그램들[7, 12]이 포함되며 선택한 프로그램들은 Table 2에서 보이는 바와 같다.

Table 2에서 trityp, enumeration은 정적 분기와 동적 분기가 동일한 프로그램이고 나머지 sample, counter, loop_assign은 정적 분기로부터 많은 수의 동적 분기가 생성되는 경우이다. 즉, 분기 조건에 동적으로 다른 메모리 로케이션이 바인딩 되는 프로그램들이다. 또한 프로그램 trityp를 제외하고는 모두 플래그 조건을 가지고 있다. 플래그 조건을 가지고 있는 대상 프로그램에 대해서는 해당 플래그 조건이 만족되었을 때만 실행되는 목표 문장을 설정하여 어떤 테스트 방법이목표 문장을 수행할 수 있는 테스트 데이터를 생성할 수 있는지에 대한 실험도 수행하였다.

각 프로그램에 대해 랜덤테스팅, BrGen, DyBrGen을 5번씩 반복 수행하였으며 랜덤 테스팅은 DyBrGen이 생성한 테스트 데이터 개수까지 생성을 하도록 하였다. BrGen과 DyBrGen은 DFS와 BFS 탐색 전략을 사용하여 테스트 데이터를 각각 생성하여 실험을 수행하였다. 실험은 64 bit Windows Vista 운영체제, CPU는 Intel(R) Core(TM) 6420 @ 2.13Ghz, 메모리는 4GB에서 수행하였다.

Table 2. Programs used in the experiment

program		# of branches	
		static	dynamic
trityp	This is a triangle classification program. The target is executed when the type of the triangle is EQUILATERAL.	16	16
sample	program of (Fig. 2A)	12	48
counter	The program increments its counter variable only under special circumstances - when values of the inputted integer array are equal to zero. The target is executed when the counter variable becomes a certain value.	6	24
enumeration	This decides whether three inputted colour intensity values (integers in the range 0 to 255) represents one of the colours in an enumeration {UNKNOWN, WHITE, RED, YELLOW, MAGENTA, BLUE, CYAN, GREEN BLACK}. The target statement is executed when the inputs represent the colour BLACK	30	30
loop_assign	This program takes an array of ten integer values. A flag is initially set to 0, but becomes 1 when one or more of the array values is zero. This assignment occurs within a loop body. When the flag is set to 1, the target is executed.	8	26

실험은 각 테스트 데이터 생성 방법을 5개의 대상 프로그램에 대해 적용하였을 때 정적 분기 커버리지 및 동적 분기 커버리지를 구하도록 하였다. 실험에서는 동적 분기 커버리지 기준으로 3.3절의 ALC를 사용하여 측정하였다.

Table 3은 5개의 방법(랜덤, BrGen/DFS, BrGen/BFS, DyBrGen/DFS, DyBrGen/BFS)을 적용한 결과를 보여주고 있다.

Table 3. Experimental Results

trityp					
coverage	Random	BrGen		DyBrGen	
		DFS	BFS	DFS	BFS
static	81.25	100	100	100	100
dynamic(ALP)	81.25	100	100	100	100

sample					
coverage	Random	BrGen		DyBrGen	
		DFS	BFS	DFS	BFS
static	35.7	91.67	91.67	100	100
dynamic(ALP)	44	60.42	60.42	100	100
target executed	X	X	X	O	O

counter					
coverage	Random	BrGen		DyBrGen	
		DFS	BFS	DFS	BFS
static	83	83	83	100	100
dynamic(ALP)	62.8	58.33	58.33	100	100
target executed	X	X	X	O	O O

enumeration					
coverage	Random	BrGen		DyBrGen	
		DFS	BFS	DFS	BFS
static	10	100	100	100	100
dynamic(ALP)	10	100	100	100	100
target executed	X	O	O	O	O

loop_assign					
coverage	Random	BrGen		DyBrGen	
		DFS	BFS	DFS	BFS
static	50	100	86	100	100
dynamic(ALP)	45.33	65.39	61.53	100	100
target executed	X	O	X	O	O

우선 BRGen과 DyBrGen이 랜덤 테스팅에 비해 확연하게 높은 정적 및 동적 분기 커버리지를 달성하고 있음알 을 수 있다. 랜덤 테스팅은 평균 약 52%와 49% 정적 분기 커버리지와 동적 분기 커버리지를 각각 달성하였는데 비해

BrGen/DFS는 95%와 77%, BrGen/BFS 92%와 76%를 달성하였다. 이는 정적 분기 커버리지 관점에서 랜덤 테스팅 결과 대비 약 77%((95-52)/52*100) 향상되었고 동적 분기 커버리지에서도 약 55%의 향상되었음을 알 수 있다. 이 결과는 BrGen이 랜덤 테스팅에 비해 매우 효과적으로 분기 커버리지를 달성할 수 있다는 것을 확인할 수 있다. 또한 정적 분기와 동적 분기가 동일한 경우에는 모두 100% 커버리지를 달성함을 볼 수 있다.

그러나 BrGen이 이와 같이 매우 효과적인 방법임에도 불구하고 플래그 조건을 가지고 있는 4개의 프로그램에 대해서 2개만이 목표 문장을 실행할 수 있는 테스트 데이터를 생성하였다. 이에 반해 DyBrGen은 탐색 전략에 상관없이 5개의 모든 대상 프로그램에서 100% 정적 분기 커버리지 및 ALC를 달성하였으며 모든 목표 문장들이 실행되었다.

이러한 차이는 달성한 커버리지에 의해 설명될 수 있다. DyBRGen/DFS와 DyBRGen/BFS가 정적 분기 커버리지 관점에서 BRGen/DFS와 BRGen/DFS 결과 대비 각각 약 5.2%와 8.4%향상되었음을 알 수 있다. 이에 반해 동적 분기 커버리지는 DyBRGen/DFS와 DyBRGen/BFS가 BRGen/DFS와 BRGen/DFS 결과 대비 각각 약 30%와 32%향상되었음을 알 수 있다. 일반적으로 커버리지 차가 적어도 10% 이상 돼야 커버리지 차이가 실제적 의미를 가진다고 받아들여지기 때문에 정적 분기 커버리지에 비해 동적 분기 커버리지가 BrGen과 DyBrGen의 차이를 명확하게 보여줌을 알 수 있다.

Table 4는 테스트 효율성 측면에서 DyBrGen을 분석한 결과이다. 테스트 효율성은 생성한 테스트 데이터 중에서 실제적으로 테스트 커버리지에 영향을 미치는 테스트 데이터의 비율로 정의하였다. 만약 새롭게 생성되는 테스트 데이터가 새로운 분기를 실행하는 비율이 많을수록 테스트 효율성은 좋아질 것이다. 위의 5개의 프로그램에 대해 5번씩 반복하여 테스트 효율성의 평균과 100% ALP를 만족할 때까지 생성된 테스트 데이터 개수의 평균을 계산하였다.

Table 4. Experimental Results of Test Efficiency

program	trityp	sample	counter	enumeration	loop_assign
test efficiency (%)	11.2	3.2	3.6	8.3	4.1
# of tests	171	1565	802	435	556

5개의 프로그램 중에서 삼각형 분류 프로그램인 trityp가 가장 테스트 효율성이 좋았다. 이 프로그램은 플래그 문제가 없는 프로그램이다. enumeration도 결과가 그리 테스트 효율성이 다른 프로그램에 비해 좋은 결과를 보여준다. 이 프로그램은 trityp 프로그램과 함께 정적 분기와 동적 분기가 동일하다. 이 결과만 보면 정적 분기와 동적 분기가 같은 프로그램이 다른 프로그램에 비해 테스트 효율성이 좋다고 할 수 있다.

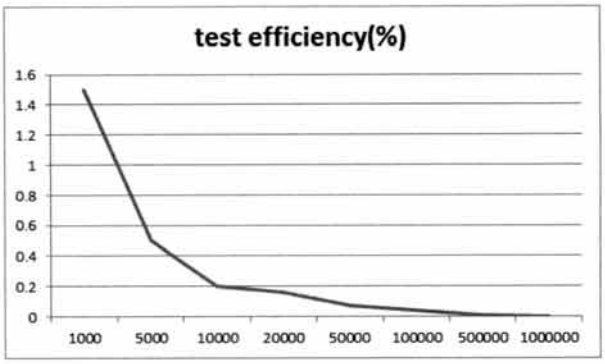


Fig. 5. Test efficiency of random testing for the Sample program with respect to the number of generated test data

동적 심볼릭 실행을 통해 제약식 해결 기법을 사용하는 콘콜릭 테스팅은 이 논문의 실험 대상 프로그램들에 대해서 100% 테스트 효율성을 보인다. 그 이유는 콘콜릭 테스팅은 이전에 실행된 프로그램 경로와는 다른 경로를 실행하기 위해 이전 경로를 표현한 심볼릭 표현식에서 하나를 선택하여 부정하여 새로운 경로 제약식을 생성하기 때문이다. 이렇게 생성된 경로 제약식을 만족하는 입력 데이터는 이전에는 실행하지 않았던 분기를 실행할 수 있는 테스트 데이터가 된다. 콘콜릭 테스팅과 비교했을 때 Table 4의 결과를 보면 테스트 효율성이 매우 좋지 않은 것처럼 보인다. 그러나 이는 반드시 사실이 아니다. Fig. 5는 생성한 테스트 데이터의 개수에 따른 sample 프로그램에 대해서 수행한 랜덤 테스팅의 테스트 효율성을 보여준다. 그래프에서 볼 수 있듯이 테스트 효율은 약 1000개의 생성된 테스트 데이터에 대해서는 1.5%로 출발하지만 생성된 테스트 데이터의 개수가 많아질수록 테스트 효율성은 떨어지는 것을 확인할 수 있다. 1,000,000개의 테스트 데이터 생성되어도 77%의 커버리지만 달성하며 이 때 거의 0에 가까운 테스트 효율성을 보임을 확인할 수 있었다.

5. 결론 및 향후 연구

논문에서 제안한 테스트 데이터 생성 방법 DyBrGen은 Korel의 경로 지향 테스트 데이터 생성 방법을 기반으로 하는 기존의 방법 'BrGen'을 동적 분기 개념을 이용하여 개선하였다. 기존의 방법에서는 분기 조건에서 사용되는 변수에 바인딩 되는 메모리 로케이션이 항상 동적으로 변경될 수 있는 상황을 고려하지 않았기 때문에 정교한 테스트 데이터 식별이 어려웠다. 제안된 방법에서는 실제 프로그램을 실행하여 동적 분기를 식별한 후에 가능한 많은 동적 분기를 실행하는 테스트 데이터를 생성한다. 기존의 방법에서는 정적인 분기에서만 분기 함수를 평가 하였으나 제안된 방법에서는 식별된 모든 동적 분기에서 분기 함수를 평가한다. 분기 함수 평가 결과에 따라 입력 값을 조정하여 아직 실행되지 않은 동적 분기를 실행되도록 하는 테스트 데이터를 생성한다. 이 때 사용되는 입력 값 조정 방법은 기존의 방법과 동

일하게 함수 최소화 기법에 바탕을 두고 있으며 어떠한 심볼릭 실행 기법도 이용되지 않는 순수한 동적 분석 방법이다. 따라서 심볼릭 실행 도구에 의존적인 문제 및 수행에 수반되는 비용을 절감할 수 있다.

또한 이 논문에서는 동적 분기 개념을 기반으로 다양한 동적 분기 커버리지 기준을 제안하였다. 이 기준들은 기존의 분기 커버리지에서는 식별되기 어려운 테스트 데이터를 식별하는 것을 가능하게 할 것으로 기대된다. 이러한 동적 분기 커버리지 기준들을 만족하기 위해서는 동적 분기들을 우선 식별하는 작업이 선행되어야 한다. 이 논문에서는 실제 프로그램의 실행을 통하여 식별하는 방법을 사용하였다. 그러나 현재 사용하는 방법은 분기 조건에서 사용되는 변수에 바인딩 가능한 모든 메모리 로케이션을 식별하는 것을 보장하지 못한다. 특히, 현재의 테스트 데이터 생성 도구가 정수형 및 정수형 배열만 지원이 가능하기 때문에 이를 구조체 및 포인터를 포함한 타입들도 지원이 가능하도록 확장하였을 때 동적 분기를 식별하는 문제에 대한 연구가 필요하다. 또한 동적 심볼릭 실행 기법을 사용하는 콘콜릭 테스팅에 비해 테스트 효율성이 매우 낮으므로 이를 개선할 수 있는 방법에 대한 연구가 필요하다.

참 고 문 헌

[1] J. Edvardsson, "A Survey on Automatic Test Data Generation", in Proceedings of the Second Conf. on Computer Science and Engineering, 1999, pp.21-28.

[2] J. Clarke, J. Dolado,, M. Harman,, R. Hierons, B. Jones, M. Lumkin, B. Mitchell, S. Mancoridis, K. Rees, M. Roper, and M. Shepperd, "Reformulating Software Engineering as a Search Problem", *IEE Proceedings-Software*, Vol.5, No.1, pp.161-175, 2003.

[3] J. Wegener, A. Baresel, and H. Sthamer, "Evolutionary Test Environment for Automatic Structural Testing", *Information and Software Test Technology*, Vol.43, No.14, pp.841-854, 2001.

[4] L. Bottaci, "Instrumenting Programs with Flag Variables for Test Data Search by Genetic Algorithm", in *Proc. of the Genetic and Evolutionary Computation Conf.(GECCO'02)*, pp. 1337-1342, NY, USA, July, 2002.

[5] M. Harman, R. Hu, R. Hierons, A. Baresel, and M. Sthamer, "Improving Evolutionary Testing by Flag Removal", *Information and Software Test Technology*, Vol.43, No.14, pp.841-854, 2001.

[6] M. Alshraideh, L. Bottachi, B. Mahafzah, Using program data-state scarcity to guide automatic test data generation, Software Quality Journal, Vol.18, No.1, pp.109-144, 2010.

[7] P. McMinn, "Evolutionary Search for Test Data in the Presence of State Behaviour", Ph.D. dissertation, The University of Sheffield, 2005.

[8] P. Godefroid, N. Klarlund, K. Sen, "DART: Directed automated random testing", in *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, Chicago, Illinois, 2005, pp.213-223.

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

[10] K. Lakhotia, P. McMinn and M. Harman, "An Empirical Investigation Into Branch Coverage for C Programs Using CUTE and AUSTIN", *Journal of Systems and Software*, Vol.83, No.12, pp.2379 - 2391, 2010.

[11] I. S. Chung, "Automated Test Data Generation Based on Branch Coverage for Testing C Programs", *Journal of the Korea Contents Association*, Vol.12, No.11, pp.39-48, 2012.

[12] B. Korel, "Automated Software Test Data Generation", *IEEE Trans. onSoftware Eng*, Vol.16. No.8. pp.870-879, 1990.

[13] Y. H. Kim, M. Z. Kim, and Y. K. Jang, "CREST-BV: An Improved Concolic Testing Technique with Bitwise Operations Support for Embedded Software", in *Proceedings of KCC*, 2012, pp.120-122.

[14] A. Baresel, and H. Sthamer, "Evolutionary Testing of Flag Conditions", in *Proc. of the Genetic and Evolutionary Computation Conf.(GECCO'03)*, Chicago, USA, July 2003, pp.2442-2454.

[15] A. Baresel, D. Binkley, M. Harman. and B. Korel, "Evolutionary Testing in the Presence of Loop Assigned Flags: A Testability TRansformation Approach", in *Proc. of the ACM SIGSOFT International Symp. on Software Testing and Analysis(ISSTA'04)*, Boston, USA, July 2004, pp.108-118.

[16] M. J. Gallagher, M. J., V. L. Narasimhan. "ADTEST: A Test Data Generation Suite for Ada Software Systems", *IEEE Trans. on Software Eng*, Vol.23, No.8, pp.473-484, 1997.

[17] http://www.pairwise.org.

[18] I. S. Chung. "HanCC: A Transformation and Instrumentation Tool for Automated Test Data Generation of C Programs", *Hansung University Engineering Research*, Vol.5, 2012.

정 인 상

e-mail : insang@hansung.ac.kr

1987년 서울대학교 컴퓨터공학과(학사)

1989년 한국과학기술원(KAIST) 전산학과 (석사)

1993년 한국과학기술원(KAIST) 전산학과 (박사)

1999년~현 재 한성대학교 컴퓨터공학과 교수

관심분야 : 소프트웨어 공학, 소프트웨어 테스팅