http://dx.doi.org/10.5392/JKCA.2012.12.11.039

C 프로그램을 테스팅하기 위한 분기 커버리지에 기반을 둔 자동 테스트 데이터 생성

Automated Test Data Generation Based on Branch Coverage for Testing C Programs

정인상
한성대학교 컴퓨터공학과

In-Sang Chung(insang@hansung.ac.kr)

요약

소프트웨어 테스팅이 소프트웨어 개발 비용의 상당 부분을 차지하는 것은 잘 알려진 사실이다. 소프트웨어 테스팅 비용을 줄이기 위해 소프트웨어 테스트 데이터를 자동으로 생성하는 방법에 많은 연구가 이루어지고 있다. 일반적으로 테스트 데이터 자동 생성을 지원하기 위해 심볼릭 실행기나 제약 해결기와 같은 정교한 도구들을 요구한다. 그러나 이와 같은 도구들을 개발하거나 구입하는 것은 소프트웨어 테스트 관련 비용을 증가시키는 또 다른 요소로 작용된다. 이 논문에서는 심볼릭 실행이나 제약 해결에 의존되지 않는 동적 테스트 데이터 방법을 제안한다. 제안된 방식은 분기 커버리지 기준을 효과적으로 만족하도록 Korel의 경로 지향 테스트 데이터 생성 방법을 확장한다. 이 논문에서는 삼각형 분류 프로그램에 대한 실험을 통하여 제안된 방법이 분기 커버리지를 매우 효과적으로 달성함을 보인다.

■ 중심어 : | 테스트 데이터 자동 생성 | 분기 커버리지 | 소프트웨어 테스트 |

Abstract

It is well known that software testing amounts for a significant portion of software development cost. In order to reduce the cost of software testing. a lot of researches on automated test data generation have been performed. Sophisticated tools for performing symbolic execution or solving a system of path constraints are required to support automated test data generation. Developing or purchasing those tools leads to another factor of increasing the cost involving software testing. In this paper, we propose a dynamic test data generation approach that does not depend on symbolic execution or constraint solving at all. The proposed approach extends Korel's path-oriented method to satisfy the branch coverage criterion effectively. We conducted an experiment to evaluate the effectiveness of the proposed technique with a triangle classification program to show that branch coverage can be easily achieved.

■ keyword : | Automated Test Data Generation | Branch Coverage | Software Testing |

I. 서론

소프트웨어 테스팅 비용이 소프트웨어 개발 비용의 50%이상 차지한다는 사실로부터 소프트웨어 테스팅 프로세스를 자동화하기 위해 많은 연구가 이루어지고 있다.

* 본 연구는 한성대학교 교내연구장려금 지원 과제임
접수번호 : #120904-004
접수일자 : 2012년 09월 04일
심사완료일 : 2012년 10월 11일
교신저자 : 정인상, e-mail : insang@hansung.ac.kr

지금까지 프로그램 테스트에 관한 연구는 효과적인 테스트 데이터를 선정하기 위하여 적합성 기준(adequacy criteria) 개발에 많은 노력을 했지만 상대적으로 테스트 데이터를 적합성 기준에 따라 자동으로 생성하는 기술에 관한 연구는 미진한 것이 사실이다.

이 논문에서는 C 프로그램을 대상으로 분기 커버리지를 만족하기 위한 테스트 데이터를 효과적으로 수행할 수 있는 방법을 제안한다.

지금까지 테스트 데이터를 생성하는 연구들은 다음과 같이 두 그룹으로 분류할 수 있다[1][2]:

- 경로 지향적 테스트 데이터 생성(path-oriented test data generation): 테스트할 프로그램 경로를 선정한 후에 이를 실행하는 입력 값을 식별하는 방법이다. 이 방법은 기존의 많은 방법들이 사용한 방법이다. 만약 분기 커버리지를 테스트 기준으로 선정하였다면 분기 커버리지를 만족하는 프로그램 경로들을 생성하고 각 생성된 프로그램 경로를 실행할 수 있는 테스트 데이터를 생성한다. 그러나 주어진 프로그램 경로가 실행이 불가능한 경우, 즉 경로를 실행할 수 있는 입력 값이 존재하지 않는 경우에는 입력 값을 찾기 위해 많은 시간과 노력이 소요될 수 있다.
- 목적 지향(goal-oriented) 테스트 데이터 생성: 특정 프로그램 경로를 제공하는 대신에 블록이나 분기와 같은 프로그램 상의 특정 프로그램 포인트를 주고 이를 실행할 수 있는 테스트 데이터를 생성하는 방법이다. 따라서 사용자가 일일이 프로그램 경로를 선정하는 부담이 없다. 경로 지향 방법에서는 사용자가 정한 프로그램 경로가 실행 불가능하다면(즉, 주어진 경로를 실행할 수 있는 입력 값이 존재하지 않는다면) 사용자가 다른 경로를 선택해야 하였으나 목적 기반 테스트에서는 주어진 프로그램 포인트(i.e., 블록 또는 분기)를 실행할 수 있는 다른 경로를 탐색하여 입력 값을 생성할 수 있다.

최근에 위 두 방법이외에 프로그램의 모든 경로들을 탐색하는 콘콜릭 테스팅(concolic testing)이 제안되었다[3][4]. 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.

이 논문에서는 C 프로그램을 테스트하기 위해 분기 커버리지 기준을 만족하는 테스트 데이터를 자동으로 생성하는 방법을 제안한다. 이를 위해 목적 코드를 테스트하기 위해 [6]에서 사용한 방법을 적용한다. [6]에서 제안한 방법은 경로 지향 테스트 데이터 생성을 위해 개발된 Korel의 방법[5]을 확장하였다.

Korel의 방법을 기반으로 하는 이유는 다음과 같다. 우선 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.

이 논문은 다음과 같이 구성된다. 2장에서는 테스트 데이터 생성을 위해 사용되는 방법에 대한 배경 설명을 한다. 3장에서는 Korel의 방법을 기반으로 분기 커버리지 기준을 만족하는 테스트 데이터를 생성하는 방법에 대해 설명한다. 4장에서는 이 논문에서 제안한 방법을 구현 기술에 대해 설명한다. 또한 삼각형 분류 프로그램에 대한 간단한 실험을 통하여 분기 커버리지를 매우 효과적으로 달성함을 보인다. 마지막으로 5장에서 결론 및 향후 연구에 대해 기술한다.

II. 배경

1장에서 완전한 프로그램 경로가 요구되는지에 따라 테스트 데이터 생성 방법을 경로 지향 방법과 목적 지

향 방법으로 분류하였다. 이 장에는 실제 테스트 데이터를 생성하기 위해 프로그램을 실행하느냐에 따라 다음과 같이 테스트 데이터 생성 방법들을 정적 테스트 데이터 생성, 동적 테스트 데이터 생성, 혼합(hybrid) 테스트 데이터 생성 방법으로 분류한다.

정적 방법은 테스트 데이터를 생성하기 위해 프로그램 실행을 요구하지 않는다. 대표적인 정적 기법으로 심볼릭 실행을 들 수 있다. 이 방법은 테스트하고자 하는 프로그램 경로에 대한 제약식을 추출하기 위해 프로그램을 어떤 특정한 값으로 실행하기보다는 모든 입력 도메인에 있는 값들을 대표할 수 있는 심볼릭 값을 사용하여 프로그램을 실행하는 방식이다. 과거의 많은 테스트 데이터 생성 방법은 주어진 적합성 기준을 만족하는 경로(들)에 대한 경로 조건(path condition)들을 추출하기 위해 심볼릭 실행을 이용하였다[1][7]. 그러나 심볼릭 실행 기법은 예를 들어 “x=TestGen[i+j]” 와 같이 배열의 원소가 간접적으로 참조될 때 변수 ‘i’와 'j' 가 특정한 값으로 바운드 되지 않기 때문에 실제 어느 배열의 원소를 참조하는지 알 수가 없다.

동적 테스트 데이터 생성은 실제 입력 값을 사용하여 프로그램을 실행하여 원하는 테스트 데이터를 생성한다. 최근에 제안된 많은 테스트 데이터 생성 방법들은 함수 최소화 기법에 기반을 두고 있으며 대표적인 동적 테스트 데이터 생성 기술이라 할 수 있다. 예를 들면 TESTGEN[5]이나 ADTEST[8]와 같은 테스팅 시스템은 특정한 입력 값을 사용하여 주어진 프로그램 경로에 따라 실제로 프로그램을 실행하는 방식을 취한다.

하이브리드 방식은 심볼릭 실행 기법을 프로그램 실행할 때 수행한다. 이런 이류로 이 방법이 동적 심볼릭 실행 기법이라고 불린다. 이 방식은 프로그램 실행 정보를 이용하여 심볼릭 실행을 하기 때문에 정적 심볼릭 실행과는 달리 배열 참조가 정확하게 어디에서 이루어졌는지를 알 수 가 있다.

대표적인 동적 심볼릭 실행은 콘콜릭 테스팅 방법[3][4]이다. 콘콜릭 테스팅 방법은 이 논문에서와 같이 특정 프로그램 경로나 목표 블록(또는 분기) 대신에 프로그램의 모든 분기들을 탐색한다. 최근에는 콘콜릭 테스팅을 특정 목표에 제한하는 목적 지향 콘콜릭 테스팅 방법이 제안되기도 하였다[9].

III. 테스트 데이터 생성

이 장에서는 분기 커버리지를 만족하는 테스트 데이터를 생성하는 방법에 대해 기술한다. 3.1절에서는 이 논문에서 기반으로 하고 있는 Korel의 방법에 대해 기술한다. 3.2절에서는 Korel의 방법을 분기 커버리지를 만족하는 테스트데이터를 생성하기 위해 확장한다. 지금부터 제안된 테스트 데이터 생성 방법을 ‘BrGen’이라 명명한다. 3.3절에서는 제안된 방법을 예를 들어 설명한다.

1. Korel의 테스트 데이터 생성 방법

일반적으로 정적 방식과 하이브리드 방식은 심볼릭 실행과 제약식 해결을 위한 과정이 관여되기 때문에 이를 위한 도구들의 개발이나 구입이 선행되어야 한다. 이에 반해 동적 방식에서 주로 이용하고 있는 함수 최소화 기법은 이러한 도구들이 요구되지 않고 간단한 입력 값의 조정으로 테스트 데이터를 생성할 수 있다. 이러한 대표적인 방법으로 Korel의 방법[5]이 있다.

Korel의 방법은 실제 프로그램을 주어진 경로에 따라 실행하여 테스트 데이터를 탐색하는 경로 지향 방법이다. 예를 들어 입력 값이 주어진 경로와 다른 경로를 실행하는 경우 입력 값을 조정하여 원하는 방향으로 실행할 수 있도록 유도한다.

Korel의 방법에서는 원하는 테스트 데이터를 찾기 위해 우선 랜덤하게 생성된 초기 입력 데이터로 주어진 경로를 따라 프로그램을 실행한다. 만약 프로그램 실행 도중에 분기 조건문을 만난 경우에 현재의 입력 값이 주어진 프로그램 경로를 따라서 적절한 분기가 일어난다면 입력 값을 변경하지 않는다. 만약 주어진 프로그램 경로와는 다른 분기가 일어난다면 (즉, 실제 프로그램 실행 경로와 주어진 프로그램 경로가 다른 경우) 실행의 흐름을 바꾸도록 현재 분기문의 분기 함수에 대해 함수 최소화 기법을 이용하여 현재의 입력 데이터를 수정한다. 만약 이러한 과정을 거치고도 적절한 입력 데

이터를 찾지 못한다면 프로그램 경로 상에서 현재 분기 조건문에 바로 앞서 실행된 조건문으로 되돌아가 다른 입력 값을 찾는 과정을 되풀이해야 한다.

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.

2. 분기 커버리지 기반 테스트 데이터 생성

이 논문의 목적은 프로그램의 실행 가능한 분기를 가능한 많이 실행하게 하는 테스트 데이터를 생성하는 것이다. 이를 위해 [6]에서 제안한 실행가능한 목적 코드에 대해 테스트 데이터를 생성하는 방법을 C 프로그램에 대해 분기 커버리지를 달성하도록 적용하였다.

[6]에서는 목적 코드를 대상으로 하였기 때문에 'cmp'와 같은 비교 연산 명령어의 결과가 다양하게 설정될 수 있도록 분기 함수들을 cmp 명령어가 수행될 때 평가하도록 하였으나 이 논문에서는 C 프로그램을 대상으로 하기 때문에 각 분기 조건 문 "a op b"이 실행될 때 아직 실행되지 않은 분기에 대한 분기 함수 F rel 0을 추출하여 이를 최소화 하는 입력 데이터를 생성한다. [표 1]은 분기 함수 F와 rel을 보여준다.

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 이 논문에서는 모든 분기에 대한 분기 함수를 추출하여 이를 최소화 하는 입력 데이터를 식별한다.

표 1. 분기 함수

F	rel
B-A	〈
B-A	≤
A-B	〈
A-B	≤
abs(A-B)	=
-abs(A-B)	≠

예를 들면, "if x>5"의 분기 조건에 대해 다음과 같은 두 개의 분기 함수를 생성한다: (1) $5-x<0$ (2) $x-5\leq0$. (1) 분기 함수를 최소화하는 테스트 데이터는 참인 분기를 실행하고 (2) 분기 함수를 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 ($rel \in \{<. \leq, =\}$)
- 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 이 논문에서는 특정 프로그램 경로를 실행해야 하는 제약조건 대신에 목표 분기의 시작 블록을 실행해야 하는 제약조건만이 있다. 즉, 테스트 데이터가 어떤 경로를 실행해도 상관없다는 의미이다.

[그림 1]은 분기 함수 최소화 방법을 이용하여 가능한 많은 분기들을 실행할 수 있는 테스트 데이터를 생성하는 프로시저를 보여준다.

```
Initialize v, Stack;

Generate random value v=(v1, v2, ..., vn)
for input variable x=(x1, x2, ..., xn);

execute(P, v, Stack);

while (Stack<>∅) do {
   _id = pop() from Stack;
   if (t=minimize(_id, v) then
       print( "found test data" , t);
}
```

그림 1. 테스트 데이터 생성 프로시듀어

이 프로시저는 기본적으로 깊이 우선 탐색(depth first search)에 바탕을 두고 있으며 'execute' 함수와 'minimize' 함수를 사용하여 테스트 데이터를 생성한다.

- execute(P, v, Stack): 프로그램 P를 입력 v로 실행한다. 이 때 분기가 실행되는 경우에 관련된 분기함수들을 평가한 후에 분기 함수 평가 결과를 갱신한다. 아직 거짓으로 평가된 분기 함수가 남아있는 경우에만 해당 분기를 스택 Stack에 넣는다.
- minimize(id, v):이 함수는 분기 조건 id에서 아직 실행 안 된 분기에 대한 분기 함수 F를 최소화하는 입력 값을 v를 기반으로 생성한다. 성공적으로 생성되었다면 이 함수는 해당 분기를 실행하는 테스트 데이터를 반환한다. minimize 함수는 'execute' 함수를 이용하여 스택 정보를 갱신한다.

우선 필요한 변수 및 자료 구조들을 초기화 한다. Stack은 앞으로 실행되어야 할 분기 정보를 가지고 있는 스택 자료 구조이다. 임의의 입력 값 v를 생성한 후에 대상 프로그램 P를 실행한다. 이 때 실행되는 분기 조건들을 스택에 저장한 후에 분기 함수에 대한 평가를 수행한다.

이 논문에서는 분기 함수의 평가를 위한 정보를 획득하기 위해 'HanTestCC' 도구를 이용하여 분기 조건 'if (x op y)' 실행에 앞서 다음 함수를 호출하도록 탐침 한다[10]

HanTestGen(id, op_id, x, y);

여기에서 'id'는 대상 분기 조건을 나타내는 고유 식별번호이며 'op_id'는 op에 해당하는 관계 연산자 식별번호이다. 4장에서 HanTestCC가 제공하는 기능들에 대한 자세하게 설명한다.

```
void HanTestGen(int id, int op_id, long x, long y) {
 improved=false;
 executed = false;
if (not visited[id]) {
     visited[id]=true;
     push(id);
}
if (not finished[id]) {
   dist = x-y;
   case op_code {
   '>' :
       if (dist <0) {
          br[id].fbr=1;
          dist = y-x;
       }
       else br[id].tbr=1;

   case op_id {
   '<' :
      if (dist < 0) {
           br[id].tbr=1;
           dist = y-x;
      }
      else br[id].fbr=1;
   ...
   end case

   if (imp[id]>dist) {
     imp[id]=dist;
     if (i == target_br) {
              improved=true;
              executed = true;
          }
   }
   finished[id] = br[id].tbr && br[id].fbr;
   }
```

그림 2. 탐침 함수 HanTestGen

[그림 2]는 탐침 함수 'HanTestGen'의 구현 내역을 보여준다. 이 탐침 함수는 분기 id가 실행될 때 다음 기능을 수행 한다.

- 해당 분기가 처음 방문된 분기라면 이를 스택에 저장한다.
- 관계 연산자에 따른 분기 함수를 평가하여 개선된 경우에 한하여 분기 함수 평가 결과를 distance[_id]에 저장한다.
- 해당 분기가 목표 분기이고 실행되었다면 'executed'를 참으로 설정한다.
- 해당 분기가 목표 분기이고 분기 함수가 개선되었다면 'improved'를 참으로 설정한다.

```
int[] minimize(int _id, int[]  v) {
     int t[NumofInputs];
     for each input variable inv do {
       delta = exploratoryMove(inv, v)
       t = patternMove(_id, inv, v, delta, 1);
       if (finished[_id]) return t;
      }
     return null;
}
int exploratoryMove(inv, v) {
     execute(P, v/inv+SmallDelta);
     if (improved) return Delta;
     else {
            execute(P, v/inv-SmallDelta);
            if (improved) return -Delta;
      }
     return 0;
}
int[] patternMove(int _id, int inv, int [] v, int delta, int cnt)
{
   if (delta==0) return null;
   execute(P, v/(inv+cnt*delta));
   if (finished[_id]) return v/(inv+cnt*delta);
   else if (improved) {
       delta = delta + cnt*delta;
     return patternMove(inv, v/inv, delta, cnt+1);
    }
    else {
      delta = exploratoryMove(inv, v);
      return patternMove(_id, inv, v, delta, 1.0);
   }
  }
```

그림 3. 분기 함수 최소화 : ‘minimize’ 함수

[그림 3]의 ‘minimize()’ 함수는 탐침 함수 ‘HanTestGen’의해 생성된 정보들을 이용하여 분기 함수 F를 최소화하는 입력 값 v를 생성한다. [그림 3]에서 볼 수 있듯이 ‘minimize()’ 함수는 각 입력 변수에 대해 탐색이동(exploratoryMove)과 패턴이동(patternMove)을 반복한다. [그림 3]의 exploratoryMove()와 patternMove() 함수에서 ‘v/inv+(-)k’는 입력 벡터 v에서 입력변수 inv에 해당하는 값을 k만큼 증가(감소)하였다는 의미이다.

함수 exploratoryMove()는 현재 입력값의 해당 입력 변수의 값을 SmallDelta 만큼 증가 또는 감소하여 탐색이 어느 방향으로 진행되어야 하는지 검사한다. 이 논문에서는 상수 SmallDelta를 1로 설정하였다. 만약 증가하는 방향으로 탐색하여야 한다면 양수 값(Delta)을 반환하고 감소하는 방향으로 탐색이 진행되어야 한다면 음수 값(-Delta)을 반환한다. 만약 해당 입력 변수가 목표 분기 실행에 전혀 영향을 주지 못한다면 0을 반환하도록 하였다.

함수 patternMove()는 목표 분기를 실행하는 해당 입력 변수 값을 식별할 때 까지 반복한다. 분기 함수의 값이 개선될수록 이와 비례하여 동일 방향으로 더 많은 이동이 이루어진다. 그러나 분기 함수의 값이 더 이상 개선되지 않는다면 이전 분기 함수의 값을 유지한 상태에서 탐색 이동을 다시 수행하게 된다.

이러한 과정을 목표 분기를 수행하는 테스트 데이터(즉, 분기 함수의 값이 음수이거나 0이 될 때까지)를 식별하거나 모든 입력 변수에 대해 분기 함수의 값이 더 이상 어떤 개선이 없을 때 까지 반복한다.

3. 예제

이 절에서는 앞 절에서 기술한 Korel 방법을 분기 커버리지를 달성하기 위해 확장한 방법 ‘BrGen’에 대해 구체적으로 기술한다. 이를 위해 우선 [그림 4]에 있는 MID 함수를 대상으로 BrGen을 설명한다. MID 함수는 세정수를 입력으로 받아들여 중간 값을 출력한다. [그림 5]는 [그림 4]의 프로그램에 대한 제어흐름 그래프를 보여준다.

BrGen은 우선 초기 무작위로 값을 생성하여 프로그래a을 실행한다. 만약 (x:0, y:0, z:0)이 생성되어 실행되었다고 가정할 때 경로 (n0, n1, n2, n4, n10, n11)가 실행됨을 알 수 있다. 이 때 스택은 [그림 6](a)와 같다.

따라서 스택 탑에 있는 ‘x<y’가 참이 되도록 입력 값을 조정하여야 한다. 일반적으로 ‘x<y’와 같은 분기 술어(branch predicate)가 ’a op b’가 표현되었다고 가정하자. 여기에서 a와 b는 산술식이고 op는 관계연산자이다. 이 때 이 분기술어로부터 [표 1]에 따라 ‘F rel 0’ 형태의 목적 함수를 도출할 수 있다.

```
int MID(int x, int y, int z)
{
   int midVal;
   midVal=z;
   if (y<z) {
           if (x<y) midVal = y;
           else if (x<z) midVal = x;
}
     else {
           if (x>=y)      midVal = y;
           else if (x>z) midVal = x;
}
            return midVal;
  }
```

그림 4. 예제 프로그램

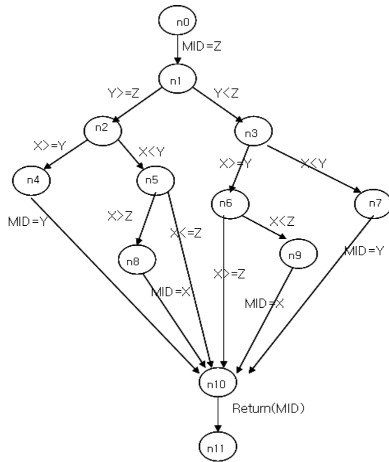

그림 5. MID 프로그램의 제어 흐름 그래프

분기 함수라 불리는 F는 함수 최소화 대상이 되며 분기 술어가 거짓일 때 양수 값(또는 rel이 '<'이라면 0)을 갖고 참일 때 음수 값(또는 rel이 '≤'이거나 '='이라면 0)을 갖는다. 현재 n1에서 참인 분기 'x<y'에 대해 'x-y<0'인 목적 함수가 도출된다. 입력 (x:0, y:0, z:0)에 대한 이 함수의 값은 0-0=0이다.

Korel의 탐색 이동 단계에서 x의 값을 증가하거나 감소하여 분기 함수의 값을 살펴본다. 이 논문에서는 탐색 단계에서 1만큼 각 입력 변수를 증가하거나 감소한다. 입력 x의 값을 증가시키면 분기 함수의 값이 더 나빠지지만[그림 6](b) 증가시키면 좋아질 뿐만 아니라 분기 술어가 만족됨을 알 수 있다[그림 6](c). 이렇게 구해진 입력 (x:-1, y:0, z:0)은 경로 (n0, n1, n2, n5, n10, n11)가 실행되고 [그림 6](c)의 스택이 구해진다. 이 경우 n2의 모든 분기들이 실행되었기 때문에 n2가 스택에서 제거되었음을 볼 수 있다.

스택 탑에 있는 분기 술어 'x>z'에 해당하는 목적함수는 'z-x<0'에 대해 현재 분기 함수의 값은 0-(-1)=1이다. x의 값을 감소하면 분기 함수의 값은 더 나쁘게 되지만 x의 값을 증가하면 경로의 변화 없이 분기 함수의 값이 개선됨[그림 6](d)을 알 수 있다. 따라서 변수 x에 대해 증가하는 방향으로 'x>z'가 만족될 때까지 패턴 이동을 수행 한다 이렇게 구해진 입력은 (x:5, y:0, z:0)[그림 6](e)이다. n5의 모든 분기들이 수행되었기 때문에 n5가 [그림 6](e)에서 볼 수 있듯이 제거되었음을 알 수 있다.

(a)	(b)	(c)
n2(x<y) n1(y<z) (0,0,0)	n2(x<y) n1(y<z) (1,0,0)	n5(x>z) n1(y<z) (-1,0,0)

(d)	(e)	(f)
n5(x>z) n1(y<z) (0,0,0)	n1(y<z) (5,0,0)	n1(y<z) (6,0,0)

(g)	(h)	(i)
n1(y<z) (4,0,0)	n1(y<z) (5,1,0)	n6(x<z) n3(x<y) (5,-1,0)

그림 6. 스택의 변화

현재 스택 탑에 있는 분기 술어 'y<z'에 해당하는 목적함수는 'y-z<0'에 대해 입력 변수 x에 대한 탐색이동은 아무런 영향이 없음을 알 수 있다[그림 6](f)와 [그림 6](g). 따라서 입력 변수 y에 대해 탐색이동을 수행하여야 한다. y의 값을 증가하는 경우에는 분기 함수의 값이 개선되지 않지만[그림 6](h) 감소하는 경우에 개선됨과 동시에 분기 술어가 참이 된다[그림 6](i). 이렇게 식별된 입력 값 (x:5, y:-1, z:0)에 대해 프로그램을 실해하면 경로 (n0, n1, n3, n6, n10, n11)가 실행되며 스택은 [그림 6](i)와 같이 갱신된다. 이와 같은 과정을 반복하면 모든 분기 커버리지를 만족하는 테스트 데이터 집합을 생성할 수 있다.

IV. 구현 및 평가

이 논문에서 제안된 테스트 데이터 생성 방법은 C 프로그램을 대상으로 한다. 그러나 C 언어로 작성된 프로그램은 C 언어의 여러 특성 때문에 프로그램 분석 도구를 개발하기가 용이하지 않다. 이런 이유로 C 프로그램을 대상으로 테스트 데이터 자동 생성을 지원하기 위한 도구 HanTestCC를 개발하였다[10].

HanTestCC는 CIL[11]의 하나의 OCaml 모듈로 개발되었다. CIL은 버클리 대학에서 개발된 C프로그램 변환 및 분석 도구이다. C 언어는 저수준의 레벨에서 프

로그래밍 하는 유연성도 제공하는 반면에 사람이 이해하거나 자동화된 도구에 의해 분석하기 어렵다는 것도 잘 알려진 사실이다.

CIL은 C 프로그램을 분석하고 변환하기 용이하게 하기 위해 개발된 일종의 중간 코드 형식을 갖는다. CIL에서 제공하는 프로그램 구조물(construct)은 매우 적은 수의 구조물만 가지고 있으며 C의 복잡한 구조를 훨씬 단순하게 표현하게 해준다. 반면에 어느 정도의 추상성을 지니고 있어 3 주소 코드(3-address code)와 같은 중간 코드 형식보다는 원래의 C 프로그램 구조를 많이 훼손하지 않는다. 이와 같은 CIL의 특성은 프로그램 분석할 때 고려해야 하는 경우의 수를 줄여 프로그램 분석 및 변환 작업을 매우 효과적으로 수행할 수 있게 한다.

HanTestCC기 테스트 데이터를 생성을 위해 다음과 같은 기능들을 제공한다:

- 인터페이스 추출 기능: 테스트 대상 함수의 인터페이스 정보를 바탕으로 생성해야 하는 테스트 데이터의 타입 정보 및 입력 인자의 개수 등을 식별할 수 있다. 개수 및 타입 정보를 획득할 수 있다. 이 인터페이스 정보로부터 테스트 드라이버를 자동으로 생성한다.
- 테스트 데이터 실행 정보 획득 기능: HanTestCC는 이를 위해 프로그램에 탐침 작업을 수행한다. 우선 탐침 작업을 용이하게 하기 위해 테스트 대상 프로그램을 매우 간단한 구조로만 이루어진 프로그램으로 변환한다. 현재 HanTestCC에서는 'while(1), if (condition) {...} else{...}, 'goto' 등으로만 이루어진 함수로 변환하고 프로그램의 모든 분기 조건식에서 'AND'나 'OR' 논리 연산자를 제거한다.

```
#define NOT_A_TRIANGLE 0
#define SCALENE 1
#define ISOSCELES 2
#define EQUILATERAL 3
int tri_type(int a, int b, int c) {
  int type;
  if(a > b){
          int t = a;
          a = b;
          b = t;
  }
  if(a > c){
          int t = a;
          a = c;
          c = t;
  }
  if(b > c){
          int t = b;
          b = c;
          c = t;
  }
  if(a+b<= c){
          type = NOT_A_TRIANGLE;
  } else {
          type = SCALENE;
          if(a == b && b == c){
            type = EQUILATERAL;
          } else
          if (a == b || b == c){
                  type = ISOSCELES;
              }
  }
  return type;
}
```

그림 7. 삼각형 분류 프로그램

예를 들면 [그림 7]의 삼각형 분류 프로그램의 정삼각형이나 이등변 삼각형으로 결정하는 부분 즉, "if(a == b && b == c){ type = EQUILATERAL; }else if(a == b || b == c){ type = ISOSCELES;}"이 [그림 8]과 같이 변환 된다:

```
if (a == b) {
     {
        if (b == c) {
      type = 3;
     } else {
      goto _L;
     }
     }
   } else {
     _L: /* CIL Label */
     {
        if (a == b) {
      type = 2;
     } else {
      {
        if (b == c) {
         type = 2;
      } else { ...
```

그림 8. 변환된 조건문

또한 HanTestCC는 'if (!cond))...'와 같이 not 연산이 있는 경우에는 '!' 연산을 제거하고 조건식의 의미가 보존되도록 cond를 부정한다. 예를 들어'if !(x >y)' 경우에는 'if (x<=y)'로 변환한다.

HanTestCC는 이와 같이 변환된 프로그램을 바탕으로 테스트 데이터 실행 정보를 획득하기 위해 3.2절에서 설명하였듯이 탐침을 수행한다. 탐침 함수 HanTestGen의 구현 내역은 테스트 데이터 생성 알고리즘에 따라 달라질 것이다.

- 제어 흐름 그래프 생성 기능: HanTestCC는 테스트 대상이 되는 함수의 제어 흐름에 관한 정보를 생성한다. 이 정보를 바탕으로 실제 테스트 데이터 생성 도구에서는 테스트 대상 함수의 제어 흐름 그래프를 생성할 수 있다. 더 나아가 이를 바탕으로 생성된 테스트 데이터가 실행하는 프로그램 경로를 시각적으로 확인 할 수 있도록 해줄 수 있도록 한다.

제안된 방법의 타당성 측정을 위해 테스트 벤치 마크 프로그램으로 많이 사용되는 [그림 6]의 삼각형 분류 프로그램(Tri)과 특정일 사이의 날짜수를 계산하는 프로그램(Days)에 대해 달성된 분기 커버리지를 측정하여 랜덤 테스팅과 비교하였다. Tri는 분기의 개수가 12개이고 Days는 분기의 개수가 36개인 매우 복잡한 프로그램이다. 가능한 정확한 결과를 얻기 위해 10번 반복하여 테스트 데이터를 생성하였으며 랜덤 테스팅은 분기 커버리지가 더 이상 향상이 없을 때 테스트 데이터 생성을 종료하도록 하였다. 이 실험은 CPU (2.66GHz), RAM 4GB, 윈도우 XP를 탑재한 PC에서 수행되었다.

표 2. 실험 결과

(a) Tri 프로그램

Tri	Cov.(%)	테스트 수
제안된방법	100	181
Random	81	201

(b) Days 프로그램

Days	Cov.(%)	테스트 수
제안된방법	80	73
Random	50	7

[표 2]는 실험 결과를 보여준다. 실험해 본 결과 이 논문에서 제안된 방법은 커버리지 측면에서 Tri 프로그램에 대해 10번 반복에 대해서도 100% 분기 커버리지를 달성하는 놀라운 결과를 보여 주었다. 또한 랜덤 테스팅 결과 대비 약 24%((100-81)/81*100)의 향상된 결과를 보여준다. Days 프로그램에 대해서도 랜덤 테스팅 대비 60% 향상되었음을 알 수 있다. Days 프로그램에서 생성된 랜덤 테스트 데이터 개수가 7개로 작은 이유는 아주 빠르게 분기 커버리지가 달성되고 더 이상 커버리지 향상이 없기 때문이다.

V. 결론 및 향후 연구

이 논문에서는 가능한 많은 분기를 실행하는 테스트 데이터를 생성하기 위하여 분기 함수들의 평가 결과에 따라 입력 값을 조정하여 아직 실행되지 않은 분기를 실행되도록 하는 테스트 데이터를 생성한다. 이 때 사용되는 입력 값 조정 방법은 함수 최소화 기법에 바탕을 두고 있으며 어떠한 심볼릭 실행 기법도 이용되지 않는 순수한 동적 분석 방법이다. 따라서 심볼릭 실행 도구에 의존적인 문제[12] 및 수행에 수반되는 비용을 절감할 수 있다.

이 연구에서 개발된 테스트 데이터 생성 전략은 깊이 우선 탐색에 기반을 둔 방법이다. 즉, 이전에 실행되는 경로 중에서 가장 나중에 나온 미실행 분기를 최우선으로 선정하여 테스트 데이터를 생성한다. 보다 심도 있는 테스트 데이터 생성 전략을 식별하기 위해 다양한 방법으로 테스트 데이터를 생성하여 비교할 필요가 있다. 예를 들면 실행할 분기를 무작위로 선정하여 해당 분기 함수를 최소화하는 테스트 데이터를 생성하는 랜덤 생성 방법이나 이전에 실행된 경로에서 가장 먼저 방문되는 미실행 분기를 최우선으로 선정하여 테스트 데이터를 생성하는 방법 등과 같은 여러 테스트 데이터 생성 전략을 고려해 볼 수 있다.

현재의 테스트 데이터 생성 도구가 기반으로 하고 있는 HanTestCC는 정수형 및 정수형 배열만 지원이 가능하기 때문에 이를 구조체 및 포인터를 포함한 타입들도 지원이 가능하도록 확장할 필요가 있다.

참 고 문 헌

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

[2] P. McMinn, "Search-based Software Test Data Generation: A Surveym," Software Testing, Verification and Reliability, Vol.14, No.2, pp.105-156, 2004.

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

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

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

[6] 정인상, "실행가능 목적 코드를 기반으로 하는 자동 테스트 데이터 생성," 한국인터넷방송통신학회 논문지, 제12권, 제2호, pp.189-197, 2012.

[7] L. A. Clarke, "A System to Generate Test Data and Symbolically Execute Program," IEEE Trans. on Software Eng, Vol.2, No.3, pp.215-222, 1976.

[8] M. J. Gallagher and 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.

[9] 정인상, 박정규, "목적 지향 콘콜릭 테스팅", 정보과학회논문지, 제37권, 제10호, pp.768-772, 2010.

[10] 정인상, "HanTestCC: C 프로그램의 자동 테스트 데이터 생성을 위한 프로그램 변환 및 탐침 도구", 한성대학교 공학연구 논문집, 5월호, 2012.

[11] http://cil.sourceforge.net/

[12] 김윤호, 김문주, 장윤규, "CREST-BV: 임베디드 소프트웨어를 위한 Bitwise 연산을 지원하는 Concolic 테스팅 기법", 한국정보과학회 2012한국컴퓨터종합학술대회 논문집.

저 자 소 개

정 인 상(In-Sang Chung) 정회원

• 1987년 : 서울대학교 컴퓨터공학과 졸업(학사)
• 1989년 : 한국과학기술원(KAIST) 전산학과 졸업(석사)
• 1993년 : 한국과학기술원(KAIST) 전산학과 졸업(박사)
• 1999년 ~ 현재 : 한성대학교 컴퓨터공학과 교수

<관심분야> : 소프트웨어 공학, 소프트웨어 테스팅