http://dx.doi.org/10.7236/JIWIT.2012.12.2.189
JIWIT 2012-2-24

실행가능 목적 코드를 기반으로 하는 자동 테스트 데이터 생성

Automated Test Data Generation based on Executable Object Codes

정인상*

Insang Chung

요 약 고수준의 명세나 고수준의 프로그래밍 언어로 작성된 원시 코드를 이용하여 테스트 데이터를 생성하는 것이 일반적이다. 그러나 어떤 상황에서는 이러한 테스트 데이터 생성 정보가 항상 이용가능하지 않을 수 있다. 이 논문에서는 실행가능 목적코드를 바탕으로 테스트 데이터를 생성하는 방법을 제안한다. 제안된 방법은 정교한 목적 코드 분석을 필요로 하지 않은 매우 간단한 함수 최소화 기법을 사용하여 동적으로 테스트 데이터를 생성한다. 삼각형 분류 프로그램에 대한 실험을 통하여 분기 커버리지를 매우 효과적으로 달성함을 보인다.

Abstract It is usual for test data generation to be performed using either high-level specifications or source codes written in high-level programming languages. In certain circumstances, however, such information is not always available. This paper presents a technique that generates test data based on executable object codes. The proposed technique makes use of a very simple function minimization technique without sophisticated object code analysis and produces test data dynamically. We have conducted a simple experiment to evaluate the effectiveness of the proposed test data generation technique with a triangle classification program to show that branch coverage can be easily achieved.

Key Words : 테스트 데이터 자동 생성, 실행 가능 목적 코드, 분기 커버리지, 함수 최소화 기법

Ⅰ. 서 론

프로그램 테스팅은 프로그램의 품질을 제고하기 위한 일반적인 방법으로 사용하고 있지만 많은 시간과 자원을 소모하는 단점이 있다. 소프트웨어 테스트 비용을 줄이기 위해서 테스트 데이터를 자동으로 생성하는 것이 효과적이다.

테스트 데이터 생성 방법들은 사용자가 테스트할 프로그램 경로를 제공하는지에 따라 경로 지향(path-oriented) 방법과 목적 지향 방법(goal-oriented)으로 분류할 수 있다. 경로지향 방법은 프로그램의 제어흐름이 복잡하거나 반복문이 존재하는 경우에 경로를 선정하는 작업 자체가 용이하지 않을 뿐만 아니라 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 있다는 단점이 있다[1][2].

*정회원, 한성대학교 컴퓨터공학과
접수일자 2012.1.2, 수정일자 2012.2.27.
게재확정일자 2012.4.13

Received: 2 January 2012, Revised: 27 February 2012
Accepted: 13 April 2012
*Corresponding Author: insang@hansung.ac.kr
Dept. of Computer Engineering, Hansung University, Korea

이와는 달리 목적 지향(goal-oriented) 테스트 데이터 생성 방법은 특정 프로그램 경로를 제공하는 대신에 프로그램 상의 한 블록 또는 분기를 주고 이를 실행할 수 있는 테스트 데이터를 생성하는 방법이다. 따라서 사용자가 일일이 프로그램 경로를 선정하는 부담이 없으며 경로 기반 테스트에서는 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 자동으로 탐색하여 입력 값을 생성할 수 있다.

최근에는 특정 프로그램 경로나 포인트 대신에 프로그램의 모든 경로들을 탐색하는 콘콜릭 테스트가 제안되었다[3][4]. 이 방법은 동적 테스트 방법과 심볼릭 수행을 결합하여 높은 테스트 커버리지를 달성하기 위해 개발되었다. 콘콜릭 테스트는 우선 무작위로 생성된 입력으로 프로그램을 수행한다. 이 때 입력에 의해 실행된 경로를 따라 심볼릭 수행을 하여 프로그램 경로 제약 조건을 생성한다. 이렇게 생성된 경로 제약 조건은 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.

이러한 테스트 생성 방법들은 모두 기본적으로 프로그램의 원시 코드 정보가 이용가능하다는 사실을 전제로 하고 있다. 예를 들어 테스트를 수행할 때 소스 코드를 이용할 수 없는 경우는 벤더가 기술 보안상의 이유로 소스 코드를 제공하지 않을 수도 있으며 또한 부주의로 인해 소스 코드가 분실되는 경우도 있을 수 있다. 이러한 상황은 오류가 있는 컴파일러로 개발되는 임베디드 시스템 개발에서 전형적으로 발견된다. 뿐만 아니라 소스 코드가 있다할지라도 컴파일 과정 자체가 신뢰가 없을 경우에는 소스 코드 분석 결과를 테스트 과정에 이용할 때 부정확한 결과가 초래될 여지가 있다[5].

이와 같이 소스코드를 이용할 수 없는 상황에서는 실행 가능한 목적 코드를 바탕으로 테스트 데이터를 생성해야 한다. 이를 위해서는 매우 정교하고 복잡한 목적 코드 분석을 통하여 원하는 테스트 정보를 추출하여야 한다. 이 논문에서는 정교한 목적 코드 분석을 필요로 하지 않으면서도 효과적인 테스트 데이터 생성 방법을 소개한다.

제안된 방법은 주어진 경로를 실행하기 위한 테스트 데이터를 생성하기 위해 경로 제약 조건을 생성하여 해결하는 기존의 전통적인 방식 대신에 매우 간단한 함수 최소화 기법을 사용하여 동적으로 테스트 데이터를 생성한다.

본 논문은 다음과 같이 구성된다. 2장에서는 기존의 테스트 데이터를 자동으로 생성하는 방법들에 대해 간략하게 소개한다. 3장에서는 본 논문에서 제안한 방법에 대해 기술한다. 4장에서 다른 테스트 생성 방법들고 비교한 후에. 간단한 예제 프로그램을 사용한 실험 결과를 기술한다. 마지막으로 5장에서 결론 및 향후 연구에 대해 언급한다.

II. 관련 연구

이 장에서는 테스트 데이터를 자동으로 생성하는 기술들을 정적 기법과 동적 기법으로 분류한다. 대표적인 정적 기법으로 심볼릭 수행(symbolic execution)을 들 수 있다[6]. 심볼릭 수행 기법은 테스트하고자 하는 프로그램 경로에 대한 제약식을 추출해기 위해 실제 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 수행을 이용하였다.

그러나 심볼릭 수행 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 수 있는 방법이 없다. 또한 포인터를 사용하는 경우에도 동일한 기억 장소를 다른 변수 이름을 이용하여 접근하는 경우(aliasing problem)에도 포인터 변수가 실제 어떤 기억장소를 가리키고 있는지에 대한 정보가 부정확할 수 있다

최근에 제안된 많은 테스트 데이터 생성 방법들은 함수 최소화 기법에 기반을 두고 있다. 함수 최소화 기법은 실제 입력 값을 사용하여 프로그램을 실행하는 대표적인 동적 기반 테스트 데이터 생성 기술이다. 예를 들면 TESTGEN[7]이나 ADTEST[8]와 같은 테스팅 시스템은 특정한 입력 값을 사용하여 주어진 프로그램 경로에 따

라 실제로 프로그램을 실행하는 방식을 취한다.

이러한 시스템에서는 프로그램의 경로 상에 있는 분기 조건문을 함수로 간주한다. 예를 들면, 프로그램에서 분기 조건 x>=10이 참이 되게 하는 테스트 데이터를 구하는 문제를 생각해보자. 이와 같은 분기 조건문은 'F(x) = 10-x if x<10, 0 otherwise'와 같은 함수로 생각할 수 있다. 이때 x를 문장 2번에 도달할 때의 변수 x에 저장되어 있는 값이라고 가정하면 함수 F(x)를 최소화하는 입력 데이터 x는 분기 조건 x>=10을 참이 되게 하는 입력 데이터 값이 된다.

이러한 입력 값을 찾기 위해 TESTGEN과 같은 시스템에서는 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. 그러나, 만약 주어진 프로그램 경로와는 다른 분기가 일어난다면 (즉, 실제 프로그램 실행 경로와 주어진 프로그램 경로가 다른 경우) 실행의 흐름을 바꾸도록 함수 최소화 기법을 이용하여 현재의 입력 데이터를 수정한다. 만약 이러한 과정을 거치고도 적절한 입력 데이터를 찾지 못한다면 프로그램 경로 상에서 현재 분기 조건문에 바로 앞서 실행된 조건문으로 되돌아가 다른 입력 값을 찾는 과정을 되풀이해야 한다.

콘콜릭 테스팅 방법은 이 논문에서와 같이 특정 프로그램 경로나 목표 블록(또는 분기) 대신에 프로그램의 모든 경로들을 탐색한다. 최근에는 콘콜릭 테스팅을 특정 목표에 제한하는 목적 지향 콘콜릭 테스팅 방법이 제안되기도 하였다[3][4].

콘콜릭 방법을 통해 테스트 데이터 생성을 위한 첫 번째 단계로 우선 각 입력 변수에 무작위 값이 할당되어 프로브가 삽입된 프로그램이 실행된다. 이 때 실행된 경로를 따라 심볼릭 수행을 하며 심볼릭 수행을 한 결과는 심볼릭 맵을 통해 관리된다. 심볼릭 맵은 각 변수에 대한 심볼릭 표현식이 있으며 배정문을 만날 때마다 삽입된 프로브를 통해 이 표현식은 갱신된다. 물론 프로그램 실행이 처음 시작된 경우에는 각 입력변수에 대한 심볼릭 표현식은 어떤 배정문도 실행되기 전이기 때문에 단순한 심볼릭 값이 된다.

심볼릭 맵외에도 콘콜릭 테스팅은 실행 경로에 따른 경로 제약식을 저장하기 위한 자료 구조 Φ를 관리한다. Φ는 초기 값으로 true가 주어진다. 만약 조건문 'if p then ...'을 처리하는 경우에 실제 실행된 경로가 p가 참이 될 때에는 조건식 p에 해당하는 심볼릭 표현식 ξ(p)이 현재 Φ에 있는 식과 and로 결합되어 Φ에 저장된다. 실행된 경로가 p가 거짓인 경우에는 심볼릭 표현식 'not ξ(p)'이 결합된다. 즉, Φ를 만족하는 입력 값은 지금까지 실행된 경로를 실행하는 테스트 데이터가 된다.

콘콜릭 테스팅은 이전에 실행된 프로그램 경로와는 다른 경로를 실행하기 위해 Φ를 구성하는 심볼릭 표현식에서 하나를 선택하여 부정하여 새로운 경로 제약식을 생성한다. 예를 들어 $<p_0, p_1,..., p_{k-1}>$이 실제 실행된 경로라고 하자. 콘콜릭 테스팅은 분기 $p_j(0\le j<k)$를 선정한다. 이 때 ϕ_i을 이 분기를 수행하기전의 경로 제약식이라 하고 ϕ_t를 이 분기에 의해 생성된 심볼릭 표현식이라고 하자. 콘콜릭 테스팅은 이전 경로와는 다른 경로를 실행하는 테스트 데이터를 찾기 위해 $(\wedge_{\phi\in\Phi i}\phi)\wedge(\text{not } \phi_t)$를 만족하는 해를 구한다. 만약 이 새로운 경로 제약식을 만족하는 입력 값으로 프로그램을 실행하면 실행 경로는 $<p_0, p_1,...,\text{paired}(p_j), p'_{j+1}, ..., p'_m>$일 것이다. 여기에서 paired(pj)는 p_j의 짝 분기를 나타낸다. 즉, 어떤 조건식에서 p_j가 참인 분기라면 paired(p_j)는 거짓인 분기를 나타내며 p_j가 거짓인 분기라면 paired(p_j)는 참인 분기를 나타낸다. 이와 같은 과정을 모든 경로들에 대한 테스트 데이터를 찾을 때까지 반복하거나 지정한 회수만큼 반복한다.

III. 테스트 데이터 생성

1. 배경 지식

이 논문의 목적은 프로그램의 실행 가능한 모든 경로를 가능한 많이 실행하게 하는 테스트 데이터를 생성하는 것이다. 이를 위해 프로그램 명령어 n_i에서 명령어 n_j로 제어가 이전되는 분기(branch)들을 가능한 많이 실행하여야 한다. C와 같은 고수준 언어에서 분기는 조건문이나 반복문에서 찾아볼 수 있다. 가장 대표적으로 나타나는 형태가 if-else문이다. if-else와 같은 분기 조건문이 컴파일러에 의해 구현될 때 분기 (n_i, n_j)에서 명령어 n_i는 jmp, jne, je 등과 같은 명령어에 해당되고 명령어 n_j는 n_i에 실행된후에 다음에 실행되는 명령어에 해당된다.

분기가 컴파일러에 의해 어떻게 번역되는지 예를 들어 구체적으로 살펴보자.

if (foo>10) {
block1
else
block2
}
	1: cmp [foo] 10
	2: jle elseblock
	3: assmebly code for block1
	4: elseblock:
	5: assembly code for block2

(a)	(b)

그림 1. (a) 예제 C 조건문 (b) 어셈블리코드
Fig. 1. (a) Example C conditional statement (b) assembly code for the (a) conditional statement

그림 1은 C언어로 작성된 조건문과 이를 어셈블리어로 번역된 결과를 보여준다. 이 조건문에서는 두 개의 분기 (2, 3), (2, 5)가 존재한다. 변수 foo의 값이 10이하이면 (2, 3)분기가 실행되고 변수 foo의 값이 10보다 크면 (2, 5)분기가 실행된다. 따라서 가능한 많은 분기를 실행하기 위해서는 변수 foo의 값을 한번은 10보다 큰 값 한번은 10보다 작은 값으로 설정되도록 입력 값을 조정할 필요가 있다.

보통 분기를 실제 수행하는 jmp류의 명령어들은 OF (오버플로우 플래그), ZF(제로 플래그), SF(부호 플래그)와 같은 특정 플래그에 있는 값을 참고하여 분기를 수행한다. 이러한 플래그들은 일반적으로 cmp와 같은 비교 연산을 수행하는 명령어 의해 설정된다. cmp,명령은 첫 번째 인자에서 두 번째 인자를 뺀 후 결과 값은 버린다. 빼기 연산의 결과에 따라 플래그 값이 변경된다. 예를 들어 그림 1의 foo 변수가 음수 -5가 현재 설정되어 있다고 가정하자. 이 경우 1번 cmp 명령어가 실행된 후에는 플래그들이 OF=0, SF=1, ZF=0으로 설정되어 변수 foo의 값이 10보다 작음을 알 수 있다. 따라서 jle 명령어에 의해 분기 (2, 5)가 수행된다. 따라서 두 개의 분기 중 하나의 분기 (2, 5)가 수행되었으므로 분기 (2, 3)을 실행하는 테스트 데이터를 더 식별할 필요가 있다.

2. 분기 함수(branch function)

테스트 데이터를 자동으로 생성하는 것은 소프트웨어 테스팅 비용을 줄이기 위한 매우 효과적인 방법이다. 제안된 많은 테스트 데이터 자동 생성 방법들은 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. 예를 들면, 어떤 프로그램에서 'x==5'인 분기 조건이 참이 되게 하는 테스트 데이터를 구하는 문제를 생각해보자. 이와 같은 분기 조건은 'F(x) = |5-x|'와 같은 함수로 간주한다. 이때 함수 F(x)를 최소화하는 입력 데이터 x는 분기 조건 'x==5'를 참이 되게 하는 입력 값이 원하는 테스트 데이터가 된다. 여기에서 분기 조건 'x==5'의 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 경우보다 분기 조건 "x==5"을 참이 되게 하는 경우에 보다 더 가까이 접근했음을 의미한다. 만약 탐색 알고리즘이 10을 먼저 생성하였다면 이를 감소하는 방향으로 탐색을 진행할 것이다. 그림 2는 분기 조건 'x==5'가 참이 되게 하는 x의 값에 따른 적합성 함수 그래프를 보여준다.

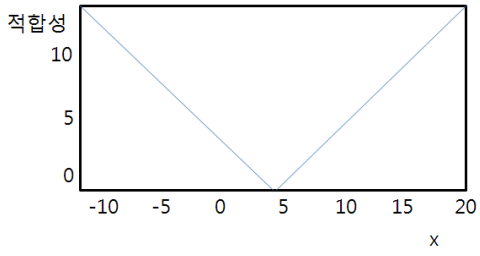

그림 2. 분기 조건 'x==5'를 참이 되게 하는 x에 대한 적합성 그래프
Fig. 2. Fitness function landscape for evaluating 'x==5' to true

이 논문에서는 이러한 적합성 함수 개념을 이용한 목적 코드 기반 테스트 데이터 생성 방법을 제안한다. 이 논문에서는 적합성 함수를 분기 함수(branch function)라 한다. 그 이유는 ET와 달리 특정 프로그램 목표를 실행하는 보다 적합한 테스트 데이터를 생성하기 보다는 가능한 많은 분기를 실행하기 위한 테스트 데이터를 생성하기 때문이다.

이 논문에서 제안하는 방법은 cmp 비교 연산 명령어의 결과가 ZF, OF, SF와 같은 플래그들이 다양한 방식으로 설정될 수 있도록 하여 가능한 많은 분기를 실행하는 테스트 데이터를 생성한다. 이를 위해 이 논문에서는 cmp 명령어 비교 연산 실행 뒤에 어떤 형태의 jmp 문이 뒤 따르는지에 관계없이 가능한 많은 분기가 수행되도록 분기 함수(distance function)들을 cmp 명령어가 수행될 때 평가하도록 한다. 분기 함수들의 평가 결과에 따라 입력 값을 조정하여 아직 실행되지 않은 분기를 실행되도록 하는 테스트 데이터를 생성한다.

각 'cmp A B' 명령어가 실행될 때 다음과 같은 형태의 분기 논리식이 고려된다.

F rel 0

여기에서 F와 rel은 표 1에 주어진다.

표 1. 분기 함수
Table 1. Branch function

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

위 논리식에서 F가 분기 함수이다. 분기 함수 F는 해당 논리식이 참일 때 음수 값이나 0을 가지며 거짓인 경우에는 양수 값을 갖는다. 분기 함수 F를 심볼릭 실행 기법을 통해 입력 변수들 $x=(x_1, x_2, ..., x_n)$의 함수 $F(x)$로 명시적으로 표현하는 것이 가능하다. 그러나 실제 프로그램에서는 이는 복잡한 대수적 조작을 요구한다. 이 논문에서는 어떠한 대수적 조작을 요구하지 않는 동적 분석 기법을 통해 프로그램을 실행하여 분기 함수가 평가되도록 하는 방식을 취한다.

표 2. 분기 함수 평가 예
Table 2. Example of evaluating branch functions

F	분기 함수	평가 값
10-[foo]	5	F
10-[foo]	5	F
[foo]-10	-5	T
[foo]-10	-5	T
abs([foo]-10)	5	F
-abs([foo]-10)	-5	T

표 2는 그림 1의 프로그램에서 foo 변수의 값을 5로 설정되게 하는 어떤 입력에 대해 실행되었을 때의 분기 함수 및 해당 논리식의 진리 값을 보여준다. 이 경우에 분기 (2, 5)가 실행됨을 알 수 있다. 나머지 분기 (2, 3)을 실행하는 테스트 데이터를 식별하기 위해 이 논문에서는 표 2에서 평가 값이 거짓인 나머지 분기 함수를 함수 최소화 기법으로 참이 되게 하는 입력 값을 찾는다. 이러한 입력 값을 찾는 방법은 다음 절에서 자세하게 기술된다.

3. 분기 함수의 최소화

이 절에서는 각 cmp 문에서 현재 평가 값이 거짓인 분기 함수에 대해 참이 되게 하는 테스트 데이터를 어떻게 식별하는 방법에 대해 기술한다.

테스트 데이터 식별을 위해 기본적으로 Korel이 제안한 방법[7]을 이용한다. 이 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 방법은 각 입력 변수를 차례대로 선정하여 값을 조정한다. 이 때 선정되지 않은 입력 값들은 변경하지 않는다. 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 감소 되었을 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 cmp 명령을 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.

예를 들어 표 2에서 분기함수 '10-[foo]'를 대상으로 이를 참이 되게 하는 foo 변수의 값을 식별하는 과정을 살펴보자. 여기에서 foo 변수는 이 프로그램에서 하나 뿐

인 입력변수라 가정하고 어떤 프로그램도 foo 변수의 값을 변경시키는 문장이 없다고 가정한다. 이러한 가정은 방법의 제약 요건이 아닌 단지 설명의 단순함을 위해서이다.

우선 가장 먼저 foo 변수에 대해 탐색 이동을 수행한다. 만약 foo 변수가 0으로 초기화 되어 있을 때 1만큼 감소시킨다. 이 경우 분기 함수의 값은 11이 되어 이전 값 10보다 값이 개선이 되지 않는다. 반면에 1만큼 증가 시키는 경우에는 분기 함수의 값이 9가 되어 분기 함수의 값이 개선됨을 알 수 있다. 따라서 증가시키는 방향으로 패턴 이동을 수행한다. foo 값을 5로 하였을 때 분기 함수의 값은 분기 함수의 값은 5가 되어 이전보다 개선됨을 알 수 있다. 한 번 더 foo의 값을 15로 하는 패턴 이동을 수행하면 분기 함수의 값이 음수가 되어 함수가 최소화 되었음을 알 수 있다. 여기에서 해당 변수의 값의 증가 (또는 감소) 분이 패턴 이동의 횟수에 비례하도록 하는 것이 일반적이다.

4. 테스트 데이터 생성

특정 분기를 실행하는 테스트 데이터를 생성하는 문제는 목표 분기와 관련 있는 분기 함수 F(x)를 최소화하는 문제로 간주할 수 있다. 따라서 분기 (bi, bj)에서 분기 함수 F가 주어진 경우에 테스트 데이터 생성 문제는 다음과 같이 분기 함수 최소화 문제로 형식화 된다:

- 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 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. 이 논문에서는 목표 분기의 'cmp' 명령어와 같이 비교 연산 명령어가 위치한 곳을 분기의 시작 블록으로 간주한다. 따라서 3절에서 기술한 함수 최소화 기법을 통해 생성된 테스트 데이터는 어떤 경로를 통하든지 상관없이 해당 비교 연산 명령어만을 실행하면 된다. 이 논문에서는 비교 연산 명령어로 'cmp' 명령어만을 고려한다.

```
Initialize v, Stack, H;

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

execute(P, v, Stack);

while Stack<>∅ do {
   pop(F) from Stack;
   if not minimize(F, v) then
            put F in H;
}
```

그림 3. 테스트 데이터 생성 프로시듀어
Fig. 3. Test data procedure

그림 3은 분기 함수 최소화 방법을 이용하여 가능한 많은 분기들을 실행할 수 있는 테스트 데이터를 생성하는 프로시듀어를 보여준다. 이 프로시듀어는 기본적으로 깊이 우선 탐색(depth first search)에 바탕을 두고 있다.. 또한 다음 execute 함수와 minimize 함수를 사용하여 테스트 데이터를 생성한다:.

- execute(P, v, S): 프로그램 P를 입력 v로 실행한다. 이 때 'cmp' 명령어가 실행되는 경우에 관련된 분기함수들을 평가한 후에 분기 함수 평가 결과를 갱신한다. 아직 거짓으로 평가된 분기 함수가 남아있는 경우에만 명령어 주소와 같은 관련정보를 스택 S에 넣는다.
- minimize(F, v): 3절에서 시술한 방법에 따라 분기 함수 F를 최소화하는 입력 값 v를 생성한다. 성공적으로 생성되었다면 이 함수는 참을 반환한다. 만약 F를 최소화하는 값을 찾지 못했다면 거짓을 반환한다. minimize 함수는 execute 함수를 이용하여 스택 정보를 갱신한다.

우선 필요한 변수 및 자료 구조들을 초기화 한다. Stack은 스택 자료 구조이며 H는 minimize 함수에 의해 최소화되는 값을 찾지 못한 분기 함수가 있는 명령어 주소를 저장하는 자료 구조이다. 임의의 입력 값 v를 생성한 후에 대상 프로그램 P를 실행한다. cmp 명령어가 실행되면 관련된 분기함수들을 평가한 후에 거짓이 되는

분기 함수가 있는 cmp 명령어 주소를 스택 Stack에 넣는다. 프로그램 실행이 완료 된 후에 Stack으로부터 가져온 cmp 명령어 주소로 아직 처리 안된 분기 함수를 선정한다. 다음 단계로 선정된 분기 함수를 최소화하는 있는 입력 값을 구하여 프로그램을 수행한다. 만약 분기 함수를 최소화 할 수 있는 값을 식별하지 못하였다면 처리 마크를 한 후에 이를 H에 넣는다. 테스트 데이터 실행 절차 후에 H에 있는 분기 함수들은 어떤 분기들이 아직 실행되지 않았는지에 대한 보다 구체적인 정보를 얻게 해주는 역할을 할 수 있다. 이러한 과정을 Stack에 아무것이 남아있지 않을 때까지 반복한다.

Ⅳ. 평 가

표 3. 테스트 데이터 생성 도구와의 비교
Table 3. Comparison results with other test data generation techniques

방법	코드	테스트 데이터 생성 전략			구현 방법		
		경로	목적	모든 경로	정적	동적	복합
[7]	X	X				X	
[3]	X			X			X
[4]	X			X			X
[9]	X		X		X		
[10]	X		X		X		
[5]			X	X			X
제안 방법				X		X	

표 3에서 이 논문에서 제안한 테스트 데이터 생성 방법을 다른 테스트 데이터 생성도구와 비교하였다. 이를 위해 우선 테스트 데이터 생성 정보가 원시 코드에 바탕을 두느냐 아니면 목적 코드에 바탕을 두느냐에 따라 구분하였다. 또한 테스트 데이터 생성 전략을 경로 지향, 목적 지향 및 모든 경로로 분류하였으며 구현 방법의 비교를 위해 테스트 데이터를 생성하기 위해 실제 프로그램을 수행하느냐 여부에 따라 정적 방법, 동적 방법 및 복합 방식으로 분류하였다. 복합 방식이란 테스트 데이터 생성을 위해 부분적으로는 심볼릭 실행과 같은 정적 방식을 사용하고 부분적으로는 프로그램을 직접 실행하는 동적 방식을 혼합한 방법을 의미한다.

표 3에서 볼 수 있듯이 대부분의 테스트 데이터 생성 방법들은 원시 코드에 바탕을 두고 있다는 사실을 알 수 있다. 다만 OSMOSE[5]에서는 이 논문과 같이 목적 코드을 이용하여 테스트 데이터 생성을 한다. 그러나 이 방법은 콘콜릭 테스팅과 같이 심볼릭 수행 기법을 이용한다.

또한 이 논문에서 제안한 방법의 성능 평가를 위해 CPU (2.0Ghz), RAM 4GB, 윈도우 XP를 탑재한 PC에서 테스트 환경을 구현하였다. 테스트 환경은 PyEmu[11]을 이용하였다. PyEmu는 개발자가 파이썬을 사용하여 CPU 에뮬레이션 작업을 수행할 수 있게 해준다. 이 논문에서는 에뮬레이션을 수행하기 위해 IDA Pro[12]와 같은 디스어셈블작업을 위한 도구에 의존하지 않기 위해 독립적인 정적 분석 라이브러리인 PEPyEmu에뮬레이터를 이용하였다.

이 논문에서 PyEmu를 이용하여 테스트 환경을 구현한 이유는 PyEmu가 어떤 프로그램의 특정한 실행지점을 관찰하거나 변경할 수 있게하는 매우 유연하고 강력한 콜백(callback) 메커니즘을 제공하기 때문이다. PyEmu는 레지스터 핸들러, 라이브러리 핸들러 등과 같은 여러 핸들러를 제공하는데 이 논문에서는 특정 명령이 실행되기 전에 해당 명령의 실행을 모니터링하는 명령 핸들러를 이용하여 cmp 명령어에 대해 핸들러를 설치하였다. 이 핸들러에서 분기 함수 평가를 수행하고 제약조건이 위반되었는지를 검사한다.

제안된 방법의 타당성 측정을 위해 테스트 환경에서 그림 4의 프로그램에 대해 달성된 분기 커버리지를 측정하였다. 그림 4는 세 개의 정수를 받아들여 삼각형인지 검사한 후에 삼각형인 경우 정삼각형, 이등변 삼각형, 부등변 삼각형, 삼각형이 안 되는 경우로 분류하는 프로그램을 C로 작성한 후에 실행 파일을 디스어셈블 시킨 결과를 제어 흐름 그래프로 표현한 것이다. 10여번 반복하여 테스트 데이터를 생성하여 실험해 본 결과 평균 98%의 커버리지를 달성하였다. 이는 평균 83%의 분기 커버리지를 달성한 랜덤 테스팅에 비해 매우 탁월한 성능을 보여주는 것을 확인할 수 있었다.

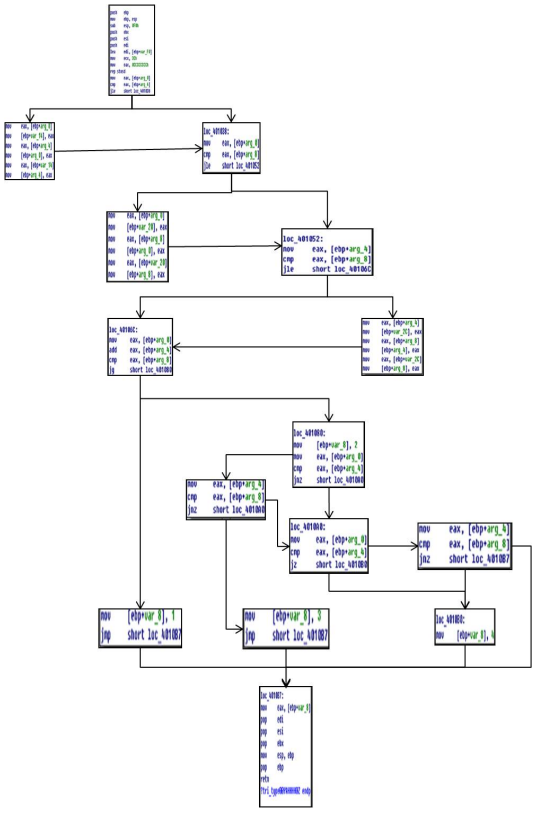

그림 4. 삼각형 분류 프로그램 제어 흐름 그래프
Fig. 4. Control flow graph of a trianle classification program

V. 결 론

이 논문에서는 실행 기반 목적코드만 존재하는 경우에 이를 기반으로 화이트박스 테스트 데이터를 자동으로 생성하는 방법을 제안하였다. 기존 대부분의 화이트박스 테스트 방법들은 원시 코드가 존재하는 경우에만 적용가능한 한계점을 지니고 있었다. 따라서 벤더가 기술 보안상의 이유로 소스 코드를 제공하지 않거나 소스 코드가 분실되는 경우에는 화이트박스 테스팅을 수행하는 것이 사실상 불가능하였다. 뿐만 아니라 소스 코드가 있다할지라도 컴파일 과정 자체가 신뢰가 없을 경우에는 소스 코드 분석 결과를 테스트 과정에 이용할 때 부정확한 결과가 초래될 여지가 있다.

이 논문에서는 가능한 모든 경로를 실행하는 테스트 데이터를 생성하기 위하여 'cmp'와 같은 비교 연산 명령어의 결과가 다양하게 설정될 수 있도록 분기 함수들을 cmp 명령어가 수행될 때 평가하도록 한다. 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 수행에 수반되는 비용을 절감할 수 있다.

이 연구에서 개발된 테스트 데이터 생성 도구는 아직까지는 제안된 방법의 효과성만을 판별하는 개념적 도구 수준이다. 따라서 보다 실효성 있는 실험을 위해서는 안정화 작업이 필요하다.

References

[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, P, "Search-based Software Test Data Generation: A Survey", Software Testing, Verification and Reliability, vol. 14, no. 2, pp. 105-156, 2004.

[3] P. Godefroid, N. Klarlund, 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, K. Sen, "Heuristics for dynamic test generation", Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering , pp. 443-446, 2008.

[5] S. Bardin, P. Herrmann, OSMOSE: automatic structural testing of executables", Software Testing, Verification and Reliability, vol. 21, pp. 29-54, 2011.

[6] 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.

[7] B. Korel, "Automated Software Test Data Generation", IEEE Trans. onSoftware Eng, vol. 16. no. 8. pp. 870-879, 1990.

[8] 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.

[9] A. Gotlieb, B. Botella,and M. Rueher, "Automatic Test Data Generation using Constraint Solving Techniques", Proceedings of ACM ISSTA, pp. 53-62, 1998.

[10] A. Gotlieb, T. Denmat, and B. Botella, "Goal-oriented Test Data Generation for Pointer Programs", Information and Software Technology, Vol. 49, Issues 9-10, pp. 1030-1044, 2007.

[11] http://code.google.com/p/pyemu/

[12] http://www.idabook.com/

※ 본 연구는 한성대학교 교내연구장려금 지원과제임.

저자 소개

정 인 상(Insang Chung)(정회원)

- 1987년 서울대학교 컴퓨터공학과 졸업(학사)
- 1989년한국과학기술원(KAIST) 전산학과 졸업(석사)
- 1993년한국과학기술원(KAIST) 전산학과 졸업(박사)
- 1999~현재 한성대학교 컴퓨터공학과 교수

<관심분야 : 소프트웨어 공학, 소프트웨어 테스팅>