DOI: 10.3745/KIPSTD.2009.16-D.3.371

SAT를 기반으로 하는 플래그 변수가 있는 프로그램 테스팅을 위한 테스트 데이터 자동 생성

정 인 상†

요 약

최근에 테스트 데이터를 자동으로 생성하는 방법에 관한 연구가 활발하게 진행되고 있다. 그러나 이러한 방법들은 플래그 변수가 프로그램에 존재하는 경우에는 효과적이지 못함이 밝혀졌다. 이는 엔진 제어기와 같은 내장형 시스템들이 전형적으로 디바이스 관련 상태 정보를 기록하기 위해 플래그 변수를 많이 이용한다는 점을 고려할 때 문제가 된다. 이 논문에서는 플래그 변수가 있는 프로그램에 대하여 효과적으로 테스트 데이터를 생성할 수 있는 방법을 소개한다. 이 방법은 테스트 데이터 생성 문제를 SAT(SATisfiability) 문제로 변환하고 SAT 해결도구를 이용하여 자동으로 테스트 데이터를 생성한다. 이를 위해 프로그램을 1차 관계 논리 언어인 Alloy로 변환하고 Alloy 분석기를 통하여 테스트 데이터를 생성한다.

키워드 : 프로그램 테스트, 자동 테스트 데이터 생성, Alloy, 플래그 변수

Automated Test Data Generation for Testing Programs with Flag Variables Based on SAT

In-Sang Chung†

ABSTRACT

Recently, lots of research on automated test data generation has been actively done. However, techniques for automated test data generation presented so far have been proved ineffective for programs with flag variables. It can present problems when considering embedded systems such as engine controllers that make extensive use of flag variables to record state information concerning devices. This paper introduces a technique for generating test data effectively for programs with flag variables. The presented technique transforms the test data generation problem into a SAT(SATisfiability) problem and makes advantage of SAT solvers for automated test data generation(ATDG). For the ends, we transform a program under test into Alloy which is the first-order relational logic and then produce test data via Alloy analyzer.

Keywords : Program Testing, Automated Test Data Generation. Alloy, Flag Problem

1. 서 론

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[1].

ET는 후보 테스트 데이터와 원하는 테스트 데이터간의 차이를 평가하기 위해 적합성 함수(fitness function)를 이용한다[2]. 예를 들면, 어떤 프로그램에서 “x==5”인 분기 조건이 참이 되게 하는 테스트 데이터를 구하는 문제를 생각해 보자. 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

※ 본 연구는 2008년도 한성대학교 교내연구비 지원과제임.
† 정 회 원 : 한성대학교 컴퓨터공학과 교수
논문접수 : 2009년 2월 17일
수 정 일 : 1차 2009년 4월 22일
심사완료 : 2009년 5월 4일

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==5”을 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[3, 4].

플래그 문제를 해결하기 위해 많은 방법들이 제안되었지만 지금까지 만족할만한 해결책은 없는 실정이다. 이 논문에서는 플래그 문제의 해결을 위한 효과적인 목적 지향(goal-oriented) 테스트 데이터 생성방법을 소개한다. 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[5]. 따라서 사용자가 일일이 프로그램 경로를 선정하는 부담이 없으며 사용자가 정한 프로그램 경로가 실행 불가능하다면(즉, 주어진 경로를 실행할 수 있는 입력 값이 존재하지 않는다면) 사용자가 다른 경로를 선택해야 하였으나 목적 지향 테스트에서는 주어진 프로그램 블록을 실행할 수 있는 다른 경로를 자동으로 탐색하여 입력 값을 생성할 수 있다.

프로그램의 실행을 요구하는 기존의 목적 지향적 방법과는 달리 이 논문에서 소개하는 방법은 테스트 데이터 생성 문제를 SAT(SATisfiability)로 변환하여 테스트 데이터를 생성하는 정적 방법이다. SAT는 어떠한 이진 논리식(Boolean expression)이 있을 때 그 논리식을 참이 되도록 하는 모델(model)이 존재하는지를 검사하는 문제를 말한다. 여기에서 모델이란 주어진 논리식을 참이 되게 하는 변수들의 값의 조합이다. 현재 zChaff[6], BerkMin[7] 등과 같은 SAT 해결기(SAT solver)들이 개발되어 사용되고 있다.

이 논문에서는 플래그 문제를 효과적으로 접근하기 위해 [8]에서 제안된 방법을 기반으로 하고 있다. [8]에서는 모양 문제(shape problem)를 해결하기 위한 방안으로 SAT 기반 테스트 데이터 생성 방법을 사용하였다. 모양 문제란 프로그램이 포인터를 사용하는 경우에는 테스트 데이터는 리스트, 트리 또는 그래프와 같은 2차원적인 자료구조로 표현되며 이와 같이 입력 자료 구조의 모양을 식별하는 문제를 말한다. 이 논문에서는 [8]에서 제안한 프레임워크가 모양 문제뿐만 아니라 플래그 문제에도 효과적으로 사용될 수 있음을 보인다.

SAT를 이용하여 테스트 데이터 생성을 하기 위해서는 다음과 같은 점이 고려되어야 한다. 우선 현재의 SAT 도구들은 일반적인 논리식 대신에 논리곱 정규식(Conjunctive Normal Form, CNF)만을 입력으로 받기 때문에 프로그램을 CNF로 변환할 수 있는 방법이 있어야 한다. 또한 CNF로 표현된 논리식을 만족하는 모델이 프로그램의 특정 블록에 도달할 수 있는 경로 및 입력 값에 대한 정보를 포함하여야 한다.

이를 위해 [8]에서 C 프로그램을 직접적으로 CNF로 직접 변환하지 않고 Alloy 명세로 변환하는 방법을 기술하였다. Alloy는 1차 관계 논리(first-order relational logic)에 기반 한 모델링 언어이다[9-11]. 입력 프로그램을 Alloy 명세로 변환하는 이유는 Alloy가 기본적으로 SAT에 기반을 둔 제약식 해결도구(solver)이기 때문이다. Alloy는 Alloy 명세를 입력으로 받아 우선 명제 논리식(propositional logic formula)으로 변환한 후에 이를 CNF로 바꾸는 과정을 거친다. 또한 현재 Alloy 분석도구는 기본적으로 여러 SAT 해결 방법들이 제공되기 때문에 이를 별다른 노력 없이 이용할 수 있다는 이점이 있다.

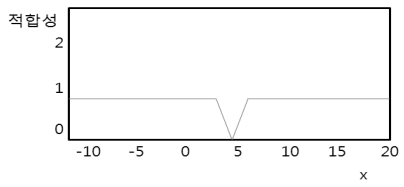
본 논문은 다음과 같이 구성된다. 2장에서는 기존의 테스트 데이터를 자동으로 생성하는 방법들이 어떻게 플래그 문제를 처리하는지에 대해 기술한다. 3장에서는 [8]에서 제안한 SAT 기반 테스트 데이터 생성을 위한 프로그램 변환에 대해 기술한다. 4장에서는 대표적인 플래그 변수를 사용하는 프로그램들에 대해 제안된 방법에 대한 실험 결과를 기술한다. 마지막으로 결론 및 향후 연구에 관해 기술한다.

2. 플래그 문제

프로그램의 분기 조건에 플래그를 사용하였을 때 플래그 문제가 발생한다. 플래그 문제를 구체적으로 이해하기 위해 (그림 1)에 주어진 간단한 프로그램을 생각해보자.

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

(a)



(b)

(그림 1) (a) 예제 프로그램 (b) 적합성 함수

목표 문장을 실행하기 위한 테스트 데이터를 자동으로 생성하기 위해서는 플래그 변수 'flag'가 참이 되도록 하는 입력 변수 x의 값이 필요하다. 지금까지 제안된 테스트 데이터 방법은 이를 위해 분기 거리를 테스트 데이터의 적합성 함수로 사용하여 평가하였다. 그러나 플래그 변수의 값은 참 또는 거짓만을 가질 수 있으므로 "|flag|"로 주어지는 분기 함수는 부울리언 함수가 된다. 이 경우에는 (그림 1)(a)와 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 진화 알고리즘에 바탕을 둔 테스트 방법에 많은 연구가 진행되었다. Harman은 분기 조건에 있는 플래그 변수를 제거하기 위해 해당 플래그 변수의 정의로 대치하는 프로그램 변환 방법을 제안하였다[4]. 예를 들어 Harman이 제안한 변환 방법을 사용하면 (그림 2)(a)의 flagProc2는 플래그 변수가 제거된 (그림 2)(b)의 flagProc3과 같이 변환할 수 있다. 이 경우에 분기 조건 "x==5"에 대한 적합성 함수는 "|5-x|"로 주어지며 입력 값 x에 따른 적합성 함수의 값은 (그림 2)(c)와 같다. 즉, 요구되는 테스트 데이터 (i.e., x=5)를 찾기 위한 충분한 정보가 제공되는 것이다. (그림 1)(b)와 비교하여 (그림 2)(c)는 입력 값을 증가하거나 감소하는 것에 따라 적합성 값이 증가 하거나 감소가 되므로 어느 방향으로 테스트 데이터를 탐색해야 하는지를 알 수 있다. 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[3]. (그림 1)(a) flagProc1에서 2번 조건에서 사용되는 플래그 변수 flag는 1번 문장에서 정의된다. 따라서 1번 문장에서 플래그 변수를 정의할 때 분기 거리 "|5-x|"를 미리 계산하여 두고 이를 2번 분기 조건에서 적합성 값으로 사용한다. 이 경우에 적합성 함수는 (그림 2)(c)와 같게 된다.

또 다른 방법으로 Baresel과 Sthamer이 제안한 자료 흐름 분석을 이용한 방법이 있다[12]. (그림 2)(a)의 flagProc2에서 목표 문장이 실행되기 위해 3번 분기 조건이 참이 되어야 한다. 이를 위해 3번 문장이 실행되기 전에 2번 문장이 실행되어 변수 flag가 참이 될 필요가 있다. 이러한 정보는 자료 흐름 분석 기법을 이용하여 얻을 수 있다. 따라서 목표 문장을 실행하는 테스트 데이터를 찾기 위해 우선 1번 분기를 참이 되게 하는 테스트 데이터를 우선 찾을 필요가 있다. 1번 분기 조건의 분기거리로 주어지는 적합성 함수 "|5-x|"는 (그림 2)(c)와 같으며 이를 최소화하는 입력 x는 앞에서 설명한바와 같이 쉽게 탐색할 수 있다.

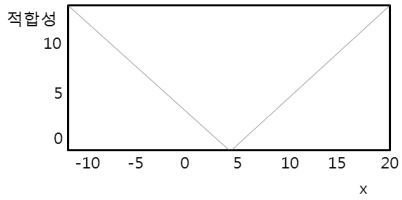
이 모든 방법들은 불행하게도 (그림 3)(a)에 주어진 프로그램과 같이 반복문내에서 플래그 변수가 정의되는 경우에는 적용할 수 없다. Baresel 등은 이 문제를 해결하기 위해 새로운 프로그램 변환 방법을 제안하였다[13]. 이 방법은 두 개의 변수 fitness와 counter를 사용한다. (그림 3)(b)는 변환된 프로그램을 보여준다. 변수 fitness는 목표 문장을 실행할

```
void flagProc2(int x) {
    int flag=0;
1:  if (x==5)
2:      flag=1;
3:  if (flag)  { // 목표 문장 ... }
}
```

(a)

```
void flagProc3(int x) {
    int flag=0;
1:  if (x==5)
2:      flag=1;
3:  if (x==5)  { // 목표 문장 ... }
}
```

(b)



(c)

(그림 2) (a) 예제 프로그램 (b) 변환된 예제 프로그램 (c)적합성 함수

```
void flagProc4(int a[], int n) {
    int flag=1;
    int i;

    for (int i=0; i<n; i++)  {
        if (a[i]!=0)
flag=0;
    }

    if (flag) { // 목표 문장 }
}
```

(a) 변환 전

```
void flagProc5(int a[], int n) {
    int flag=1;
    int i;
    int counter=0;
    double fitness=0.0;

    for (int i=0; i<n; i++)  {
        if (a[i]!=0) {
            flag=0;
        else fitness+=1.0;
        counter++;
    }

    if (counter==fitness) { // 목표 문장 }
}
```

(b) 변환 후

(그림 3) 반복문 내에서 플래그 변수 정의 예제 프로그램

수 있도록 반복이 성공적으로 수행될 때 마다 1만큼 증가하고 변수 counter는 매 반복마다 1만큼 증가한다. (그림 3)(b)에서와 같이 플래그 변수를 사용한 분기 조건 “if (flag)”를 “if (fitness==counter)”로 대치하여 목표 문장을 실행하는 테스트 데이터를 찾기 위한 적합성 함수를 개선한다.

3. SAT를 기반으로하는 테스트 데이터 생성

이 장에서는 Alloy에 대해 간단하게 소개한 후에 테스트 데이터를 자동으로 생성하기 위하여 [6]에서 제안된 C 프로그램을 Alloy로 변환하는 방법에 대하여 기술한다.

3.1 Alloy

Alloy는 MIT의 Daniel Jackson과 그의 동료들에 의해 개발된 1차 관계 논리(first-order relational logic)에 기반을 둔 모델링 언어이다[9]. 기본적으로 Alloy는 형식 명세 언어인 Z에 의해 영향을 받았으며 단순히 시스템의 명세를 기술하기 위한 언어가 아니라 기술된 모델을 분석 할 수 있도록 개발되었다. Alloy는 집합과 관계에 기반을 두고 시스템을 기술하며 분석기능을 자동화하기 위해 Alloy 분석기가 개발되었다. Alloy 분석기는 여러 논리식으로 구성된 Alloy 명세를 참이 되게 하는 모델을 탐색하는 도구이다. 여기에서 모델은 명세에 있는 논리식들을 참이 되도록 명세에 나타난 집합들과 관계들에게 구체적인 값들을 바인딩 하는 것을 말한다. Alloy 명세는 크게 두 부분, 시그너쳐 단락(signature paragraph)과 제약식 단락으로 구성된다.

시그너쳐(signature)를 통해 새로운 형을 정의한다. 다음은 People과 Fish를 정의한 예이다:

```
sig People {}        sig Fish {}
```

이를 People과 Fish라는 개체들의 집합들을 각각 정의한 것으로 볼 수 있다. 또한, 개체들 간의 관계도 시그너쳐의 필드들을 통해 정의할 수 가 있다. 예를 들어, 사람들이 물고기를 잡을 수 있다는 사실은 People과 Fish 개체들 간의 관계(i.e., People->Fish)를 People에 catch라는 필드를 정의하여 다음과 같이 표현할 수 있다:

```
sig People { catch: Fish }
```

만약 물고기의 종류를 세분화하여 표현할 필요가 있는 경우는 다음과 같이 시그너쳐 확장(signature extension)을 통해 기술 한다:

```
sig CatFish, Mullet        extends Fish {}
```

이 예는 CatFish와 Mullet은 공통 원소를 지니지 않은 Fish의 부분집합임을 나타낸다. 만약 Fish가 “abstract sig Fish {}”와 같이 선언되어 있다면 Fish는 CatFish와 Mullet으로만 구성된 집합임을 나타낸다.

제약식들은 ‘fact’와 술어(predicate)를 사용해서 표현한다. ‘fact’는 항상 참이 되야 하는 속성, 즉 불변성(invariant)을 표현한다. 다음은 “모든 물고기는 길이가 0이상이어야 한다”는 사실과 “어느 한 물고기는 기껏해야 한사람만이 잡을 수 있다.”는 불변성을 표현한 것이다:

```
fact {
        all f: Fish | int f.len >0
        all f: Fish | lone d: People |  f in p.catch
}
```

이는 Fish에 필드 len이 다음과 같이 정의되어 있음을 가정한 것이다:

```
sig Fish { len: Int }.
```

여기에서 ‘in’은 멤버쉽관계를 나타내는 연산자이며 ‘Int’는 정수형 개체를 나타낸다. ‘Int’ 정수형 개체에서 실제 정수 값은 ‘int’ 키워드를 사용하여 구할 수 있다.

Alloy에서 술어는 ‘pred’를 사용하여 표현한다. 예를 들어 “사람들은 최소한 한 마리 이상 물고기를 잡는다”라는 사실을 Alloy 술어를 사용하여 나타내면 다음과 같다 :

```
pred doFishing {
   all p: People | some d: Fish |  p->d in catch
}
```

여기에서 ‘p->d’는 p와 d로 구성된 튜플 (p, d)을 나타낸다.

이와 같이 작성된 Alloy 명세를 분석하기 위해 Alloy 분석기를 이용한다. Alloy 분석기는 Alloy 언어의 의미적 분석을 자동화하는 도구이다. Alloy 분석기는 논리식을 참으로 만드는 모델을 탐색하기 때문에 기본적으로 Alloy 분석기는 모델 탐색기라 말할 수 있다. 모델 탐색의 완전 자동화를 위해 Alloy 분석기는 사용자가 제공하는 일정 범위 안에서 분석을 수행한다.

예를 들어 앞에서 주어진 술어 “doFishing”을 참으로 만드는 인스턴스를 발견하기 위해서 다음과 같은 명령을 수행할 수 있다.

```
run doFishing for exactly 3 People, 4 Fish
```

이 명령어는 People 개체는 정확하게 3개를 사용하고 Fish 개체는 4개를 사용하여 “doFishing” 술어를 참으로 만드는 모델을 탐색하라는 의미이다. 이와 같이 각 시그너쳐에 제공하는 크기를 영역(scope)이라 부른다. 즉 People, Fish의 영역은 각각 3, 4가 되며 이는 People, Fish집합들의 크기라고 생각해도 무방하다. (그림 1)은 위의 명령에 대한 Alloy 분석기의 결과를 보여 준다:

```
sig People extends univ = {People_0, People_1, People_2}
        catch : some models/shape/paper1/Fish =
        {
              People_0 -> Fish_0,
              People_1 -> Fish_1,
              People_2 -> {Fish_2, Fish_3}
        }
sig Fish extends univ = {Fish_0, Fish_1, Fish_2, Fish_3}
len : alloy/lang/Int/Int =
{
        Fish_0 -> 1,   Fish_1 -> 3,   Fish_2 -> 3,   Fish_3 -> 2
}
```

(그림 4) Alloy 명세의 분석 결과(모델)

(그림 4)에서 볼 수 있듯이 Alloy 분석기는 People에서 3개의 개체(i.e., People_0, People_1, Peopl_3) Fish의 4개의 개체(i.e., Fish_0, Fish_1, Fish_2, Fish_3) 그리고 Int(i.e., 1, 2, 3)에서는 3개의 개체를 사용하여 주어진 논리식을 참으로 만드는 바인딩을 보여준다.

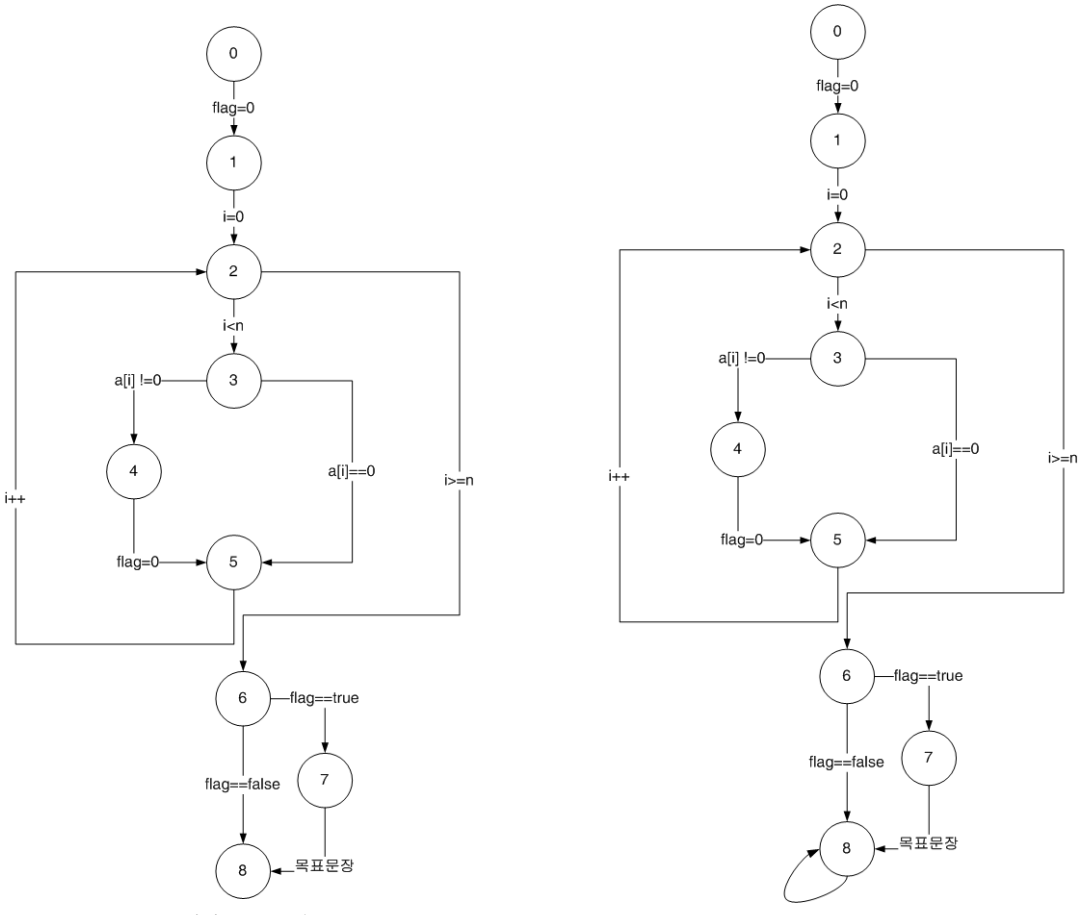
이와 같이 사용자가 제공한 영역 내에서만 탐색을 하기 때문에 설령 모델을 찾을 수 없다할지라도 주어진 논리식을 만족할 수 없다고 결론내릴 수 없다. 이 경우에는 영역의 크기를 더 늘려 탐색을 다시 시도할 수 있지만 보통 작은 영역에서 논리식을 만족할 수 있다는 연구 결과가 있다[14].

3.2 Alloy 기반 테스트 데이터 생성

이 절에서는 테스트 데이터를 자동으로 생성하기 위하여 C 프로그램을 Alloy로 변환하는 방법에 대하여 기술한다. 우선 테스트 대상이 되는 프로그램으로부터 제어 흐름 그래프를 추출하고 이를 상태 전이 모델로 변환하는 방법을 소개하고 상태 전이 모델로 부터 Alloy 명세로 변환하기 위한 방법을 소개한다. 이를 위하여 우선 프로그램 변수들을 Alloy로 표현할 필요가 있다. 현재 이 논문에서는 정수형, 구조체, 포인터만 고려하기 때문에 이들 변수들에 대해 Alloy로 모델링하는 방법에 대해 설명한다.

3.2.1 상태모델링

(그림 5)는 (그림 3)(a)의 'flagProc4'의 제어 흐름 그래프와 상태 전이 모델을 보여준다. 제어 흐름 그래프의 각 노드(node)는 프로그램상의 제어점(control point)을 나타내고 간선(edge)은 프로그램의 각 문장이나 조건식을 나타낸다. 제어 흐름 그래프로부터 상태 전이 모델을 구축하는 작업은 매우 간단하다. 제어 흐름 그래프의 각 노드는 상태 전이 모델의 상태로 변환되고 간선은 상태 간의 전이로 표현된다. 상태간의 전이를 표현할 때 간선의 문장이나 조건이 Alloy 술어로 변환되어 표현된다. 다만 주의할 점은 제어 흐



(a) 제어흐름 그래프　　(b) 상태 전이 모델

(그림 5) 함수 'flagProc4'의 제어 흐름 그래프와 상태 모델

름 그래프에서 진출 간선(out-going edge)이 없는 노드(i.e., 종료 노드)에 해당하는 상태는 자신으로의 전이가 존재한다.

상태 전이 모델에서 상태는 프로그램상의 각 제어점에서 변수가 가질 수 있는 값을 나타낸다. Alloy로 이와 같은 사실을 표현하기 위해서 프로그램의 각 변수에 해당하는 필드들을 갖는 시그너쳐를 정의함으로써 나타낼 수 있다. 예를 들어 (그림 3)(a)의 'flagProc4' 함수는 정수형 배열 a, 정수형 변수, flag, i, n이 선언되어 있으므로 상태는 (그림 6)과 같이 이 변수들에 해당하는 필드를 가지는 시그너쳐를 정의하여 표현된다.

```
abstract sig State {
        a: seq Int,
        n: one Int,
        flag: one Int,
        i : one Int
}
sig  S0, S1, S2, S3, S4, S5, S6,S7, S8 extends State { }
```

(그림 6) 함수 'flagProc4'의 상태 표현

시그너쳐 'State'에서 a, n, flag, i 필드는 상태와 프로그램 변수를 연결하는 관계들이며 정확하게 각 상태에서 각 변수의 인스턴스는 하나만 존재해야 한다는 사실을 한정자 'one'을 사용하여 표현하였다. 이 때 정수형 변수는 Alloy에서 기본적으로 제공하는 'Int'를 이용하여 모델링하고 정수형 배열은 시퀀스를 나타내는 'seq'를 사용하여 모델링하였다. 'State' 시그너쳐를 확장한 S0, ..., S10은 각각 (그림 5)의 상태 0, ..., 8에 해당하는 시그너쳐들이다. 예를 들어, s가 시그너쳐 S3의 한 개체라면 's.i'는 (그림 5)에서 제어가 상태 3 (또는 제어점 3)에 이르렀을 때 변수 i의 값을 나타낸다.

3.2.2 전이 모델링

유한 상태 모델에서 상태 간의 전이는 프로그램의 해당 문장(배정문이나 조건식)을 변환한 Alloy 술어에 의해 표현된다. (그림 7)의 'doAssign'은 (그림 2)의 'flagProc2' 함수에서 'flag=1'를 Alloy 술어로 변환한 예를 보여준다. 예에서 볼 수 있듯이 변환된 Alloy 술어는 선행 상태(s)와 후행상태(s′)를 인자로 가진다.

```
pred doAssign(s, s′: State) {
        int s′.flag = 0                    (1)
        s′.x=s.x                           (2)
}
```

(그림 7) 배정문을 Alloy로 변환한 예

(그림 7)에서 (1)은 배정문에 의해 변수 flag가 갱신되기 때문에 배정문이 실행된 후의 상태, 즉 후행 상태 s′에서 flag의 값은 정수 0과 같아야 한다는 사실을 표현한다. (2)는 해당 배정문에 의해 영향을 받지 않는 변수(들)는 선행 상태와 후행 상태에서 변하지 않아야 한다는 사실을 표현한 것이다. 현재의 경우에는 변수 x가 이에 해당하기 때문에 선행 상태(s)에서의 x 값과 후행상태(s′)에서의 x의 값이 변하지 않아야 한다는 사실을 (2)에서 표현하고 있다, 만약 이를 명시적으로 표현하지 않으면 Alloy 분석기가 변수 x에 임의의 값을 후행 상태에 할당하게 되어 원하지 않은 결과가 발생할 수 있다. 이러한 조건을 프레임 조건(frame condition)이라 한다. (그림 8)은 배정문 "v_1=exp"에 대해 Alloy 술어로 변환하는 일반적인 규칙을 보여준다. 여기에서 v_2, ..., v_n은 해당함수에서 v_1과 함께 사용되는 변수들이다.

조건식도 거의 동일한 방식으로 변환할 수 있다. 배정문을 변환할 때와 다른 점은 조건식에 의해 갱신된 변수가 없다는 점이다. 따라서 해당 프로그램에서 사용되는 모든 변수에 대해 프레임 조건을 기술한다는 점이다. (그림 9)(a)는 일반적으로 v_1 relop exp으로 표현되는 조건식을 Alloy로 변환한 규칙을 보여준다. 여기에서 relop는 관계연산자 {>, <, ==, >=, <=}를 나타낸다. 해당 프로그램에서 사용되는 변수 (v_1, v_2, ..., v_n)들이 조건식을 실행하여도 아무런 영향을 받지 않는다는 사실을 프레임 조건을 통해 표현한 것에 주목할 필요가 있다. (그림 9)(b)는 flagProc2 함수의 조건식 'x==5'를 Alloy로 변환한 예를 보여준다.

이 논문에서는 1차원 정수형 배열도 고려하기 때문에 이를 Alloy로 모델링할 규칙이 필요하다. 3.2.1에서 이미 보았듯이 배열은 Alloy에서 시퀀스로 표현될 수 있다. 만약 배열 a의 i번째 원소, 즉, 'a[i]'는 Alloy에서 시퀀스 a의 i번째 원소로 모델링 되며 이는 '(s.arr)[s.i]' 표현된다. 여기에서 s는 'a[i]'가 사용되는 상태 전이 모델에서의 상태를 나타낸다. 예를 들어 상태 s에서 'a[i]=exp'를 수행한 후의 상태가 s′라면 Alloy로 'int (s′.arr)[s.i]=int exp'로 모델링된다. 물론 이 배정

```
pred ruleForAssignment(s, s′: State) {
        int s′v1 = int s.exp
        s′.v2=s.v2
        ...
        s′.vn=s.vn
}
```

(그림 8) 배정문을 Alloy로 변환하는 규칙

```
pred ruleForCondition(s, s′: State) {
      int s′.v1 relop int s.exp
      s′.v1=s.v1
      ...
      s′.vn=s.vn
}
```

(a) 조건식 변환 규칙

```
pred performComp(s, s′: State) {
      int s′.x = 5
      s′.flag=s.flag
      s′.x=s.x
```

(b)'x==5'를 변환한 예

(그림 9) 조건식을 Alloy로 변환하는 규칙 및 예

식에 의해 여향 받지 않은 모든 변수들에 대한 프레임 조건도 더불어 기술되어야 한다.

배열이 관련된 조건식도 일반 조건식 경우와 같이 동일하게 처리된다. 함수 flagProc4에서 'a[i] !=0'은 다음과 같이 Alloy로 모델링된다. (1)~(3)은 프레임 조건이다.

$$(s'.arr)[s.i] \mathrel{!}= 0$$
$$s'.arr=s.arr \quad (1)$$
$$s'.i=s.i \quad (2)$$
$$s'.flag=s.flag \quad (3)$$

또한 배열의 크기는 시퀀스에서 사용되는 모든 인덱스들의 크기에 대한 제약을 기술하여 표현한다. 예를 들면 배열의 크기가 5인 경우에 다음과 같이 표현한다.

```
fact {
        all s: State | #(s.arr).inds = 5
}
```

즉, 모든 상태에서 배열 'arr'의 크기는 5라는 사실을 표현한 것이다. 여기에서 'inds'는 시퀀스의 인텍스 집합을 계산하는 함수이며 연산자 '#'를 통해 인덱스 집합의 크기를 구한다.

3.2.3 시퀀스 모델링

이 절에서는 유한 상태 모델의 제어 흐름을 Alloy로 표현하는 방법에 대해 기술한다. 만약 S_{j1}, S_{j2}, S_{j3}가 대상 프로그램에서 순차적으로 실행된다고 가정하자. 즉, S_{j1}; S_{j2}; Sj3. 이와 같이 순차적으로 실행되는 문장들은 (그림 10)과 같은 형태로 상태 전이가 이루어진다. 여기에서 P_{i1}, P_{i2}, P_{i3}는 문장 S_{j1}, S_{j2}, S_{j3}를 앞 절에서 설명한 변환 규칙에 따라 변환한 Alloy 술어들이다.

(그림 10)에서 나타난 상태 전이 흐름은 매우 간단하게 다음과 같이 Alloy로 변환될 수 있다.

$$s \text{ in } S_{j0} \Rightarrow P_{i1}[s, s'] \;\&\&\; s' \text{ in } S_{j1}$$
$$s \text{ in } S_{j1} \Rightarrow P_{i2}[s, s'] \;\&\&\; s' \text{ in } S_{j2}$$
$$s \text{ in } S_{j2} \Rightarrow P_{i3}[s, s'] \;\&\&\; s' \text{ in } S_{j3}$$

위 Alloy 식은 만약 현재 상태가 S_{jk}(k=1, 2, 3)라면 다음 상태 S_{jk}으로 가기 위해서는 P_{ik}을 만족해야한다는 사실을 표현하고 있다. 여기에서 =>, &&은 각각 implication, and를 나타내기 위해 Alloy에서 사용하는 논리 연산자이다.

상태 전이는 순차적으로 발생하기도하지만 선택적으로 발생할 수도 있다. 프로그램에 'if 조건문'이 있는 경우에는 현재 상태에서 갈 수 있는 상태가 여러 개 존재한다. 예를 들어 프로그램이'if C_i then S_{j1} else S_j'와 같은 조건문을 포함하고 있다면 이 조건문은 (그림 11)과 같이 표현된다.

(그림 11)에서 P_i는 조건문 C_i에 해당하는 Alloy이 술어이며 Q_i는 조건문 C_i가 만족되지 않는 경우를 모델링한 Alloy 술어이다. 또한 P_{j1}과 P_{j2}는 S_{j1}과 S_{j2}를 Alloy로 변환한 결과이다. (그림 11)에서 묘사된 상태 전이를 Alloy 논리식으로 변환하면 다음과 같다:

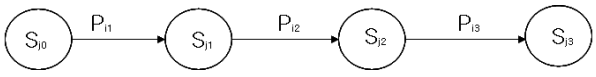
$$s \text{ in } S_i \Rightarrow P_i[s, s'] \;\&\&\; s' \text{ in } Si_1 \;||\; Q_i[s, s'] \;\&\&\; s' \text{ in } Si_2$$
$$s \text{ in } S_{i1} \Rightarrow P_{j1}[s, s'] \;\&\&\; s' \text{ in } S_{j1}$$
$$s \text{ in } S_{i2} \Rightarrow P_{j2}[s, s'] \;\&\&\; s' \text{ in } S_{j2}$$

첫 번째 Alloy 식은 현재 상태가 Si라면 다음 상태는 S_{i1}이거나 S_{i2}이다. 이와 같이 선택적으로 상태 전이가 되는 경우는 논리연산자 '||'(Alloy에서 사용하는 or에 해당하는 논리연산자)를 사용하여 표현한다. 함수 flagProc4에서 조건문 'if (a[i] !=0) flag=0;'을 Alloy로 변환한 결과가 다음과 같다(상태 번호는 (그림 5)(b)에 주어진 상태 전이 모델을 참조).

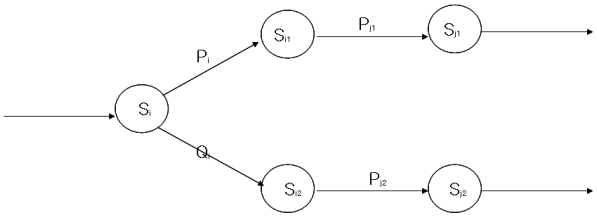
```
s in S3 => doComp34[s, s'] && s' in S4 || doComp35
           [s,s'] && s' in S5
s in S4 => doAssign45[s, s'] && s' in S5

pred doComp34[s, s': State] {
        (s'.arr)[s.i] !=0
        doNothing[s, s']
}
pred doComp35[s, s': State] {
        (s'.arr)[s.i] =0
        doNothing[s, s']
}
pred doAssign45[s, s': State] {
        int s'.flag=0
        s'.arr=s.arr
        s'.i=s.i
}

pred doNothing[s, s': State] {
        s'.arr=s.arr
        s'.i=s.i
        s'.flag=s.flag
  }
```



(그림 10) 순차적 상태 전이의 예



(그림 11) 선택적 상태 전이의 예

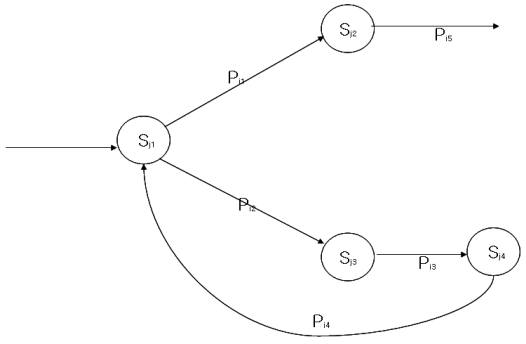
위 변환 결과에서 doComp34는 조건 'a[i]!=0'을 Alloy로 변환한 결과이고 doAssgn45는 배정문 'flag=0'을 변환한 Alloy 술어이다. Alloy 술어 doComp35는 원래의 프로그램 조건 'y<p->data'가 만족되지 않는 경우에 해당하는 Alloy 술어이다.

그런데 여기에서 주목할 점은 doComp34와 doComp35 간의 관계이다. 언뜻 보면 doComp34는 !doComp35(여기에서 ! 은 not의 의미를 갖는 Alloy 논리연산자이다)와 같아야 하지만 실제로는 그렇지 않음을 위 변환 결과에서 확인할 수 있다. 그 이유는 프레임 조건 때문이다. doComp34는 조건 'a[i]!=0'에 해당되는 술어이고 이 조건에 의해 어떤 변수도 영향을 받지 않기 때문에 이를 명시적으로 기술할 필요가 있다. 이 때문에 Alloy 술어 doNothing이 도입되었다. 즉, doNothing은 모든 변수가 상태 전과 후에도 변경되지 않아야한다는 사실을 표현한다. 이는 조건 'a[i]!=0'을 만족하지 않는 경우에도 모든 변수가 영향을 받지 않아야 하므로 단순하게 's in S3 => doComp34[s, s′] && s′ in S4 || !doComp35[s, s′] && s' in S5라고 변환할 수 없다. 만약 이와 같이 변환하였다면 프레임 조건을 기술한 Alloy 술어 doNothing[s, s′]도 !doNothing[s, s′]로 변환될 것이다. 이는 어떤 프로그램 변수는 조건문의 실행으로 인해 영향을 받을 수도 있다는 점을 의미하게 되므로 잘못된 결과를 초래할 수 있다.

(그림 12)는 이전에 이미 방문했던 상태로 다시 전이가 발생하는 상황을 묘사한 것이다. 이러한 경우는 프로그램에 반복문이 존재할 때 나타난다. (그림 12)를 Alloy로 변환하는 것은 (그림 11)에서 설명한 것과 큰 차이가 없다. 만약 현재 상태가 S_{j4}라면 다음 상태는 이미 방문한 상태 S_{j1}이어야 하며 전이가 일어나기 위해서는 P_{i4}를 만족해야 한다. (그림 12)의 상태 전이 모델을 Alloy로 표현하면 다음과 같다:

s in S_{j1} => P_{i1}[s, s′] && s′ in S_{j2} || P_{i2}[s, s′] && s′ in S_{j3}
s in S_{j3} => P_{i3}[s, s′] && s′ in S_{j4}
s in S_{j4} => P_{i4}[s, s′] && s′ in **S_{j1}**
s in S_{j2} => P_{i5}[s, s′] && s′ in ...

여기에서 한 가지 주목할 점이 있다. 반복되는 구간내의 상태들은 최소한 반복 회수만큼의 개체를 자신의 원소로 가져야 한다. 예를 들면 (그림 12)에서 <S_{j1}, P_{j2}, S_{j3}, P_{j3}, S_{j4}, P_{j4}>가 최소한 n번 반복된다고 가정할 때 상태 S_{j1}, S_{j3}, S_{j4}는 각각 최소한 n개의 개체들을 포함해야 한다. 그러나 다행히도 일일이 각 상태에 대해 몇 개의 개체가 할당되어야 하는지 알 필요가 없다. 다만 상태 전이 모델에 나타난 모든 상태들이 필요로 하는 전체 개수만을 기술하면 된다. Alloy 분석기의 SAT 해결도구가 상태 전이 모델로 표현된 논리식을 만족시킬 수 있도록 자동적으로 각 상태에 적절한 개수의 개체들을 분배하기 때문이다. 이러한 점은 SAT를 통해 얻을 수 있는 중요한 이점이다. 이에 반해 경로 기반 테스트에서는 완전한 경로를 명시해야 하므로 반복문이 수행되는 회수를 제공할 필요가 있다.



(그림 12) 반복의 예

3.3 목표 문장의 표현

프로그램의 특정 문장, 즉 목표 문장을 실행하는 테스트 데이터를 생성하기 위해서는 목표 문장을 Alloy 명세에 기술하는 것이 필요하다. 예를 들면 (그림 3)(a)에서 목표 문장을 실행하는 테스트 데이터를 생성하기 위해서는 (그림 13)과 같이 S0를 초기 상태로 주고 S7을 목적 상태로 명시하면 된다. 여기에서 S0는 (그림 5)(b)의 0에 해당하고 S7은 7에 해당하는 상태이다.

```
sig S0, S1, S2, ... extends State {}
fact {
        so/first()  in S0
}
pred testIt() {
        so/last() in S7
}

run testIt for 6 State, 5 Integer, 5 Int
```

(그림 13) 초기 상태 및 목적 상태를 Alloy로 표현한 예

S0를 'fact'를 사용하여 초기 상태임을 기술하였고 S7은 'pred'를 사용하여 기술하였다. 그 이유는 초기 상태는 항상 변함이 없지만 목적 상태는 달라질 수 있기 때문이다. (그림 13)과 같이 명세를 주었을 때 Alloy 분석기는 여러 가지 정보를 생성하며 특히 S7에 도달하기 위한 경로와 입력 값 정보를 계산한다.

4. 실험 결과

이 절에서는 이 논문에서 제안한 SAT 기반 테스트 데이터 생성방법의 효용성을 보이기 위해 (그림 3)(a)에 주어진 함수 "flagProc4"에 대해 랜덤 테스트 데이터 생성(간단히 랜덤 테스팅)과 비교하여 실험을 수행한 결과를 기술한다. 실험에 사용된 함수는 기존의 방법들이 효과적으로 테스트 데이터를 생성하지 못한 대표적인 프로그램이며 비교를 위하여 랜덤 테스팅을 선택한 이유는 1장에서 언급했듯이 기존의 테스트 데이터 생성 방법들이 플래그 변수가 있는 프

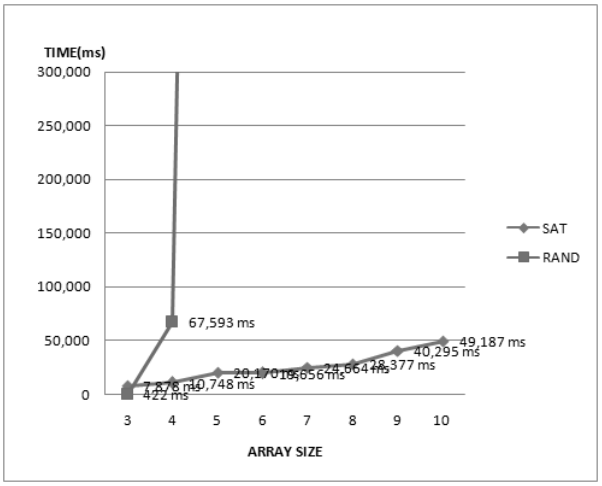
로그램을 테스트하는 경우에는 랜덤 테스팅을 수행한 경우와 동일하기 때문이다.

함수 “flagProc4”는 정수형 배열을 입력으로 받는다. 7번 문장이 목표 문장이며 이를 실행하기 위해서는 입력 배열 값들이 모두 0이 되는 경우뿐이다. 이는 플래그 변수 ‘flag’의 값을 통해 알 수 있다. 플래그 변수 ‘flag’는 참(즉, 1)으로 초기화 되어 있으며 반복문에서 배열의 어느 한 값이라도 0이 아니라면 ‘flag’가 0이 되기 때문에 6번 조건식이 거짓이 되어 7번 문장을 실행하지 못한다.

실험에서는 입력 정수의 값을 7비트 즉 (-64 ~ 63)의 범위로 제한하였으며 배열의 크기도 3부터 10까지로 하였다. 따라서 최소 입력 공간의 크기는 배열의 크기가 3일 때이며 이 경우에는 128^3이다. 반면에 최대 입력 공간의 크기는 배열의 크기가 10일 때 $128^{10}(\approx 10^{20})$이 된다. 7번 문장을 실행할 수 있는 입력 값은 배열의 모든 원소가 0(a[0]=a[1]=...=a[8]=a[9]=0)인 경우 밖에 없기 때문에 이를 발견할 확률은 $1/10^{20}$으로 매우 희박하다.

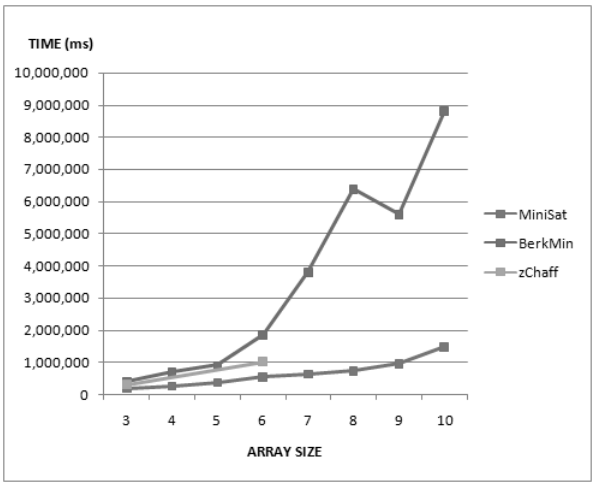
<표 1>은 랜덤 테스팅(RAND)과 이 논문에서 제안한 SAT 기반 테스팅(SAT)을 배열의 크기에 따라 원하는 테스트 데이터를 생성한 시간을 측정한 결과이며 각 배열 크기에 대해 5번씩 수행하여 평균값을 기록한 것이다. 표에서 ‘-’는 1시간이 경과하여도 원하는 테스트 데이터를 찾지 못한 경우를 나타낸다. (그림 14)는 <표 1>의 결과를 그래프로 표현한 것이다.

<표 1>에서 볼 수 있듯이 입력 탐색 공간이 비교적 작은 경우(배열의 크기가 3일 때)에는 랜덤 테스팅이 보다 빠르게 원하는 테스트 데이터를 생성하였지만 배열의 크기가 6부터(즉, 탐색공간의 크기가 $128^6 \approx 10^{12}$)는 아예 탐색에 실해한 것을 볼 수 있다. 반면에 SAT 기반 테스트 데이터 생성 방법은 모든 경우에서 원하는 테스트 데이터를 생성하였고 배열 크기가 3인 경우를 제외한 나머지 모든 경우에서 랜덤 테스팅보다는 빠르게 생성한 것을 볼 수 있다.



(그림 14) 랜덤 테스팅과의 비교 그래프

위 결과는 SAT 해결기로 MiniSat를 사용한 것이다. SAT 해결기에 따른 성능에 변화가 있는지를 알아보기 위해 Alloy 분석기에서 지원하는 BerkMin과 zChaff를 사용하여 테스트 데이터를 생성하여 성능 비교를 하였다. (그림 15)는 3개의 SAT 해결기-MiniSat, BerkMin, zChaff-를 사용한 테스트 데이터 생성 시간을 비교한 그래프이다. 그래프에서 볼 수 있듯이 플래그 문제에는 MiniSat가 가장 좋은 결과를 보였다. BerkMin은 MiniSat 보다는 원하는 테스트 데이터를 생성하는 시간이 보다 많이 소요 되었지만 배열의 크기가 10일 때 까지 모든 경우에 원하는 테스트 데이터를 생성하였다. 반면에 zChaff는 배열의 크기가 6이 넘어가는 경우에는 원하는 테스트 데이터를 주어진 시간(1시간)에 생성하지 못하였다. 물론 이 결과에 대한 원인은 명확하지 않고 일반화 할 수 없지만 플래그 문제의 해결에는 MiniSat가 다른 SAT 해결기에 비해 탁월한 성능을 보인 것은 확실해 보인다.



(그림 15) SAT 해결기에 따른 성능 비교

5. 결 론

이 논문에서는 플래그 문제를 효과적으로 다루기 위한 SAT 기반 테스트 데이터 생성 방법을 소개하였다. 이 방법은 기존의 방법과는 다르다. 기존의 방법은 플래그 변수가

〈표 1〉 랜덤 테스팅과의 비교(시간 단위는 ms)

	배열 크기							
	3	4	5	6	7	8	9	10
SAT	7,878	10,748	20,170	19,656	24,664	28,377	40,295	49,187
RAND	422	67,593	2,247,860	-	-	-	-	-

있는 프로그램의 테스트 용이성이 낮기 때문에 탐침을 하거나 일련의 규칙을 통해 프로그램 변환을 한 후에 테스트 데이터를 생성하는 방식을 취한다. 그러나 이러한 방식은 이 논문에서 사용했던 프로그램과 같이 반복문안에서 플래그 변수를 사용하는 경우에 랜덤 테스팅과 동일하게 수행되고 프로그램의 의미와는 다른 문장들이 다만 테스트 용이성을 높일 목적으로 삽입된다. 따라서 변환된 프로그램은 원래의 프로그램과는 기능적으로 다른 프로그램이다. 반면에 SAT 기반 테스트 데이터 생성 방법은 프로그램을 추가된 코드 없이 그대로 변환되기 때문에 테스트 데이터 생성 할 목적뿐만 아니라 특정 프로그램의 속성을 검증할 목적으로도 사용할 수 있다.

SAT 기반 테스트 데이터 생성 방법은 기존의 테스트 데이터 자동 생성 방법보다는 플래그 문제 해결에 보다 효과적인 테스트 데이터를 생성하지만 보완할 점도 있다. 우선 현재 수작업을 통해 프로그램을 Alloy로 변환한다. 따라서 변환 과정이 번거럽고 변환 중에 오류가 도입될 여지가 있기 때문에 프로그램을 자동으로 Alloy 명세로 변환하는 도구의 개발이 요구된다. 현재 이 도구를 개발 중이다. 향후의 연구로 자동화 도구를 구현하여 보다 다양한 프로그램에 대해 SAT 기반 테스트 데이터 생성 방식에 대해 평가할 필요가 있다. 두 번째로 현재 이 논문에서는 플래그 변수를 지닌 프로그램에 대한 테스트 데이터 생성을 검증하였으나 일반적인 프로그램에 대해 얼마나 효과적으로 테스트 데이터를 생성할 수 있는지에 대한 연구가 필요하다. 아직까지 테스트 데이터 자동 생성에 대한 연구가 미진하기 때문에 SAT 기반 테스트 데이터 생성 방법은 기존 방법의 대안 또는 보완 수단이 될 수 있는지에 대한 보다 심도 깊은 검증이 필요하다. 마지막으로 이 논문에서 기술한 방법은 단위 테스트만을 지원한다. 물론 현재 대부분의 테스트 데이터를 자동으로 생성하는 방법들이 단위 모듈만을 지원하는 것도 사실이지만 보다 실효성이 있는 테스트 데이터를 생성하기 위해서는 모듈이 하나 이상 통합되어 있는 경우도 지원이 되어야 할 것이다.

참 고 문 헌

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] J. Edvardsson, “A Survey on Automatic Test Data Generation”, In *Proc. the Second Conf. on Computer Science and Engineering*, pp.21-28, 1999.

[6] M. W. Moskewicz, Y. Zhao, L. Zhang, and Malik, “Chaff: Engineering an Efficient SAT solver”, In *Proc. 38th Design Automation Conference(DAC)*, pp.530-535, 2001.

[7] E. Goldgerg, and Y. Nivikov, “BerkMin: A Fast and Robust SAT solver”, In *DATE*, pp.142-149, 2002.

[8] 정인상, “SAT에 기반한 포인터가 있는 프로그램을 위한 목적 지향 테스트 데이터 생성”, 인터넷정보학회논문지, 제9권 2호, pp.89-105, 2008.

[9] D. Jackson, “Alloy: A light weight object modeling notation”, Technical Report 797, MIT Lab for Computer Science, Feb., 2000.

[10] D. Jackson, Alloy 3.0 Reference Manual, http://alloy.mit.edu, 2004.

[11] D. Jackson., Alloy 4.0 Quickstart guide, http://alloy.mit.edu, 2007.

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

[13] 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)*, pp. 108-118, Boston, USA, July, 2004.

[14] A. Andoni, D. Daniliuc, S. Khurshid, and D. Marinov, “Evaluating the Small Scope Hypothesis”, Technical Report 921, MIT Lab for Computer Science, Feb., 2003.

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