# Alloy 명세 기반 자동 테스트 데이터 생성 기법

정 인 상†


## 요 약

일반적으로 테스트 데이터 생성 방법들은 테스트 데이터를 자동으로 생성하기 위해서 완전한 프로그램 경로를 기술할 것을 요구한다. 이 논문에서는 프로그램 경로를 완전하게 명시하지 않아도 테스트 데이터를 자동으로 생성하는 새로운 방법을 제안한다. 이를 위해 이 논문에서는 테스트 대상 프로그램을 1차 관계 논리 언어인 Alloy로 변환하고 Alloy 분석기를 통하여 테스트 데이터를 생성하는 방법을 제안한다. 제안된 방법은 사용자로 하여금 프로그램 경로를 선택하도록 하는 부담을 덜어줄 뿐만 아니라 다양한 테스트 적합성 기준에 따라 테스트 데이터를 생성하는 일을 용이하게 한다. 간단하지만 설명에 도움이 될 수 있는 예들을 통하여 제안한 방법에 대해 설명한다.


**키워드 : 프로그램 테스팅, 자동 테스트 데이터 생성, Alloy**

# An Alloy Specification Based Automated Test Data Generation Technique

In-Sang Chung†


## ABSTRACT

In general, test data generation techniques require the specification of an entire program path for automated test data generation. This paper presents a new way for generating test data automatically even without specifying a program path completely. For the ends, this paper presents a technique for transforming a program under test into Alloy which is the first-order relational logic and then producing test data via Alloy analyzer. The proposed method reduces the burden of selecting a program path and also makes it easy to generate test data according to various test adequacy criteria. This paper illustrates the proposed method through simple, but illustrative examples.


**Key Words : Program Testing, Automated Test Data Generation, Alloy**

## 1. 서 론

소프트웨어가 실생활과 밀접한 관계를 유지함에 따라 소프트웨어의 신뢰성과 같은 품질 제고가 주요 관심사가 되고 있으며 프로그램의 신뢰성을 높이는 여러 방법들 중에서 프로그램 테스팅은 아직까지 가장 현실적인 방법으로 널리 사용되고 있다. 그러나 프로그램 테스팅이 많은 자원과 비용을 소모하는 것도 사실이다.

프로그램을 테스트하는 절차는 보통 다음 단계들로 구성된다. 우선 사용할 테스트 적합성 기준(test adequacy criterion)을 선정한다. 현실적으로 프로그램의 모든 입력 데이터를 사용하여 프로그램을 테스트할 수 없기 때문에 테스트 데이터를 선정 또는 생성하는 기준을 정하여 이 기준을 만족하는 입력 데이터만을 테스트 데이터로 사용한다. 적합성 기준이 선정되었으면 그 기준을 만족하는 입력 값들을 생성해야 한다. 이 단계가 바로 테스트 데이터를 생성하는 단계이며 프로그램 테스트 단계 중에서 가장 노력이 많이 들고 자동화하기 어렵다[1]. 지금까지 프로그램 테스트에 관한 연구는 효과적인 테스트 데이터를 선정하기 위하여 적합성 기준 개발에 많은 노력을 했지만 상대적으로 테스트 데이터를 적합성 기준에 따라 자동으로 생성하는 기술에 관한 연구는 미진한 것이 사실이다. 마지막 단계에서는 생성된 테스트 데이터를 가지고 실제 프로그램을 실행하여 그 결과를 분석한다. 만약 오류가 발견되었다면 오류의 원인을 찾아 제거한다. 프로그램을 테스트하는 과정에서 가장 어려운 문제는 테스트 결과가 올바른지를 판단하는 문제이다. 테스트 오라클(test oracle)은 프로그램의 실제 실행결과가 올바른 결과인지를 판단하는 메커니즘이다[2]. 프로그램을 테스트할 때 테스트 오라클 역할은 일반적으로 수작업으로 수행한다. 만약 테스트 오라클을 자동화하고 싶다면 테스트 대상이 되는 프로그램을 개발한 알고리즘과는 다른 알고리즘을 사용하여 테스트 오라클을 만들 필요가 있다.

소프트웨어 테스트 비용을 줄이기 위해서 위에서 언급한 단계들 중에서 두 번째 단계, 즉 테스트 데이터를 생성하는 단계를 자동화하는 것이 효과적이다. 일반적으로 테스트 데이터를 자동으로 생성하는 문제는 주어진 프로그램 경로를 실행할 수 있는 입력 값을 찾아내는 문제로 변환될 수 있다. 이

※ 본 연구는 2006학년도 한성대학교 교내연구비 지원과제임
† 정 회 원 : 한성대학교 컴퓨터공학과 교수
논문접수 : 2006년 9월 1일, 심사완료 : 2007년 3월 12일

러한 테스트 데이터 생성 방법을 경로 지향적 테스트 데이터 생성(path-oriented test data generation) 방법이라고 한다.

경로 지향적 테스트 데이터 생성 방법은 테스트 데이터를 생성하기 전에 테스트하고자 하는 프로그램 경로를 선택하고 선택된 프로그램 경로를 실행하는 입력 값을 식별하는 방법이다. 이 방법은 기존의 많은 방법들이 사용한 방법이다. 그러나 이 방법은 사용자로 하여금 테스트할 프로그램 경로를 선택하도록 하는 부담이 있다. 또한 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[3, 4].

경로 지향적 방법과는 달리 목적 지향적 (goal-oriented) 테스트 데이터 생성 방법은 특정 프로그램 경로를 주는 대신에 프로그램 상의 한 제어점(control point)을 주고 이를 실행할 수 있는 입력 값을 생성하는 방법이다[3, 4]. 지금까지 대부분의 목적 지향 테스트 데이터 생성 방법은 테스트 데이터를 생성하기 위해 프로그램을 탐침(instrumentation)하여 프로그램의 실행을 모니터링하는 동적 기법에 바탕을 두고 있다. 따라서 내장형 시스템과 같이 절대적으로 메모리와 같은 자원의 제약이 있는 환경에서는 사용하기 힘든 단점이 있다.

이 논문에서 제안하는 방법은 기본적으로 목적 지향적 테스트 데이터 생성 방법이다. 그러나 프로그램을 실제 실행하여 테스트 데이터를 생성하는 일반적인 목적 지향적 방법과는 달리 프로그램의 실행을 요구하지 않는 정적 기법이다. 따라서 제안된 방법은 프로그램의 실행 과정을 모니터링하기 위해 프로그램에 탐침을 하지 않는다. 또한 일반적인 목적 지향적 방법은 하나만의 제어점을 제공받는 것에 반해 제안된 방법은 하나 이상의 제어점을 입력받아 그들을 실행할 수 있는 테스트 데이터를 생성할 수 있다. 이러한 특징은 다양한 테스트 적합성 기준에 따라 테스트 데이터를 용이하게 생성할 수 있게 한다. 예를 들면, 자료 흐름에 기반을 테스트 적합성 기준(test adequacy criteria based on data flow)[5]들은 변수가 정의 되는 문장과 변수가 사용되는 문장, 즉 두 개의 프로그램 제어점을 요구하기 때문에 일반적인 목적 지향적 테스트 데이터 생성 방법을 직접적으로 사용할 수 없다.

이 논문에서 제안하는 방법은 입력 프로그램을 Alloy 명세로 변환한다. Alloy는 1차 관계 논리(first-order relational logic)에 기반 한 모델링 언어이다[6, 7]. 입력 프로그램을 Alloy 명세로 변환하기 위해서는 우선 프로그램으로부터 계산 그래프(computation graph)를 추출하고 이를 유한 상태 모델로 변환하는 과정을 거친다. 계산 그래프는 프로그램에서 반복문이 나타나는 경우 이를 특정 횟수만큼 펼친 제어 흐름 그래프이다[8]. 프로그램으로부터 계산 그래프를 구성한 후 목적 노드 즉, 특정 프로그램 제어점을 실행할 수 있는 테스트 데이터를 탐색할 수 있도록 유한 상태 모델(finite state model)로 변환한다. 이 상태 모델로부터 Alloy 분석기를 통해 목적 노드에 도달 가능한 프로그램 경로 및 이를 실행할 수 있는 입력 값을 생성한다.

이 논문은 다음과 같이 구성된다. 2장에서는 기존의 테스트 데이터 생성 방법들과 Alloy에 대해 소개한다. 3장에서는 프로그램을 Alloy로 변환하는 과정에 대해 기술한다. 특히 각 프로그램 문장의 타입에 대해 Alloy 논리식(formulae)으로 변환하는 방법에 대해 기술한다. 4장에서는 테스트 적합성 기준에 따른 테스트 데이터 생성 방법에 대해 기술한다. 5장에서는 결론 및 향후 연구에 관해 기술한다.

## 2. 관련 연구

### 2.1 테스트 데이터 생성

테스트 데이터를 생성하는 대표적인 방법으로 심볼릭 실행(symbolic execution)을 들 수 있다. 심볼릭 실행 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 실행을 이용하였다[12, 13].

그러나 최근에 제안된 많은 테스트 데이터 생성 방법들은 함수 최소화 기법에 기반을 두고 있다[14, 15]. 함수 최소화 기법은 실제 입력 값을 사용하여 프로그램을 실행하는 대표적인 동적 기반 테스트 데이터 생성 기술이다. 예를 들면 TESTGEN[15]이나 ADTEST[14]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>=10"이 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>=10"을 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 만난다면 현재의 입력 값이 주어진 프로그램 경로를 따라서 적절한 분기가 일어나는 경우에는 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 현재의 입력 값을 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 되풀이 한다.

그러나 정적 기법이 프로그램의 실행을 모니터링하고 제어하는 어떤 수단도 필요하지 않는데 반해 동적 기반 테스트 데이터 생성 방법은 프로그램을 원하는 경로(함수의 값

이 최소가 되는 방향)를 따라 수행할 수 있도록 탐침할 필요가 있다. 또한 원하는 테스트 데이터를 생성하기 위해 제공되는 초기 값에 따라 테스트 데이터 생성 비용이 영향을 받을 수 있다.

국내에서는 테스트 데이터를 자동생성하기 위한 정적 프로그램 테스트 도구인 'SGEN'이 개발되었다 [17, 18]. 'SGEN'은 프로그램 경로가 주어지면 그 경로를 실행할 수 있는 입력 값을 찾는 것을 목적으로 한다. 다른 일반적인 테스트 데이터 자동 생성 도구와는 달리 'SGEN'은 포인터가 있는 프로그램도 처리할 수 있도록 개발되었다. 주어진 경로가 포인터가 있는 경우에는 해당 경로를 실행하기 위해서는 보통 입력 값이 리스트나 트리와 같은 2차원적인 자료구조를 가져야 하는 경우가 일반적이다. 'SGEN'은 주어진 경로를 실행할 수 있는 자료구조가 존재한다면 이를 자동으로 생성할 수 있다.

위에서 언급한 방법들은 테스트 데이터를 생성하기 위해 완전한 프로그램 경로를 사용자가 제공하여야 한다. 이러한 방식은 사용자가 일일이 프로그램 경로를 명시해야 하는 불편이 있을 뿐만 아니라 주어진 경로가 실행 불가능한 경우에 테스트 데이터를 생성하는 노력 및 비용의 증가를 가져온다.

이와는 대조적으로 Roger 등[16]은 특정 프로그램 문장을 명시하면 사용자가 완전한 프로그램 경로를 제공하지 않아도 주어진 프로그램 문장(i.e., 목적 노드)에 도달할 수 있는 테스트 데이터를 생성하는 방법을 제안하였다. 이 방법은 프로그램 상의 분기들을 목적 노드에 관해 다음과 같이 세 가지로 분류하였다:

- 임계 분기(critical branch): 프로그램이 이 분기를 따라 실행되면 목적 노드에 도달할 수 있는 가능성이 전혀 없다.
- 반임계 분기(semi-critical branch): 프로그램이 이 분기를 따라 실행되면 목적 노드에는 도달할 수는 있지만 제어흐름 그래프 상에서 반복문의 역 간선(backward edge)을 따라 도달할 수 있다.
- 비필수적 분기(non-essential branch): 임계 분기도 아니고 반임계 분기도 아닌 분기를 비필수적 분기라 한다.

이 방법은 우선 무작위로 생성된 입력 값으로 프로그램을 실행시킨다. 만약 실행과정에서 임계 분기를 따라 실행되고 있으면 실행을 중단하고 앞서 언급한 함수 최소화 기법을 이용하여 입력 값을 수정하여 다시 프로그램을 실행한다. 만약 반임계 분기를 따라 실행되고 있다면 함수 최소화 기법을 이용하여 새로운 입력 값을 찾는다. 그러나 이 경우에 적절한 입력 값을 찾지 못했다할지라도 프로그램의 실행을 중단하지 않고 프로그램의 실행을 계속한다. 비필수적 분기는 말 그대로 목적 노드에 도달할지를 결정하는데 영향을 주지 않기 때문에 실행을 중단하지 않고 계속 실행하게 한다.

그러나 이 방법은 앞서 설명한 동적 기반에 방식을 둔 테스트 데이터 생성 방법과 같이 프로그램의 실행을 모니터링하고 제어하기 위해 탐침을 할 필요가 있으며 분기 적합성 기준이나 자료 흐름 적합성 기준과 같이 목적 노드가 두 개 이상인 경우에는 적용할 수 없다.

### 2.2 Alloy

Alloy는 MIT의 Daniel Jackson과 그의 동료들에 의해 개발된 1차 관계 논리(first-order relational logic)에 기반을 둔 모델링 언어이다[6]. 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 인스턴스(또는 모델)를 탐색하는 도구이다. 여기에서 인스턴스는 명세에 있는 논리식들을 참이 되도록 명세에 나타난 집합들과 관계들에게 구체적인 값들을 바인딩하는 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 모델(model) 또는 인스턴스(instance)을 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, 4 Int
```

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

그림 1에서 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.

```
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 1) Alloy 명세의 분석 결과(모델)

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[9].

## 3. 테스트 데이터 생성을 위한 Alloy로의 변환

이 장에서는 테스트 데이터를 자동으로 생성하기 위하여 C 프로그램을 Alloy로 변환하는 방법에 대하여 기술한다. 우선 테스트 대상이 되는 프로그램으로부터 일종의 제어 흐름 그래프인 계산 그래프(computation graph)를 추출하고 이를 유한 상태 모델로 변환하는 방법을 소개한다. 또한, Alloy 분석기를 통해 테스트 데이터를 생성하기 위해 유한 상태 모델로 부터 Alloy 명세로 변환하기 위한 방법을 기술한다.

### 3.1 계산 그래프의 추출

계산 그래프는 비순환 방향 그래프(Directed Acyclic Graph, DAG)이다[8]. 계산 그래프의 각 노드(node)는 프로그램상의 제어점(control point)을 나타내고 간선(edge)은 프로그램의 각 문장이나 조건식을 나타낸다. 계산 그래프가 일반적인 제어 흐름 그래프와 다른 점은 프로그램이 반복문을 포함한 경우에 주어진 일정한 횟수만큼 반복문을 펼친(unfold)다는 점이다. 예를 들면, "a; while (p) s; b"로부터 한 번만 반복문을 펼친 계산 그래프는 'a; if (p) s; b'의 제어 흐름 그래프와 동일하다. 따라서 프로그램이 만약 반복문을 포함하고 있지 않다면 제어 흐름 그래프와 같게 된다.

계산 그래프는 반복문을 포함하는 경우 주어진 수만큼만 반복문을 펼치기 때문에 모든 가능한 실행 경로를 표현하지 않는다. 그러나 현실적으로 테스트 대상이 되는 프로그램의 모든 실행 가능한 경로들을 시험하는 것이 가능하지 못하기 때문에 이러한 접근 방식은 프로그램의 오류를 간과할 수 있는 단점이 있지만 현실적인 프로그램 검증 수단이 될 수 있다. 이와는 대조적으로 반복문을 직접적으로 처리하는 정적 분석 방법들은 실제 일어날 가능성이 없는 프로그램 오류도 프로그램 오류로 보고할 수 있다(i.e., false alarm).

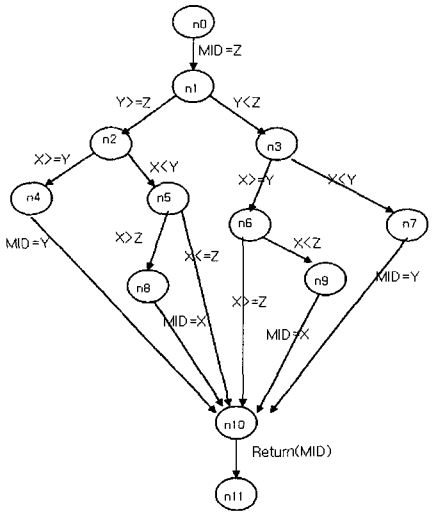
그림 2는 세 개의 정수 값들(i.e., x, y, z)을 입력으로 받아 중간 값을 구하는 프로그램[10]과 계산 그래프를 보여준다. 프로그램이 반복문을 포함하지 않기 때문에 계산 그래프는 일반적인 제어 흐름 그래프와 다르지 않다는 것을 알 수 있다.

### 3.2 유한 상태 모델

프로그램으로부터 계산 그래프를 추출한 후에 이를 바탕으로 유한 상태 모델을 구축한다. 계산 그래프의 노드는 유한 상태 모델의 상태로 변환되고 간선은 상태간의 전이로 변환된다. 다만 주의할 점은 계산 그래프에서 진출 간선(out-going

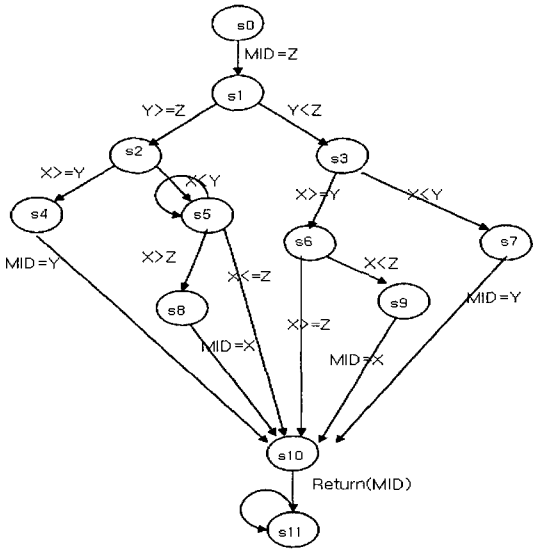
```
int mid(int x, int y, int z) {
    int mid
    mid = z;
    if (y<z) {
            if (x<y)
                    mid = y;
            else if (x<z)
                    mid = x;
    } else {
            if (x>=y)
                    mid = y;
            else if (x>z)
                    mid = x;
 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

(a) 예제 프로그램




(b) 계산 그래프

(그림 2) 예제 프로그램과 계산 그래프




(그림 3) 유한 상태 모델

edge)이 없는 노드(i.e., 종료 노드)에 해당하는 상태와 사용자가 최종적으로 실행하고 싶은 노드(i.e., 목적 노드)에 해당하는 상태는 자신으로의 전이가 존재한다. 물론 사용자가 목적 노드이외에 목적 노드를 실행하기 이전에 반드시 실행해야할 노드들을 명시할 수 있지만 유한 상태 모델에서는 목적 노드에 대응되는 상태, 즉 목적 상태와 종료 노드에 대응되는 종료 상태만 자신으로의 전이가 있다.

그림 3에서 s11은 계산 그래프에서 진출 간선이 없는 n11에 해당되는 종료 상태를 나타낸다. 만약 사용자가 그림 2에서 n5가 나타내는 프로그램 제어점을 반드시 실행하는 테스트 데이터를 생성하기를 원한다면 n5에 대응되는 s5가 목적 상태가 된다. 따라서 그림 3에서 볼 수 있듯이 s5와 s11은 자신으로의 전이를 갖는다. 자신으로의 전이를 만드는 구체적인 이유에 대해서는 3.3절에서 기술한다.

### 3.3 Alloy로의 변환 규칙

유한 상태 모델에서 상태 간의 전이는, 프로그램의 각 문장이나 조건식이 Alloy 술어로 변환되어 표현된다. 이 절에서는 유한 상태 모델를 구성하는 상태 및 전이를 Alloy로 표현하는 규칙에 대해서 구체적으로 기술한다.

#### 3.3.1 상태 모델링

이 논문에서는 SSA(Static Single Assignment) 폼[11]을 기반으로 프로그램의 문장이나 조건식을 Alloy로 변환한다. SSA 폼에서는 모든 변수는 유일한 정의를 가지고 있으며 각 변수의 참조는 해당 변수를 정의하는 문장에 의해 도달이 가능하다는 특징을 지니고 있다. 따라서 각 프로그램의 변수를 논리 변수(logical variable)로 처리하는 것을 가능하게 해주기 때문에 프로그램에 있는 각 문장이나 조건식을 제약식(등식이나 부등식)으로 다룰 수 있다. 예를 들어 그림 4는 일련의 프로그램 문장들을 SSA 폼으로 변환한 결과를 보여준다.

그림 4(b)에서 볼 수 있듯이 변수가 새로 정의될 때 마다 변수의 첨자가 1만큼 증가됨을 볼 수 있다. 이는 배정문(assignment statement)에 의해 변수가 정의되면 새로운 값을 가지게 되므로 제약식(equality)으로 취급하기 위해서는 갱신된 변수가 다른 변수 이름을 사용할 필요가 있다. 이는 배정문에 의해 갱신되기 전의 변수와 갱신된 후의 변수를 다르게 취급함으로써 배정문을 등식으로 다룰 수 있게 하기 위함이다. 그림 4(a)의 첫 번째와 세 번째에 배정문에서 정의된 변수 X가 SSA 폼에서는 다른 변수로 취급됨을 알 수 있다. 즉, 변수가 갱신될 때마다 갱신된 변수의 새로운 인스턴스가 생성된다고 볼 수 있다. 이러한 관점에서 이 논문에서는 각각의 프로그램 변수를 집합으로 간주하여 변수가 갱신될 때 마다 새로운 원소를 사용하는 방식으로 프로그램 변수를 Alloy로 모델링한다.

그림 5는 그림 2에 주어진 예제 프로그램의 변수 x, y, z,

```
X=10;                    X1=10;
Y=20;                    Y1=20;
X=X+Y+30;                X2=X1+Y1+30;
```

(a) 예제 프로그램 (b) 변환된 SSA 폼

(그림 4) SSA 폼 변환 예

```
abstract sig Integer {}

sig X, Y, Z, MID extends Integer {
        val: Int
}
```

(그림 5) Alloy로 모델링한 프로그램 변수 예

```
sig State {
        x: one X,
        y: one Y,
        z: one Z,
        mid: one MID
}
```

(그림 6) Alloy로 모델링한 상태 정보

mid를 Alloy로 모델링한 것이다. 여기에서 볼 수 있듯이 시그너쳐 확장을 통해 X, Y, Z, MID는 Integer의 공통 개체들이 없는 부분집합들로 선언하며 변수가 배정문에 의해 갱신될 때마다 해당 집합으로부터 새로운 개체를 선택하여 사용하도록 한다. Integer의 val 필드는 x, y, x, mid가 정수형 변수이기 때문에 변수가 갖는 값을 모델링하기 위해 도입되었다. 예를 들어, 변수 x의 값이 10과 같다는 사실은 'int x.val = 10'으로 나타낼 수 있다.

변수를 이와 같이 모델링한다면 유한 상태 모델의 각 상태는 현재 변수의 인스턴스에 대한 정보를 가지고 있어야 한다. 그림 6은 유한 상태 모델의 상태를 Alloy로 표현한 것이다. State에서 x, y, z, mid 필드는 상태와 프로그램 변수를 연결하는 관계들이며 정확하게 각 상태에서 각 변수의 인스턴스는 하나만 존재해야 한다는 사실을 한정자 'one'을 사용하여 표현하였다. 예를 들면, 'int s.x.val=10'는 상태 s에서 x의 값은 10과 같다는 사실을 표현한다.

### 3.3.2 전이 모델링

유한 상태 모델에서 상태 간의 전이는 프로그램의 해당 문

```
pred doAssign1(s, s': State) {
        some mid' | s.mid !=mid' && mid'=s'.mid -------(1)
        int s'.mid.val = int s.z.val ----------------------(2)
        s'.x=s.x      ----------------------------------(3)
        s'.y=s.y      ----------------------------------(4)
        s'.z=s.z      ----------------------------------(5)
}
```

(그림 7) 배정문을 Alloy로 변환한 예

```
pred ruleForAssignment(s, s': State) {
        some a' | s.a !=a' && a'=s'.a
        int s'.a.val = int exp
        s'.x=s.x
         ...
        s'.z=s.z
}
```

(그림 8) 배정문을 Alloy로 변환하는 규칙

장(배정문이나 조건식)을 변환한 Alloy 술어에 의해 표현된다. 그림 7의 'doAssign1'은 그림 2의 프로그램에서 'mid=z'(문장 1) 를 Alloy 술어로 변환한 예를 보여준다. 예에서 볼 수 있듯이 변환된 Alloy 술어는 선행 상태(s)와 후행상태(s')를 인자로 가진다.

그림 7에서 (1)은 배정문에 의해 변수 MID가 갱신되기 때문에 MID에서 새로운 개체를 선택하여 이를 배정문이 실행된 후의 상태, 즉 후행 상태 s'에서 새롭게 선택된 변수 mid'에 대한 정보를 가지도록 한다. (2)는 새로운 MID 개체의 값(i.e., int s'.mid.val)과 배정문을 실행하기 전의 상태, 즉 선행 상태 s에서의 변수 Z의 값(i.e., int s.z.val)은 같아야 한다는 제약식이다. 이는 배정문 'mid=z'를 수행 한 후에는 변수 MID값과 Z의 값이 같게 되는 사실로부터 쉽게 알 수 있다. (3)~(5)는 해당 배정문에 의해 영향을 받지 않는 변수들은 선행 상태와 후행 상태에서 변하지 않아야 한다는 사실을 표현한 것이다. 만약 이를 명시적으로 표현하지 않으면 Alloy 분석기가 X, Y, Z에 포함된 임의의 개체들을 후행 상태에 할당하게 되어 원하지 않은 결과가 발생할 수 있다. 이러한 조건을 프레임 조건(frame condition)이라 한다. 그림 8은 배정문 "a=exp"에 대해 Alloy 술어로 변환하는 일반적인 규칙을 보여준다.

조건식도 거의 동일한 방식으로 변환할 수 있다. 배정문을 변환할 때와 다른 점은 조건식에 의해 갱신된 변수가 없

```
pred ruleForCondition(s, s': State) {
        int s'.a.val relop int exp
        s'.x=s.x
         ...
        s'.z=s.z
}
```

(a)

```
pred performComp(s, s': State) {
        int s.y.val > int s.z.val
        s'.x = s.x
        s'.y = s.y
        s'.z=s.z
        s'.mid = s.mid
}
```

(b)

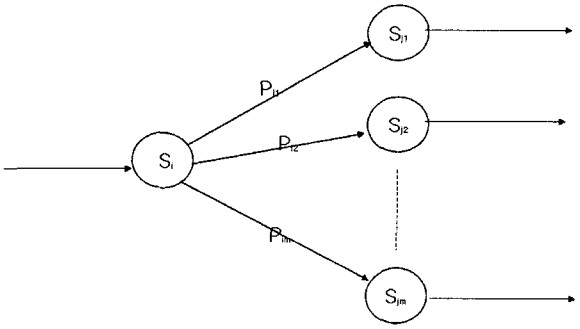
(그림 9) (a) 조건식을 Alloy로 변환하는 규칙 (b) 'y>z'를 변환한 예

기 때문에 새로운 변수 개체를 선택할 필요가 없다. 그림 9(a)는 일반적으로 조건식을 Alloy로 변환한 규칙을 보여주고 그림 9(b)는 'y>z' 를 Alloy로 변환한 예를 보여준다.

3.3.3 시퀀스 모델링

이 절에서는 유한 상태 모델의 제어 흐름을 Alloy로 표현하는 방법에 대해 기술한다. 그림 10에서 볼 수 있듯이 현재 상태가 $S_i$라면 $S_{j1}$, ..., $S_{jm}$ 상태로 전이할 수 있다. 만약 $S_i$에서 $S_{jk}$간에 전이가 일어나기 위해서는 $S_i$와 $S_{jk}$ 전이에 있는 Alloy 논리식 $P_{ik}$를 만족해야 한다.

그림 10을 Alloy 논리식으로 변환하면 다음과 같다.




(그림 10) 상태 전이의 예

$s \text{ in } S_i \Rightarrow P_{i1}(s, s') \;\&\&\; s' \text{ in } S_{j1} \;||\; \ldots \;||\; P_{im}(s, s') \;\&\&\; s' \text{ in } S_{jm}$

$s \text{ in } S_i \Rightarrow \ldots \;||\; P_{im}(s, s') \;\&\&\; s' \text{ in } S_{jm} \;||\; \underline{doNothing(s, s') \;\&\&\; s' \text{ in } Si}$

```
doNothing(s, s': State) {
        s'.x1 = s.x1
        s'.x2 = s.x2
         ...
        s'.xn = s.xn
}
```

(그림 11) 상태 변화의 표현

여기에서 =>, &&, || 은 각각 implication, and , or을 나타내기 위해 Alloy에서 사용하는 논리 연산자이다. 만약 $S_i$가 자신으로의 전이가 존재한다면 위의 논리식에 다음과 같은 논리식을 추가해야 한다(그림 11 참조).

자신으로 전이할 때 변수에 아무런 변화가 없어야 한다는 사실을 그림 11에서 볼 수 있듯이 'doNothing' 술어를 사용하여 표현하였다. 이는 'skip' 연산에 대응되는 것으로 간주할 수 있다. 유한 상태 모델의 모든 상태에 대해 위에서 주어진 논리식을 기술하여 이를 결합(conjunction)하여 표현한다.

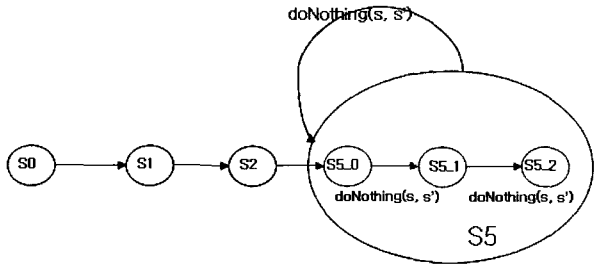
그림 12는 그림 3에 주어진 유한 상태 모델의 전이에 대한 규칙을 기술한 것이다. 초기 상태와 목적 상태가 주어져 있다면 이 전이 규칙은 어떤 상태에 대해 다음상태가 어떻게 되어야 목적 상태에 도달할 수 있는지를 결정하게 된다.

그림 12에서 'doAssign1', 'Comp1'등은 각각 'mid=z', 'y>=z', ... 에 해당하는 Alloy 술어들이다. 유한 상태 모델을 Alloy로 변환할 때 기본적으로 Alloy의 'ordering' 유틸리티를 이용한다(그림 12의 첫 번째 문장 참조). Alloy의 ordering 유틸리티는 사용자가 명시한 수만큼 개체를 포함한다. 예를 들면 State의 영역(scope)의 크기가 6이라면 정확하게 6개의 State 개체들에 대해 순서를 생성한다. 이러한 제약 때문에 유한 상태 모델에서 자신으로의 전이가 없으면 특정 상태에 도달하는 상태들의 경로를 생성할 때 문제가 발생할 수 있다.

예를 들면 그림 3에서 S5 상태를 목적 상태라 가정하고 상태 자신으로의 전이가 없다면 초기 상태 S0로부터 도달할 수 있는 경로는 <S0, S1, S2, S5>가 유일하며 이 경로의

```
open util/ordering[State] as so // import ordering utility
...
fact doTrans {
        all s: State-so/last() | 
                let s'=so/next(s) | {
                        s in S0 => doAssign1(s, s') && s' in S1
                         s in S1 => Comp1(s, s') && s' in S2 || Comp2(s, s') && s' in S3
                         s in S2 => Comp3(s, s') && s' in S4 || Comp4(s, s') && s' in S5
                         s in S3 =>...
                         s in S4 =>...
                         s in S5 => Comp5(s, s') && s' in S8 || Comp6(s, s') && s' in S10 || doNothing(s, s') && s' in S5
                        s in S6 => ...
                        s in S7 => ...
                        s in S8 => ...
                        s in S9 => ...
                        s in S10 => ...
                        s in S11 => doNothing(s, s') && s' in S11
        }
}
```

(그림 12) 전이 규칙




(그림 13) 목적 상태에서 자신으로의 전이의 효과

크기는 4라는 것을 알 수 있다. 만약 State의 영역으로 4보다 큰 값을 주었다면 경로의 크기가 4보다 크면서 S5에 도달할 수 있는 경로를 찾아야 한다. 그림 3에서 볼 수 있듯이 이러한 경로는 존재하지 않기 때문에 Alloy 분석기는 원하는 경로를 탐색하지 못한다.

이를 해결하기 위해 목적 상태에서 자신으로의 전이를 허용한다. 따라서 목적 상태에 도달할 수 있는 실제 경로의 크기가 State의 영역의 크기보다 적다할 지라도 State 영역의 크기와경로의 크기 차이만큼 목적 상태에서 자신으로의 전이를 실행하게 된다. 즉, S5에 도달할 수 있는 경로는 <S0_0, S1_0, S2_0, S5_0, S5_1, S5_2>가 된다. 여기에서 S0_0, S1_0, S2_0, S5_0, S5_1, S5_2는각각 S0, S1, S2에 속한 개체들이며 S5_0, S5_1, S5_2는 모두 S5에 속한 개체들이다.

S5_0, S5_1, S5_2는 비록 서로 다른 개체들일지라도 S5에서 S5로 전이를 수행 할 때 doNothing(그림 11 참조)을 만족해야하기 때문에 S5_0, S5_1, S5_2들은 동일한 상태 정보를 표현하는 것으로 간주할 수 있다. 그림 13은 목적 상태에서 자신으로 전이를 Alloy에서 어떻게 처리하는지를 도식화 한 것이며 S0, S1, S2는 실제로는 S0_0, S1_0, S2_0를 각각 나타낸다.

지금부터는 복잡성을 피하기 위해 상태들의 경로를 나타낼 때 실제 개체가 속한 상태로 나타내며 자신으로의 전이가 있는 경우는 자신으로의 전이가 발생한 해당 상태만을 나타내기로 한다. 따라서 Alloy 분석기가 S5에 도달하기 위한 경로로 <S0_0, S1_0, S2_0, S5_0, S5_1, S5_2>을 생성하였다 할지라도 <S0, S1, S2, S5>로 축약하여 표시한다.

이와 같이 목적 상태에서 자신으로의 전이를 허용하면 사용자가 정확하게 특정 상태에 도달하는 경로의 크기를 일일이 명시할 필요가 없으며 목적 상태에 도달할 수 있을 정도의 충분한 상태들의 개수를 State의 영역으로 제공하면 된다. 가장 쉬운 방법은 유한 상태 모델의 상태의 개수를 제공하는 것이다. 만약 목적 상태에 도달할 수 있는 가장 짧은 경로를 테스트하고 싶다면 Alloy 분석기가 경로를 찾을 수 있는 한 State 영역의 크기를 하나씩 줄어가는 방법을 사용하면 된다.

## 4. 테스트 적합성 기준과 테스트 데이터 생성

이 절에서는 실제적으로 Alloy 분석기를 이용하여 다양한 테스트 적합성 기준에 따라 테스트 데이터를 생성하기 위한

```
sig S0, S1, S2, ..., S11 extends State {}
fact {
        so/first() in S0
}
pred testIt() {
                so/last() in S9
}

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

(그림 14) 초기 상태 및 목적 상태를 Alloy로 표현한 예

부가적인 정보를 어떻게 명시하는지에 대해 구체적으로 기술한다. 이를 위해 문장 적합성 기준(statement adequacy *criterion*), 분기 적합성 기준(*branch adequacy criterion*) 및 자료 흐름 적합성 기준에 따라 테스트 데이터를 생성하는 방법을 제시한다.

### 4.1 문장 적합성 기준에 따른 테스트 데이터 생성

테스트 데이터가 프로그램 상에 존재하는 모든 문장을 최소한 한번은 실행할 수 있게 하는 입력 값들로 구성되어 있다면 문장 적합성 기준을 만족한다. 이를 유한 상태 모델 관점에서 다시 표현하면 유한 상태 모델의 모든 상태들을 최소한 한 번은 실행하게 하는 입력 데이터들을 생성하는 문제로 변환할 수 있다. 이를 위해 유한 상태 모델의 각 상태를 목적 상태로 명시하고 초기상태로부터 도달할 수 있는지를 Alloy 분석기를 이용하여 탐색하는 것이 필요하다.

예를 들어 그림 2에서 노드 n9를 실행하는 테스트 데이터를 생성하기 위해서는 그림 14와 같이 S0를 초기 상태로 주고 S9를 목적 상태로 명시하면 된다.

S0를 'fact'를 사용하여 초기 상태임을 기술하였고 S9는 'pred'를 사용하여 기술하였다. 그 이유는 초기 상태는 항상 변함이 없지만 목적 상태는 달라질 수 있기 때문이다. 그림 14와같이 명세를 주었을 때 Alloy 분석기는 여러 가지 정보를 생성한다(부록 1참조). 결과에서 볼 수 있듯이 Alloy 분석기는 S9에 도달하기 위한 경로로 <S0, S1, S3, S6, S9>를 생성하였다. 이는 부록 1의 하단 부분의 State의 순서 관계로부터 알 수 있다. 또한 분석결과에서 이 경로를 실행하기 위해 제공되어야 하는 입력 값들도 알 수 가 있는데 초기상태, 즉 S0에서 X, Y, Z의 개체들이 갖는 값들이 원하는 테스트 데이터들이다. S0의 X의 개체는 X_0, Y의 개체는 Y_0, Z의 개체는 Z_0이므로 X_0, Y_0, Z_0의 val 필드 값들을 조사하면 된다. 이들은 각각 -1, -4, 0으로 바인딩 되었음을 Alloy 분석 결과에서 알 수 있다. 결론적으로 프로그램 제어점 n9를 실행하려면 X, Y, Z,에 -1, -4, 0을 각각 제공하면 된다.

### 4.2 분기 적합성 기준에 따른 테스트 데이터 생성

분기 적합성 테스트 기준은 프로그램에 존재하는 모든 분기를 최소한 한번은 실행하게 하는 입력 데이터를 테스트 데이터로 제공하여야 한다. 분기는 노드들의 쌍으로 기술될

```
pred testIt() {
        so/last() in S8
        some S5
}
run testIt for 6 State, 6 Integer, 5 Int
```

(그림 15) 분기를 Alloy로 명시한 예

〈표 1〉 분기 적합성 기준을 만족하는 테스트 데이터

| 분기 | 테스트 데이터 <X, Y, Z> | 경로 | 테스트되는 분기 |
|---|---|---|---|
| <n2, n4> | <2, 2, 0> | <n0, n1, n2, n4> | <n1, n2><br><n2, n4> |
| <n5, n8> | <2, 3, -3> | <n0, n1, n2, n6, n8> | <n1, n2><br><n2, n5><br><n5, n8> |
| <n5, n10> | <-4, 2, 2> | <n0, n1, n2, n5, n10> | <n1, n2><br><n2, n5><br><n5, n10> |
| <n6, n10> | <3, -4, -3> | <n0, n1, n3, n6, n10> | <n1, n3><br><n3, n6><br><n6, n10> |
| <n6, n9> | <0, 0, 1> | <n0, n1, n3, n6, n9> | <n1, n3><br><n3, n6><br><n6, n9> |
| <n3, n7> | <-4, 0, 2> | <n0, n1, n3, n7> | <n1, n3><br><n3, n6> |

수 있기 때문에 Alloy로 테스트할 분기를 명시하기 위해서는 한 개의 목적 상태만을 기술하는 문장 적합성 기준과는 달리 두 개의 상태를 명시한다. 예를 들면 그림 2에서 분기 <n5, n8>을 테스트하기 위한 테스트 데이터를 구하기 위해서는 그림 15에서 기술한 것과 같이 S8을 목적 상태로 하고 S5를 반드시 S8에 도달하는 경로에서 반드시 포함해야할 상태로 기술한다.

표 1은 그림 2의 예제 프로그램에 대해 분기 적합성 기준을 만족하는 테스트 데이터를 Alloy를 이용하여 생성한 결과를 보여준다. 첫 번째 열(분기)은 사용자가 명시한 분기를 기술한 것이며 두 번째 열과 세 번째 열은 각각 해당 분기를 실행하는 테스트 데이터와 실행 경로이다. 마지막 열은 생성된 테스트 데이터에 의해 실제로 실행되는 분기들을 열거한 것이다. 예를 들어 그림 2(b)에서 분기 <n2, n4>를 실행하는 테스트 데이터는 X=2, Y=2, Z=0이며 이 분기는 경로 <n0, n1, n2, n4>를 통해 실행된다. 이 경로는 또한 분기 <n1, n2>도 포함하고 있으므로 <n1, n2>를 실행하기 위한 테스트 데이터를 따로 생성할 필요가 없다.

### 4.3 자료 흐름 적합성 기준에 따른 테스트 데이터 생성

자료 흐름 적합성 기준은 앞 절들에서 살펴 본 문장 적합성 기준이나 분기 적합성 기준과 같이 테스트 데이터의 적합성을 판별할 때 프로그램의 제어 흐름에 바탕을 둔 것과는 달리 변수의 정의 및 사용 정보에 기반을 둔 적합성 기준이다. 기본적으로 자료 흐름에 기반을 둔 적합성 기준들은 변수가 정의된 곳에서 사용된 곳까지의 프로그램 경로를 실행할 수 있는 테스트 데이터를 생성하는 것이 목적이다.

```
int dftest(int x, y) {
        int v, t;
        1: t = 3;
        2: if (x>y)
        3:t = 5;
        4: v = t+x+y
        5: return v;
}
```

(그림 16) 자료 흐름 테스팅의 예제 프로그램

```
abstract sig State {
        x: one X,
        y: one Y,
        v: one V,
        t: one T,
        DEF : set Vars,
        USE: set Vars
}
```

(그림 17) 자료 흐름 테스팅을 위한 State 확장

```
pred dfAssign(s, s': State) {
        some v': V | s.v != v' && v'=s'.v
        int s'.v.val= int s.x.val + int s.y.val + int s.t.val
        s'.DEF = s.v
        s'.USE = s.x+s.y+s.t
        s.x=s'.x
        s.y=s'.y
        s.t=s'.t
}
```

(그림 18) 자료 흐름 테스팅을 위한 배정문의 변환

우선 Alloy 분석기를 사용하여 자료 흐름 적합성 기준을 만족하는 테스트 데이터를 생성하기 위해서는 각 상태에서 어떤 변수가 정의되는지 또는 사용되는지를 명시할 필요가 있다. 그림 17은그림 16에 주어진 프로그램을 대상으로 상태 정보를 표현한 것이다.

그림 17에서 DEF와 USE는 각 상태에서 정의되거나 사용되는 변수들을 알려주는 관계들이다. DEF 와 USE의 실제 사용 예를 보기 위해 그림 16의 4번 문장 "v=t+x+y"을 예로 들어보자. 이 배정문은 t, x, y를 사용하여 v를 정의한다. 따라서 이 배정문을 실행하고 난후의 상태 s'에서 DEF와 USE는 "s'.DEF = s.v"와 "s'.USE = s.x+s.y+s.t"이다. 이를 바탕으로 이 배정문을 Alloy 술어로 변환하면 그림 18과 같다.

여기에서 한 가지 주의할 점은 변수 가 정의된 곳에서 사용된 곳까지 모든 프로그램 경로가 테스트 대상이 되는 것이 아니라는 점이다. 테스트 대상이 되는 경로는 변수 X가 정의된 위치에서 사용된 곳까지의 경로 상에 변수 X의 재정의가 없는 경로만 테스트 대상이 된다는 점이다[5].

```
fact dfTesting {
    some d: D | some u: U | no s : State | {
        gt(s, d)                          ------------ (1)
        lt(s, u)                          ------------ (2)
        some s.DEF                        ------------ (3)
        s.DEF in V                        ------------ (4)
    }
}
```

(그림 19) 자료 흐름 테스팅을 위한 불변성 명세

예를 들면 그림 16에서 1번 문장의 t의 정의가 4번 문장에서 올바르게 사용되는지를 테스트 하기위해서 프로그램 경로 <1, 2, 3, 4>를 실행하는 테스트 데이터를 생성하였다면 자료 흐름에 기반을 둔 적합성 기준을 만족하지 못한다. 그 이유는 3번 문장에서 t에 대한 재정의가 이루어지기 때문이다. 따라서 변수가 정의되는 곳에서 사용되는 곳까지의 경로에 대한 제약을 Alloy로 표현할 필요가 있다.

그림 19는 이런 제약을 변수 v에 대해 불변성으로 나타낸 것이다. 여기에서 D는 프로그램에서 변수 v가 정의된 위치에 대응되는 상태(State) d를 나타내고 U는 변수가 사용된 위치에 대응되는 상태 u를 나타낸다. (1)~(4)는 d와 u사이에 있는 상태들에서 변수 v를 재정의 한 상태가 없어야 된다는 사실을 나타낸다. 이와 같은 제약이 있는 상태에서 그림 17에 주어진 예제 프로그램의 변수 t에 대해 d를 1번 문장에 대응되는 상태라 하고 u를 4번 문장에 대응되는 상태라고 할 때 Alloy 분석기는 테스트 데이터로 x=0, y=2 값을 생성하였다. 이 값은 프로그램 경로 <1, 2, 4, 5>를 실행하는 것을 알 수 있다.

## 5. 결 론

이 논문에서는 테스트 데이터를 생성하기 위해 완전한 프로그램 경로를 제공하는 일반적인 방법과는 달리 프로그램상의 특정 제어점 또는 경로상의 일부분만 주어져도 테스트 데이터를 자동으로 생성할 수 있는 방법을 제안하였다. 이 논문에서는 테스트 데이터를 자동으로 생성하기 위해 우선 프로그램을 1차 관계 논리에 바탕을 둔 형식 명세 언어 Alloy로 변환하는 방법을 제시하였다. 또한 여러 가지 테스트 데이터 적합성 기준에 따라 테스트 데이터를 생성하는 방법에 대해서도 기술하였다.

제안된 방법은 기존의 테스트 데이터 자동 생성 방법보다는 많은 기술적인 진보가 있었지만 보완할 점도 많이 있다. 우선 이 논문에서 기술한 방법은 단위 테스팅만을 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. 두 번째로 이 논문에서 제안된 방법을 자동화 하는 것이 필요하다. 현재 이 논문의 목적은 프로그램을 Alloy 명세로 변환하여 테스트 데이터를 자동으로 생성하는 프레임워크를 구축하는 것이다. 그러나 향후의 연구로 자동화 도구를 구현하여 보다 다양한 프로그램에 대해 이 논문에서 제안한 방식에 대해 평가할 필요가 있다. 마지막으로 이 논문에서 지원하지 못하고 있는 다양한 고급 프로그래밍 언어의 특성들을 지원할 수 있는 방안이 강구되어야 할 것이다. 특히 실제 프로그램에 자주 사용되고 있는 포인터 등의 지원은 필수적이다. 현재 이에 대한 연구가 진행 중에 있다.

## 참 고 문 헌


[1] Ince, D., "The Automatic Generation of Test Data", *The Computer Journal*, Vol.30, No.1, pp.63-69, 1987.

[2] Weyuker, E. J.., "Testing Non-testable Programs", *The Computer Journal*, Vol.25, No.4, pp.465-470, 1982.

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

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

[5] Frankl, P. G. and Weyuker, E. J., "An Applicable Family of Data Flow Testing Criteria", *IEEE Trans on Software Eng.*, Vol.14. No.10. pp.1483-1498, 1988.

[6] Jackson, D., "Alloy: A light weight object modeling Notation", Technical Report 797, MIT Lab for Computer Science, Feb., 2000.

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

[8] Jackson, D. and Vaziri, M., "Finding Bugs with a Constraint Solver", In *Proc. InternationalConf. on Software Testing and Analysis*, 2000.

[9] Andoni, A., Daniliuc, D., Khurshid, S., and MariNov, D., "Evaluating the Small Scope Hypoththesis", Technical Report 921, MIT Lab for Computer Science, Feb., 2003.

[10] Offutt, J., Pan, J., "The Dynamic Domain Reduction Approach to Test Data Generation", *Software-Practice and Experience*, Vol.29. No.2. pp.167-193, 1997.

[11] Cytron, R. Ferrante, J., Rosen, B. K., Wegman, M. N., Zadeck, F. K., "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph", *ACM Trans on Programming Languages and Systems*. Vol. 13. No. 4. pp. 451-490, 1991.

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

[13] Howden, "Symbolic Testing and the DISSECT Symbolic Evaluation System", *IEEE Trans on Software Eng.*, Vol.4. No.4. pp.266-2781,1977.

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

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

[16] Roger, F., Korel, B, "The Chaining Approach for Software Test Data Generation", ACM *Trans. on Soft. Eng. Methodology*, Vol.5. No.1. pp.63-86, 1996.

[17] 정인상, "자동화된 프로그램 시험을 위한 입력 자료구조의 모양 식별", 한국정보과학회 논문지, 제31권 제10호, pp.1304-1319, 2004.

[18] 정인상, "SGEN: 자동 프로그램 테스팅을 위한 입력 자료 구조 생성기", 한국정보과학회, 소프트웨어공학회지, 18권 4호, pp. 39-50, 2005.

**정 인 상**

1987년 서울대학교 컴퓨터공학과(학사)
1989년 한국과학기술원 전산학과(석사)
1993년 한국과학기술원 전산학과(박사)
1994 ~ 1998 한림대학교 교수
1997 ~ 1998 미 Purdue 대학 방문교수
2004 ~ 2005 미 CSU 방문교수
1999 ~ 현재 한성대학교 컴퓨터공학과 교수
관심분야 : 소프트웨어공학, 프로그램 테스팅, 정형검증 및 명세

## 부록 1. Alloy 분석기의 실행 결과

```
sig Integer extends Integer = {M_0, M_1, X_0, Y_0, Z_0}
 val : alloy/lang/Int/Int =
  {M_0 -> 0,
   M_1 -> -1,
   X_0 -> -1,
   Y_0 -> -4,
   Z_0 -> 0}
...
sig State extends univ = {S0_0, S1_0, S3_0, S6_0, S9_0, S9_1}
 x : models/shape/mid/X =
  {S0_0 -> X_0,
   S1_0 -> X_0,
   S3_0 -> X_0,
   S6_0 -> X_0,
   S9_0 -> X_0,
   S9_1 -> X_0}
 y : models/shape/mid/Y =
  {S0_0 -> Y_0,
   S1_0 -> Y_0,
   S3_0 -> Y_0,
   S6_0 -> Y_0,
   S9_0 -> Y_0,
   S9_1 -> Y_0}
 z : models/shape/mid/Z =
  {S0_0 -> Z_0,
   S1_0 -> Z_0,
   S3_0 -> Z_0,
   S6_0 -> Z_0,
   S9_0 -> Z_0,
   S9_1 -> Z_0}
 m : models/shape/mid/M =
  {S0_0 -> M_1,
   S1_0 -> M_0,
   S3_0 -> M_0,
   S6_0 -> M_0,
   S9_0 -> M_0,
   S9_1 -> M_0}
...

module util/ordering[models/shape/mid/State]
sig Ord extends univ = Ord_0
 first_ : models/shape/mid/State =
  {Ord_0 -> S0_0}
 last_ : models/shape/mid/State =
  {Ord_0 -> S9_1}
 next_ : ( models/shape/mid/State) ->lone ( models/shape/mid/State) =
  {Ord_0 -> {S0_0 -> S1_0, S1_0 -> S3_0, S3_0 -> S6_0, S6_0
   -> S9_0, S9_0 -> S9_1}}
 prev_ : ( models/shape/mid/State) ->lone ( models/shape/mid/State) =
  {Ord_0 -> {S1_0 -> S0_0, S3_0 -> S1_0, S6_0 -> S3_0, S9_0
   -> S6_0, S9_1 -> S9_0}}
```