SAT에 기반한 포인터가 있는 프로그램을 위한 목적 지향 테스트 데이터 생성☆

A Goal-oriented Test Data Generation for Programs with Pointers based on SAT

정 인 상*
In-Sang Chung

요 약

지금까지 테스트 데이터를 자동으로 생성하기 위한 대부분의 연구는 프로그램에 포인터가 존재하지 않는 경우만을 대상으로 하였다. 최근에 포인터가 있는 경우에도 테스트 데이터를 자동으로 생성할 수 있는 방법들이 제안되었지만 테스트할 프로그램 경로를 완전하게 명시해야 하는 경로 기반 방법이거나 프로그램을 실제 실행해야 하는 방법들이다. 이 논문에서는 프로그램 경로를 완전하게 명시하지 않아도 포인터가 있는 프로그램에 대하여 테스트 데이터를 생성할 수 있는 새로운 방법을 제안한다. 제안된 방법은 테스트 데이터 생성 문제를SAT(SATisfiability) 문제로 변환하고 SAT 해결도구를 이용하여 자동으로 테스트 데이터를 생성하는 정적 방법이다. 이를 위해 프로그램을 1차 관계 논리 언어인 Alloy로 변환하고 Alloy 분석기를 통하여 테스트 데이터를 생성한다.

Abstract

So far, most of research on automated test data generation(ATDG) deals with programs without pointers. Recently, few works have been done on ATDG in the presence of pointers, but they are path-oriented techniques which require the specification of an entire program path to be tested or a program to be executed. This paper presents a new technique for generating test data even without specifying a program path completely. The presented technique is a static technique which transforms the test data generation problem into a SAT(SATisfiability) problem and makes advantage of SAT solvers for 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.

☞ keyword : Program Testing, Automated Program Test Data Generation, SAT, First-order Relational Language, Alloy, 프로그램 테스트, 자동 테스트 데이터 생성, SAT, 1차 관계 논리 언어

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 테스트는 아직까지 가장 현실적인 방법으로 널리 사용되고 있다. 그러나 한편으로 프로그램 테스트를 수행하는데 많은 자원과 비용을 소모하는 것도 사실이다.

소프트웨어 테스트 비용을 줄이기 위해서는 테스트 데이터를 자동으로 생성하는 것이 효과적이다. 지난 수 년 동안 테스트 데이터를 자동으로 생성하기 위한 기술 (ATDG: Automated Test Data Generation) 개발에 많은 연구가 있어왔지만 대부분 포인터가 없는 프로그램을 대상으로 하였다. 프로그램에 포인터가 존재하는 경우에 테스트 데이터는 리스트, 트리 또는 그래프와 같은 2차원적인 자료구조로 표현된다. 따라서 포인터를 고려한 테스트 데이터 자동생성 방법은 특정 프로그램

* 정 회 원 : 한성대학교 컴퓨터공학과 교수
insang@hansung.ac.kr
[2007/09/10 투고 - 2007/09/17 심사 - 2007/10/02 심사완료]
☆이 연구는 2007년 한성대학교 교내연구비 지원과제임

경로나 특정 프로그램 블록을 실행할 수 있는 입력 자료 구조 모양을 결정하는 것이 주요 목적이라 할 수 있다. 이와 같이 입력 자료 구조의 모양을 식별하는 문제를 모양 문제(shape problem)라고 한다[1].

비록 최근에 포인터를 고려한 테스트 데이터 자동 생성에 관한 연구가 몇몇 제안되었지만 이 방법들은 사용자가 테스트할 프로그램 경로를 제공해야 하는 경로 지향(path-oriented) 방법들이다. 이 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].

이와는 달리 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[1,2]. 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 블록을 실행할 수 있는 다른 경로를 자동으로 탐색하여 입력 값을 생성할 수 있다. 지금까지 대부분의 목적 지향 테스트 데이터 생성 방법은 테스트 데이터를 생성하기 위해 프로그램을 탐침(instrumentation)하여 프로그램의 실행을 모니터링 하는 동적 기법에 바탕을 두고 있다. 따라서 내장형 시스템과 같이 절대적으로 메모리와 같은 자원의 제약이 있는 환경에서는 사용하기 힘든 단점이 있다.

이 논문에서는 포인터가 있는 프로그램에 대하여 테스트 데이터를 자동으로 생성할 수 있는 목적 지향적 방법을 제안한다. 프로그램의 실행을 요구하는 기존의 목적 지향적 방법과는 달리 이 논문에서 제안하는 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 zschaff[3], BerkMin[4], GRASP[5] 등과 같은 SAT 도구들이 개발되어 사용되고 있다.

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.

이 논문에서는 프로그램을 직접적으로 CNF로 직접 변환하지 않고 Alloy 명세로 변환한다. Alloy는 1차 관계 논리(first-order relational logic)에 기반 한 모델링 언어이다[6,7,8]. 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.

이 논문은 C 프로그램을 대상으로 하고 있으며 C 프로그램을 Alloy 명세로 변환하는 방법은 기본적으로 [9]에서 제안된 방법을 기반으로 하고 있다. 그러나 [9]에서 제안된 방법은 포인터 및 구조체에 대한 고려가 없었다. 또한 프로그램에 반복문이 포함될 때 반복횟수를 명시하기를 요구했다, 이 논문에서는 이러한 점들을 보완한 테스

트 데이터 생성 방법을 제안한다.

본 논문은 다음과 같이 구성된다. 2장에서는 기존의 테스트 데이터를 자동으로 생성하는 방법들에 대해 간략하게 소개하며 포인터를 사용하는 프로그램을 대상으로 하는 경우에 어떤 문제점이 발생하는 지에 대해 기술한다. 3장에서는 포이터 및 구조체를 사용하는 프로그램을 Alloy 명세로 변환하는 방법에 대해 기술한다. 4장에서는 변환된 Alloy 명세를 기반으로 목표 블록을 실행하는 테스트 데이터를 자동으로 생성하는 방법에 대해 기술한다. 마지막으로 5장에서는 결론 및 향후 연구에 대해 언급한다.

2. 관련 연구

2.1 테스트 데이터 생성

포인터를 고려하여 테스트 데이터를 생성하는 방법은 가장 먼저 Korel에 의해 제안되었다[10]. 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 실행 흐름을 변경 시킬 필요가 있다. Korel의 방법은 이를 위해 동적 자료 흐름 분석 기법을 이용한다. 동적 자료 흐름 분석을 통하여 원하는 방향으로 분기가 발생하지 않은 분기문에 영향을 미칠 수 있는 입력 변수들의 값을 변경하여 분기가 다른 방향으로 일어날 수 있도록 유도한다. 만약 현 상태에서 가능한 어떠한 입력 값도 원하는 방향으로 분기가 일어나지 않는다면 이전 분기로 되돌아가 새로운 입력 값을 다시 탐색하는 절차를 반복한다.

Visvanathan과 Gupta도 프로그램이 포인터를 사용하는 경우에 테스트 데이터를 생성하는 경로 지향적 방법을 제안하였다[11]. 그들이 제안한 방식은 두 단계 방법(Two phase approach)이라고 하는데 그 이유는 이 방식이 두 단계로 구성되어져 있기 때문이다. 첫 번째 단계에서 주어진 경로에 사용되는 포인터 값에 대한 제약식을 구성하고 이들을 만족하는 입력 자료 구조의 모양을 결정하는 단계이다. 이런 이유로 첫 번째 단계를 모양 식별 단계(shape identification phase)라 한다. 두 번째 단계에서는 포인터가 아닌 입력 변수나 구조체의 데이터 값을 계산하는 단계이다. 이 단계를 데이터 값 생성(data value generation phase) 단계라 한다.

국내에서도 포인터를 사용하는 프로그램을 테스트하기 위한 SGEN이라는 도구가 개발되었다[12, 13]. SGEN은 앞서 언급한 두 단계 방법과 매우 유사하지만 다음과 같은 차이가 있다. 두 단계 방식은 주어진 경로가 포인터를 사용하는 문장이 없는 경우에는 모양 식별 단계 즉, 첫 번째 단계가 필요가 없다. 반면에 SGEN에서는 포인터와 관련된 문장이 전혀 없다 할지라도 포인터 형이 아닌 입력 변수에 대한 제약식을 구하는 작업을 수행하므로 보다 효율적이라 할 수 있다. 또한, SGEN에서는 포인터 타입이 아닌 입력 변수에 대한 제약식을 만들 때 이미 이명 정보(aliasing information)가 반영된 상태로 생성되기 때문에 별다른 노력 없이 기존의 제약식 해결 시스템(constraint solving system)을 사용할 수 있다. 반면에 두 단계 방식에서는 이명 정보를 계산하지만 제약식을 만들 때 이를 직접적으로 활용하지는 않는다. 따라서 기존의 제약식 해결 시스템을 이용하기 위해서는 추출한 이명 정보를 반영한 제약식을 따로 만들 필요가 있다. 마지막으로 SGEN은 실행 불가능한 경로를 보다 효과적으로 찾을 수 있다. 두 단계 방식에서는 모양 식별 단계를 수행 한 후에만 포인터 타입이 아닌 입력 변수에 대한 제약식이 생성된다. 이 의미는 두 번째 단계에서 생성된 제약식을 만족하는 해가 없어 주어진 프로그램 경로를 실행 할 수 있는 입력 값을 찾지 못한 경우에는 두 단계 방식이 효율적이지 못하다는 사실이다. 즉, 주어진 경로가 실행 불가능한 지의 여부는 두 단계를 거친 후에

만 알 수 있기 때문이다. 반면에 SGEN에서는 모양 생성과 동시에 포인터 타입이 아닌 입력 값에 대한 제약식이 동시에 생성되므로 보다 효과적으로 실행 불가능한 지의 여부를 검사할 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[14]은 특정 프로그램 블록을 명시하면 사용자가 완전한 프로그램 경로를 제공하지 않아도 주어진 프로그램 블록에 도달할 수 있는 테스트 데이터를 생성하는 목적 지향적 테스트 데이터 생성 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 블록에 도달할 수 있는 가능성이 전혀 없다.
- 반임계 분기(semi-critical branch): 프로그램이 이 분기를 따라 실행되면 목적 노드에는 도달할 수는 있지만 제어 흐름 그래프 상에서 반복문의 역 간선(backward edge)을 따라 도달할 수 있다.
- 비필수적 분기(non-essential branch): 임계 분기도 아니고 반임계 분기도 아닌 나머지 분기를 비필수적 분기라 한다.

이 방법은 우선 무작위로 생성된 입력 값으로 프로그램을 실행시킨다. 만약 실행과정에서 임계 분기를 따라 프로그램이 실행되면 실행을 중단하고 입력 값을 수정하여 다른 분기를 따라 실행되도록 다시 프로그램을 실행한다. 만약 반임계 분기를 따라 실행되고 있다면 새로운 입력 값을 찾는다. 그러나 이 경우에 적절한 입력 값을 찾지 못했다할지라도 프로그램의 실행을 중단하지 않고 프로그램의 실행을 계속한다. 비필수적 분기는 말 그대로 목적 블록에 도달할지를 결정하는데 영향을 주지 않기 때문에 실행을 중단하지 않고 계속 실행하게 한다.

프로그램의 실행을 요구하는 외국의 목적 기반 테스트 방법에 관한 연구와는 달리 국내에서는 프로그램을 실행하지 않고 테스트 데이터를 생성하는 목적 지향 테스트 방법이 제안되었다[9]. 이 방법은 입력 프로그램을 Alloy 명세로 변환한다. 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[15]. 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. 그러나 이 방법은 구조체나 포인터가 없는 프로그램만을 대상으로 하였다.

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 모델을 탐색하는 도구이다. 여기에서 모델은 명세

에 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 1은 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 1) Alloy 명세의 분석 결과(모델)

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.

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[16].

3. Alloy에 기반한 테스트 데이터 자동 생성

이 장에서는 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[9]. 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.

3.1 변수 모델링

정수형 변수는 Alloy에서 기본적으로 제공하는 'Int'를 이용하여 모델링한다. 구조체는 Alloy 시그너쳐를 통하여 모델링하며 구조체의 필드들은 시그너쳐의 필드들로 매핑된다. 예를 들어 그림 2(a)의 세 개의 정수형 필드 x, y, z를 갖는 구조체는 그림 2(b)와 같이 Alloy로 변환된다.

이와 같이 변환하는 이유는 Alloy에서는 시그너쳐의 필드들을 통하여 개체들을 연관시키기 때문이다. 그림 2에서 시그너쳐 필드 p, q, r은 각각 foo에 속한 한 개체와 Int의 한 개체를 연관시키는 관계(relation) 이름으로 간주된다. 만약 f가 foo의 한 개체라면 관계 p에 의해 연관된 Int 객체는 f,p 또는 p[f]를 통해 구할 수 있다.

```
struct foo {
        int p;
        int q;
        int r;
}
```

(a) 구조체

```
sig foo {
        p: Int,
        q: Int,
        r: Int
}
```

(b) 구조체에 대응되는 Alloy 시그너쳐

(그림 2) 구조체를 Alloy 시그너쳐로 변환한 예

포인터는 한 개체가 다른 객체의 메모리 주소를 저장함으로써 두 개체를 연결할 수 있다. 따라서 포인터도 두 개체의 관계를 표현하는 시그너쳐의 필드를 통하여 표현할 수 있다. 그림 3은 그림 4에서 사용된 이진 탐색 트리의 노드를 나타내는 구조체를 Alloy로 모델링한 예를 보여준다. left, right 필드들은 Node의 한 개체와 Node의 한 개체를 연결해주는 관계들이며 여기에서 한정자 lone을 사용하는 이유는 어떤 Node의 개체에 대해서는 연결되는 Node의 개체가 없을 수도 있다는 사실을 나타내기 위함이다. 즉, lone을 사용하여 NULL 포인터를 표현하였다.

```
sig Node {
        data: Int,
        left: lone Node,
        right: lone Node
}
```

(그림 3) 이진 탐색 트리의 구조체 포인터 모델링

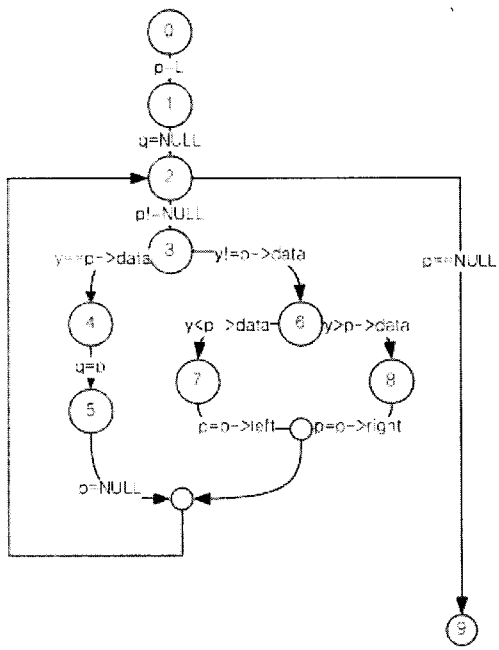
만약 Node에 속한 개체 n1과 n2가 관계 left에 속한다는 사실은 Alloy로 'n1->n2 in left'로 표현된다. 여기에서 n1과 연관된 개체를 구하기 위해서는 앞에서 언급한 대로 n1.left나 left[n1]을 사용한다.

```
struct Node {
          int data;
          struct Node *left;
          struct Node *right;
};

void Find(struct Node *L, int y, struct Node *q) {
        struct Node *p;
1:      p = L;
2:      q = NULL;
3:      while (p != NULL) {
4:      if (y == p->data) {
5:              q = p;
6:              p = NULL;
        }
7:      else {
8:              if (y < p->data)
9:                      p = p->left;
10:             else p= p->right;
11:     }

}
```

(그림 4) 이진 탐색 함수와 제어 흐름 그래프

3.2 상태 전이 모델링(State Transition Modeling)

그림 5는 그림 4의 이진 탐색 함수의 제어 흐름 그래프를 나타낸다. 제어 흐름 그래프의 각 노드(node)는 프로그램상의 제어점(control point)을 나타내고 간선(edge)은 프로그램의 각 문장이나 조건식을 나타낸다.



(그림 5) 이진 탐색 함수의 제어 흐름 그래프

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 흐름 그래프에서 진출 간선(out-going edge)이 없는 노드(i.e., 종료 노드)에 해당하는 상태는 자신으로의 전이가 존재한다[9]. 이 절에서는 상태 전이 모델을 구성하는 상태 및 전이를 Alloy로 표현하는 규칙에 대해서 구체적으로 기술한다.

3.2.1 상태 모델링

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 4에 주어진 이진 탐색 프로그램의 상태는 그림 6과같이 표현된다.

```
abstract sig State {
        y : Int,
        L : lone Node,
        q : lone Node,
        p : lone Node
}
```

(그림 6) 이진 탐색 함수의 상태 표현

시그너쳐 State에서 y, L, q, p 필드는 주어진 상태에서 해당 프로그램 변수가 갖는 값을 연결하는 관계들이며 정확하게 각 상태에서 각 변수가 취할 수 있는 값은 기껏해야 하나만 존재해야 한다는 사실을 한정자 'lone'을 사용하여 표현하였다. 예를 들면, 's.y=10'는 상태 s에서 y의 값은 10과 같다는 사실을 표현한다.

3.2.2 전이 모델링

상태 전이 모델에서 상태 간의 전이는 프로그램의 해당 문장(배정문이나 조건식)을 변환한 Alloy 술어에 의해 표현된다. 변환된 Alloy 술어는 해당 프로그램 문장이 실행되기 전의 상태와 실행된 후의 상태에서 변수들 간의 연관 관계를 기술한다.

```
pred doAssign2[s, s': State] {
        no s'.q ---------- (1)
        s'.y=s.y ---------- (2)
        s.'L=s.L ---------- (3)
        s'.p=s.p ---------- (4)
}
```

(그림 7) 배정문을 Alloy로 변환한 예

그림 7의 'doassign2'는 그림 4의 프로그램에서 'q=NULL'을 Alloy 술어로 변환한 예를 보여준다. 예에서 볼 수 있듯이 변환된 Alloy 술어는 선행 상태(s)와 후행상태(s')를 인자로 갖는다. (1)은 배정문에 의해 포인터 q가 NULL이 되기 때문에 배정문이 실행된 후의 상태, 즉 후행 상태 s'에서는 포인터 q가 아무것도 가리키지 않아야 된다는 사실을 나타낸다. 여기에서 주목할 것은 선행 상태 s에서 포인터 p가 가리키는 것에 대한 어떤 언급도 없다는 점이다. 즉, 해당 배정문이 실행되기 전의 상태에서 q가 가리키는 개체에 상관없이 해당 배정문이 실행된 후에는 q는 아무 개체도 가리키지 않는다는 점을 나타낸다. (2)~(4)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 y, L, p에 임의의 개체들을 후행 상태에 할당하게 되어 원하지 않은 결과가 발생할 수 있다. 이러한 조건을 프레임 조건(frame condition)이라 한다.

그림 8은 프로그램 문장을 Alloy로 변환하는

규칙들을 보여준다. 여기에서는 포인터와 직접적으로 관련이 있는 변환 규칙만을 보여준다. 변환 함수 Σ는 프로그램 문장에 적용되는 함수이고, Φ는 조건 그리고 Ξ는 수식에 적용되는 함수이다.

Σ : Statement→State→State→Alloy Formula
Φ : Predicate→State→State→Alloy Formula
Ξ : Expression→State→Alloy Expression
V: 모든 포인터들의 집
M(p) : 포인터 p가 가리키는 구조체의 필드들의 집합; rel: 관계 연산자

Σ[p = NULL]ss′	= no Φ[v]s′ && (all r: V-p | s′.r=s.r)
Σ[p = q->f]ss′	= some Ξ[q]s && Φ[v]s′ = Ξ[q->f]s && (all r: V-p | s′.r=s.r)
Σ[p->f = NULL]ss	= some Ξ[p]s && no Ξ[p->f]s′ && (all r: V | s′.r=s.r)&& (all m: M(p)-f |m[Ξ[p]s′] = m[Ξ[p]s′)
Σ[p->f = q]ss′	= some Ξ[p]s && Ξ[p->f]s′ = Ξ[q]s && (all r: V | s′.r=s.r)&& (all m: M(p)-f | m[Ξ[p]s′] = m[Ξ[p]s′)

Φ[p == NULL]ss′	= no Φ[p]s && (all r: V | r[s′] = r[s])
Φ[p == q]ss′	= Φ[p]s = Φ[q]s && (all r: V | r[s′] = r[s])
Φ[p != NULL]ss′	= some Φ[p]s && (all r: V | r[s′] = r[s])
Φ[p != q]ss′	= Φ[p]s != Φ[q]s && (all r: V | r[s′] = r[s])
Φ[p->f rel e]ss	= some Ξ[p]s && Ξ[q->f]s rel Ξ[e]s && (all r: V | s′.r=s.r)

Ξ[v]s	= v[s]
Ξ[p->f]s	= f[Φ[p]s]

Σ[p malloc(...)]ss′	= no p[s′] && (all r: V-p | r[s′] = r[s])
Σ[free(p)]ss′	= no p[s′] && (all r: V-p | r[s′].ptr = r[s].ptr)

(그림 8) Alloy 변환규칙

그림 8에 주어진 Alloy 변환 규칙의 이해를 위해 그림 4에서 'p=p->left'를 Alloy로 변환해보자. 즉, Σ[p = p->left]ss'. 이 배정문의 오른쪽에 있는 'p->left'는 포인터 p가 가리키는 구조체의 left 필드를 가리킨다. 이 배정문이 실행되기 전의 상태를 s라 할 때 p가 가리키는 구조체는 Alloy로 p[s]로 표현된다. 그러나 포인터 p가 상태 s에서 NULL이 되어서는 안된다 (i.e., some Ξ[p]s). 즉, 'some p[s]'를 만족하여야 한다. 포인터 p가 가르키는 구조체의 left 필드는 p[s].left로 표현된다. 따라서 이 배정문이 실행된 후의 상태(s')에서 포인터 p가 가리키는 개체는 p[s].left가 가리키는 개체와 같아야 하므로 'p[s'] = p[s].left'를 만족하여야 한다 (i.e., Φ[p]s' = Ξ[p->left]s). 또한 이 배정문에 의하여 포인터 p 이외의 다른 변수들의 값은 변경되지 않아야 하므로 (i.e., all r: V-p | s'.r=s.r) 프레임 조건 s'.y=s.y &&s'.L=s.L&&s'.q=s.q이 만족되어야 한다. 그림 9는 최종적으로 변환된 Alloy 술어를 보여준다.

```
pred doAssign5[s, s': State] {
        some p[s]
        p[s'] = p[s].left
        s'.y=s.y
        s.'L=s.L
        s'.p=s.p
}
```

(그림 9) p=p->left를 Alloy로 변환

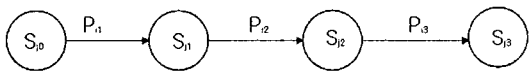
조건도 배정문과 거의 동일한 방식으로 변환할 수 있다. 배정문을 변환할 때와 다른 점은 조건에 의해 영향을 받는 변수가 없기 때문에 모든 변수가 프레임 조건을 만족해야 된다는 사실이다. 그림 10은 그림 4에서 조건 'y<p->data'을 변환 규칙 'Φ[p->f rel e]ss'에 따라 Alloy로 변환한 예를 보여준다.

```
pred doComp5[s, s': State] {
        some p[s]
        s'.y < p[s].data
        s'.y=s.y
        s.'L=s.L
        s'.p=s.p
        s'.q=s.q
}
```

(그림 10) 'y<p->data'를 변환한 예

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}; S_{j3}. 이와 같이 순차적으로 실행되는 문장들은 그림 11과 같은 형태로 상태 전이가 이루어진다.



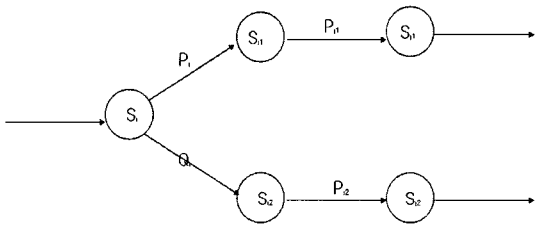
(그림 11) 순차적 상태 전이의 예

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 11에서 나타난 상태 전이 흐름은 매우 간단하게 다음과 같이 Alloy로 변환될 수 있다.

```
s in Sj0 => Pi1[s, s'] && s' in Sj1
s in Sj1 => Pi2[s, s'] && s' in Sj2
s in Sj2 => Pi3[s, s'] && s' in Sj3
```

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 조건문이 있는 경우에는 현재 상태에서 갈 수 있는 상태가 여러 개 존재한다. 예를 들어 프로그램이 "if C_i then S_{j1} else S_{j2}"와 같은 조건문을 포함하고 있다면 이 조건문은 그림 13과 같이 상태 전이가 표현된다.



(그림 12) 선택적 상태 전이의 예

그림 12에서 P_i는 조건문 C_i에 해당하는 Alloy 이 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 12에서 묘사된 상태 전이를 Alloy 논리식으로 변환하면 다음과 같다:

```
s in Si => Pi[s, s'] && s' in Si1 || Qi[s, s'] &&
s' in Si2
s in Si1 => Pj1[s, s'] && s' in Sj1
s in Si2 => Pj2[s, s'] && s' in Sj2
```

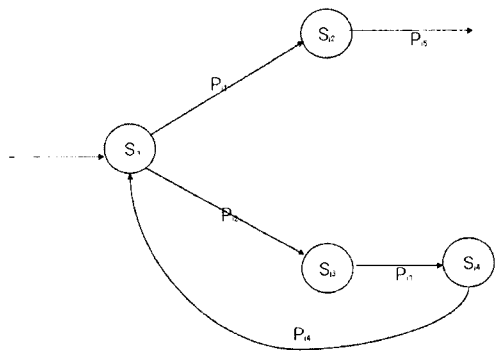
첫 번째 Alloy 식은 현재 상태가 Si라면 다음 상태는 S_{i1}이거나 S_{i2}이다. 이와 같이 선택적으로 상태 전이가 되는 경우는 논리연산자 '||'(Alloy에서 사용하는 or에 해당하는 논리 연산자)를 사용하여 표현한다. 그림 4의 프로그램에서 라인 8-10에 주어진 조건문- if (y < p->data) p = p->left; else p= p->right;-을 Alloy로 변환한 결과가 그림 13에 있다.

```
s in S6 => doComp5[s, s'] && s' in S7 ||
doComp6[s, s'] && s' in S8
s in S7 => doAssign5[s, s'] && s' in S2
s in S8 => doAssign6[s, s'] && s' in S2

pred doComp5[s, s': State] {
        doNothing[s, s']
        some p[s]
        s.y < (p[s]).data
}
pred doComp6[s, s': State] {
        doNothing[s, s']
        some p[s]
        s.y >= (p[s]).data
}
pred doAssign5[s, s' : State] {
        some p[s]
        p[s'] = p[s].left
        s'.y=s.y
        s'.L=s.L
        s'.q=s.q
}
pred doAssign6[s, s' : State] {
        some p[s]
        p[s'] = p[s].right
        s'.y=s.y
        s'.L=s.L
        s'.q=s.q
}
pred doNothing[s, s' : State] {
        s'.y = s.y
        s'.L = s.L
        s'.q = s.q
        s'.p = s.p
}
```

(그림 13) 선택적 상태 전이에 대한 Alloy 변환 예

그림 13에서 doComp5는 조건 'y<p->data'를 Alloy로 변환한 결과이고 doAssign5, doAssign6는 'p=p->left'와 'p=p->right' 문장들에 해당하는 Alloy 술어이다. Alloy 술어 doComp6는 원래의 프로그램 조건 'y<p->data'가 만족되지 않는 경우에 해당하는 Alloy 술어이다.

그런데 여기에서 주목할 점은 doComp5와 doComp6간의 관계이다. 언뜻 보면 doComp6은 !doComp5(여기에서 !은 not의 의미를 갖는 Alloy 논리연산자이다)와 같아야 하지만 실제로는 그렇지 않음을 그림 14에서 볼 수 있다. 그 이유는 프레임 조건 때문이다. doComp5는 조건 'y<p->data'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 'y<p->data'을 만족하지 않는 경우 즉, 'y>=p->data'을 실행하는 경우에도 모든 변수가 영향을 받지 않아야 하므로 단순하게 s in S6 => doComp5[s, s'] && s' in S7 || !doComp5[s, s'] && s' in S8라고 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 14) 반복의 예

그림 14는 이전에 이미 방문했던 상태로 다시 전이가 발생하는 상황을 묘사한 것이다. 이러한 경우는 프로그램에 반복문이 존재할 때 나타난다.

그림 14를 Alloy로 변환하는 것은 그림 12에서 설명한 것과 큰 차이가 없다. 만약 현재 상태가 Sj4라면 다음 상태는 이미 방문한 상태 Sj_1이어야 하며 전이가 일어나기 위해서는 P_{i4}를 만족해야 한다. 그림 14를 Alloy로 표현하면 다음과 같다:

s in Sj1 => P_{i1}[s, s'] && s' in S_{j2} || P_{i2}[s, s'] && s' in S_{j3}
s in Sj3 => P_{i3}[s, s'] && s' in S_{j4}
s in Sj4 => P_{i4}[s, s'] && s' in S_{j1}
s in Sj2 => 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 15에서 <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 우리가 일일이 각 상태에 대해 몇 개의 개체가 할당되어야 하는지 알 필요가 없으며 상태 전이도에 나타난 모든 상태들이 필요로 하는 전체 개수만을 기술하면 된다. Alloy 분석기의 SAT 해결도구가 상태전이도로 표현된 논리식을 만족시킬 수 있도록 자동적으로 각 상태에 적절한 개수의 개체들을 분배한다. 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 15는 그림 4에 주어진 이진 탐색 트리의 탐색 함수를 위에서 언급한 변환 과정을 거쳐 변환된 결과를 보여준다. 그림 15에서 'doAssign1', 'doComp1'등은 각각 'p=L', 'p!=NULL', ... 에 해당하는 Alloy 술어이다. 프로그램의 처음 시작은 S0부터라는 사실과 프로그램 종료는 S9이라는 사실은 각각 fact 'InitialAndFinal'을 통해 표현하고 있다.

```
fact doTrans {
        all s : State-so/last |
          let s' = so/next[s] | {
                s in S0 => doAssign1[s, s'] &&
                s' in S1
                s in S1 => doAssign2[s, s'] &&
                s' in S2
                s in S2 =>doComp1[s, s'] && s'
                in S3 || doComp2[s, s'] && s'
                in S9
                 s in S3 => doComp3[s, s'] &&
                 s' in S4 || doComp4[s, s'] &&
                 s' in S6
                 s in S4 => doAssign3[s, s'] &&
                 s' in S5
                 s in S5 => doAssign4[s, s'] &&
                 s' in S2
                 s in S6 => doComp5[s, s'] &&
                 s' in S7 || doComp6[s, s'] &&
                 s' in S8
                 s in S7 => doAssign5[s, s'] &&
                 s' in S2
                 s in S8 => doAssign6[s, s'] &&
                 s' in S2
                 s in S9 => doNothing[s, s'] &&
                 s' in S9
        }
}
fact InitialAndFinal{
        so/first in S0
        so/last in S9
}
```

(그림 15) 변환 결과

4. 테스트 데이터 생성

이 장에서는 변환된 Alloy를 기반으로 목표 블

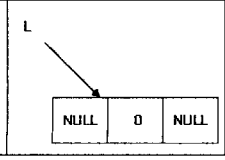
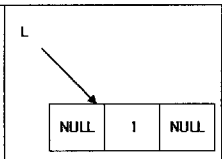
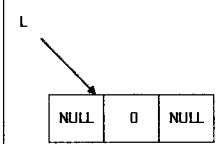
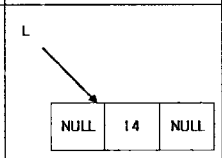
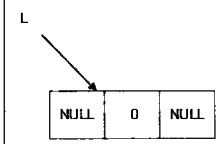
록을 실행하는 테스트 데이터를 자동으로 생성하는 방법에 대해 기술한다. 이를 위하여 우선 목표 블록을 변환된 Alloy 명세에 기술하는 방법과 목표 블록을 실행하는 경로에 대한 제약을 주는 방법에 대해서도 기술한다. 또한 입력 데이터에 대한 제약이 없는 경우와 제약이 존재하는 경우에 대해서도 그림 5의 이진 탐색트리로 테스트 데이터를 직접 생성하여 그 차이에 대해 기술한다.

4.1 경로에 대한 제약

목표 지향 프로그램 테스트는 기본적으로 프로그램의 특정 블록을 실행하는 입력 데이터를 구하는 것이다. 따라서 실행하고자 하는 프로그램 블록을 명시하는 것이 필요하다. 프로그램 시작점과 종료점을 fact를 사용하여 명시하는 것에 반해 실행하고자 하는 프로그램 블록은 'pred'를 사용하여 표현한다. 그 이유는 프로그램 시작점과 종료점은 테스트 데이터 생성 문제에 있어 변경되지 않지만 실행 프로그램 블록은 테스트 목적에 따라 변경되기 때문이다. 예를 들어 그림 17은 그림 5에서 5번 문장 'q=p'를 실행하는 테스트 데이터를 생성하기 위해 작성된 Alloy 술어이다.

```
pred testIt {
        some S4
}
```

(그림 17) 프로그램 목표 블록 명시 예

프로그램 문장	실행경로	입력 변수	
		L	y
p=L, q=NULL	0-1-2-9	NULL	1
p!=NULL	0-1-2-3-6-8-2-9		1
y==p->data, q=p, p=NULL	0-1-2-3-4-5-2-9		1
y!=p->data,	0-1-2-3-6-8-2-9		1
y<p->data, p=p->left	0-1-2-3-6-7-2-9		0
y>p->data, p=p->right	0-1-2-3-6-8-2-9		1
p==NULL	0-1-2-9	NULL	-16

(그림 18) 이진 탐색 함수 테스트 데이터

그림 17에서 'some S4'는 상태 S4는 공집합이 아니어야 된다는 사실을 나타낸다. 따라서 이 술어(e.g, testIt)를 만족하기 위해서는 Alloy 분석기가 적어도 하나이상의 개체를 S4에 할당해야 한다. 즉, InitAndFinal과 더불어 testIt 술어를 만족하기 위해서는 Alloy 분석기가 S0부터 출발하여 최소한 한번은 S4를 지나고 S9에서 끝나는 프로그램 경로(e.g., 상태들의 시퀀스)를 검출할 수 있으면 된다. S4를 실행하면 자동적으로 목표 배정문 'q=p'가 실행되게 된다. 이 때 찾아낸 경로에서 이 경로를 실행하기 위해 제공되어야 하는 입력 값들도 알 수 가 있는데 초기상태, 즉 S0에서 y, L의 필드들이 갖는 값들이 5번 문장을 실행하기 위해 제공해야 하는 테스트 데이터들이다.

그림 18은 그림 5의 이진 탐색 함수의 각 프로그램 문장에 대해 Alloy 분석기가 생성한 테스트 데이터와 실행 경로를 보여준다. 그림 18의 '프로그램 문장'열은 실행하기를 원하는 목적 프로그

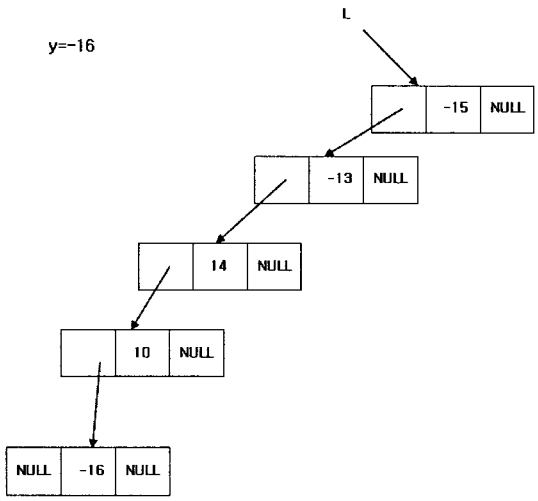
램 문장이다. 여기에 여러 문장들이 있는 경우는 Alloy 분석기가 이 문장들에 대해 동일한 실행 경로와 동일한 입력 L, y 값을 생성하였기 때문이다. 이는 매우 자연스러운 현상이다. 그림 5에서 볼 수 있듯이 'p=L'이 실행되면 'p=NULL'은 자동적으로 실행된다. 따라서 'p=L'에 대해 생성된 테스트 데이터는 'p=NULL'을 실행하기 위해 사용될 수 있다.

그러나 그림 18에서 볼 수 있듯이 입력 트리가 NULL이거나 단지 하나의 노드로만 구성된 매우 간단한 자료 구조들만을 생성함을 볼 수 있다. 물론 이와 같이 생성된 자료 구조가 원하는 목적 프로그램 문장을 실행할 수 있다할지라도 프로그램의 많은 부분들이 테스트 대상에서 누락될 수 있는 위험이 있다. 그림 18에서 실행된 프로그램 경로를 보면 그림 4의 반복문이 기껏해야 한번만 수행되는 경우만을 테스트했음을 알 수 있다. 따라서 보다 다양한 경우에 대해 프로그램을 테스트하기 위해서는 목적 프로그램 문장만을 명시할 뿐만 아니라 실행 경로에 대한 제약도 명시적으로 줄 필요가 있다.

```
pred pathConstraint {
        #S7 > 2
}
pred testIt {
        some S4
}
```

(그림 19) 경로 제약 예

실행 경로에 대한 제약은 Alloy로 쉽게 표현할 수 있다. 예를 들어, 그림 19에 주어진 Alloy 명세를 생각해보자. 이 명세는 최소한 'p=p->left' 가 두 번 이상 수행되어야 하며 동시에 'q=p' 문장을 수행하는 테스트 데이터를 구하는 문제를 Alloy 논리식으로 작성한 것이다. 이 명세를 주었을 때 Alloy 분석기는 입력 변수 L, y에 대해 그림 20에 보여진 입력 트리 구조 및 데이터를 생성한다. 생성된 테스트 데이터가 0-1-2-3-6-7-2-3-6-7-2-3-6-7-2-3-6-7-2-3-4-5-2-9를 수행한다. 이 실행 경로는 그림 19에 기술된 경로 제약을 모두 만족한다.



(그림 20) 그림 19의 경로제약을 만족하는 테스트 데이터

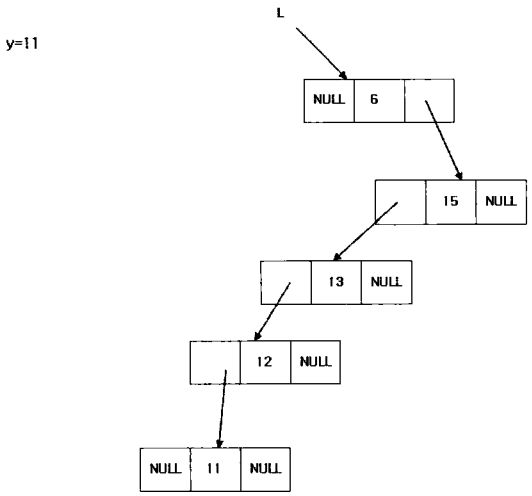
4.2 입력에 대한 제약

그림 4의 함수는 입력이 이진 탐색 트리를 가정하고 있다. 그러나 그림 20에 보여진 트리는 왼쪽 서브 트리의 키(e.g., data 필드의 값)들이 루트 노드의 키보다 보다 작아야 하는 기본적인 이진 탐색 트리의 구성 요건을 만족하지 못하고 있다. 경우에 따라서는 프로그램 테스트를 수행할 때 입력에 대한 제약들을 고의적으로 만족하지 않는 데이터를 테스트 데이터로 사용하는 것이 보다 효과적일 수 있다. 그러나 이러한 테스트 방법도 우선은 입력에 대한 제약을 만족하는 테스트가 수행되고 난 후에 이를 보완하는 수단으로서 사용되는 것이 일반적이다.

```
fact OnlyOneParent {
        all n: Node | lone (n.~left +n.~right)
}
```

```
fact BST {
        all n: Node | {
            some n.left => n.data > data[n.left]
            some n.right => n.data < data[n.right]
        }
}
```

(그림 21) 이진 탐색 트리



(그림 22) 입력 제약을 만족하는 테스트 데이터

이 논문에서는 입력에 대한 제약사항들이 존재하는 경우에 이들을 명시적으로 기술할 것을 요구한다. 그림 21은 이진 탐색 트리에 대한 제약사항을 Alloy로 기술한 것을 보여준다. OnlyOneParent는 모든 노드들은 기껏해야 하나의 부모 노드만을 가진다는 일반적인 트리 속성을 기술하며 BST는 왼쪽 서브트리의 키들은 루트의 키보다 작아야 한다는 사실과 오른쪽 서브트리의 키들은 루트의 키보다 커야 한다는 이진 탐색 트리가 만족해야 하는 속성을 표현한다. 이러한 제약과 그림 19의 경로 제약을 동시에 만족하도록 Alloy 명세을 주었을 때 Alloy 분석기는 그림 22의 테스트 데이터를 생성한다.

그림 22의 y이 값과 L이 가리키는 트리가 'p=p->left'가 최소한 두 번 수행되고 동시에 'q=p'를 수행하도록 하는 테스트 데이터이며 이진 탐색트리의 모든 요건들을 만족하는 사실을 쉽게 알 수 있다.

5. 결 론

이 논문에서는 테스트 데이터를 자동으로 생성하기 위한 목적 지향적 방법을 제안하였다. 이는 테스트 데이터 생성을 위해 완전한 프로그램 경로를 제공해야 하는 경로 지향 테스트 데이터 생성 방법과는 달리 프로그램 상의 특정 제어점 또는 경로상의 일부분만 주어져도 테스트 데이터를 자동으로 생성할 수 있는 기존의 방법을 프로그램이 구조체 및 포인터를 사용하는 경우에도 적용할 수 있도록 확장한 방법이다.

이 논문에서 제안한 방법은 기본적으로 테스트 데이터 생성 문제를 SAT(SATisfiability)로 변환하여 테스트 데이터를 생성하는 정적 방법이다. 그러나 SAT 해결도구를 사용하여 직접적으로 사용하는 방법 대신에 우선 프로그램을 1차 관계 논리에 바탕을 둔 형식 명세 언어 Alloy로 변환하여 Alloy 분석기를 통하여 테스트 데이터를 생성하는 방법을 제안하였다.

제안된 방법은 기존의 테스트 데이터 자동 생성 방법보다는 많은 기술적인 진보가 있었지만 보완할 점도 많이 있다. 우선 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. 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.

참고문헌

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

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

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

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

[5] Marques-Silva, J. P. and Sakallah, K. A., "GRASP: A Search Algorithm for Propositional Satisfiability", *IEEE Trans.. on Computers,* vol. 48, pp. 506-521, 1999.

[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., Alloy 4.0 Quickstart guide, http://alloy.mit.edu, 2007.

[9] 정인상, "Alloy 명세 기반 자동 테스트 데이터 생성 기법", 한국정보처리학회 논문지 D, 14권 2호, pp. 191-202, 2007.

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

[11] Visvanathan, S. and Gupta, N. "Generating Test Data for Functions with Pointer Inputs", *In Proc. 17th IEEE International Conf. Automated Software Eng.*, pp. 149-160.

[12] 정인상, "자동화된 프로그램 시험을 위한 입력 자료구조의 모양 식별", 한국정보과학회 논문지, 31권 10호, pp. 1304-1319, 2004

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

[14] 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.

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

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

◐ 저 자 소 개 ◑

정 인 상(In-Sang Chung)
1987년 서울대학교 컴퓨터공학과 졸업(학사)
1989년 한국과학기술원(KAIST) 전산학과 졸업(석사)
1993년 한국과학기술원(KAIST) 전산학과 졸업(박사)
1999~현재 한성대학교 컴퓨터공학과 교수
관심분야 : 소프트웨어 공학, 소프트웨어 테스팅
E-mail : insang@hansung.ac.kr