DOI: 10.3745/KIPSTD.2010.17D.4.297

다단계 스택 지향 포인터가 있는 프로그램 테스트를 위한 테스트 데이터 자동 생성

정 인 상†

요 약

최근에 콘콜릭 테스트이라 불리는 새로운 테스팅 방법이 많은 관심을 받고 있다. 콘콜릭 테스팅은 높은 테스트 커버리지를 달성하기 위해 실제 프로그램 수행과 심볼릭 수행을 결합하여 테스트 데이터를 생성한다. CREST는 콘콜릭 테스팅을 구현한 대표적인 open-source 테스트 도구이다. 그러나 현재 CREST는 입력으로 정수형만 다룬다. 이 논문은 포인터형인 입력이 존재하는 경우에 자동 테스트 데이터 생성을 위한 새로운 규칙을 제안한다. 이 규칙들은 C 프로그램에서 주로 사용되는 다단계 스택 지향 포인터를 효과적으로 처리한다. 또한, 이 논문에서는 제안된 규칙을 구현한 vCREST라 불리는 도구에 대해 기술하고 C 프로그램에 적용한 결과도 함께 기술한다.

키워드: 콘콜릭 테스팅, 자동 테스트 데이터 생성, 포인터, CREST

Automated Test Data Generation for Testing Programs with Multi-level Stack-directed Pointers

In-Sang Chung†

ABSTRACT

Recently, a new testing technique called concolic testing receives lots of attention. Concolic testing generates test data by combining concrete program execution and symbolic execution to achieve high test coverage. CREST is a representative open-source test tool implementing concolic testing. Currently, however, CREST only deals with integer type as input. This paper presents a new rule for automated test data generation in presence of inputs of pointer type. The rules effectively handles multi-level stack-directed pointers that are mainly used in C programs. In addition, we describe a tool named vCREST implementing the proposed rules together with the results of applying the tool to some C programs.

Keywords: Concolic Testing, Automated Test Data Generation, Pointer, CREST

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 있어왔다. 그 중에서 최근에 관심을 끄는 연구로 콘콜릭 테스트(concolic test)라 불리는 방법이 제안되었다[1]. 콘콜릭 테스트는 동적 테스트 방법과 심볼릭 수행을 결합하여 높은 테스트 커버리지를 달성하기 위해 개발되었다.

가장 기본적인 동적 테스트 방법으로 무작위 테스트 방법이 있다. 이 방법은 프로그램에 구조에 대한 지식이 없이 임의적으로 테스트 데이터를 선정하기 때문에 동일 프로그램 경로를 실행하는 입력들을 중복되게 선정할 수 있는 가능성이 많으며 이로 말미암아 테스트 커버리지도 낮다. 이에 반해 심볼릭 수행은 테스트하고자 하는 프로그램 경로에 대한 제약 조건을 추출하기 위해 실제 프로그램을 어떤 특정한 값으로 실행하기보다는 모든 입력 도메인에 있는 값들

※ 본 연구는 2010년도 한성대학교 교내연구비 지원과제임
† 정 회 원: 한성대학교 컴퓨터공학과 교수
논문접수: 2010년 6월 14일
수 정 일: 1차 2010년 7월 26일
심사완료: 2010년 7월 26일

을 대표할 수 있는 심볼릭 값을 사용하여 프로그램을 (정적으로) 실행하는 방식이다[2]. 심볼릭 수행의 결과로 프로그램의 모든 실행 경로들이 트리 구조로 표현된다. 이 때 트리의 간선(edge)들은 프로그램 각 조건문에서 가능한 분기들이다. 이렇게 구해진 트리의 각 경로에 대해 경로 제약 조건을 추출하고 제약 조건을 만족하는 입력 변수의 값을 구하면 해당 경로를 수행할 수 있는 테스트 데이터가 구해지게 되는 것이다.

그러나 심볼릭 수행은 배열의 첨자를 처리할 때 문제가 발생한다[3]. 예를 들어 "x=TestGen[i+j]" 와 같은 문장에서 배열 "TestGen"의 원소가 간접적으로 참조 될 때 변수 'i'와 'j'가 특정한 값으로 바운드 되지 않기 때문에 실제 어느 배열의 원소를 참조하는지 알 수 있는 방법이 없다. 또한 프로그램에서 포인터를 사용하여 동일한 기억 장소를 다른 변수 이름을 이용하여 접근하는 경우(aliasing problem)와 비선형 연산을 처리할 때 문제가 될 수 있다.

콘콜릭 테스트는 심볼릭 수행의 이 같은 문제를 실제 프로그램의 수행을 통해 처리하고자 하는 시도이다. 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 분기가 실행되거나 사용자가 지정한 종료 조건을 만족할 때까지 반복된다.

CREST는 콘콜릭 테스트를 기반으로 C 프로그램을 대상으로 테스트 데이터를 자동으로 생성하는 대표적인 open-source 테스트 도구이다[4]. 그러나 현재 CREST는 프로그램의 입력을 정수형으로 제한하고 있으며 포인터가 입력이 되는 경우는 제대로 처리하지 못한다. 프로그램 함수 입력이 포인터인 경우에 테스트 데이터는 리스트, 트리 또는 그래프와 같은 2차원적인 자료구조로 표현된다. 따라서 포인터를 고려한 테스트 데이터 자동생성 방법은 모든 경로들을 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[4, 5].

모양 문제 해결을 위해 이 논문에서는 포인터가 있는 문장을 처리할 때 실행에 필요한 최소한의 'points-to'관계를 생성한다. 여기에서 'points-to' 관계는 포인터 변수가 무슨 메모리 로케이션을 가리키고 있는지에 대한 관계를 의미한다. 예를 들어 만약 주어진 프로그램 경로 상에 배정문 'p=*q'를 처리한다고 가정하자. 이 배정문이 오류 없이 실행되기 위해서는 포인터 q가 'NULL'이 되어서는 안되므로 'q ≠NULL'이라는 제약조건을 추출할 수 있다. 만약 이미 q가 'NULL' 이라면 해당 프로그램 경로는 모순되므로 실행 불가능한 경로가 된다는 사실도 알 수 있다. q의 값이 아직 결정되지 않은 상태라면 q는 'NULL'을 가질 수 있는 가능성이 배제가 된다. 따라서 q는 구체적인 기억 장소를 가리키도록 조정한다. 또한, 입력 자료 구조 모양에 영향을 주지 않은 문장들에 대해서는 'points-to' 관계를 생성하지 않기 때문에 시간 및 공간의 낭비를 최소화 할 수 있다. 이 논문에서 제안한 모양 식별 방법을 CREST와 연동하여 동작하는 vCREST라는 도구에 구현하였으며 현재는 다단계 스택 지향 포인터를 지원한다.

이 논문은 다음과 같이 구성된다. 2장에서는 모양 문제가 무엇인지 보다 자세하게 기술하고 기존의 모양 문제에 대한 접근 방식에 대해 간략하게 소개한다. 3장에서는 이 논문에서 제안한 새로운 모양 생성 방법에 대해 설명한다. 4장에서는 새롭게 개발한 vCREST에 대해 기술하며 이 도구를 다른 도구와 비교한다. 또한 간단한 실험결과에 대해 언급한다. 마지막으로 5장에서 결론 및 향후 연구에 대해 기술한다.

2. 연구 배경

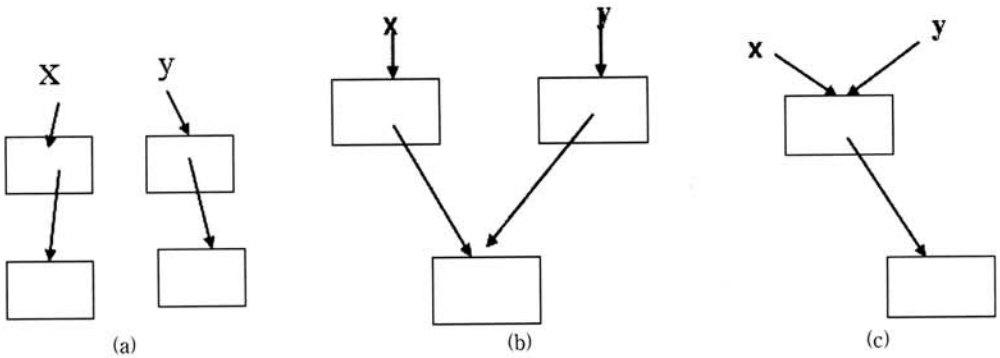
2.1 모양 문제

모양 문제를 구체적으로 알아보기 위해 (그림 1)에 있는 프로그램에서 프로그램 경로 <1,2,3,4,6,8,9,10,...> 를 실행하기 위해 어떤 입력 값(여기에서는 포인터 x와 y의 값 및 정수 v의 값)을 가져야 하는지 알아보자.

만약 포인터 x와 y가 (그림 2)(a)에 주어진 자료구조를 가리키고 있다면 주어진 프로그램 경로를 실행할 수 있을까? 명확하게 알 수 있듯이 그에 대한 답은 '아니오'이다. 그 이유는 포인터 x와 y는 또 다른 포인터 p와 q를 가리키고 있고(문장 1, 2) 문장 3의 조건이 참이 되어야 하기 때문에

```
void Example(int **x, int **y, int v) {
int *p, *q, *r, z;
1:      p = *x;
2:      q = *y;
3:      if (p == q) {
4:              if(p == NULL)
5                       *q = v;
6:              else if(q == NULL)
7                       *p = v;
                else {
8:                      r = &z;
9:                      *r = 10;
10:                     if(z == v)
11:                             ....
                }
        }
        else {
12:             *p = v;
13:             *q = v;
        }
}
```

(그림 1) 모양 분석이 필요한 예제 프로그램[5]



(그림 2) 입력 자료 구조의 모양

포인터 p와 q는 동일한 장소를 가리켜야 한다. (그림 2)(b)는 주어진 프로그램 경로를 실행할 수 있는 입력 자료 구조의 모양(shape)을 보여준다.

이 예제로부터 포인터가 있는 프로그램을 다루는 경우에는 입력 자료 구조의 모양을 정확하게 생성하는 것이 매우 중요하다는 사실을 알 수 있다. 입력 자료 구조의 모양을 결정하기 위해서는 얼마나 많은 노드가 필요한지 노드들 간의 연결이 어떻게 되어야 하는 지를 결정해야 한다.

또 생각해 보아야 할 점은 주어진 프로그램 경로를 수행할 수 있는 입력 자료 구조의 모양은 하나 이상이 될 수 있다는 사실이다. 예를 들어 (그림 2)(c)에 주어진 입력 자료 구조를 생각해보자. 이 자료 구조는 (그림 2)(b)에 주어진 입력 자료 구조와 마찬가지로 (그림 1)의 주어진 경로를 실행할 수 있다. 그 이유는 주어진 프로그램 경로 어디에서도 포인터 x와 y가 달라야 한다는 제약조건을 추출할 수 없기 때문이다. 이와 같이 주어진 하나의 프로그램 경로를 실행할 수 있는 입력 자료 구조의 모양이 하나 이상 있을 수 있는 경우에는 어떤 모양을 택할 것인지를 일관되게 선택할 수 있는 규칙이 필요하다.

다시 (그림 1)의 프로그램으로 돌아가서 조건 3은 '참', 조건 4는 '거짓' 조건 5는 '참'이 되는 프로그램 경로를 생각해보자. 이 경우는 해당 프로그램 경로를 실행 할 수 있는 입력 값들을 발견할 수 없다. 그 이유는 조건 3이 '참'이 되기 위해서는 포인터 p와 q가 동일한 값을 가져야 하는데 조건 4가 '거짓'이 되어야 하므로 포인터 p는 NULL이 되어야 한다. 한편 조건 5가 '참'이 되어야 하므로 포인터 q는 NULL이 아니어야 한다. 이는 p와 q가 동일한 값을 가져야 하는 제약 조건에 위반되므로 해당 프로그램 경로를 실행할 수 있는 입력 값이 존재하지 않음을 알 수 있다. 유용한 테스트 도구에서는 이와 같은 경로가 주어졌을 때 입력 자료 구조의 모양을 발견할 수 없다는 사실을 사용자에게 즉각적으로 알려 주는 것이 필요하다.

2.2 관련 연구

포인터를 고려하여 테스트 데이터를 생성하는 방법은 가장 먼저 Korel에 의해 제안되었다[6]. 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 일어날 수 있도록 입력 변수들의 값을 변경한다. 만약 현 상태에서 가능한 어떠한 입력 값도 원하는 방향으로 분기가 일어나지 않는다면 이전 분기로 되돌아가 새로운 입력 값을 다시 탐색하는 절차를 반복한다. 그러나 이 방법은 주어진 경로가 실행 불가능한 경로인 경우에는 이 사실을 즉각 검출할 수 없어 입력 값을 찾기 위해 많은 시간과 노력이 소요될 수 있다는 단점이 있다.

Visvanathan과 Gupta도 프로그램이 포인터를 사용하는 경우에 테스트 데이터를 생성하는 경로 지향적 방법을 제안하였다[7]. 그들이 제안한 방식은 두 단계 접근 방식(Two phase approach)이라고 하는데 그 이유는 이 방식이 두 단계로 구성되어져 있기 때문이다. 첫 번째 단계는 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 변수의 값을 계산하는 단계이다. 이 단계를 데이터 값 생성(data value generation phase) 단계라 한다.

최근에 포인터를 사용하는 프로그램을 테스트하기 위한 SGEN이라는 도구가 개발되었다[5]. 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. 따

라서 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 후에만 알 수 있기 때문이다. 반면에 SGEN에서는 모양 생성과 동시에 포인터 타입이 아닌 입력 값에 대한 제약식이 생성되므로 보다 효과적으로 실행 불가능한 지의 여부를 검사할 수 있다.

그러나 SGEN이 취한 방법은 주어진 프로그램 변수를 논리 변수(logical variable)로 다루기 위해 기본적으로 프로그램 경로의 모든 문장을 SSA(Static Single Assignment) 폼[8]으로 변환한다. 이는 각 프로그램 변수에 대해 많은 SSA 변수들이 생성되는 결과를 가져온다. 또한 입력 자료 구조의 모양에 전혀 영향을 주지 않는 문장들에 대해서도 'points-to' 관계를 유지하기 때문에 많은 공간의 낭비를 초래할 수 있다. 또한 이 방식은 기본적으로 정적 방식이기 때문에 배열이나 포인터 사용으로 인한 이명 문제를 처리하는 데에 한계가 있다. 마지막으로 SGEN은 경로지향적 방법이기 때문에 사용자가 일일이 테스트하고자하는 프로그램 경로를 제공하여야 한다.

INKA[9]라 불리는 테스트 데이터 생성도구는 이 논문에서처럼 다중 스택 지향 포인터가있는 프로그램에 대해 테스트 데이터를 자동으로 생성한다. 이 도구가 앞에서 기술한 테스트 데이터 생성도구와 다른 점은 경로 지향적 방식이 아닌 목적 지향적으로 테스트 데이터를 생성한다는 점이다.

목적 지향(goal-oriented) 테스트 데이터 생성 방법은 특정 프로그램 경로를 제공하는 대신에 프로그램 상의 한 블록 또는 분기를 주고 이를 실행할 수 있는 입력 값을 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 블록을 실행할 수 있는 다른 경로를 자동으로 탐색하여 입력 값을 생성할 수 있다. 그러나 이 도구는 프로그램에서 내부 변수로 포인터를 사용하는 경우만 고려하였고 입력 변수가 포인터인 경우에는 제외하였다. 따라서 이 논문에서 다루는 모양문제 대한 처리는 하지 못한다.

3. 새로운 모양 식별 방법

3.1절에서는 이 논문에서 제안하는 방법을 기술하는 데 필요한 몇 가지 용어에 대해 정의한다. 3.2절에서는 기존의 콘콜릭 테스트를 기반으로 포인터를 처리하는 절차에 대해 기술한다. 3.3절에서는 경로 $<s_1, ..., s_n>$의 부분 경로 $<s_1, ..., s_{i-1}>(i\le n)$를 실행할 수 있는 입력 자료 구조를 식별하기 위해 각 문장을 변환 함수(transfer function)로 간주하여 각 문장 형태에 따라 변환 함수를 정의하고 설명한다. 3.4절에서는 3.3.절에서 기술한 변환 함수를 예로 들어 설명한다.

3.1 용어 정의

이 논문에서 제안하는 방법은 경로 $<s_1, ..., s_n>(i\le n)$의 부분 경로 $<s_1, ..., s_{i-1}>$를 실행할 수 있는 입력 자료 구조를 'σ_i'로 표현한다. 즉,

$$\sigma_i \in State=Var->Loc$$

이 때 Var는 프로그램 변수들의 집합으로 표현하고 Loc는 논리적 메모리 로케이션들의 집합을 나타낸다. 이 때 $\sigma_i(p)$는 다음과 같은 의미를 갖는다:

- $\top_X$: p는 X에 속한 나머지 포인터들과 동일한 어떤 논리적 메모리 로케이션도 가리킬 수 있다. 즉 X는 p와 동일한 로케이션을 가리키는 포인터들의 집합이다. 이 때 p는 NULL도 될 수 있다.
- $\perp$:p가 포인터가 아니거나 정의되지 않았다. 예를 들면 p가 정수형 변수라면 $\sigma_i(p)=\perp$.
- l_i: p가 논리적 주소 l_i를 갖는 메모리 로케이션을 가리키고 있음을 나타낸다. 논리적 주소를 갖는 메모리 로케이션들은 서로 동일한 메모리 로케이션이 될 수 있다. 즉, $k<m$일 때 $\sigma_k(p)=l_i$, $\sigma_k(q)=l_j(i\neq j)$라면 $\sigma_m(p)=l_w$, $\sigma_m(q)=l_w$. 또한 이 논문에서 논리적 주소를 갖는 메모리 로케이션은 테스트 대상이 되는 프로그램 함수 외부에서 만들어진 메모리 영역만을 의미한다. 이런 측면에서 논리적 주소를 갖는 메모리 로케이션과 물리적 주소를 갖는 메모리 로케이션의 의미가 다르다.

각 입력 포인터 p의 값을 $\top_{(p)}$로 초기화하여 p가 어느 메모리 로케이션도 가리킬 수 있음을 나타낸다. 이 값은 프로그램의 경로가 실행됨에 따라 보다 구체화된다. 또한 '$\perp_\pi$'를 사용하여 주어진 경로 π가 실행 불가능함을 나타낸다, 즉 '$\perp_\pi$'는 π를 실행할 수 있는 입력이 존재하지 않는다는 의미이다.

또한 k_i는 물리적 주소를 나타내는데 사용하고 변수 p가 현재 가리키고 있는 메모리 로케이션은 'pts(p)'로 나타낸다. 만약 p가 포인터가 아니라면 'pts(p)=$\perp$'. 포인터 p에 대해서 함수 'pts(p)'는 '$\sigma_i(p)$'와 다를 수 있다. 'pts(p)'는 p가 가리키고 있는 현재 메모리 로케이션을 의미하며 이 로케이션은 논리적 주소나 물리적 주소를 모두 가질 수 있다. 그러나 '$\sigma_i(p)$'는 논리적 주소를 갖는 메모리 로케이션만을 가리킬 수 있으며 경로 $<s_1, ..., s_n>$의 부분 경로 $<s_1, ..., s_{i-1}>(i\le n)$를 실행할 수 있는 입력 자료 구조를 나타내는데 사용한다.

이 논문에서는 논리적 메모리 로케이션들이 시간에 따라 순서화 되어 있다고 가정한다. 즉, 프로그램 수행 중에 논리적 메모리 로케이션 l_i에 논리적 메모리 로케이션 l_j보다 값

이 시간적으로 먼저 할당되었다면 '$l_i \lesssim l_j$'로 나타내고 시간적으로 앞섰다고 말한다. 이러한 개념은 두 (논리적) 메모리 로케이션을 병합할 때 사용된다.

또한 이 논문에서는 프로그램에서 사용하는 변수들을 두 집합으로 분리한다. 프로그램 경로 $<s_1, ..., s_n>$가 주어졌을 때 $I_{i(i\leq n)}$는 (부분) 경로 $<s_1, ..., s_i>$를 실행하는 입력 자료 구조에 잠재적으로 영향을 줄 수 있는 포인터를 포함하며 N_i는 영향을 줄 수 있는 변수들을 포함한다. 만약 ψ가 모든 변수들의 집합이라고 하면 $\psi=I_i \cup N_i$, $I_i \cap N_i=\varnothing$.

I_0는 입력 포인터 변수들로 초기화 되며 이를 제외한 나머지 변수 들은 N_0에 속한다. Ii 및 Ni는 처리하는 문장 si에 따라 계산된다(3.3절 참조). 이 논문에서는 고려하는 배정문은 다음과 같다: 'x=y', 'x=*y', '*x=y', 'x=&y', 'x=NULL'. 복잡한 문장들은 위와 같은 형태로 변환하여 다룬다. 예를 들면, '*x=*y'는 't=*y;*x=t'로 분할하여 처리 할 수 있다.

이와 같은 변수의 분류에 따라 프로그램 문장들 중에서 잠재적으로 입력 자료 구조 모양에 영향을 줄 수 있는 문장을 식별할 수 있다. 만약 문장 si에 관련된 변수들 중 최소한 하나가 I_{i-1}에 속한다면 si를 잠재적 모양 생성문(potentially shape generating statement, PSGS)이라 하며 이 문장이 실행되면 입력 자료 구조의 모양이 보다 명확하게 결정될 수 있음을 의미한다.

3.2 콘콜릭 테스트를 이용한 모양 식별 절차

문장 s에 대한 변환 함수는 $\Sigma=(\sigma,\xi, I,N)$로 정의된 상태에 대한 함수로 정의된다. 즉, [s]: Σ->Σ. 여기에서 σ, I, N은 3.1절에서 정의한대로 각각 입력 자료 구조의 모양을 나타내는 입력 포인터에 대한 'points-to' 관계를 표현하고 I는 잠재적으로 입력 자료 구조에 영향을 줄 수 있는 포인터 변수들을 포함하고 있으며 N은 I에 속하지 않는 나머지 변수들을 포함한다. 'ξ'는 포인터들이 동일한 대상을 가리키고 있는지 여부를 알려주는 제약 조건이다. 제안된 방법은 다음과 같은 절차를 수행하여 입력 자료 구조 모양을 식별한다:

(1) 먼저 C 언어로 작성된 프로그램 소스 코드를 입력으로 받아 초기 상태 $\Sigma_0=(\sigma_0,\xi_0, I_0,N_0)$에서 수행한다. 초기 상태 $\Sigma_0=(\sigma_0,\xi_0, I_0,N_0)$는 다음과 같이 정의된다. 모든 입력 포인터 변수 p들은 $\top_{\{p\}}$로 초기화 되어있다고 가정한다. 즉, '$\delta_0(p)=\top_{\{p\}}$'. '$\top p$'는 포인터 집합 P에 속한 모든 포인터들의 값이 아직 정해지지 않았다는 의미이다. 문장이 실행되어 감에 따라 'NULL'이 될 수도 실제 구체적인 메모리 영역이 될 수도 있다. 또한 입력 포인터 변수들은 집합 I_0에 포함되었다고 가정하고 입력 포인터를 제외한 나머지 모든 변수들은 N_0에 포함된다고 가정한다. 포인터 형이 아닌 나머지 입력 변수들의 값은 무작위로 생성된 값을 할당받는다.

(2) 문장 si를 수행하는 경우에 si가 잠재적 모양 생성문인 경우인지를 판별하여 다음과 같이 달리 수행한다. si가 잠재적 모양 생성문인 경우에는 3.3절에 주어진 문장에 따른 변환 함수에 따라 상태 변환을 수행한 후에 콘콜릭 테스트를 수행한다. 이에 반해 문장 si가 잠재적 모양 생성문이 아닌 경우에는 상태 변환을 수행하지 않고 콘콜릭 테스트를 수행한다. 즉, 기본적으로 이 논문에서 제안하는 모양 생성 알고리즘은 각 문장이 입력 자료 구조에 영향을 미치는지를 검토하여 전혀 영향을 미치지 못하는 문장들은 고려 대상에서 제외하는 방식을 취하기 때문에 효과적으로 입력 자료 구조의 모양을 식별할 수 있다.

(3) 콘콜릭 테스트가 문장 si를 수행하면 si가 배정문인 경우에는 정의되는 변수에 해당하는 심볼릭 표현식을 갱신한다. 만약 si가 조건 분기문인 경우에는 해당 조건 분기의 조건식에 해당하는 심볼릭 표현식을 계산하여 지금까지 실행된 (부분)경로 $<s_1, ..., s_i>$에 대한 심볼릭 경로 조건식이 생성된다.

(4) 마지막 문장 s_n의 실행이 종료되면 실행된 프로그램 경로 $\pi=<s_1, ..., s_n>$를 수행할 수 있는 입력 자료 구조의 모양이 생성되고 콘콜릭 테스트에 따라 실행된 경로의 심볼릭 경로 조건식 ϕ_π이 생성된다. 이 경로 조건식 ϕ_π는 다음 입력 값을 생성하기 위해 ϕ_π의 조건 중 하나를 부정하여 새로운 경로 조건 ϕ'_π을 생성한다. ϕ'_π를 만족하는 입력 값은 다음 실행을 위한 포인터 형이 아닌 입력 변수의 값이 된다.

3.3 변환 함수

3.3.1 배정문에 대한 변환 함수

(그림 3)~(그림 6)은 배정문에 대한 변환 함수를 보여준다. 변환 함수는 '$\odot$' 는 함수 'overriding' 연산자를 사용하며 이 연산자는 다음과 같은 의미를 지닌다. f$\odot$g의 도메인은 f와 g의 두 도메인의 합집합으로 정의된다. g의 도메인에 있는 값에 대해서는 f$\odot$g는 g와 일치하며 그 외의 값에 대해서는 f와 일치한다.

(그림 3)(a)는 'p=q'로 주어진 배정문을 처리하기 위한 변환함수를 보여준다. 해당 변환 함수는 q가 어떤 메모리 로케이션을 가리키고 있는지 아직 결정되지 않았어도 포인터 p는 q가 가리키는 메모리 로케이션을 가리켜야한다는 사실을 알 수 있다. 즉, 나중에 q가 구체적으로 특정 메모리 로케이션을 가리킨다는 사실이 밝혀지면 p도 q와 동일한 메모리 로케이션을 가리켜야 한다. 이러한 사실을 나타내기 위해 q가 '$\top_X$'를 가리키고 있다면 p를 집합 x에 추가한다.

(그림 3)(a)의 마지막(다섯번 째) 경우는 q가 이미 특정 메모리 로케이션을 가리키는 경우에 해당된다. 이 경우는 해당 배정문이 입력 자료 구조의 모양을 변경하거나 전혀 영향을 주지 않는 경우에 해당되기 때문에 포인터 p만 N_i에 포함시키고 별다른 처리는 수행하지 않는다. (그림 3)(b)의 두 번째 경우도 이와 같이 간주할 수 있다. 다만 첫 번째 경우에서처럼 p가 가리키는 메모리 영역이 아직 결정되지 않은 상황이고 p와 동일한 영역을 가리켜야 하는 포인터들

$[p=q](\sigma_{i-1}, \xi_{i-1}, I_{i-1}, N_{i-1})=$

	CASE	
1	$\sigma_{i-1}(p)=\top_X \wedge \sigma_{i-1}(q)=\top_Y$	$\sigma_i=\sigma_{i-1}\odot\{(r, \top_{Y\cup\{p\}})\|\text{for } \forall r1\in Y\cup\{p\}\}\odot\{(r2, \top_{X\setminus\{p\}})\|\text{ for}\forall r2\in X\setminus\{p\}\}$ $I_i=I_{i-1}$, $N_i=N_{i-1}$, $\xi_i=\xi_{i-1}$
2	$q\in I_{i-1}\wedge\sigma_{i-1}(p)=\top_X\wedge$ $\sigma_{i-1}(p)\neq\top_X$	$\sigma_i=\sigma_{i-1}\odot\{(p,\sigma_{i-1}(q))\}\odot\{(r, \top_{X\setminus\{p\}})\|\text{ for all } r\in X\setminus\{p\}\}$ $I_i=I_{i-1}\cup\{p\}$, $N_i=N_{i-1}\setminus\{p\}$, $\xi_i=\xi_{i-1}$
3	$q\in N_{i-1}\wedge\sigma_{i-1}(p)=\top_X$	$\sigma_i=\sigma_{i-1}\odot\{(r, \top_{X\setminus\{p\}})\|\text{ for all } r\in X\setminus\{p\}\}$ $I_i=I_{i-1}\setminus\{p\}$,$N_i=N_{i-1}\cup\{p\}$, $\xi_i=\xi_{i-1}$
4	$p\in N_{i-1}\wedge\sigma_{i-1}(q)=\top_Y$	$\sigma_i=\sigma_{i-1}\odot\{(r, \top_{Y\cup\{p\}})\|\text{for}\forall r1\in Y\cup\{p\}\}$, $I_i=I_{i-1}\cup\{p\}$,$N_i=N_{i-1}\setminus\{p\}$, $\xi_i=\xi_{i-1}$
5	Otherwise	$\sigma_i=\sigma_{i-1}$, $I_i=I_{i-1}$,$N_i=N_{i-1}$, $\xi_i=\xi_{i-1}$

(a) q가 NULL이 아닌 경우

$[p=NULL](\sigma_{i-1}, \xi_{i-1}, I_{i-1}, N_{i-1})=$

	CASE	
1	$\sigma_{i-1}(p)=\top_X$	$\sigma_i=\sigma_{i-1}\odot\{(r, \top_{X\setminus\{p\}})\|\text{ for all } r\in X\setminus\{p\}\}$, $I_i=I_{i-1}\setminus\{p\}$, $N_i=N_{i-1}\cup\{p\}$, $\xi_i=\xi_{i-1}$
2	Otherwise	$\sigma_i=\sigma_{i-1}$, $I_i=I_{i-1}$, $N_i=N_{i-1}$, $\xi_i=\xi_{i-1}$

(b) q가 NULL인 경우

(그림 3) 'p=q'에 대한 변환 함수

이 존재한다면 p를 제외한 나머지 포인터들은 여전히 입력 자료 구조에 접근하여 영향을 줄 수 있기 때문에 p만 N_i에 편입한다.

(그림 4)는 'p=*q' 형태의 배정문에 관한 변환 함수를 정의한 것이다. 만약 주어진 프로그램 경로 상에 배정문 'p=*q'를 처리한다고 할 때 이 배정문이 오류 없이 실행되기 위해서는 포인터 q가 'NULL'이 되어서는 안되므로 'q≠NULL'이라는 제약조건을 추출할 수 있다. 만약 이미 q가 'NULL'이라면 해당 프로그램 경로는 모순되므로 실행 불가능한 경로가 된다는 사실도 알 수 있다. q의 값이 아직 결정되지 않은 상태라면 q는 'NULL' 이라는 가능성이 배제가 된다. 따라서 (그림 4)(a)와 (그림 4)(b)의 첫 번째 경우에서처럼 q는 구체적인 메모리 로케이션을 가리키도록 조정 한다.

(그림 4)(b)의 첫 번째 경우는 (그림 4)(a)에 비해 보다 복잡하다. 그 이유는 q가 가리키는 메모리 로케이션이 포인터가 아니기 때문이다. 즉, 이는 q를 통해서 접근 가능한 입력 자료 구조에는 아직 결정되지 않은 메모리 로케이션이 없음을 의미한다. 따라서, 일단 구체적인 메모리 로케이션이 결정되었으면 이를 직접 또는 간접으로 가리키는(i.e., $\sigma'^*(r)=l$) 모든 포인터들은 N'에 편입된다.

(그림 4)(a)의 두 번째 경우는 q가 현재 구체적인 메모리 로케이션을 가리키고 있고 이 로케이션이 포인터인 경우에 해당한다. 이 메모리 로케이션을 통해 아직 결정되지 않은 입력 자료 구조에 접근 가능하기 때문에 이를 (그림 3)에 주어진 변환 함수를 적용하여 처리한다. (그림 4)(b)의 두 번째 경우는 q가 현재 구체적인 메모리 로케이션을 가리키고 있고 이 로케이션이 포인터가 아닌 경우에 해당한다. 이 경우에는 이 배정문의 수행을 통해 아직 결정되지 않는 입력 자료 구조에 접근 할 가능성이 없기 때문에 별다른 처리 없이 콘콜릭 테스팅을 수행한다. 즉, 잠재적 모양 생성문이 아닌 경우에 해당된다.

(그림 5)는 'p=q' 형태의 배정문에 관한 변환 함수를 정의

$[p=*q](\sigma_{i-1}, \xi_{i-1}, I_{i-1}, N_{i-1})=$

	CASE	
1	$q\in I_{i-1}\wedge\sigma_{i-1}(q)=\top_Y$	Let $\sigma'=\sigma_{i-1}\odot\{(r,l)\|\text{for } \forall r\in Y\}\odot\{(l, \top_{(l)})\}$ $I'=I_{i-1}\cup\{l\}$ Then $(\sigma_i, \xi_i, I_i, N_i)=[p=l](\sigma', \xi_{i-1}, I', N_{i-1})$
2	Otherwise	$\sigma_i=\sigma_{i-1}$, $I_i=I_{i-1}$, $N_i=N_{i-1}$, $\xi_i=\xi_{i-1}$

(a) p와 *q가 포인터 형인 경우

$[p=*q](\sigma_{i-1}, \xi_{i-1}, I_{i-1}, N_{i-1})=$

	CASE	
1	$q\in I_{i-1}\wedge\sigma_{i-1}(q)=\top_Y$	Let $\sigma'=\sigma_{i-1}\odot\{(r,l)\|\text{for } \forall r\in Y\}$ and l에 (임의의) 정수값 바인딩, $I'=I_{i-1}\setminus\{r\|\ \sigma'^*(r)=l\}$, $N'=N_{i-1}\cup\{r\|\ \sigma'^*(r)=l\}$ Then $(\sigma_i, \xi_i, I_i, N_i)=[p=l](\sigma', \xi_{i-1}, I', N')$
2	Otherwise	$\sigma_i=\sigma_{i-1}$, $I_i=I_{i-1}$, $N_i=N_{i-1}$, $\xi_i=\xi_{i-1}$

(b) p와 *q가 포인터 형이 아닌 경우

(그림 4) 'p=*q'에 대한 변환 함수

한 것이다. 이 경우는 (그림 4)에 주어진 'p=*q'에 대한 변환 함수와 매우 유사하다. (그림 5)(a)와 (그림 5)(b)의 첫 번째 경우에서 p가 가리키는 메모리 로케이션이 아직 결정되지 않았을 때, i.e., '$\sigma_{i-1}(p)=\top_x$', 실제 메모리 로케이션을 하나 생성 시키고 p로 하여금 가리키게 한다. 두 번째 경우는 p가 구체적인 메모리 로케이션을 가리키는 경우에 해당되는데 이때에는 별다른 처리 없이 콘콜릭 테스트를 수행한다.

배정문의 마지막 형태로 (그림 6)은 'p=&q' 형태의 배정문에 관한 변환 함수를 정의한 것이다. 이 경우는 q를 통해 입력 자료구조의 모양에 영향을 줄 수 있는지 여부에 따라 달리 처리한다. 만약 첫 번째 경우처럼 q를 통해 아직 결정되지 않은 메모리 로케이션에 도달가능하다면 이 배정문의 수행 후에는 p를 통해서도 도달 가능하기 때문에 p를 I_i에 포함되게 한다. 그러나 q를 통해서는 아직 결정되지 않은 메모리 로케이션에 도달 가능하지 않다면 이 배정문 수행 후에는 p도 도달가능하지 않게 된다. 따라서 p를 N_i에 편입시켜 이 사실을 나타낸다. 특히 주목할 점은 이 두 가지 어떤 경우이든 배정문이 실행이 입력 자료 구조에 전혀 영향을 미치지 않는 다는 점이다(i.e., $\sigma_i=\sigma_{i-1}$).

3.3.2 술어(predicate)에 대한 변환 함수

(그림 7)은 술어가 'p==q' 형태인 경우에 대한 변환 함수를 보여준다. 이 변환 함수의 목적은 해당 술어가 참이 되도록 하는 입력 자료 구조의 모양을 식별하는 것이다. 따라서 이 경우에는 p가 현재 가리키고 있는 메모리 로케이션과 q가 현재 가리키고 있는 메모리 로케이션이 동일하도록 입력 자료 구조 모양을 조정하는 작업을 수행한다. 이를 위해 (그림 7)(a)에서 처럼 'Merge'함수를 사용한다.

(그림 8)은 'Merge' 함수의 자세한 내역을 보여준다. 1번 조건에 의해 메모리 로케이션 p와 q가 동일하지 않아야 한다는 제약이 없는 경우에만 이 함수를 수행하며 그렇지 않은 경우는 'Error'로 판단한다. 간단한 예로 'if (p<>q) {A; if (p==q) B}'를 생각해보자. 이 코드에서 A와 B는 여러 개의 문장들로 구성된 블록들을 나타내며 A 블록에서 p와 q

$[*p=q](\sigma_{i-1},\xi_{i-1},I_{i-1},N_{i-1})=$

	CASE	
1	$\sigma_{i-1}(p)=\top_x$	Let $\sigma'=\sigma_{i-1}\odot\{(r,l)\|\text{for } \forall r\in X\}\odot\{(l,\top_{(l)})\}$ Then $(\sigma_i, \xi_i, I_i, N_i)=[l=q](\sigma', \xi_{i-1}, I_{i-1}, N_{i-1})$
2	Otherwise	$\sigma_i=\sigma_{i-1}$, $I_i=I_{i-1}$, $N_i=N_{i-1}$, $\xi_i=\xi_{i-1}$

(a) *p와 q가 포인터 형인 경우

$[*p=q](\sigma_{i-1},\xi_{i-1},I_{i-1},N_{i-1})=$

	CASE	
1	$\sigma_{i-1}(p)=\top_x$	Let $\sigma'=\sigma_{i-1}\odot\{(r,l)\|\text{for } \forall r\in X\}\odot\{(l,\top_{(l)})\}$ $I'=I_{i-1}\setminus\{r\|\ \sigma'^*(r)=l\}$, $N_i=N_{i-1}\cup\{r\|\ \sigma'^*(r)=l\}$, $\xi_i=\xi_{i-1}$
2	Otherwise	$\sigma_i=\sigma_{i-1}$, $I_i=I_{i-1}$, $N_i=N_{i-1}$, $\xi_i=\xi_{i-1}$

(b) *p와 q가 포인터 형이 아닌 경우

(그림 5) '*p=q'에 대한 변환 함수

$[p=\&q](\sigma_{i-1},\xi_{i-1},I_{i-1},N_{i-1})=$

	CASE	
1	$q\in I_{i-1}$	$\sigma_i=\sigma_{i-1}$, $I_i=I_{i-1}\cup\{p\}$, $N_i=N_{i-1}\setminus\{p\}$, $\xi_i=\xi_{i-1}$
2	Otherwise	$\sigma_i=\sigma_{i-1}$, $I_i=I_{i-1}\setminus\{p\}$, $N_i=N_{i-1}\cup\{p\}$, $\xi_i=\xi_{i-1}$

(그림 6) 'p=&q'에 대한 변환 함수

$[p==q](\sigma_{i-1},\xi_{i-1},I_{i-1},N_{i-1})=$

	CASE	
1	$pts(p)\neq k_m$ or $pts(q)\neq k_n$	$\Sigma_i=\text{Merge}(pts(p), pts(q), \Sigma_{i-1})$
2	Otherwise	$\sigma_i=\sigma_{i-1}$, $I_i=I_{i-1}$, $N_i=N_{i-1}$, $\xi_i=\xi_{i-1}$

(a) q가 NULL이 아닌 경우

$[p==NULL](\sigma_{i-1},\xi_{i-1},I_{i-1},N_{i-1})=$

	CASE	
1	$pts(p)=\top_X$	$\sigma_i=\sigma_{i-1}\odot\{(r,NULL)\|\text{for } \forall r\in X\}$ for all y such that $\xi_{i-1}\models(y\neq\top_X)$ $[y!=NULL](\sigma_{i-1},\xi_{i-1},I_{i-1},N_{i-1})$
2	Otherwise	$\sigma_i=\sigma_{i-1}$, $I_i=I_{i-1}$, $N_i=N_{i-1}$, $\xi_i=\xi_{i-1}$

(b) q가 NULL인 경우

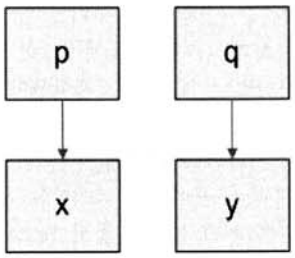
(그림 7) 술어 'p==q'에 대한 변환 함수

```
State Merge(Var p, Var q, State Σ)
begin
Let Σ=(σ,ξ, I,N)
1: if (ξ⊨!(p ≠q)) then
2:      if (p==q) then return Σ
3:      else if (p== T_X) and (q== T_Y) then
        begin
4:              σ'=σ⊙{(r, T_{X∪Y})|for ∀r∈X∪Y};
5:              Let Σ'=(σ',ξ, I,N)
6:              return Σ'
        end
7:      else if (p== T_X) and (q==l) then
        begin
8:              σ'=σ⊙{(r,l)| for all r∈X};
9:              Let Σ'=(σ',ξ, I,N)
10:             return Σ'
        end
11:     else if (p==l) and (q== T_Y) then begin
12:             σ'=σ⊙{(r,l)| for all r∈Y};
13:             Let Σ'=(σ',ξ, I,N)
14:             return Σ'
        end
15:     else if (p==l_i) and (q==l_j) then
        begin
16:             σ''=Merge(pts(p), pts(q), Σ);
17:             if (p≲_t q) then
18:                     R={r| σ(r)=q};
19:             else    R={r| σ(r)=p};
20:             return σ''⊙{(r, σ(p))| for ∀r ∈R}
21:     else if (p==⊥) or (q==⊥) then return Σ
22:     else    Error
23: else Error
end
```

(그림 8) Merge 함수

에 대한 변경이 없다고 가정하자. 이 경우에 B 블록을 실행하기 위한 p와 q 값은 존재하지 않게 되어 B 블록을 실행할 수 없다. 이러한 제약은 (그림 10)(a)에 보여진 'p!=q' 형태의 술어를 처리할 때 생성된다.

'Merge' 함수의 2번 조건에 의해 두 메모리 로케이션이



(그림 9) 이명(aliasing) 문제

이미 동일한 경우를 처리한다. 이 경우는 별다른 처리를 수행하지 않고 그대로 현재 상태를 되돌려 준다.

3번 조건에 의해 메모리 로케이션 모두가 결정되지 않았을 때(i.e., T_X와 T_Y)를 처리한다. 이때에는 메모리 로케이션을 병합(i.e., $T_{X\cup Y}$)하고 이 병합된 로케이션이 여전히 아직 결정되지 않았다는 사실을 상태 정보에 반영한다.

7번과 11번 조건은 두 메모리 로케이션들 중 하나만 결정된 경우에 해당되는 경우를 처리한다. 이 경우는 간단하게 아직 결정 안 된 메모리 로케이션(i.e, T_X나 Ty)을 결정이 된 로케이션(i.e., l)으로 변환하면 된다. 이를 위해 아직 결정 안 된 메모리 로케이션을 가리키는 포인터(i.e., X나 Y)들로 하여금 이미 결정이 된 메모리 로케이션을 가리키게 하면 된다.

15번 조건은 대상 메모리 로케이션들이 모두 결정이 된 경우(i.e., l_i와 l_j)에 해당된다. 이 경우에는 자식 로케이션(i.e., pts(p)와 pts(q))을 우선 병합시킨 후에 대상 로케이션을 병합한다. 이 때 주의할 점은 이 문장을 실행하기 전까지는 대상 메모리 로케이션들이 달리 취급되었다는 사실이다. 예를 들어 '*p=x; *q=y; if (p==q)... '를 생각해보자. 여기에서 p와 q는 입력 포인터이고 x와 y는 정수형 변수라고 가정하자. (그림 5)에 주어진 변환 함수에 따라 '*p=x; *q=y;' 수행 후에는 p와 q가 가리키는 자료 구조는 (그림 9)와 같을 것이다.

그런데 현재 조건이 참이 되어야 하는 경우에는 p와 q가 가리키는 메모리 로케이션을 병합할 필요가 있다. 그러나 이때에는 p가 가리키는 메모리 로케이션과 q가 가리키는 메모리 로케이션에 어떤 값이 이미 기록되어 있으므로 시간상 나중에 기록된 값이 있는 메모리 로케이션으로 병합할 필요

$[p!=q](\sigma_{i-1},\xi_{i-1},I_{i-1},N_{i-1})=$

	CASE	
1	$pts(p)\neq k_m$ or $pts(q)\neq k_n$	$\sigma_i=\sigma_{i-1}$, $I_i=I_{i-1}$, $N_i=N_{i-1}$ $\xi_i=\xi_{i-1}\wedge(pts(p)\neq pts(q))$
2	Otherwise	$\sigma_i=\sigma_{i-1}$, $I_i=I_{i-1}$, $N_i=N_{i-1}$, $\xi_i=\xi_{i-1}$

(a) q가 NULL이 아닌 경우

$[p!=NULL](\sigma_{i-1},\xi_{i-1},I_{i-1},N_{i-1})=$

	CASE	
1	$\sigma_{i-1}(p)=T_X$	$\sigma_i=\sigma_{i-1}\odot\{(r,l)\|for\ \forall r\in X\}$,$I_i=I_{i-1}\cup\{l\}$, $N_i=N_{i-1}$, $\xi_i=\xi_{i-1}$
2	Otherwise	$\sigma_i=\sigma_{i-1}$, $I_i=I_{i-1}$, $I_i=I_{i-1}$, $N_i=N_{i-1}$, $\xi_i=\xi_{i-1}$

(b) q가 NULL인 경우

(그림 10) 술어 'p!=q'에 대한 변환 함수

가 있다. 이 예에서는 q가 가리키는 메모리 로케이션이 나중에 기록되어 있으므로 p로 하여금 q가 가리키는 메모리 로케이션을 가리키게 한다. 이러한 처리를 수행하는 부분이 17번-19번 문장들이다.

21번 문장은 더 이상 병합할 자료 구조가 존재하지 않는 경우를 다룬다. 이때에는 아무런 상태의 변화가 없다. 22번은 메모리 로케이션들이 병합할 수 없는 경우를 처리한다. 이는 포인터 하나는 물리적 메모리 로케이션을 가리키고 있고 다른 하나는 논리적 메모리 로케이션을 가리키고 있는 경우에 해당한다. 그 이유는 물리적 메모리 로케이션은 대상 프로그램 함수에서 생성된 메모리 로케이션이고 논리적 메모리 로케이션은 대상 프로그램 함수 외부에서 생성되는 메모리 로케이션을 나타내기 때문이다.

(그림 10)는 술어 'p!=q'에 대한 변환 함수를 보여준다. 이 변환 함수의 목적은 포인터들이 서로 다른 메모리 로케이션을 가리켜야 한다면 이를 제약 'ξ_i'에 반영하여 후에 아직 결정되지 않은 메모리 로케이션들을 결정할 때 제약에 부합되게 결정되게 하는 것이다. 예를 들어 변환 함수 첫 번째 경우를 보자. 이 경우는 p나 q가 논리적 메모리 로케이션을 가리키거나 아직 결정되지 않은 자료 구조를 가리키고 있을 때 (i.e., $pts(p) \neq k_m$ or $pts(q) \neq k_n$) 제약식 ξ_i에 두 로케이션은 서로 다른 로케이션이어야 한다는 사실이 반영된다:"$\xi_i = \xi_{i-1} \wedge (pts(p) \neq pts(q))$ '. 그 이유는 기본적으로 이 논문에서는 논리적 메모리 로케이션들이 서로 같지 않다고 판단되지 않은 이상 서로 동일한 메모리 로케이션이 될 가능성을 배제하지 않기 때문이다. 따라서 자료 구조를 병합할 때 우선 병합이 가능한지를 검사하여야 한다. 이러한 검사는 위해 'Merge'의 첫 번째 조건(라인 1)에서 수행되며 만약 병합이 가능하지 못하는 경우에는 'Error'로 판명되어 입력 자료 구조를 식별할 수 가 없다.

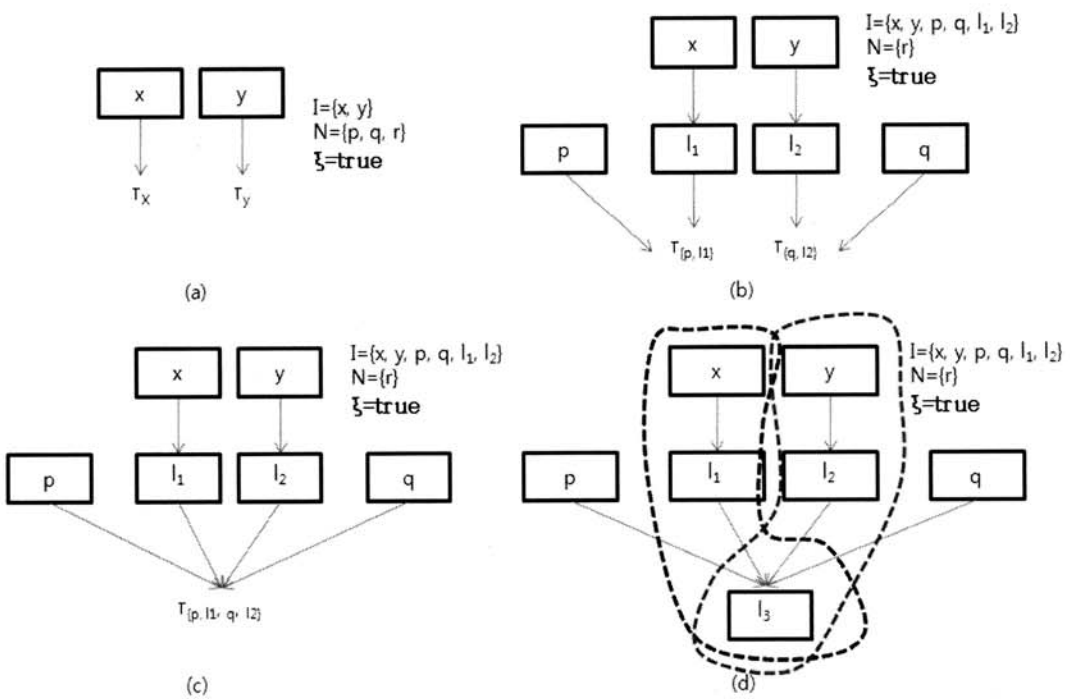
3.4 예 제

3.3절에서 기술한 변환 함수에 대한 이해를 위해 (그림 1)의 프로그램을 예로 들어 설명해보자. (그림 11)은 프로그램 경로 <1, 2, 3, 4, 6, 8, 9, 10,...>의 각 문장이 실행됨에 따라 3.3절에서 정의된 변환함수에 따라 입력자료 구조가 어떻게 식별되는지를 차례대로 보여준다.

(그림 11)(a)는 초기상태 $\Sigma_0=(\sigma_0, \xi_0, I_0, N_0)$를 보여준다. 즉, $\sigma_0(x)=\top_x$, $\sigma_0(y)=\top_y$, $I_0=\{x, y\}$, $N_0=\{p, q, r\}$, $\sigma_0(x)=\top_x$, $\sigma_0(y)=\top_y$, ξ_0=True. 문장 1, 'p=*x'에 관련된 변환 함수는 x에 관한 'points-to'정보를 이용하여 포인터 역참조 연산을 제거하려고 시도한다. 이 경우에는 x는 현재 $\top_x$를 가리키고 있기 때문에 (그림 4)(a)의 첫 번째 경우에 해당된다. 이 경우에는 새로운 메모리 영역(i.e., l_1)을 할당하고 x로 하여금 가리키게 한다. 이 새롭게 생성된 메모리 영역은 '$\top_{l1}$'로 초기화 한다. 아직까지는 무엇을 가리킬지 모르기 때문이다. 이 정보를 σ_0에 반영한 후에 변환 함수 [p=l_1]를 수행한다.

'$p \in N_0 \wedge \sigma_0(l_1)=\top_{l1}$'이기 때문에 변환 함수 [p=$l_1$]의 네번째 경우에 해당된다. 이 경우는 p와 l_1이 동일한 개체를 가리키도록 조정하고 p도 I_1에 포함되도록 한다. 문장 2, 'q=*y'도 문장 1과 동일한 과정을 거쳐 나온 상태가 (그림 11)(b)에 있다.

조건 3은 p가 q가 동일한 자료 구조를 가리키도록 요구한다. 현재 p와 q가 모두 물리적 메모리 로케이션이 아닌 논리적 메모리 로케이션을 가리키고 있기 때문에 (그림 7)(a)의 변환 함수의 첫 번째 경우에 해당되며 입력 자료 구



(그림 11) 예제 프로그램(그림 1)에 대한 입력 자료 구조의 식별

조의 병합이 필요하다. 이 경우는 (그림 8)의 병합 함수에서 3번째 조건(라인 3-라인 6)에 해당되는 처리를 수행하며 결과는 (그림 11)(c)와 같다.

4번 조건은 p가 'NULL'이 아니어야 한다는 사실을 나타내기 때문에 p, l_1, l_2, q는 새로운 객체 l_3를 가리키도록 조정된다(그림 11)(d).

6번 조건은 q가 'NULL'이 아니어야 한다는 사실을 나타낸다. 현재 (그림 11)(d)에서 볼 수 있듯이 이미 현재 상태에 의해 이 조건은 만족되므로 상태에 변화가 없다. 변환함수 (그림 10)(b)의 두 번째 경우에 해당된다.

문장 8은 변수 z가 N에 속하기 때문에 현재 상태의 변화를 가져오지 않고 문장 9, 10, 11도 입력 자료 구조의 모양에 전혀 영향을 미치지 않기 때문에 별다른 처리를 하지 않는다. 이 점은 [5]에서 제안했던 방법과 큰 차이를 보인다. 기존의 방법은 입력 자료 구조의 모양에 영향을 미치지 않더라도 불필요하게 부가적인 자료 구조를 형성하는 단점이 있었다.

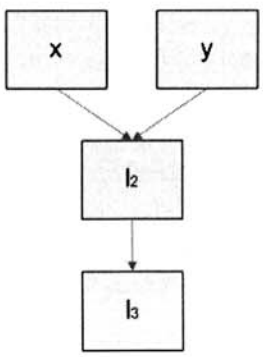
최종적으로 입력 포인터가 가리키는 자료 구조의 모양이 목표 프로그램 경로를 실행할 수 있는 자료 구조이다. 따라서 (그림 11)(d)에서 입력 포인터 x, y가 가리키는 자료 구조를 점선 영역으로 표시하였으며 이는 프로그램 경로 <1, 2, 3, 4, 6, 8, 9, 10, 11>를 실행할 수 있는 입력 자료 구조 모양이다.

입력 자료 구조의 병합에 대한 보다 명확한 이해를 돕기 위해 (그림 1)의 프로그램에서 (그림 12)과 같이 14번 15번 문장을 추가한 후에 프로그램 경로 <1, 2, 3, 4, 6, 8, 9, 10, 14, 15,...> 를 실행할 수 있는 입력 자료 구조를 식별해보자. 이 경우는 (그림 8)의 'Merge' 함수에서 조건 15가 참이 되는 경우에 해당되며 포인터 x와 y가 어떤 메모리 영역을 가리키고 있지만 이 메모리 영역이 동일한 객체인지가 아직은 결정이 내려지지 않은 상태를 처리한다. 예를 들어, (그림 11)(d)를 보자. 이 자료 구조에서 포인터 x와 y가 서로 상이한 메모리 영역을 가리키는 것처럼 보인다. 그러나 실제 x, y가 가리키는 메모리 영역 l_1, l_2는 동일한 메모리 영역일 수도 있다. 이 논문에서는 어떤 메모리 영역들이 서로 동일하다고 결정되기 전까지는 서로 상이한 영역으로 간주하고 우선 처리한다. 만약 이들이 동일해야 한다면 이 경우에 'Merge' 함수가 메모리 영역을 동일하게 해부는 역할을 수행한다.

14번 조건은 (그림 11)(d)에서 포인터 x와 y가 가리키는 메모리 영역이 동일할 것을 요구하며 l_2가 l_1보다 나중에 사용되었기 때문에 l_1과 l_2가 나타내는 (논리적) 메모리 로케이션은 l_2로 병합된다. (그림 13)은 'Merge' 함수가 수행된 결과를 보여준다.

```
void Example(int **x, int **y, int v) {
int *p, *q, *r, z;
1:      p = *x;
2:      q = *y;
3:      if (p == q) {
4:              if(p == NULL)
5:                      *q = v;
6:              else if(q == NULL)
7:                      *p = v;
                else {
8:                      r = &z;
9:                      *r = 10;
10:                     if(z == v)
11:                             ...
                }
        }
        else {
12:             *p = v;
13:             *q = v;
        }
14: if (x==y)  {
15:             ...
        }
}
```

(그림 12) 자료 구조의 병합 예



(그림 13) 자료 구조 병합을 보여주는 예

4. 구현 및 평가

이 논문에서는 제안된 테스트 데이터 생성 방식을 vCREST라 불리는 도구에 구현하였다. vCREST는 C 프로그램을 입력으로 받아 3.3절에서 기술한 변환 함수를 수행하기 위해 프로그램 코드에 탐침(instrument)을 한다. 이를 위해 CREST에서 이미 제공하는 CIL[16]을 다단계 스택 지향 포인터를 지원할 수 있도록 확장하였다. 일단 탐침된 프로그램은 3.2절에 기술된 테스트 데이터 생성 절차에 따라 수행된다. 현재 vCREST는 Cygwin에서 개발되었으며 안정화 과정 중에 있다. 조만간 open-source로 공개될 예정이다.

vCREST는 현재 다음과 같은 기능을 수행한다.

- 다단계 스택 지향 포인터를 사용하는 프로그램에 대해 테스트 데이터를 생성한다. 이에 반해 기존의 도구들은 (i.e. CREST[4]나 INKA[9]) 정수형 입력 변수만을 지원한다.
- 프로그램에 존재하는 모든 분기들을 생성할 수 있는 테

스트 데이터를 생성한다. 이 점은 vCREST가 기존의 CREST 도구와 연동하여 동작하기 때문이다. 따라서 SGEN[5]을 포함한 기존의 대부분의 도구들이 일일이 프로그램 경로를 제공해야 하는 것에 비해 매우 편리하게 분기 커버리지를 달성할 수 있다.

- 만약 입력 변수가 포인터인 경우에 생성되는 입력 자료 구조를 시각화 하여 볼 수 있다. 현재 vCREST는 GraphViz를 이용하여 제어 흐름 그래프와 생성된 테스트 데이터를 시각화 하는 기능을 제공한다. 따라서 테스트할 때 어떤 입력 자료 구조가 필요한지 매우 쉽게 알 수 있다. 또한 어떤 분기가 실행되었는지 여부를 제어 흐름그래프에 나타내어 표시하는 기능을 제공한다.
- 실행 불가능한 경로를 처리하는 경우에는 즉각적으로 실행할 수 없음을 판단하여 다른 경로를 선택하여 수행을 계속한다. 이에 반해 Korel의 방법은 이전 분기로 되돌아가 다시 탐색하는 작업을 반복하기 때문에 많은 시간이 소요될 수 있다.

<표 1>에서 vCREST를 다른 테스트 데이터 생성도구와 비교하였다. 이를 위해 테스트 데이터 생성 전략을 경로 지향, 목적 지향 및 모든 경로(또는 분기)로 분류하였으며 구현 방법의 비교를 위해 테스트 데이터를 생성하기 위해 실제 프로그램을 수행하느냐 여부에 따라 정적 방법, 동적 방법 및 하이브리드 방식으로 분류하였다. 하이브리드 방식이란 테스트 데이터 생성을 위해 부분적으로는 심볼릭 실행과 같은 정적 방식을 사용하고 부분적으로는 프로그램을 직접 실행하는 동적 방식을 혼합한 방법을 의미한다. 또한 모양 문제를 직접적으로 다루는지에 따라서도 비교하였다.

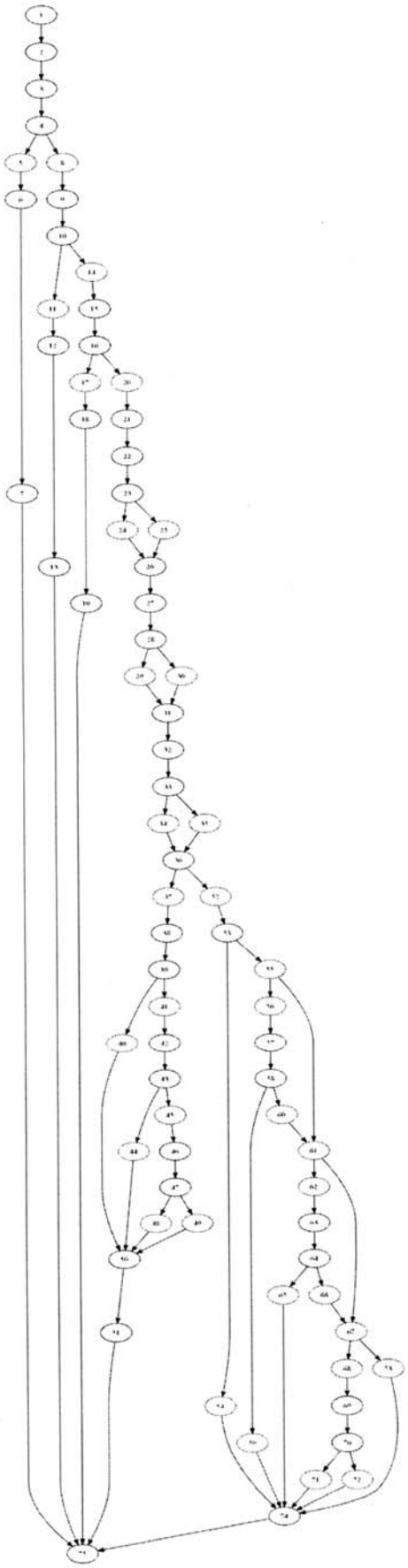
특히 <표 1>에서 열거한 도구 중에서 CUTE[18]는 포인터를 처리할 수 있는 방법을 제공하고 있다. 그러나 이 논문처럼 직접적으로 모양 문제를 다루지는 않는다. 만약 (그림 1)에 주어진 프로그램을 대상으로 CUTE를 실행하였을 경우는 적절한 입력 자료 구조를 만들어내지 못한다. 그 이유는 우선 포인터 x, y의 초기 값으로 NULL을 할당하게 되는데 이 경우에 1번의 배정문에서 NULL을 참조하게 되어 CUTE는 종료가 된다. 이에 반해 이 논문에서 제시한 방법은 1번 문장을 처리할 때 x가 NULL이 되지 않아야 한다는 사실을 유추하고 실제 x로 하여금 메모리 로케이션을 가리키도록 한다. 이러한 차이는 CUTE가 포인터를 다른 자료형과 동일하게 콘콜릭 테스트를 수행하도록 하기 때문이다. 따라서 CUTE가 처리할 수 있도록 하려면 1번 문장과 2번 문장 앞에 다음과 같은 조건식이 선행되도록 할 필요가 있다: 'if (x<>NULL) p=*x;'. 물론 2번 문장도 동일하게 'if (y<>NULL)' 조건이 2번의 배정문 'q=*y;'에 선행되도록 해야 한다.

또한 이 논문에서 제안한 테스트 데이터 생성 방법의 타당성을 검사하기 위해 vCREST를 사용하여 (그림 1)의 프로그램에 대해 테스트 데이터를 생성하였다. 여기에서 제안된 방법의 타당성은 다음의 측면들에서 살펴보았다. 우선 프로그램에 실행 불가능한 경로가 존재할 때 이를 식별할 수 있는가? 그리고 프로그램 경로가 실행 가능할 때 올바른 입력 자료 구조와 입력 값을 생성할 수 있는가? 마지막으로 프로그램에 존재하는 모든 실행 가능한 분기들을 수행할 수 있는 테스트 데이터를 생성하는가?

(그림 1)의 프로그램은 두 개의 실행 불가능한 경로를 포함하고 있다: p1=<1, 2, 3, 4, 5, ...>와 p2=<1, 2, 3, 6, 7, ...>. 경로 p1은 p가 NULL인데 5번 문장(i.e., *q=v)을 수행하려고 하면 오류가 발생한다. 그 이유는 포인터 q도 p와 동일하게 NULL이 되어야 하기 때문이다. 같은 이유로 p2도 실행 가능한 입력 값(또는 구조)이 존재 하지 않는다. vCREST는 이러한 경로에 대해 5번 또는 6번 문장을 처리하는 순간 실행이 가능하지 않음을 식별하고 오류 메시지를 출력하고 아직 실행되지 않은 다른 분기를 탐색한다.

〈표 1〉 테스트 데이터 생성 도구와의 비교

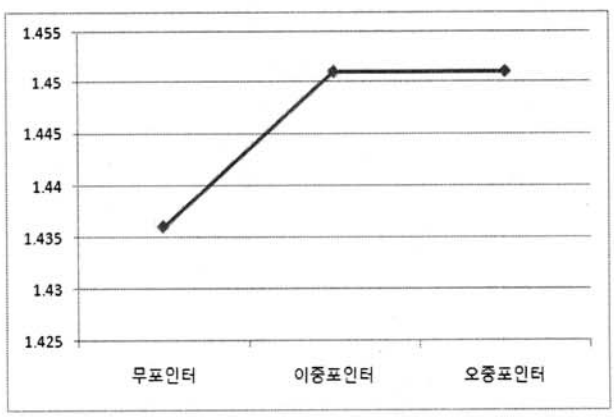
도구 및 방법		테스트 데이터 생성 전략			구현 방법			모양 문제	관련 문헌
		경로지향	목적지향	모든 경로 (분기)	정적	동적	하이브리드		
1	TESTGEN	X				X		X	[6]
2	SGEN	X			X			X	[5]
3	CREST			X			X		[4]
4	Visvanathan의 방법	X			X			X	[7]
5	vCREST			X			X	X	-
6	INKA		X		X				[9]
7	DDR	X			X				[10]
8	ATGen	X			X				[11]
9	Chaining 방식		X			X			[12]
11	GADGET								[13]
12	ADTEST								[14]
13	TGen								[15]
14	CUTE			X			X		[18]



(그림 14) triangle 프로그램 제어 흐름 그래프

또한 vCREST는 입력 자료 구조를 점진적으로 생성한다. 여기에서 '점진적'이라는 의미는 입력 경로가 모두 제공되지 않고 부분 경로만 제공된다 할지라도 해당 부분 경로를 실행할 수 있는 테스트 데이터를 생성할 수 있다는 의미이다.

vCREST의 성능 평가를 위해 CPU (2.0Ghz), RAM 6GB, 윈도7 64bit, Cygwin 1.7.3-1를 사용한 테스트 환경에서 (그림 1)의 프로그램에 대해 실행가능한 모든 분기들을 실행할 수 테스트 데이터를 생성하는데 걸리는 시간을 측정하였다. 그 결과 총 7회의 실행으로 10개의 분기 중 실행 불가능한 2개의 분기를 제외한 8개의 분기가 0.733초에 실행되었다. 보다 명확한 성능 평가를 위해 [17]에 기술되어 있는 triangle 프로그램을 포인터를 사용하지 않은 버전, 입력 인자들을 이중 포인터로 변경한 버전 그리고 오중 포인터로 변경한 버전을 만들어 실행 시간을 측정하였다. 이중 포인터 버전이란 입력 인자의 자료형을 int **로 변형한 버전을 의미하며 오중 포인터 버전이란 int *****로 입력 인자의 자료형을 변형한 버전을 의미한다. 실험 결과로 14회 실행으로 34개의 분기 중 34개의 분기 모두가 실행되었으며 포인터를 사용하지 않은 버젼은 1.436초, 이중 포인터 버전과 오중 포인터 버전은 동일하게 1.451초가 측정되었으며 (그림 15)에 이 결과를 그래프로 나타내었다. 이 결과는 포인터를 사용하지 않은 버전과 포인터를 사용하는 버전 그리고 포인터를 사용한다 하더라도 포인터의 깊이에 그다지 실행 시간에 큰 영향이 없음을 알 수 있다. 그 이유는 triangle 프로그램이 입력 자료 구조 모양에 따라 프로그램 실행을 달리하는 함수가 아니라 입력 값에 따라 각기 다른 분기를 실행하는 특징을 지니고 있기 때문이라고 판단한다. 보다 실질적인 성능 평가는 입력 자료 구조 모양에 따라 다른 기능을 수행하는 프로그램에 대해 수행할 때 계산될 수 있으리라 기대하며 이를 위해서는 지금 현 논문에서 다루고 있는 스택 지향 포인터 보다는 힙(Heap) 포인터를 사용하는 프로그램을 처리할 수 있도록 vCREST가 확장될 필요가 있다. (그림 14)는 vCREST에서 출력한 triangle 프로그램의 제어 흐름 그래프이다.



(그림 15) Triangle 프로그램의 여러 버전에 대한 테스트 데이터 생성 시간

5. 결 론

이 논문에서는 프로그램에서 포인터를 사용하는 프로그램의 테스트를 위해 테스트 데이터를 자동으로 생성하는 도구인 vCREST에 대해 기술하였다. 기존의 대부분의 테스트 도구들은 포인터를 고려하지 않고 테스트 데이터를 생성하였다. 포인터를 고려하였다 하더라도 대부분이 프로그램 경로를 명시적으로 제공하여야 한데 비해 vCREST는 콘콜릭 방식을 기반으로 하기 때문에 사용자가 프로그램 경로를 명시할 필요가 없이 프로그램에 존재하는 모든 분기들을 테스트할 수 있는 테스트 데이터를 식별할 수 있다. 또한 동적 방식으로 테스트 데이터를 생성하기 때문에 이명(aliasing) 문제를 정적 방식보다 쉽게 처리할 수 있다.

vCREST는 기존의 테스트 데이터 자동생성 도구보다는 많은 기술적인 진보가 있었지만 보완할 점도 많이 있다. 우선 vCREST는 단위 테스트만을 지원한다. 통합 테스팅을 지원할 수 있도록 한 개 이상의 프로시듀어에 걸친 프로그램 경로를 수행하는 테스트 데이터에 대한 연구가 필요하다. 이를 위하여 현재 프로그램 테스트 데이터 정보를 매번 생성하지 않고 재사용이 가능하도록 프로그램 요약 정보를 활용하는 방법에 중점적인 연구를 수행하고 있다.

현재의 vCREST는 스택 지향 포인터만을 지원한다. 따라서 이를 동적으로 할당되는 힙(heap)에 대한 메모리 포인터로 확장할 필요가 있다. 이러한 확장은 그리 큰 문제로 보이지 않는다. 이 논문에서 기술한 스택 지향 포인터와 거의 같은 방법으로 다룰 수 있으며 현재 vCREST에 이를 구현 중에 있다. 또한 vCREST를 open-source로 하기 위한 안정화 작업이 필요하다. 마지막으로 vCREST를 C++나 Java와 같은 객체지향 프로그래밍 언어에도 적용될 수 있는지를 검토 중이다. 일반적으로 객체 지향 프로그램의 테스트 케이스들은 객체들의 구성(object configuration)으로서 표현 될 수 있으며 있는 기본적으로 본 논문에서 기술한 입력 자료 구조의 모양과 근본 개념이 다르지 않기 때문이다.

참 고 문 헌

[1] P. Godefroid, N. Klarlund, and K. Sen. "DART: Directed automated random testing," In *Proc. of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation* (PLDI), 2005.

[2] 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.

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

[4] J. Burnim, K. Sen, "Heuristics for Dynamic Test Generation," In *the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2008.

[5] I. S. Chung and J. M. Bieman, "Generating Input Data Structures for Automated Program Testing," *Software Testing, Verification and Reliability*, Vol.19, No.1, pp.3-36, 2009.

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

[7] S.Visvanathan and N. Gupta., "Generating Test Data for Functions with Pointer Inputs", n *Proc. the 17th IEEE International Conf. Automated Soft. Eng.*, pp.149-160, 2002.

[8] C. Lapkowski, and L. J. Hendren, "Extended SSA Numbering: Introducing SSA properties to Languages with Multi-level Pointers," ACAPS Tecnical Memo 102. School of Computer Science. McGill Univ. Canada. 1996.

[9] A. Gotlieb, T. Denmat, and B. Botella, "Goal-oriented Test Data Generation for Pointer Programs," *Information and Software Technology*, Vol.49, Issues9-10, pp.1030-1044, 2007.

[10] A. J. Offutt, Z. Jin, and J. Pan:, "The Dynamic Domain Reduction Procedure for Test Data Generation." *Softw., Pract. Exper.* Vol.29, No.2, pp.167-193, 1999.

[11] C. Meudec, "ATGen:Automatic Test Data generation using Constraint Logic Programming and Symbolic Execution," In *Proc. IEEE/ACM Int. Workshop on Automated Program Analysis Testing and Verfication.* pp.22-31, 2000.

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

[13] C. C. Michael and G. McGraw, "Automated Software Test Data Generation for Complex Programs," In *proceedings of 13th International conference on Automated Software Engineering*, pp.136-146, 1998.

[14] M. J. Gallagher and V. L. 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] R. P. Pargas, M. J. Harrold, and R. R. Peck. "Test Data Generation using Genetic Algorithms". *Software Testing, Verification and Reliability* 9, 263-282, 1999.

[16] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer, "CIL: Intermediate Language and Tools for Analysis and transformation of C Programs." In *Proc. of Conference on compiler Construction*, pp.213-228, 2002.

[17] P. Ammann and J. Offutt, Introduction to Software Testing, Cambridge University Press, Cambridge, UK, ISBN 0-52188-038-1, 2008.

[18] K. Sen, D. Marinov, and G. Agha, "CUTE: A Concolic Unit Testing Engine for C," In *Proc. of ESEC-FSE*, pp.263-272, 2005.

정 인 상

E-mail : insang@hansung.ac.kr

1987년 서울대학교 컴퓨터공학과(학사)

1989년 한국과학기술원(KAIST) 전산학과 (석사)

1993년 한국과학기술원(KAIST) 전산학과 (박사)

1994년~1999년 한림대학교 교수

1999년~현 재 한성대학교 컴퓨터공학과 교수

관심분야 : 소프트웨어 공학, 소프트웨어 테스팅