□ 기술해설 □

객체지향 프로그램의 테스팅 방법†

한림대학교 정인상*

● 목 차 ●

1. 서 론

최근에 이르러 객체지향 기술에 대한 관심이 높아지고 객체지향 분석 및 설계 방법론에 대한 기술이 성숙되면서 객체지향 소프트웨어 개발방법론을 이용한 프로그램의 개발이 가속화되고 있다.

객체지향 소프트웨어 개발의 주요 관심사는 기존의 클래스에 정의된 데이타나 연산들과 같은 속성들을 상속관계를 통하여 하위 클래스에서 이용하도록 하는 것이다. 만약 기존의 클래스들이 효과적으로 검증되지 않았다면 이미 개발된 클래스들을 다양한 응용분야에 재사용하는 것이 어려울 것이다. 이러한 측면에서 클래스를 설계하거나 개발할 때 클래스의 신뢰성을 확립하는 작업은 필수적이다. 이에 따라 객체지향 프로그램의 테스팅의 중요성이 강조되고 있다[24].

종래의 소프트웨어 테스팅 방법에 대한 연구는 구조적 분석 및 설계 기법이나 단계적 상세화(stepwise refinement)와 같은 전통적인 소프트웨어 개발 방법론에 의해 설계되고 절차적(procedural) 언어로 구현된 프로그램을 주대상으로 하고 있다. 절차적 프로그램을 대상으로 하는 테스팅 방법을 객체지향 프로그램에 그대로 적용시키기에는 어려움이 있다[5]. 왜냐하면 종래의 테스팅 방법은 프로시듀어나 함수를 테스팅 단위로 하여 입력 데이타 값들을 테스트케이스로 선정하는데 반해 객체지향 프로그램의 테스팅 단위는 프로시듀어나 함수가 아닌 클래스이기 때문이다[21].

이러한 테스팅 단위의 차이 때문에 지금까지 제안된 대부분의 테스팅 방법들은 클래스를 추상자료형으로 모델링하고 이를 기반으로 클래스에서 제공하는 멤버 함수들의 상호작용을 테스팅하는 통합테스팅에 중점을 두고 있다[7,8,10,13,18,23]. 클래스에 속한 각 멤버 함수에 대한 단위 테스팅이 적절한 방법에 의하여 이루어졌다고 가정할 때 클래스 테스팅을 위해 사용되는 테스트케이스는 클래스에서 제공하는 멤버 함수들의 조합, 즉 멤버 함수의 특정 호출 순서로 정의 된다. 예를 들어, *"Newstack"*, *"Push"*, *"Pop"*, *'Top"*들을 멤버함수로 하는 스택(*STACK*) 클래스를 생각해보자. 이러한 멤버 함수들의 상호작용을 테스팅하기 위해서는 *"Top(Push(Push(Newstack(), 7), 9))"*와 같은 멤버함수들의 조합들을 생성하여 테스팅 대상이 되는 클래스에 대하여 수행한다. 이 경우에

† 본 연구는 '95 한국과학재단의 핵심전문연구과제(과제번호 : 951-0908-022-2)의 부분적인 지원을 받았음.

*종신회원

"Top(Push(Push(Newstack(), 7), 9))"

가 스택 클래스에 대하여 하나의 테스트케이스를 구성한다. 테스트케이스를 멤버 함수의 호출 순서로 간주하는 중요한 이유 중의 하나는 테스팅 대상이 되는 클래스에서 정의된 각 멤버 함수의 실행 경로는 입력 인자의 값(input parameter value)들에 의해서만이 아니라 객체의 상태에 의해서 결정되기 때문이다. 보통, 객체의 상태를 결정하는 데이타 멤버는 대상 클래스의 다른 멤버 함수의 실행에 의존한다는 것을 주목할 필요가 있다. 따라서, 클래스를 테스팅하기 위해서는 사용자가 원하는 멤버 함수들의 조합 또는 호출 순서로 구성되는 테스트케이스를 다양하게 생성하여 멤버 함수들의 상호작용에 의한 클래스의 행태를 효과적으로 검증할 필요가 있다.

그러나 여러개의 프로시듀어로 구성되어 있고 이들 간의 호출 순서가 정적으로 정해진 절차적 프로그램 테스팅과는 달리 클래스의 멤버 함수는 다음과 같이 임의의 순서에 의하여 호출될 수 있다.

```
Pop(Push(Newstack(), 100));
Push(Push(Pop(Push(Newstack(), 100)), 101),
102));
Top(Push(Push(Push( ... (Newstack(), 100) ...
));
...;
```

그러므로 클래스의 멤버 함수들의 상호 작용을 통하여 클래스를 테스팅 할 때에는 이러한 무한한 갯수의 호출 순서에 대한 고려가 있어야 한다.

또한, 객체지향 프로그램을 테스팅할 때에는 절차적 프로그램에서 볼 수 없는 상속성(inheritance), 자료추상화(data abstraction), 동적바인딩(dynamic binding), 다형화(polymorphism)와 같은 특성들을 고려해야 한다. 객체지향 프로그램의 상속성은 이미 테스트가 완료된 클래스로 부터 상속된 멤버 함수들 중에서 어떤 멤버 함수들을 하위 클래스(subclass) 환경에서 다시 테스트를 수행할 것인가를 결정하는 문제를 발생시키며 이는 기존의 절차적 프로그램에 적용한 테스팅 방법들이 다루지 않았던 점이다[11,19]. 동적바인딩과 다형성은 프로그램의 실행 시간에 한 멤버 함수의 호출이 여러 멤버 함수로 바인딩 될 수 있다는 것을 의미한다. 이 때문에 주어진 멤버 함수의 호출이 실제적으로 어떤 멤버 함수로 바인딩 된다는 사실을 정적으로는 알 수 없다. 따라서 정적 분석을 통하여 테스팅 정보를 얻는 테스팅 방법들은 이러한 불확실성을 다룰 수 있는 수단을 제공하여야 한다[20]. 이와 같이 객체지향 프로그램의 여러 특성들은 프로그램을 설계하고 개발하는데 많은 잇점을 주는 것이 사실이지만 프로그램의 테스팅 작업이나 유지 보수 활동의 측면에서는 어려움을 야기시키고 있다[19].

본 고에서는 지금까지 제안된 객체지향 프로그램을 테스팅 하는 방법들에 대하여 개괄적으로 고찰한다. 제2장에서는 객체지향 프로그램을 테스팅할 때 고려할 여러 가지 사항을 Weyuker가 제안한 적합성 공리 관점[22]에서 살펴보고 제3장에서는 기존의 객체지향 프로그램 테스팅 기법들을 분류하고 대표적인 테스팅 기법들을 간략하게 소개한다. 제4장에서 결론을 맺는다.

2. 객체지향 프로그램을 테스팅할 때 고려할 사항

테스팅의 주요 목적은 테스트케이스들을 선택하고 선택된 테스트케이스들이 얼마나 효과적으로 오류를 검출하는지를 판별하는 작업이다. 테스트케이스를 선택 또는 생성하는 작업은 보통 어떤 주어진 기준에 따라서 이루어지게 된다. 만약, 테스트케이스를 선택하는 기준이 없거나 잘 정의되어 있지 않다면 테스팅을 종료할 시점을 결정하는데 어려움이 있을 뿐만 아니라 주어진 프로그램이 얼마나 적절하게 테스팅이 되었는지를 판별하는 것도 불가능할 것이다. 이 때문에 테스트케이스의 선택 기준은 테스트케이스의 적합성을 판별하는 규칙으로도 사용된다. 본 고에서는 테스트케이스의 선택 기준을 테스트케이스의 적합성을 판별하는 규칙과 같은 의미로 사용할 것이다.

지금까지 절차적 프로그램에 대하여 수 많은

테스트케이스의 선택 기준이 제안되었다. 이러한 테스트케이스 선택 기준의 비교 및 분석을 위하여 [22]에서 Weyuker는 테스트케이스의 선택 기준이 만족하여야 하는 11개의 공리를 제시하였으며 이미 잘알려진 테스트케이스 선택 기준들의 단점을 보이는데 사용하였다. 또한 Perry와 Kaiser는 Weyuker가 제안한 공리들 중 몇가지를 객체지향 프로그램의 테스팅 관점에서 재해석하였다[19]. 객체지향 프로그래밍과 직접적으로 관련된 공리들은 다음과 같다.

● 반확장성(antiextensionality) 공리 : 두 개의 프로그램이 똑같은 기능을 수행한다 할지라도 한 프로그램에 적합한 테스트케이스 집합은 나머지 다른 프로그램에 대해서 반드시 적합하지는 않다. 이는 동일한 기능을 서로 다른 많은 알고리즘에 의해서 구현될 수 있기 때문이다. 객체지향 프로그래밍에서 한 클래스에서 정의된 멤버 함수가 하위 클래스에서 동일한 명세에 바탕을 두고 재정의되는 경우에 각기 다른 테스트케이스 집합이 필요하다는 것을 의미한다. 즉, 상속 받는 클래스의 멤버 함수들을 다시 테스팅을 할 필요가 있다.

● 일반적인 다중 변경(general multiple change) 공리 : 두 개의 프로그램이 구조적으로 유사하더라도 한 프로그램에 적합한 테스트케이스 집합은 나머지 다른 프로그램에 대해서 반드시 적합하지는 않다. 예를 들면, 객체지향 프로그래밍에서 다중 상속성(multiple inheritance)을 이용하여 클래스 계층 구조를 구성할 때 멤버 함수들을 상위 클래스에서 상속받는 순서를 조금만 변경한다 할지라도 의미적으로는 상당한 변화가 일어나는 경우에 발생한다. 즉, 상속 우선 순위에 따라 상속받는 멤버 함수가 다르기 때문에 이전의 테스트케이스 집합과 다른 테스트케이스 집합이 필요하는 경우가 발생한다.

● 반합성(anticomposition) 공리 : 개개의 프로그램 요소(component)들을 독립적으로 적합하게 테스팅을 했을지라도 그들을 합성한 후의 프로그램에 대해서 테스팅이 적합하게 수행된다는 것을 보장하지 못한다. 객체지향 프로그램을 테스팅하는 경우에서는 만약 상위 클래스에 변경이 일어난다면 모든 하위 클래스는 상위 클래스와 함께 다시 테스팅이 되어야 함을 의미한다. 즉, 변경된 클래스와 변경된 클래스에 의존하는 모든 클래스들은 다시 테스팅 되어야 한다.

● 반분해(antidecomposition) 공리 : 프로그램 구성 요소들이 원래 사용되는 환경에서는 적합하게 테스팅되었다 할지라도 다른 환경에서 사용될 때에는 다시 테스팅이 되어야 한다. 객체지향 프로그램을 테스팅하는 관점에서 모든 상속 받는 멤버 함수들은 새로운 환경(예를 들면 새로 추가되는 자식 클래스)에서 다시 테스팅이 되어야 함을 의미한다. 특히, 하위 클래스가 상위 클래스의 데이타 멤버들에 접근할 수 있다면 예측하지 못한 종속 관계(dependence relation)가 발생할 수 있기 때문에 상위 클래스에서 적합하게 테스팅된 멤버 함수일지라도 하위 클래스에서 다시 테스팅 되어야 한다.

위의 공리들로 부터 객체지향 프로그래밍에서 제공하는 상속성과 같은 특성들이 오히려 테스팅 작업을 어렵게한다는 것을 살펴볼 수 있다. 즉, Perry와 Kaiser의 테스팅 이론은 잘 설계되고 테스팅이 된 클래스로 부터 상속받는 멤버 함수들이라도 하위 클래스 환경에서 다시 테스팅이 되어야 한다는 것이다. 따라서 하위 클래스를 테스팅(subclass testing)하는 경우에는 새롭게 추가되거나 상속받는 멤버 함수나 데이타 멤버들의 상호작용을 철저하게 검증하여야 한다. 실제로 Harrold와 그의 동료들은 Perry와 Kaiser의 테스팅 이론에 바탕을 둔 새로운 클래스 테스팅 기법을 제시하였다[11]. 그들 연구의 주된 내용은 상속 관계의 계층성을 이용하여 클래스 내의 속성들을 테스팅하는 방법과 실제로 테스팅을 수행하는데 필요한 테스트케이스를 결정하는 문제에 중점을 두고 있다. 전자의 문제를 다루는 경우에서 특히 주목할 점은 이미 테스트가 완료된 클래스로 부터 상속된 멤버 함수들 중에서 어떤 멤버 함수들을 하위 클래스로 부터 생성된 새로운 환경에서 다시 테스팅을 수행할 것인가를 결정하는 문제이다. Perry와 Kaiser의 이론은 모든 상속 받는 멤버 함수들은 새로운 환경에서 다시

테스팅이 되는 것을 요구하는데 반해 Harrold 등의 테스팅 기법은 선택적으로 테스팅을 한다는 점이다. 이는 객체지향 프로그램을 분석하여 얻은 자료 흐름 정보 및 클래스 속성들의 특성을 이용하였기 때문에 가능하였다. 후자의 경우는 테스트케이스를 처음부터 만드는 것은 테스팅 비용을 높이는 주요 요인이므로 이미 만들어진 테스트케이스의 재사용성 여부에 중점을 두고 있다. 즉, 하위 클래스에 대해 테스팅을 할 때 상위 클래스에서 사용된 테스트케이스들의 재사용이 가능한 경우에는 속성들을 상속받는 것처럼 테스트케이스도 상속받아 사용하게 하여 테스트케이스의 개발 비용을 절감하게 하였다.

3. 테스팅 방법의 분류

객체지향 프로그램의 테스팅 방법은 종래의 소프트웨어 테스팅 방법과 마찬가지로 테스트케이스를 선정하는 바탕이 되는 문서(분석 및 설계 명세, 프로그램 코드등)의 종류나 수명주기 단계에 따라 분류할 수 있다. 본 장에서는 객체지향 프로그램의 테스팅 방법을 다음과 같은 3개의 범주로 나누어 기술한다 :

- 명세 기반 테스팅(specification-based testing)
- 구현 기반 테스팅(implementation-based testing)
- 재검증(revalidation)을 위한 테스팅

3.1 명세 기반 테스팅

명세 기반 테스팅은 프로그램 구현 내역을 참조하지 않고 명세서에 나타난 정보만을 이용하는 방법이다. 이러한 형태의 테스팅 방법은 실제 프로그램의 구현 내역이 변한다 할지라도 명세서에 나타난 내용이 변하지 않는한 이미 개발한 테스팅 정보(예를 들면, 테스트케이스)를 수정 없이 이용할 수 있다는 잇점을 지니고 있다. 클래스를 모델링하고 테스팅하기 위해 주로 사용하는 명세는 대수적 명세(algebraic specification), 모델 기반 명세(model-based specification)이다. 현재 명세 정보에 기반을 둔 객체지향 프로그램의 테스팅 방법은 특히 자료추상화의 특징을 고려하여 연구가 진행중에 있다[7]. 본 절에서는 명세 기반 테스팅 방법들을 대수적 명세를 이용한 테스팅과 모델 기반 명세를 이용하는 테스팅으로 분류하여 기술한다.

3.1.1 대수적 명세를 이용한 테스팅

대수적 명세는 공리(axiom)를 이용하여 클래스의 특성을 기술한다. 대수적 명세는 멤버 함수들의 서명(signature)을 나타내는 부분과 각 멤버 함수의 의미 및 멤버 함수사이의 관계를 공리들로 표현하는 부분으로 구성된다(그림 1).

실제로 구현된 클래스들은 대수적 명세의 공리 부분에 열거된 성질들을 만족하여야 한다. 예를 들면, 그림 1의 (b)에서 *"Pop(Push(s, i))=s"*가 의미하는 것은 주어진 스택 *(STACK)* 클래스의 객체(s)에 멤버 함수 열 *"Pop(Push(s, i))"*을 실행시킨 후의 스택 객체의 상태는 변경되지 않아야 한다는 것을 의미한다.

Newstack . →STACK,
Push : STACK, integer→STACK;
Pop . STACK→STACK,
Top . STACK→Integer,
Empty STACK→Bool;

(a) 서명 부분

Pop(Push(s, i)) = s,
Top(Push(s, i)) = i;
Empty(Newstack) = true,
Empty(Push(s, i)) = false,
Pop(Newstack) = undefined;
Top(Newstack) =undefined;

(b) 공리 부분

그림 1 스택 클래스 대수적 명세

Gannon등이 개발한 DAISTS 시스템은 사용자가 제공한 테스트케이스의 수행 결과가 대수적 명세서에 기술된 공리와 일치하는가를 검사하는 시스템이다[10]. 예를 들어, 그림 1의 공리 *"Top(Push(s, i)) = i"*는 그림 2와 같은 프로그램으로 변경되어 테스팅이 행해진다.

```
void AxiomCheck(STACK s, int x)
        if (s->Push(x)->Top() == x) then
                printf("Ok");
        else    printf("Axiom failed");
}
```

그림 2 공리 검사 프로그램

Bouge등은 멤버 함수들의 조합에 대한 복잡도를 정의하고 주어진 복잡도에 따라 멤버 함수들의 조합을 대수적 명세로부터 논리 프로그래밍을 이용하여 테스트케이스를 자동으로 생성하는 방법을 제시하였다[2]. 또한 테스트케이스 선택 기준에 대한 이론적 기반이 되는 여러 가설(hypothesis)을 제시하였다.

Doong등은 Eiffel로 작성된 프로그램을 테스팅 하기 위하여 ASTOOT(A Set of Tools for Object-Oriented Testing)이란 시스템을 개발하였다[8]. 이 시스템은 테스트케이스를 생성하기 위해 대수적 명세서에 나타난 공리의 변환 규칙을 이용한다. 즉, 대수적 명세에 나타난 공리를 이용하여 멤버 함수의 조합들, s_1, s_2, 두 개의 조합의 결과가 동일한지 여부를 나타내는 *tag*를 사용하여 순서쌍(s_1, s_2, *tag*)을 테스트케이스로 구성한 후 s_1과 s_2를 실제로 실행시킨 결과가 *tag*와 일치하는지의 여부를 검사한다. 만약 일치하지 않는다면 구현에 오류가 있음을 뜻한다. 예를 들어, 사용자가 멤버 함수의 조합 *"Pop(Push(Push(Newstack(), x), y)"*을 제공하면 시스템은 공리 변환 규칙에 의해 멤버 함수의 조합 *"Push(Newstack(), x)"*을 생성한다. 이 때 *tag*는 *true*이다. 이 경우에, 테스트케이스는 다음과 같이 구성된다 :

$(s_1 : Pop(Push(Push(Newstack(), x), y),$
$s_2 : Push(Newstack(), x), true)$

이 테스트케이스를 실행하는 순서는 멤버 함수의 조합 s_1과 s_2를 실행시키고 이들의 실제 실행 결과를 비교하여 이 결과가 *true*인지를 검사하는 단계로 구성된다. 이를 C++언어의 형태로 표현하면 그림 3과 같다 :

```
void ExecTest()
{
        STACK s1 = Newstack();
        STACK s2 = Newstack();
        s1 = s1->Push(x)->Push(y)->Pop();
        s2 = s2->Push(x);
        if (EQN(s1, s2))
                printf("Ok");
        else    printf("Something wrong");
}
```

그림 3 테스트케이스 실행 프로그램

여기에서 EQN 프로시듀어는 두 개의 스택 객체의 상태가 동일한지를 결정하는 프로시듀어이며 이는 사용자가 정의하거나 두 개의 스택 객체의 상태가 동일한지를 나타내는 공리가 있는 경우에 한하여 대수적 명세로 부터 자동적으로 생성할 수 있다.

앞에서 기술한 기법들과는 달리 Jalote는 테스트케이스를 생성하기 위하여 대수적 명세의 서명 부분만을 이용하였다[13]. 그러나 그의 테스트케이스 생성 전략은 자유 변수를 지닌 테스트케이스를 생성하기 때문에 자유변수에 어떤 멤버함수의 조합이 대입되느냐에 따라 테스트케이스의 오류를 검출할 수 있는능력이 좌우된다.예를 들면, Jalote의 방법은 *"Empty (Push(q1))"*와 같은 자유변수 *"q1"*를 지닌 테스트케이스를 생성하며 이와 같은 테스트케이스는 자유변수에 어떤 멤버함수의 조합이 제공되느냐에 따라 테스트 결과가 틀려진다.

3.1.2 모델 기반 명세를 이용한 테스팅

모델 기반 명세는 set, sequence, map 등의 수학적으로 잘정의된 모델을 사용하여 클래스를 명세한다. 그리고 선택된 모델을 사용하여

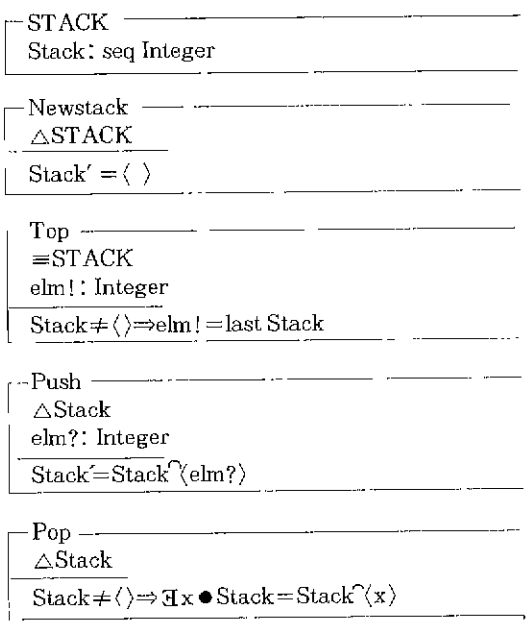


그림 4 스택 클래스의 Z 명세

각각의 멤버 함수에 대하여 선조건과 후조건을 명세하여 준다. 이러한 모델 기반 명세의 언어로는 Z, VDM, RSL 등이 있다. 그림 4는 Z 언어로 스택 클래스를 명세한 것이다.

Zweben 등은 클래스를 명세하는 모델 기반 명세로 부터 흐름 그래프(flow graph)를 구축하여 테스트케이스를 생성하는 방법을 제안하였다[23]. 흐름 그래프에서 노드는 멤버 함수를 나타내고 노드 A와 노드 B 사이의 에지(edge)는 멤버 함수 A를 실행한 후에 멤버 함수 B의 호출이 가능하다는 것을 나타낸다. 흐름 그래프에서 주어진 두 노드들 사이의 에지의 존재 여부는 모델 기반 명세에 의해 결정된다. 그림 5는 그림 4에서 주어진 스택 클래스의 Z 명세로 부터 구축된 흐름 그래프이다.

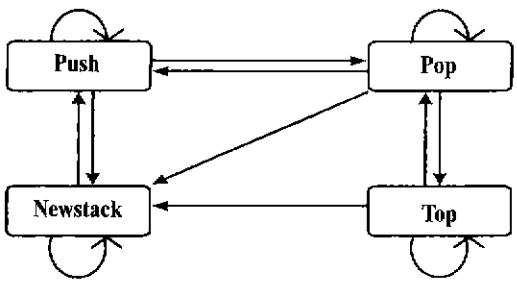


그림 5 스택 클래스의 흐름 그래프

Zweben 등은 이러한 흐름 그래프에 바탕을 둔 다양한 테스트케이스 선택 기준들을 제시하였다. 예를 들면, 노드 테스팅 완료 기준(node coverage)은 흐름 그래프에 나타나는 모든 노드, 즉 멤버 함수들을 포함하는 멤버 함수의 조합들을 테스트케이스로 요구하며 에지 테스팅 완료 기준(edge coverage)은 흐름 그래프의 모든 에지를 포함하는 멤버 함수들의 조합을 테스트케이스로 요구한다. 그림 5의 흐름 그래프에서 <*Newstack, Push, Top, Pop*>, <*Newstack, Newstack, Push, Newstack, Push, Push, Pop, Push, Top, Top, Push, Pop, Pop, Newstack, Push, Push, Push, Pop, Top, Pop, Top, Newstack*>은 각각 노드 테스팅 완료 기준과 에지 테스팅 완료 기준을 만족 하는 멤버 함수들의 열들이다. 이외에도 Zweben 등은 모델 기반 명세에 나타나는 각 멤버 함수의 인자 부분과 조건부에 나타난 객체의 정의 및 사용 정보에 바탕을 둔 다양한 테스트케이스 선정 기준들을 제안하였다.

범주 분할 테스팅 방법[17]은 비형식적인 명세가 주어졌을 때 테스트케이스를 설계하는 효과적인 방법이지만 실제로 테스팅 담당자의 능력에 좌우되는 경우가 많다. Amla 등은 Z 명세와 같은 형식적인 명세는 이미 테스트케이스를 생성할 수 있는 정보를 상당 부분 지니고 있다는 점에 바탕을 두고 Z 명세에 범주 분할 테스팅(category partition testing) 방법을 적용하여 각 멤버 함수에 대한 테스트케이스들을 생성하는 프로시듀어를 정의하였다[1].

지금까지 기술한 테스팅 방법들은 테스트케이스를 생성하는 방법에 중점을 둔 반면에 Jia가 제안한 테스팅 방법[14]은 각 멤버 함수에 대해 테스트케이스를 실행한 결과가 기대한 결과를 갖는지를 판별하는 문제(testing oracle problem)를 검색 함수(retrieve function)를 사용하여 자동화하였다는 점에서서 주목할만하다. 검색 함수의 기능은 구현된 프로그램의 상태(concrete state)를 문제 영역의 상태(abstract state)로 변환하는 역할을 한다(그림 6참조).

본래 검색 함수는 프로그램 변환이나 증명 분야에서 이론적으로 연구되고 적용되었지만 프로그램 테스팅 분야에 도입되지는 않았었다. 또한 Jia는 Zweben 등이 멤버 함수 조합들의 패턴에 중점을 두고 흐름 그래프상에서 테스트케이스 선택 기준을 정의한 것과는 달리 모델 기반 명세에서 나타나는 여러 제약 조건들이 구성되는 구조를 바탕으로 명세 적용 기준이라 불리우는 새로운 테스트케이스 선택 기준을 제

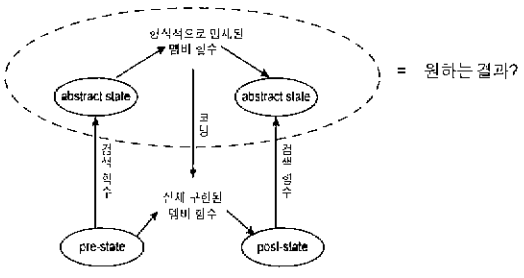


그림 6 테스팅 오라클 문제에서 검색 함수의 역할

시하였다.

3.2 구현 기반 테스팅

구현 기반 테스팅은 명세 기반 테스팅과는 달리 프로그램의 논리 구조 및 구현 내역으로부터 테스팅 정보를 추출하는 방법이다. 즉, 프로그램 코드에서 제어 흐름 정보 및 자료 흐름 정보를 분석하여 코드의 여러 특성을 검사할 수 있는 테스트케이스들을 생성하는 방법이다. 객체지향 프로그램의 테스팅에서는 명세 기반 테스팅에 비해 이 분야에 대한 연구가 상대적으로 미약한 형편이다. 대표적인 구현 기반 테스팅의 예로 1994년에 Harrold 등이 제안한 자료 흐름 정보에 기반을 둔 테스팅 방법을 들 수 있다[12]. 이 방법은 다음과 같이 3단계의 테스팅으로 나누어 클래스 테스팅을 수행한다:

- 개개의 멤버 함수를 테스팅하는 단위 멤버 함수 테스팅(intra-method testing)
- 각 멤버 함수에 대하여 해당 멤버 함수의 호출관계를 고려하여 테스팅하는 통합 멤버 함수 테스팅(inter-method testing)
- 멤버 함수들의 상호 작용을 테스팅하는 단위 클래스 테스팅(intra-class testing)

이러한 3단계의 테스팅을 위해 필요한 자료 흐름 정보를 얻기 위해 클래스의 코드로 부터 모든 가능한 멤버 함수의 조합들을 표현하는 클래스 제어 흐름 그래프(CCFG : Class Control Flow Graph)를 구축하고 프로시듀어 간

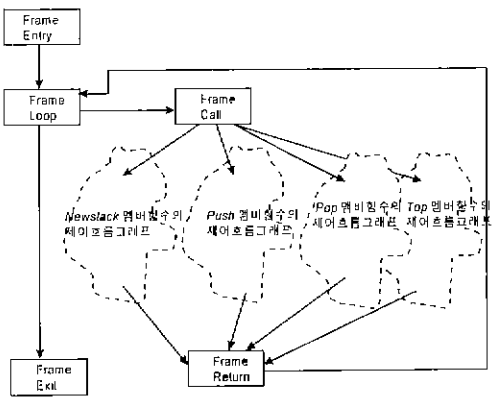


그림 7 스택 클래스의 CCFG

의 자료 흐름 정보를 분석할 수 있는 알고리즘을 적용하여 테스트케이스를 생성한다. 그림 7은 스택 클래스의 CCFG를 보여준다.

그림 7에서 각 멤버 함수의 제어 흐름 그래프에 Frame 노드들을 연결하는 이유는 호출 순서가 정적으로 정해진 절차적 프로그램과는 달리 클래스의 멤버함수는 임의의 순서에 의하여 호출될 수 있기 때문이다. 즉, Frame 노드를 사용하여 멤버 함수들이 어떠한 순서로도 호출될 수 있는 환경을 표현한 것이다.

자료 흐름을 이용한 테스팅의 또 다른 예로 Parrish 등이 제안한 방법을 들 수 있다[18]. Parrish 등은 3.1.2절에서 기술한 Zweben의 테스팅 기법을 확장하여 모델 기반 명세가 없어도 흐름 그래프를 구축하여 테스트케이스를 생성하는 방법을 제안하였다. Parrish 등이 제안한 방법과 Harrold 등이 제안한 방법의 주요한 차이점은 자료 흐름을 분석하는 대상의 차이에 있다.

Parrish 등의 방법은 분석 대상의 단위가 특정한 변수가 아닌 타입(type)이 된다. 예를 들면, 스택 클래스의 *Top* 멤버 함수는 *Integer* 인 형식 인자의 값을 변경시키기 때문에 타입 *Integer*를 정의(definition)한다고 한다. 또한, *Integer* 타입의 인자 값을 변경하기 위하여 현재 스택의 상태를 이용하므로 타입 Stack을 사용하는 연산이기도 하다. 이와 같은 자료 흐름 정보는 그림 5와 같은 흐름 그래프에 자료(실제 이 경우에서는 타입)의 정의 및 사용을 나타내는 에지를 사용하여 쉽게 첨가할 수 있다. 만약 멤버 함수 A에서 타입 T를 정의하고 멤버 함수 B가 타입 T를 사용한다면 A 노드에서 B 노드로 T로 라벨된 에지를 연결하면 된다. 이와 같이 흐름 그래프를 구성하면 절차적 프로그램의 테스팅에서 사용되었던 것과 유사한 자료 흐름 정보에 기반을 둔 테스트케이스 선택 기준들을 정의할 수 있다. 그러나, 자료 흐름 분석 대상이 타입이기 때문에 실제 프로그램에서 사용되는 변수를 대상으로 하는 Harrold 등의 방법보다 실제 프로그램 수행중에 나타나는 자료 흐름 경로들을 검증할 수 없는 경우가 발생할 수 있다.

또 하나의 대표적인 구현 기반 테스팅 방법

으로 1992년에 Harrold와 그의 동료들이 제안한 점진적 클래스 테스팅 방법을 들 수 있다[11]. 이 방법은 2장에서 간략하게 기술되었기 때문에 이 절에서는 그들이 점진적으로 클래스들을 테스팅 하기 위하여 사용하였던 방법의 요점만 기술하기로 한다. 점진적 클래스 테스팅이란 상속 계층 구조에서 우선 가장 상위에 있는 클래스를 테스팅한 후에 하위의 클래스들을 테스팅 하는 방법을 말한다. Harrold와 그의 동료들은 이와 같은 테스팅을 하기 위해 상속 계층 구조를 전체적으로 고려하지 않고 수정자(modifier) 개념을 도입하여 독립적인 구조들로 분리한다. 여기에서 수정자(M)는 부모 클래스(P)에서 자식 클래스(R)를 만들기 위해 필요로 하는 새롭게 추가되거나 재정의 되는 속성들의 집합을 말한다. 그들은 이러한 과정을 연산을 통하여 설명하였다. 즉,

$$R = P \dotplus M.$$

예를 들면, 클래스 A가 상속 계층 구조상에서 가장 상위에 위치하고 클래스 B가 A의 자식 클래스이며, 클래스 C가 B의 자식 클래스라 할 때 이러한 상속 계층 구조는 다음과 같이 분리될 수 있다. 이 때, M1을 B를 만들기 위해 필요로 하는 수정자이고 M2는 B에서 C를 생성하기 위한 수정자이다.

$$B = A \dotplus M1, C = B \dotplus M2$$

위의 식은 점진적으로 클래스들을 테스팅하기 위해서 부모 클래스를 먼저 테스팅하고 이 때 얻어진 테스팅 정보와 수정자에 대한 정보만를 가지고 자식 클래스 R에 대한 테스팅을 수행한다는 의미이다. 따라서 전체적인 상속 계층 구조를 고려할 필요가 없다.

이 외에도 Fielder의 테스팅 방법[9], Cheatham과 Mellinger의 테스팅 방법[3]이 구현 기반 테스팅에 속한다.

3.3 프로그램의 재검증을 위한 테스팅

프로그램의 재검증을 위한 테스팅은 프로그램에 대한 변경 작업이 원래의 프로그램의 기능에 영향을 주었는지를 변경 전의 프로그램에 수행했던 테스트케이스들의 부분집합을 사용하여 검사하는 방법을 말한다. 프로그램 P와 변경된 프로그램 P′가 주어졌을 때, 보통 재검증을 위해 수행하는 절차는 다음과 같다.

① P의 어느 부분이 변경되었는지를 식별한다.
② P의 테스팅에 사용했던 테스트케이스 집합을 T라 할 때 P′에 재사용될 T의 부분집합을 선택한다. 즉, T′ T.
③ T′에 대하여 P′를 검증한다.
④ T′가 P′에 대해 정의된 테스트케이스 선택 기준을 만족하지 못했다면 그 기준을 만족하도록 새로운 테스트케이스를 생성한다.
⑤ P′에 대한 새로운 테스트케이스 집합 T″를 생성한다.

위의 절차에서 첫 번째 단계를 위해 사용자가 직접 수정된 부분에 대한 정보를 제공할 수 있고 적절한 프로그램 분석 알고리즘을 사용하여 어느 부분에서 변경이 발생하였는지를 자동적으로 발견할 수도 있다. 두 번째 단계는 테스트케이스 재사용 문제(testcase reuse problem)라고 불리우는데 지금까지 제안된 대부분의 재검증을 위한 테스팅 방법들이 중점을 두고 있는 중요한 문제이다[4]. 테스트케이스의 재사용을 위하여 가장 많이 사용하는 방법은 프로그램의 변경이 발생하는 부분에 직접 또는 간접적으로 종속되어 있는 부분을 식별하여 그러한 부분들을 실행할 수 있는 테스트케이스(T′)들만 원래의 테스트케이스 집합(T)에서 선택하는 것이다. 이 때 종속되는 부분을 찾기 위하여 흔히 프로그램에 대해 자료 흐름을 분석한다.

대표적인 객체지향 프로그램의 재검증을 위한 테스팅 방법으로는 Rothermel과 Harrold가 제안한 테스팅 방법[20]과 Kung 등이 제안한 테스팅 방법[15]이 있다. Rothermel과 Harrold가 제안한 방법은 원래의 클래스(C)와 변경된(C′) 클래스가 주어졌을 때 이들 클래스에 대해서 클래스 종속 그래프(CDG : Class Dependence Graph)들을 구축하고 자료 흐름 분석을 통하여 변경된 부분 및 변경된 부분에 영향을 받는 부분을 식별한다. 이 때, C의 테스팅에 사용했던 테스트케이스 집합에서 식별

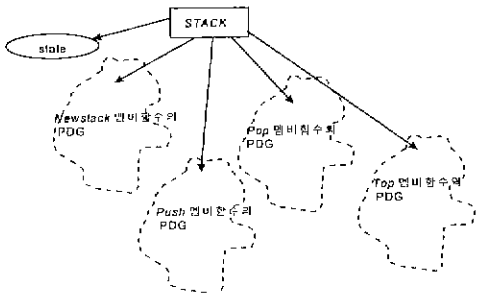


그림 8 스택 클래스의 CDG

된 부분들을 검사할 수 있는 테스트케이스 집합을 생성한다.

Rothermel과 Harrold가 사용한 CDG는 실제 절차적 프로그램의 자료 흐름 분석을 위해 사용되었던 프로그램 종속 그래프(PDG : Program Dependence Graph)를 확장한 것이다. PDG를 클래스의 재검증을 위해 직접 사용하지 못한 이유는 3.2절에서 기술한 바와 같이 Harrold가 CCFG를 도입한 이유와 같다. 그림 8은 스택 클래스에 대한 CDG를 보인 것이며 state 노드는 스택 클래스에서 정의되는 데이타 속성들의 선언부를 나타낸 것이다.

Rothermel과 Harrold 등이 클래스의 변경 작업에 영향을 받는 부분들을 자료 흐름을 분석하여 식별한 반면에 Kung 등이 제안한 방법은 클래스를 변경하였을 때 변경된 클래스(C)와 함께 다시 테스팅이 이루어져야 하는 클래스들을 식별하였으며 이러한 클래스들의 집합을 클래스 C의 방화벽(firewall)이라 정의하였다. 또한, 그들은 클래스 사이의 관계(inheritance, aggregation, association)를 객체 관계 그래프(ORG : Object Relation Graph)를 통하여 표현하였으며 주어진 클래스(C)의 방화벽은 ORG상에서 C의 이항 폐쇄 관계(transitive relation)와 같음을 증명하였다.

4. 결 론

본 고에서는 지금까지 제안된 객체지향 프로그램의 테스팅 방법들을 분류하고 대표적인 테스팅 방법들에 대해 기술하였다. 먼저, 객체지향 프로그램을 테스팅 할 때 고려해야 할 사항과 테스팅 기법들의 분류 기준에 대하여 고찰하였고 각 분류에 속한 대표적인 테스팅 방법을 소개하였다. 또한, 절차적 프로그램의 테스팅 방법들을 분류할 때와 같이 명세 기반 테스팅, 구현 기반 테스팅, 재검증을 위한 테스팅 방법으로 분류하여 객체지향 프로그램의 다양한 방법들을 기술하였다.

본 고에서 소개하지 않은 객체지향 프로그램의 테스팅 방법으로 상태 기반 테스팅(state-based testing) 방법을 들 수 있다[15,21,25]. Smith와 Robson이 제안한 상태 기반 테스팅 방법은 클래스의 각 멤버 함수에 대해 선행 상태(pre-state)와 후행 상태들을 식별(post-state)하고 주어진 선행 상태에 대해 멤버 함수의 실행 결과가 요구되는 상태인지를 검사한다[21]. 다른 형태의 상태 기반 테스팅 방법은 [15,25]사용한 방법과 유사하게 클래스의 행태를 유한 상태 기계(FSM : Finite State Automata)로 모델링하고 특정한 테스트케이스 선택 기준에 따라 테스트케이스들을 생성하는 방법 등이 있다.

문헌에 나타난 모든 테스팅 방법들이 본 고에서 기술한 어느 한 범주에 속한다는 것을 의미하지는 않는다. 예를 들면, [5]에서 제안한 테스팅 방법은 클래스를 테스팅하기 위하여 필요한 정보를 FSM으로 표현된 명세 뿐만아니라 심볼릭 실행(symbolic execution) 기법을 통하여 구현된 코드로 부터로도 얻는다. 따라서 보다 효과적으로 객체지향 프로그램을 테스팅 하기 위해서는 테스팅에 필요한 모든 정보를 가급적 일관된 틀안에서 이용할 수 있는 방법에 대한 연구가 더욱 진척되어야 할 필요가 있으며 제안된 테스팅 방법들을 비교 할 수 있는 방법이 필요하다[6].

참고문헌

[1] Amla, N. and Ammann, P., "Using Z Specifications In Category Partition Testing," in Proceedings of COMPASS, pp. 3-10, 1992.

[2] Bouge, L., Choquet, N., Epibourg, L., and

Gaudel, N.C. "Test Sets Generation from Algebraic Specifications Using Logic Programming," J. Systems and Software, vol. 6, pp. 343-360, 1986.

[3] Cheat, T.J. and Mellinger, L., "Testing Object-Oriented Systems," in Proceedings of the 18th ACM Annual Computer Science Conference, pp. 161-165, 1990.

[4] Chung, I.S. and Kwon, Y.R., "A Semantics-Based Method for Revalidating Modified Programs," Journal of Software Maintenance : Research and Practice, vol. 6, pp. 15-33, 1994.

[5] Chung, I.S, Malcolm, M., Lee, W,K, and Kwon. Y.R., "Applying Conventional Testing Techniques for Class Testing," in Proceedings of COMPSAC, 1996.

[6] 정인상, "포함, 파워 관계 및 테스트 재사용성에 기반을 둔 객체지향 프로그램을 위한 테스트 기준의 비교방법", 정보과학회 논문지, vol. 22, no. 5, pp. 693-704, 1995.

[7] 정인상, "테스트 명세서를 기반으로 하는 추상자료형의 테스팅", 정보과학회 논문지, vol. 22, no. 12, pp. 1995.

[8] Doong, R.K. and Frankl, P., "Case Studies in Testing Object-Oriented Programs," in The 4th Testing, Analysis and Verification Symposium, pp. 165-177, 1991.

[9] Fiedler, S.P., "Object-Oriented Unit Testing," Hewlett-Packard Journal, pp. 69-74, 1989.

[10] Gannon, J., McMullin, P., and Hamlet, R., "Data Abstraction and Implementation, Specification, and Testing," ACM Trans. Program. Lang. Syst., vol. 3, no. 3, pp.211-223, 1982.

[11] Harrold, M.J., McGregor, J.D. and Fitzpatrick, K.J., "Incremental Testing of Object-Oriented Programs," in 14th International Conference on Software Engineering, ACM, 1992.

[12] Harrold, M.J. and Rothermel, G., "Performing Data Flow Testing on Classes," in Proceedings of the second ACM SIGSOFT Symp. on Foundations of Software Engineering14th International Conference on Software Engineering, Dec., pp. 154-163. 1994.

[13] Jalote, P. and Caballero, M. G., "Automated Test Case Generation for Data Abstraction." in Proceedings of COMPSAC, pp. 205-210, 1988.

[14] Jia, X., "Model-Based Formal Specification Directed Testing of Abstract Data Types" in Proceedings of COMPSAC, pp. 360-366, 1993.

[15] Kung, D.C., Suchak, N. Gao, J., Hsia, P., Toyoshima, Y., and Chen, C,, "On Object State Testing," in Proceedings of COMPSAC, pp. 222-227, 1994.

[16] Kung, D.C., Gao, J., Hsia, P., Lin. J., and Toyoshima, Y., "Class Firewall, Test Order, and Regression Testing of Object-Oriented Programs," Journal of Object Oriented Programming, pp. 51-65, 1995.

[17] Ostrand. T.J. and Balcer, M.J., "The Category Partition Method for Specifying and Generating Functional Tests," Comm. ACM., vol. 31, no. 6, pp. 676-686, 1988.

[18] Parrish, A., Borie, R., and Corges, D., "Automated Flow Graph-Based Testing of Object-Oriented Software Modules," Journal of Systems and Software, no. 23, pp. 95-109, 1993.

[19] Perry, D.E. and Kaiser, G.E., "Adequate Testing and Object-Oriented Programming," Journal of Object Oriented Programming, pp. 13-19, 1990.

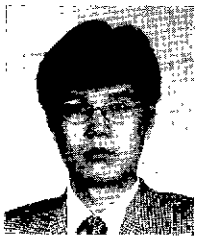
[20] Rothermel, G. and Harrold, M.J., "Selecting Regression Tests for Object-Oriented Software," Technical Report 94-104, Clemson Univ., SC, March, 1994.

[21] Smith, M.D. and Robson, D.J,, "A Suite of Tools for the State-Based Testing of Object-Oriented Programs," TR 14/92, University of Durham, England, 1992.

[22] Weyuker, E.J., "Axiomatizing Software Test Data Adequacy," IEEE Transactions on Software Engineering, vol. SE-12, no. 12, pp. 1128-1138, 1986.

[23] Zweben, S., Heym, W., and Kimich, J., "Systematic Testing of Data Abstractions Based on Software Specifications," Journal of Software Testing, Verification, and Reliability, vol. 1, pp. 39-55, 1992.
[24] Object-Oriented Software Testing, Special Issue, CACM, vol.37, no. 9, 1994.
[25] 홍형석, "유한 상태 기계를 기반으로 하는 객체지향 프로그램의 테스팅," 석사학위논문, KAIST, 1995.

정 인 상



1987 서울대학교 전자계산기공학과 학사
1989 한국과학기술원 전산학과 석사
1993. 8 한국과학기술원 전산학과 박사
1993.9～94.2 전자통신연구소 박사후연수연구원
1994.3～94.12 전자통신연구소 초빙연구원
1994.3～현재 한림대학교 컴퓨터공학과 조교수
관심분야 : 소프트웨어 테스팅, 실시간 시스템