DOI: 10.3745/KIPSTD.2009.16-D.4.517

뮤테이션 테스트를 이용한 동적 다이어그램에 근거한 테스트 케이스의 효율 비교

이 혁 수[†] · 최 은 만[††]

요 약

동적 UML 다이어그램은 객체 지향 언어로 구현된 프로그램의 복잡한 실행 동작에 대한 표현이 가능하다. 이로 인하여 동적 다이어그램 중, 순서, 상태, 액티비티 다이어그램을 이용하여 테스트 케이스를 추출하고 테스트 하는 방법이 많이 쓰이고 있다. 그러나 테스트 자원과 시간이 제한되어 있을 때 어떤 명세를 이용하여 테스트 케이스를 만드는 것이 더 효율적인지, 또한 어떤 특성이 있는지 알 필요가 있다. 이 논문에서는 ATM 시뮬레이션 프로그램을 세 가지 다이어그램으로 표현하고 이를 이용하여 서로 다른 테스트 케이스를 생성한다. 또한 뮤테이션 테스팅(Mutation Testing)을 실시하여 각 테스트 케이스에 대한 효율을 평가 하였다. 뮤턴트(Mutant) 생성은 절차적 방식과 객체 지향 방식에 의한 뮤테이션 연산자(Mutation Operator)를 구분해서 적용하였으며 뮤클립스(Muclipse)라는 이클립스(Eclipse) 기반의 플러그인 도구를 이용하였다. 생성된 테스트 케이스와 뮤턴트를 이용해서 뮤테이션 점수(Mutation Score)를 측정하고 이를 기반으로 각 테스트 케이스 및 여러 관점에서 테스트 케이스의 효율을 평가하였다. 이런 과정을 통해 테스트 케이스 생성 방식의 선택에 대한 힌트를 얻을 수 있었다.

키워드 : UML, 동적 다이어그램, 테스트 케이스 생성, 뮤테이션 테스팅

Comparison of Test Case Effectiveness Based on Dynamic Diagrams Using Mutation Testing

Lee Hyuck Su[†] · Choi Eun Man[††]

ABSTRACT

It is possible to indicate the complex design and execution of object-oriented program with dynamic UML diagram. This paper shows the way how to make several test cases from sequence, state, and activity diagram among dynamic UML diagram. Three dynamic UML diagrams about withdrawal work of ATM simulation program are drawn. Then different test cases are created from these diagrams using previously described ways. To evaluate effectiveness of test cases, mutation testing is executed. Mutants are made from MuClipse plug-in tool based on Eclipse which supports many traditional and class mutation operators. Finally we've got the result of mutation testing and compare effectiveness of test cases, etc. Through this document, we've known some hints that how to choose the way of making test cases.

Keywords : UML, Dynamic Diagram, Test Case Generation, Mutation Testing

1. 서 론

테스트 작업은 오류를 찾아낼 목적으로 개발한 소프트웨어를 시험하는 것이며 시험을 위하여 입력 값을 찾아내는 작업과 이를 실행시키고 그 결과를 체크하는 작업으로 이루어진다. 이들 작업 중 테스트 작업의 핵심은 효율적인 테스트 케이스를 찾는 것이다. 오러클(Oracle)만 잘 정의한다면 결과를 체크하는 일은 그다지 어려운 일이 아니며 같은 수의 테스트 케이스를 실행하더라도 오류를 잘 드러낼 수 있는 정도, 즉 테스트의 효율이 달라진다.

테스트 작업의 효율을 높이기 위해서는 개발 프로세스 초기에 오류를 찾을 수 있도록 명세기반의 테스트를 도입하는 것이 필수적이다. 명세기반 테스트를 개발 초기부터 도입하여 오류를 조기에 발견하고 수정한다면 코딩 후에 발견하여 수정하는 노력을 크게 절감할 수 있다. 또한 원시코드 안의 복잡한 실행 흐름 경로를 일일이 살펴보지 않고 같은 시험 효과를 보일 동치그룹의 대표값을 명세 안에서 파악하여 테스트 케이스를 찾아낼 수 있다는 점에서 명세기반 테스트는

† 정 회 원 : 휴맥스 품질경영부문 SQE팀 대리
†† 정 회 원 : 동국대학교 컴퓨터공학전공(교신저자)
논문접수 : 2008년 9월 23일
수 정 일 : 1차 2009년 3월 23일, 2차 2009년 6월 2일
심사완료 : 2009년 6월 8일

효율적이다. 명세 중에서 테스트 케이스 추출에 중요한 것은 동적 다이어그램들이다. 실행 객체들 사이의 인터랙션, 즉 메시지 교환을 순서대로 표현한 순차 다이어그램(Sequence Diagram)은 테스트 케이스를 추출하는 데 자주 사용된다. 또한 객체의 상태 변화를 나타내는 상태 다이어그램(State Diagram)이나 액티비티의 병렬적 흐름을 나타내는 액티비티 다이어그램(Activity Diagram)도 테스트 케이스를 쉽게 찾을 수 있는 명세이다. 이러한 동적 다이어그램들이 테스트 케이스를 찾는 데 많이 쓰이는 이유는 구조적 측면을 나타내는 정적 다이어그램보다 인스턴스 정보들이 나와 있는 시스템의 동적 스냅샷이기 때문이다.

따라서 UML의 동적 다이어그램으로 표현된 명세를 기초로 테스트 케이스를 생성하는 연구[4, 5, 6, 11, 12]가 많이 있다. 대부분 시스템의 동작 측면을 나타낸 것으로 객체 사이의 메시지 흐름을 따라 시험하는 테스트 케이스를 찾아내는 연구[11, 12]와 병렬적인 액티비티의 실행을 따라 시험하려는 연구[4, 5], 객체의 상태 변화를 따라 시험하려는 연구[6]가 있었다. 그러나 이들 연구에서는 단지 객체지향 프로그램의 테스트 케이스를 UML 명세로부터 생성하는 방법을 각각 설명하고 있고 생성된 테스트 케이스가 과연 어떤 특성과 어떤 효율을 보이는지 언급이 없다. 테스트 케이스는 어떤 명세를 기초로 하는가에 따라 체크되는 부분도 달라지고 오류를 검출하는 효율도 달라진다.

뮤테이션 테스트(Mutation Test)는 테스트 데이터의 효율을 점검하기 위한 좋은 도구이다. 정상 프로그램과 뮤턴트(Mutant)가 섞인 오류 프로그램을 테스트 케이스가 얼마나 잘 구별할 수 있는지 시험할 수 있기 때문이다. 테스트 케이스가 방대해지고 정해진 시간에 모두 실행하기 어려운 상황에서 테스트 케이스의 오류 검출 효율성을 비교하고 테스트 데이터의 특성을 파악하는 일은 매우 중요하다.

이 연구에서는 먼저 테스트 케이스를 UML 동적 명세에서 테스트 케이스를 생성하여 각 테스트 데이터들이 어떤 효율을 보이는지 뮤테이션 테스트를 이용하여 실험하였다. 뮤턴트는 일반적인 절차중심 프로그램에서 사용하던 연산자 및 조건부를 변형한 것과 클래스 자체의 변형 두 가지를 적용하였다. 그 이유는 어떤 명세를 근거한 테스트 케이스가 어떤 오류 타입을 찾아내는 데 효율적인지를 알아내기 위함이다.

이후 본 논문의 구성은 다음과 같다. 2장에는 관련 연구와 기존 연구 방식의 문제점을 살펴보고 3장은 동적 명세 기반 테스트 방법, 즉 순차 다이어그램, 상태 다이어그램, 액티비티 다이어그램으로부터 테스트 케이스를 생성하는 각각의 방법과 뮤테이션 테스트 방법을 살펴본다. 4장에서는 실험 대상으로 ATM(Automatic Teller Machine) 동작에 대한 각각의 다이어그램을 도식화하고 이를 통해 실제 테스트 케이스를 추출한다. 이를 바탕으로 이클립스(Eclipse) 플러그인 형태의 뮤테이션 테스팅 자동화 툴인 뮤클립스(Muclipse)[10]를 통해 얻은 뮤턴트와 뮤테이션 테스팅 실행 후의 결과를 기술하였다. 5장은 뮤테이션 테스팅을 통해 얻은 각각의 결과에 대하여 기술하였고, 얻은 결과를 바탕으로 마지막에는 어떤 식으로 테스트 케이스 생성 기법을 선택하고 적용해야 하는지를 결론으로 정리하면서 향후 연구 방향에 대해 논의하였다.

2. 관련 연구 및 문제 제기

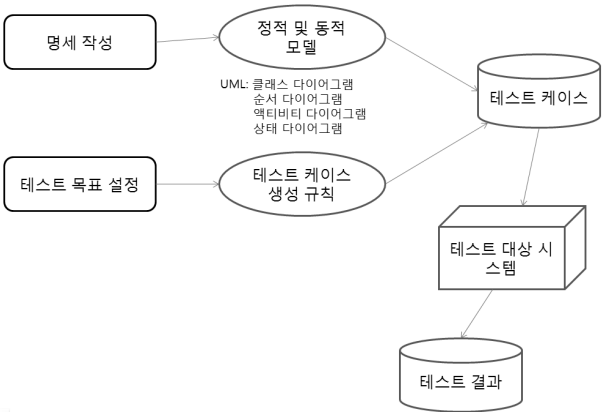
테스트 케이스를 설계하는 데 사용하는 두 가지 주된 정보는 명세나 원시 코드이다. 화이트박스 테스트에서는 원시 코드에서 파악할 수 있는 실행 경로나 자료 타입 등을 사용하지만 그 규모나 접근성 등의 이유로 인하여 단위 테스트 범위를 벗어나면 비효율적이다. 즉 테스트 대상 규모가 커지고 추상 수준이 올라갈수록 테스트 케이스를 생성할 때 원시코드보다는 명세를 사용하는 것이 효과적이다.

명세 기반 테스트는 크게 두 가지 측면이 있다. 하나는 모델이 적절히 설계되었는지 검토하는 정적인 측면과 다른 하나는 설계 모델에 적합하게 구현되었는지 테스트 케이스로 검증하는 동적인 측면이다. [18, 19]와 같은 연구는 주로 전자에 해당하는 것으로 모델 자체의 테스트나 애니메이션에 의한 적합성에 초점이 맞추어져 있다. 한편 UML 명세에 근거한 테스트 케이스의 생성은 사용 사례를[20] 비롯하여 UML의 여러 가지 동적 모델[5, 6, 11]을 이용하는 방법을 제시하고 있다. 또한 AGEDIS[21], COTE[22], Rhapsody ATG[23]와 같은 연구들이 명세로부터 테스트 케이스를 얻어내는 과정을 자동화하려는 시도들이다.

이외에도 [7]은 슬라이싱 기술을 이용해 순차 다이어그램에서 테스트 케이스를 생성했는데 본 논문에서 적용한 방식보다 적은 단계를 거치면서 구체적인 테스트 케이스 생성이 이루어지고 자동화도 비교적 쉽게 적용 가능하다는 차이점이 있었다. [14] 역시 본 논문에서 사용한 방식과는 다르게 테스트 케이스를 생성하는데, 객체의 상태 변화를 계층 구조로 표현하고 이를 이용해 테스트 케이스를 생성한다. 단위 테스트에 대한 방법을 언급한 [15]는 테스트 커버리지가 높은 테스트 케이스일수록 높은 효율을 나타낸다는 내용을 포함하고 있는데 다양한 테스트 케이스 중에서도 좀 더 효율적인 테스트 케이스를 실행하는데 대한 근거를 제시한다. [9]는 상태와 순차 다이어그램에서 얻은 테스트 케이스를 통해 단위 테스트와 통합 테스트의 효율 비교하는데 각 클래스의 특징에 따른 효율적인 테스트 케이스 선택에 대한 근거를 제시한다.

명세를 기반으로 하여 테스트 케이스를 작성하는 과정은 (그림 1)과 같이 명세를 기반으로 테스트 케이스를 만들고 이를 실행하여 결과를 검토하는 작업이다. 모델로 사용되는 것은 UML 표현 방법에서 객체의 관계를 나타내는 클래스 다이어그램, 메시지 교환을 나타내는 순차 다이어그램, 병렬성을 나타내는 액티비티 다이어그램, 시스템이나 객체의 상태를 나타내는 상태 다이어그램이 주로 쓰인다.

명세기반 테스트 방법에서 중요한 것은 테스트 케이스를 생성하는 규칙을 가이드 하는 테스트 목표이다. 예를 들어 클래스 다이어그램을 가지고 테스트 케이스를 만든다면 원



(그림 1) 명세기반 테스트 과정

하는 클래스들이 제대로 링크되어 있는지, 링크의 다중도가 적합한지, 원하는 정보를 얻기 위하여 링크를 따라 네비게이션 할 수 있는지, 또한 이를 제공하기 위한 가시성(visibility)이 확보되어 있는지 검사하기 위한 것들이다. 이렇게 클래스 다이어그램은 시스템의 정적인 관계만을 나타내므로 특정한 실행 스냅샷, 즉 동작을 체크하는 데 사용되기는 어렵다.

따라서 시스템의 어떤 외부기능이 사용되었을 때 구동되는 메시지 교환 흐름이 제대로 잘 이루어지고 있는지 테스트하는 순차 다이어그램이나 메시지 호출에 의하여 특정한 객체의 상태가 변화하는지 테스트 하는 상태 다이어그램, 스레드와 같은 병렬 수행 흐름이 잘 들어맞는지 체크하는 액티비티 다이어그램 등이 명세기반 테스트에 주로 사용된다.

하지만 서론에서도 언급했듯이 이런 명세기반 테스트, 특히 UML 동적 다이어그램의 경우는 객체 지향 프로그램의 테스트 케이스를 명세로부터 생성하는 방법 소개에 그치고 있다. 즉 UML 다이어그램으로부터 생성된 테스트 케이스가 어떤 특성과 효율을 보이는지에 대한 부분은 언급이 누락되었다는 단점이 있다. 단순히 테스트 케이스를 생성하고 이를 실행시키고 결과를 얻는 것에 그치지 않고 어떤 테스트 케이스가 어떤 상황에 더 적합한지를 알 수 있어야 한다. 왜냐하면 주어진 시간과 환경은 제한되어 있는 경우가 대부분이고 이런 상황 하에 모든 부분을 테스트 한다는 것은 불가능하기 때문이다. 결국 이런 제한된 조건에서는 적절하고 효율적인 테스트가 가능한 테스트 케이스를 추출하고 실행하는 것이 반드시 필요하다. 그렇기 때문에 본 논문에서는 동적 다이어그램으로부터 테스트 케이스만 추출하는 기존 연구에서 누락된 테스트 케이스 효율 측면을 다각적인 관점에서 고찰하고자 한다.

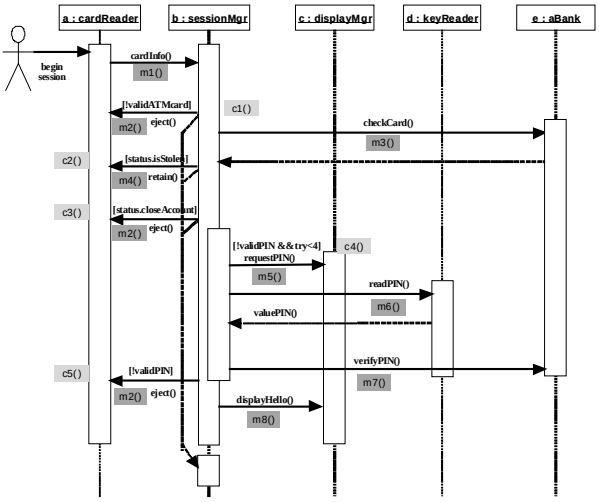
3. 명세 기반 테스트 케이스 생성과 효율 비교 방법

3.1 명세기반 테스트 케이스 생성 방법

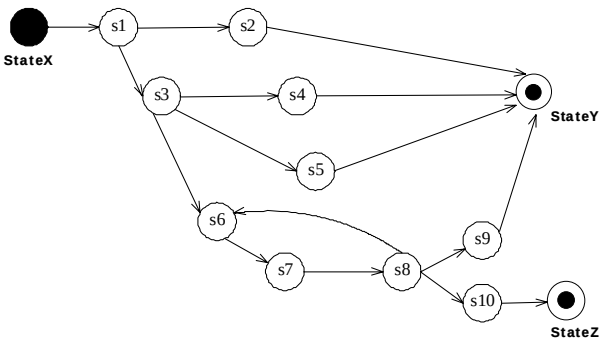
시간에 따른 객체 간 메시지의 상호 작용을 나타내는 명세인 순차 다이어그램을 이용하여 테스트 케이스를 생성하는 방법은 다음과 같다[11]. 순차 다이어그램만을 그대로 놓고 살펴보면 어떤 것을 테스트 단위로 삼을지 알아내기 어렵다. 그러므로 순차 다이어그램(Sequence Diagram) 안에 표현된 이벤트의 흐름을 그 의미를 살펴보면서 입출력을 고려한 실행 흐름을 그래프(Sequence Diagram Graph)로 변경하는 것이 [11]의 핵심 내용이다.

ATM의 PIN 인증 과정을 순차 다이어그램으로 도식화하고 순차 다이어그램 그래프 형태로 변경하는 과정을 예로 (그림 2, 3)에 표현하였다. 순차 다이어그램에 표시된 의미를 메시지 교환과 실행 결과, 상태 변화 위주로 분류하면 (그림 3)과 같이 검증 성공한 PIN <state Y>와 검증 실패한 PIN <state Z> 상태로 나누어진다. 이벤트의 흐름은 (그림 3)에서와 같이 다섯 개의 동작 시나리오 실행 흐름으로 구성되며 이 시나리오들은 적합한 초기 값과 예상 결과 값을 대입하면 테스트 케이스로 쉽게 바꿀 수 있다. 최종적으로 아래와 같은 테스트 케이스를 얻을 수 있다.

상태 다이어그램과 액티비티 다이어그램을 이용한 테스트 케이스 생성 방법도 유사하다. 근본 아이디어는 각 다이어그램이 추상의 정도나 의미는 다르지만 어느 정도의 실행 경로를 나타내고 있는데 상태 다이어그램이나 액티비티 다이어그램 안에 나와 있는 실행 경로를 발견하고 이들 경로



(그림 2) ATM 상에서 PIN 인증 과정을 표현한 순차 다이어그램



(그림 3) (그림 2)에 대한 SDG

Test case #1
Input: Card = "Not ATM"
Output: Eject card
Test case #2
Input: Card = "ATM", Status = "Stolen"
Output: Eject card
Test case #3
Input: Card = "ATM", Status = "Okay", Account = "Close"
Output: Eject card
Test case #4
Input: Card = "ATM", Status = "Okay", Account = "Open", PIN = "Invalid"
Output: Message "Invalid PIN: Try Again"
Input: Card = "ATM", Status = "Okay", Account = "Open", PIN = "Invalid"
Output: Message "Invalid PIN: Try Again"
Input: Card = "ATM", Status = "Okay", Account = "Open", PIN = "Invalid"
Output: Message "Invalid PIN: Try Again"
Input: Card = "ATM", Status = "Okay", Account = "Open". PIN = "Invalid", Try = <4>
Output: Message "Invalid PIN: Try Later" Eject card
Test case #5
Input: Card = "ATM", Status = "Okay", Account = "Open", PIN = "Valid"
Output: Display "Hello"

(그림 4) ATM 시스템의 PIN 인증 관련 테스트 케이스 예

를 구동시킬 입력 값들을 찾는 작업이다. 특히 상태 다이어그램은 루프 경로를 제외한 의미 있는 서로 다른 경로를 찾는 것이 필요하다.

3.2 뮤테이션 테스트에 의한 효율 비교 방법

1978년에 처음으로 제안된 뮤테이션 테스팅은 결함을 기반으로 테스트 케이스의 효율성을 측정하는 방법이다. 이 기법은 간단한 결함, 즉 일종의 오류인 뮤테이션 연산자(mutation operator)를 주입하여 프로그램을 실행시켰을 때, 테스트 케이스가 오류에 의한 잘못된 결과를 발견해 내는데 기여하는지 아니면 지나치는지를 판단하는 것이다. 평가하려는 테스트 케이스에 의하여 뮤턴트가 삽입된 프로그램을 실행한 후 수집된 결과를 이용해 테스트 케이스 효율을 평가하는 방식을 사용하는데 전체 뮤턴트 수에 대해 테스트 케이스 실행 후 잘못된 결과를 나타내는 뮤턴트 수를 나누고 백분율로 표시하는 간단한 연산 방식을 이용한다. 이를 뮤테이션 점수(mutation score)라고도 한다[16]. 뮤테이션 점수가 높을수록 오류를 발견할 확률이 많은 효율 높은 테스트 케이스라 할 수 있다.

전통적인 뮤테이션 테스팅은 뮤테이션 연산자 대부분이 C 언어와 같은 절차적 프로그램에 맞춰 구성되어 있다. 그러나 객체 지향 프로그램은 은닉, 상속, 다형성 등과 같은 특징을 포함하고 있어 절차적 프로그램과는 다르다. 그러므로 객체 지향 언어로 구현된 프로그램에 대해서는 절차적 프로그램에 적용하던 방식 외에도 객체 지향에 적합한 뮤테이션 연산자가 필요하다[6, 7]. 이에 속하는 것이 클래스 뮤테이션 연산자이다[8]. 이 연구에서 명세기반으로 만든 테스트 케이스의 효율을 비교하기 위하여 사용한 뮤테이션 연산자는 <표 1>과 같이 전통적인 방식과 객체 지향 프로그램의 고유한 타입을 총망라한다[7].

〈표 1〉 뮤테이션 연산자의 사례

종류	언어기능	오퍼레이터 명칭	설 명
전통적	문장	ABS	절대값 삽입(Absolute value insertion)
		AOR	산술 오퍼레이터 치환(Arithmetic operator replacement)
객체지향적	캡슐화	AMC	접근 가시성 변경(Access modifier change)
	상속	IHD	변수 삭제를 감춤(Hiding variable deletion)
		IHI	변수 추가를 감춤(Hiding variable insertion)
	다형성	PNC	자식 클래스 타입으로 메소드 호출
		PMD	부모 클래스 타입으로 멤버 변수 선언
	Java 특수 기능	JTI	this 키워드 삽입
		JTD	this 키워드 삭제

위의 뮤테이션 연산자가 포함된 뮤턴트를 생성하고 뮤턴트가 포함된 프로그램을 명세를 기반으로 생성한 테스트 케이스를 이용해 실행 해 보았다. 뮤턴트 생성은 객관성과 정확성을 위하여 뮤테이션 테스팅 자동화 도구인 이클립스(Eclipse) 기반 플러그-인 툴인 뮤클립스(MuClipse)를 이용하였다[1].

4. 동적 명세 기반 테스트 케이스 생성과 뮤테이션 테스트를 이용한 효율 비교 실험

이 논문의 실험 대상은 명세기반 테스트 케이스 추출 방법을 소개하는 연구에서 사례로 많이 쓰고 있는 입출금 기능이 포함된 간단한 가상 ATM 시뮬레이션 프로그램을 이용하였다. 여러 동작에 대한 테스트 케이스를 생성하였으나 여기서는 인출 과정에 대한 부분만을 설명한다. 2장에서 언급한 방식을 이용해 서로 다른 동적 다이어그램을 각각의 적합한 방식으로 테스트 케이스를 추출하였으며 이를 JUnit 형태로 코드화 하였다. 테스트 대상 프로그램은 총 5개의 클래스로 구성되어 있는데 이에 대한 간단한 정보는 아래 <표 2>와 같다.

〈표 2〉 실험 대상 프로그램 크기 정보

Class Names	No. of methods	Line of code	Class Information
Account	6	48	Account Information Handling
Customer	5	34	Customer Information Handling
MyBank	6	62	Customer Management
BankSystem	10	156	Main Business Logic
BankDisplay	7	178	User Interface

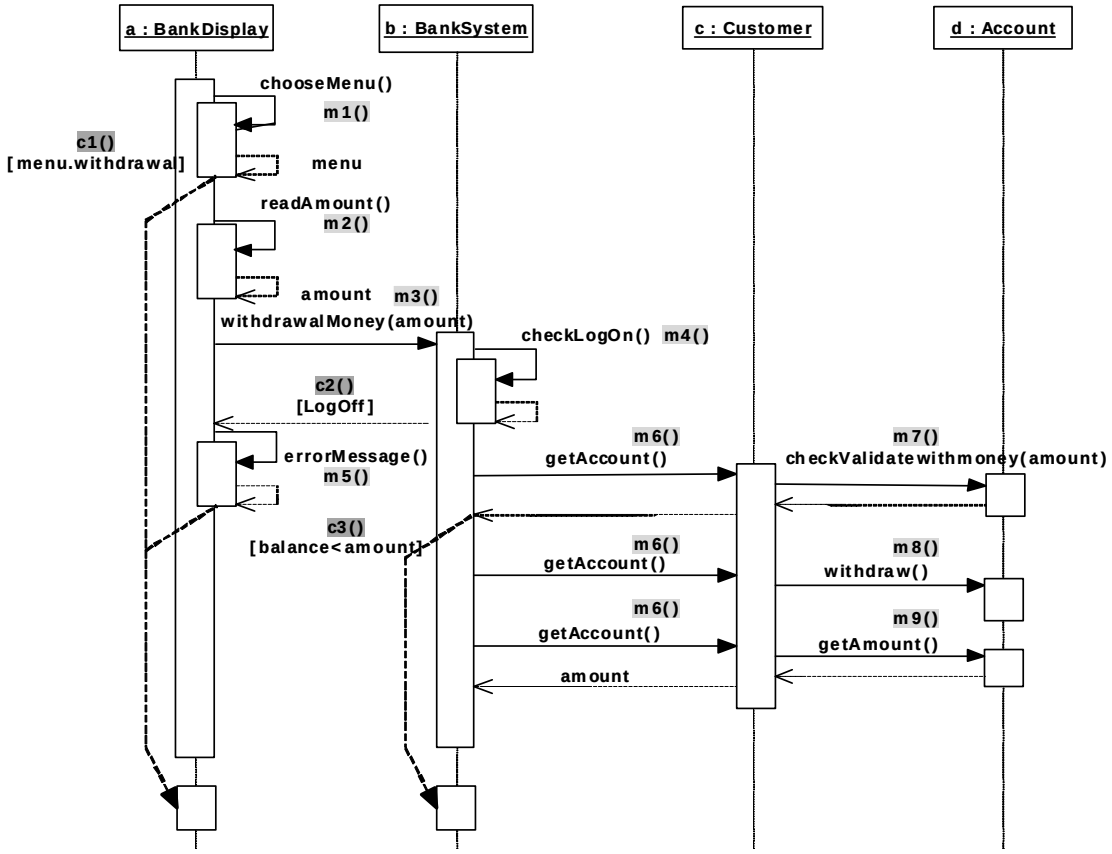
4.1 테스트 케이스 생성 과정

ATM의 인출 동작을 나타내는 순차 다이어그램은 전체 클래스 중 네 개의 클래스 간 상호 메시지 교환이 이루어진다. 이를 순차 다이어그램으로 명세화 하면 (그림 5)와 같다.

(그림 5)의 순차 다이어그램에서 2.2에서 설명한 방법으로 SDG 그래프로 표현하면 (그림 6)과 같이 총 네 가지 시나리오를 추출할 수 있다.

이렇게 얻은 시나리오 중 가장 높은 실행 커버리지를 가지는 시나리오를 선택한다. 여기서는 네 개의 동작 시나리오 중 가장 많은 메시지 교환이 이루어지는 네 번째 시나리오가 테스트 케이스 생성 후보가 된다. 이를 이용하여 실행 가능한 JUnit 코드로 표현하면 아래 (그림 7)과 같다.

ATM의 인출 과정을 상태 다이어그램으로 표현하면 (그림 8)과 같다. 상태 다이어그램은 단일 객체의 상태 변화에 초점을 맞추기 때문에 하나의 클래스(Account)의 인출 동작에 해당되는 핵심 메소드의 상태 변화를 도식화 했다. 이



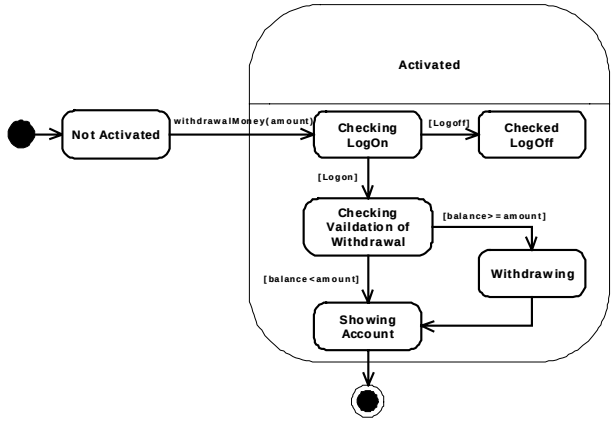
(그림 5) ATM의 인출 과정을 표현한 순차 다이어그램

<scn1	<scn2	<scn3	<scn4
StateX	StateX	StateX	StateX
s1: (m1, a, a)|c1	s2: (m1, a, a)	s2: (m1, a, a)	s2: (m1, a, a)
StateY>	s3: (m2, a, a)	s3: (m2, a, a)	s3: (m2, a, a)
	s4: (m3, a, b)	s4: (m3, a, b)	s4: (m3, a, b)
	s5: (m4, b, b)|c2	s7: (m4, b, b)	s7: (m4, b, b)
	s6: (m5, a, a)	s8: (m6, b, c)	s8: (m6, b, c)
	StateY>	s9: (m7, c, d)|c3	s10: (m7, c, d)
		StateZ>	s11: (m6, b, c)
			s12: (m8, c, d)
			s13: (m6, b, c)
			s14: (m9, c, d)
			StateW>

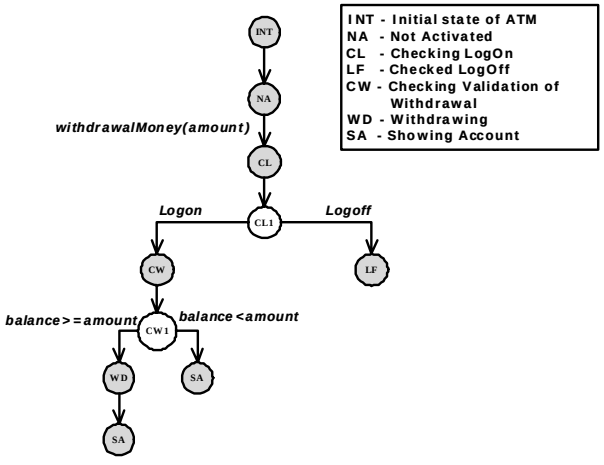
(그림 6) 네 개의 동작 시나리오

```
3 import junit.framework.TestCase;
5
6 public class TestCaseofSeq04 extends TestCase {
7
8     private BankSystem tmpbanksys;
9     private String tmpid, tmpname, tmpbalance;
10
11    public void setUp(){
12        tmpid = "rnrtn00";
13        tmpname = "hyucksu lee";
14        tmpbalance = "100000";
15        tmpbanksys = new BankSystem();
16
17        //create customer
18        assertTrue(tmpbanksys.createCustomer(tmpid, tmpname, tmpbalance));
19        //log on
20        assertTrue(tmpbanksys.logOn(tmpid));
21    }
22
23    //log on state check: current state is log on
24    public void testMethod_s4s7(){
25        String tmpwithdrawal = "10000";
26
27        assertEquals(tmpbanksys.checkLogOn(), tmpbanksys.withdrawalMoney(t
28    }
29
30    //checking validate state of withdrawal: withdrawal possible
31    public void testMethod_s8s10_1(){
32        String tmpwithdrawal = "10000";
33        long ltmpbalance = Long.parseLong(tmpbalance);
34        long ltmpwithdrawal = Long.parseLong(tmpwithdrawal);
35        boolean bvalide = false;
36        if((ltmpbalance - ltmpwithdrawal) >= 0) bvalide =  true;
37
38        assertEquals(bvalide,tmpbanksys.logOn.getAccount().checkValidateWi
39    }
40
41    public void testMethod s8s10 2(){
```

(그림 7) 순차 다이어그램에서 얻은 시나리오를 통해 생성된 JUnit 코드



(그림 8) ATM의 인출 과정을 표현한 상태 다이어그램



(그림 9) (그림 8)을 TFG로 변경한 결과

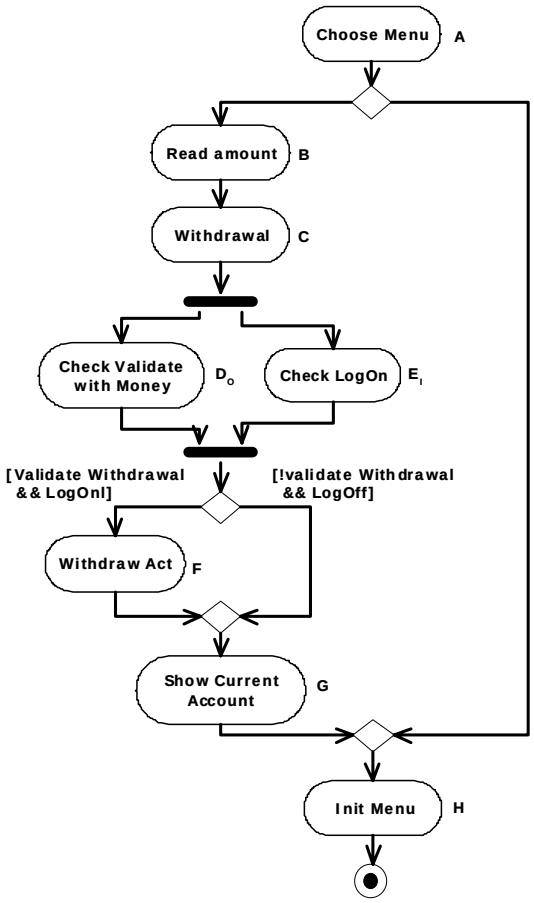
메소드는 다른 클래스의 메소드 간의 메시지 교환 작업을 포함하기 때문에 핵심 메소드의 상태 변화는 다른 클래스 메소드의 상태 변화를 의미하기도 한다.

(그림 8)의 상태 다이어그램은 (그림 9)와 같은 TFG 변경 과정을 거쳐 (그림 10)의 테스트 케이스를 얻을 수 있다 [6]. 3.1의 경우와 같은 이유로 여기서도 실행 커버리지가 가장 높은 테스트 케이스를 선택한다. 이를 이용해 (그림 7)처럼 테스트 케이스 내용에 맞도록 JUnit 코드를 생성한다.

인출 동작을 액티비티 다이어그램으로 도식화하면 (그림 11)과 같다. 일반적인 액티비티 다이어그램과의 차이점은 각 액티비티마다 알파벳 대문자가 옆에 표시되어 있다. fork-join

Test Case #1	Test Case #2	Test Case #3
withdrawalMoney (amount)	*withdrawalMoney (amount)*	*withdrawalMoney (amount)*
checkLogOn();;Logon	*checkLogOn();;Logon*	*checkLogOn();;Logoff*
checkValidatewithmoney(amount)	*checkValidatewithmoney(amount)*	*errorMessage()*
;;balance>=amount	*;;balance<amount*	
withdraw(amount)	*getAmount()*	
getAmount()		

(그림 10) 상태 다이어그램에서 얻은 테스트 케이스



(그림 11) ATM의 인출 과정을 표현한 액티비티 다이어그램

〈표 3〉 액티비티 다이어그램에서 얻은 최종 테스트 경로

No	테스트 경로	
1	A → H	O
2	A → B → C → D_o → E_I → G → H	O
3	A → B → C → E_I → D_o → G → H	X
4	A → B → C → E_I → D_o → F → G → H	X
5	A → B → C → D_o → E_I → F → G → H	O

부분의 경우 선행되어야 하는 액티비티는 아래 첨자로 I가 표시되어 있고 이후에 실행되는 액티비티는 아래 첨자로 O가 표시되어 있다.

(그림 11)의 액티비티 다이어그램에서 얻을 수 있는 총 경우의 수는 <표 3>과 같이 다섯 가지가 된다. 하지만 그 중 두 가지는 [5]에서 언급한 내용과 같이 선행 액티비티가 후행 액티비티 뒤에 위치한 경우이기 때문에 실행 가능한 테스트 경로에서 제외 된다. 결국 가능한 테스트 경로는 세 가지가 되며 이 중 실행 커버리지가 가장 높은 다섯 번째 테스트 경로를 이용해서 앞의 두 경우와 동일하게 JUnit 코드 형태로 테스트 케이스를 생성하였다.

4.2 뮤테이션 테스트

뮤클립스를 이용해서 모든 클래스에 대한 뮤턴트를 생성을 실행한 결과는 <표 4>와 같다. 이때 선택한 뮤테이션 연산자는 전통적 뮤턴트와 객체 지향 뮤턴트 모두를 선택했다. 생성된 뮤턴트는 총 277개 이고 각 뮤테이션 연산자 별, 클래스 별로 정리하였다.

이 중 BankDisplay 클래스는 객체지향 관련 문법이 존재하지 않아 객체지향 뮤턴트를 적용할 수 없었다. 이 클래스는 프로그램의 UI에 해당되는 부분으로 특별히 객체 지향 언어의 특징을 포함하고 있지 않고 UI 동작을 JUnit으로 확인하기에는 무리가 있기 때문에 뮤테이션 테스팅 실행 결과에서도 누락시켰다.

〈표 4〉 뮤클립스를 통해 생성된 뮤턴트 정보

타입	M.OP.	Account	Customer	MyBank	BankSystem	BankDisplay
전통적 뮤턴트	AOIS	36	2	16	34	2
	AOIU	5	1	5	4	1
	LOI	10	1	7	10	1
	COI	2		9	14	14
	ROR	10		22	12	8
	AORB	12		0		
	AORS			1	1	
	COR			4		
	Total	75	4	64	75	26
객체지향 뮤턴트	JSI	2	3	1		
	JTD	2	3			
	JTI	2	3			
	PRV		5			
	EAM			1	4	
	JDC			1	1	
	JID			1		
	JSI			1	3	
	Total	6	14	5	8	

5. 테스트 케이스 비교 실험 결과

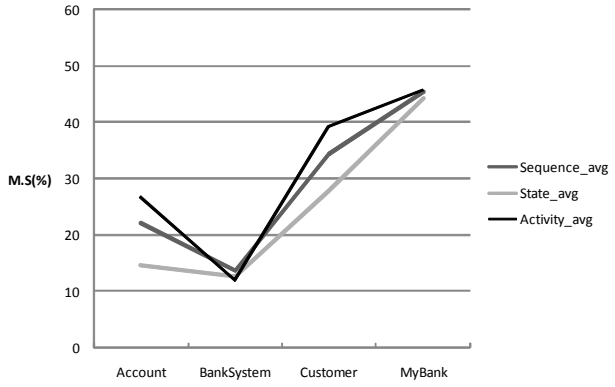
4.1 절에서 추출된 테스트 케이스와 뮤클립스를 통해 생성된 뮤턴트를 이용해 뮤테이션 테스팅을 실시한 결과는 <표 5>와 같다.

뮤테이션 테스팅 결과가 정리된 <표 5>를 바탕으로 동적 다이어그램 별, 각 클래스 별, 뮤턴트 타입 별 등의 관점에 분석하였다. 먼저 동적 다이어그램 별 관점에서 살펴보면 (그림 12)에 <표 5>의 평균값이 그래프로 표시되어 있는데, 이 그림의 다이어그램 별 그래프 모양의 변화에서도 알 수 있듯이 각기 다른 테스트 케이스에 대한 클래스 별 뮤테이션 점수는 큰 차이가 없다. 즉 세 가지 경우가 거의 유사한 결과를 보여 준다고 할 수 있다. 전체 평균적으로도 최대(30.91%)와 최소(24.85%)의 차이가 5% 정도에 그친다. 그러므로 동적 다이어그램 별로 얻은 각기 다른 테스트 케이스의 효율 비교는 크게 차이가 나지 않음을 확인할 수 있다.

다이어그램 별 점수는 큰 차이가 없지만 이에 비해 클래스 별로는 큰 점수 차이를 확인 할 수 있다. MyBank, Customer, Account, BankSystem 클래스 순으로 각 클래스마다 뮤테이션 점수가 약 10% 정도의 차이가 난다. 이는 각 클래스 별 역할과 밀접한 연관을 가진다. 가장 낮은 뮤테이션 점수를 보여주는 BankSystem 클래스의 경우 시뮬레이션 프로그램 상에서 가장 중요한 제어 부분을 주로 담당한다. 하지만 인출 동작과 연관된 부분은 BankSystem 전체 코드 상에서 일부만 차지하기 때문에 상대적으로 가장 낮은 점수를 기록하고 있다. 그 다음으로 높은 점수를 보여 주는

〈표 5〉 뮤테이션 테스팅 결과

클래스 이름	뮤턴트 타입	테스트 케이스 별 뮤테이션 점수(%)		
		순서	상태	액티비티
Account	객체지향 뮤턴트	26.67	20	33.33
	전통적 뮤턴트	17.6	9.33	20
	평균	22.13	14.67	26.67
BankSystem	객체지향 뮤턴트	12.5	12.5	12.5
	전통적 뮤턴트	14.67	12.8	11.32
	평균	13.58	12.65	11.91
Customer	객체지향 뮤턴트	28.57	25.71	28.57
	전통적 뮤턴트	40	30	50
	평균	34.29	27.86	39.29
MyBank	객체지향 뮤턴트	40	40	40
	전통적 뮤턴트	50.94	48.44	51.56
	평균	45.47	44.22	45.78
전체 평균		28.87	24.85	30.91



(그림 12) 각 테스트 케이스 별 평균 뮤테이션 점수를 나타낸 그래프

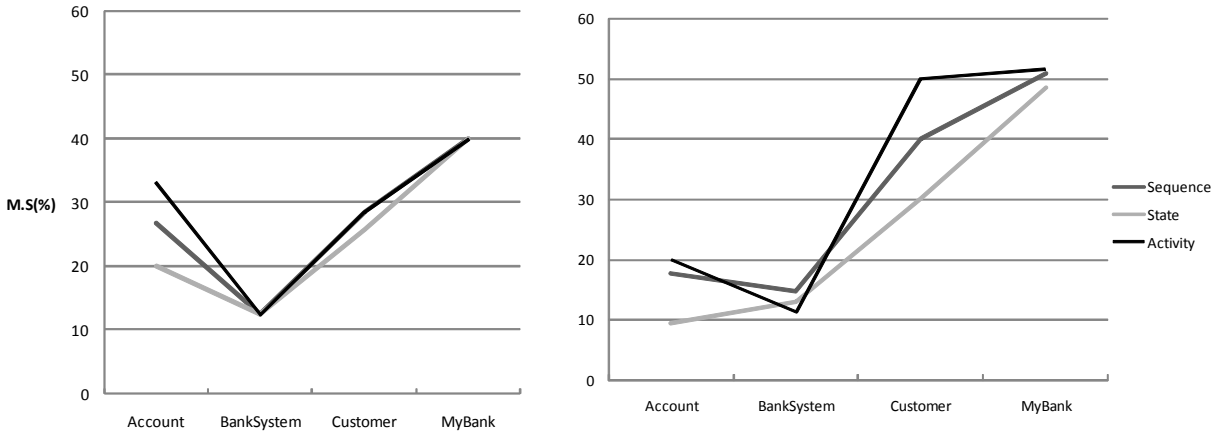
Customer, Account 클래스는 시스템의 엔티티(entity) 클래스에 해당된다. 대부분의 뮤턴트는 클래스 지역 변수 또는 지역 변수와 연관된 메소드의 변경 발생에 의한 것들이다. 즉 지역 변수나 메소드가 인출 동작과 연관 있는 경우의 뮤턴트들은 대부분 검출되었는데 Customer, Account는 기본적인 인출 동작과 연관성이 높기 때문에 BankSystem 보다는 높은 뮤테이션 점수를 보여 준다. 이와 같은 결과에도 불구하고 클래스 자체에 대한 효율적인 테스트를 위해선 클래스 다이어그램에 기초한 정적 분석 등의 방법이 더 낳은 효율을 기대할 수 있다[17]. 가장 높은 점수를 보여 주는 MyBank 클래스는 사용자 정보 관리가 주 기능이다. 인출 동작과는 직접적인 연관성이 떨어지나 인출 동작에 필요한 기본 정보 관리 기능이 주를 이루고 있기 때문에 작은 변화에도 상대적으로 민감하게 반응한다.

마지막으로 뮤턴트 타입 별 점수 비교는 절차적 프로그램 뮤턴트의 경우가 객체 지향 뮤턴트에 비해 평균적으로 약간 높지만 거의 비슷하게 나왔다(class: 26.7%, traditional: 29.7%). 이런 점은 (그림 13)의 두 개 그래프의 모양이 거의 유사하다는 것을 통해서도 확인 할 수 있다.

뮤턴트 타입 별 뮤테이션 점수 결과를 살펴보면 각 클래스 별 생성된 뮤턴트 수와는 다소 낮은 연관성을 가짐을 알 수 있다. 실제 뮤테이션 테스팅 실행을 통해 검출된 뮤턴트의 위치는 절차적 프로그램 뮤턴트와 객체지향 뮤턴트가 일치하는 경우가 많았기 때문이다. 즉, 객체지향 뮤턴트에 비해 상대적으로 절차적 프로그램 뮤턴트의 수가 훨씬 많지만 각 클래스에 대한 객체지향과 전통적 뮤턴트 두 타입 별 뮤테이션 점수는 경우에 따라 차이는 있지만 대부분 거의 유사했기 때문에 비슷한 평균 점수를 보여준다고 할 수 있다.

앞의 두 가지 관점에서 결과를 살펴본 내용을 정리하면 순서, 상태, 그리고 액티비티 다이어그램에 기초한 테스트 케이스들이 주로 메소드 호출 및 파라미터 변화에 민감하게 영향을 받지만 그 차이는 크지 않다는 것으로 요약할 수 있다. 이런 결과는 UML을 이용하진 않았으나 서로 다른 세 가지 방식을 통해 얻은 테스트 케이스를 뮤테이션 테스팅을 통해 각각의 효율을 비교한 [13]에서도 유사한 결과를 확인할 수 있다.

이번 연구로 생성된 실험 데이터는 객체 지향 프로그램의 효율적인 테스트를 위한 힌트를 준다. 우선 테스트 효율의 높고 낮음은 프로그램 내에 테스트 케이스의 실행과 연관된 뮤테이션 연산자의 적용 여부와 관련이 있다는 결과를 확인했으므로 프로그램의 어떤 부분을 점검하는지를 염두하고 이에 적합한 테스트 케이스 생성이 필요하다는 점을 들 수 있다. 그리고 객체 지향 언어로 구현된 프로그램이라도 실행 순서가 중요한 부분에는 절차적 프로그램에 적합한 테스팅 방식을 고려해야 좀 더 높은 프로그램 효율을 달성할 수 있다는 점이다. 마지막으로 클래스의 특징에 따라 동적 다이어그램에 바탕을 둔 명세 기반 테스팅이 적합한지 클래스 다이어그램에 기반한 정적 테스팅이 적합한지를 판단하고 그에 맞는 테스팅 실행이 필요하다는 점을 들 수 있다.



(a) class_mutants type (b) traditional_mutants type

(그림 13) 뮤턴트 타입 별 뮤테이션 점수를 나타낸 그래프

6. 결 론

본 논문에서는 각기 다른 동적 UML 다이어그램으로부터 서로 다른 테스트 케이스를 추출하고 각각의 효율을 비교하기 위해 뮤테이션 테스팅을 적용했다. 특히 뮤테이션 테스팅은 절차적 프로그램에 적합한 뮤턴트 연산자와 객체지향에 적합한 뮤턴트 연산자를 타입 별로 구분해서 뮤턴트를 생성하고 이를 이용해 진행되었다. 이에 대한 결과는 서로 다른 다이어그램에서 얻은 테스트 케이스에 대한 효율의 차이는 그다지 크지 않다는 점이다. 게다가 뮤턴트 연산자 타입 별로도 상호간 큰 효율 차이를 보여 주지 못했다. 그러나 클래스 별 효율 차이가 비교적 크게 발생했는데 이는 테스트 케이스에 대한 부분 보다는 해당 클래스의 어떤 부분에 뮤테이션 연산자가 적용되었는지가 더 큰 이유로 작용했다. 결국 어떤 다이어그램으로부터 테스트 케이스를 얻느냐 보다는 어떤 관점에서 어떤 대상을 테스트 할 것인지 계획하고 이에 적합한 테스트 케이스를 생성하고 테스트를 실행하는 것이 더 좋은 테스트 효율을 보여 줄 수 있다는 점을 이번 논문의 실험을 통해 확인할 수 있었다.

이 논문의 실험에서 프로그램의 사용자 인터페이스에 대한 부분은 누락되었다. 왜냐하면 사용자 주관의 다양한 동적 실행이 이루어지기 때문에, 이와 관련된 테스트에 대한 다양한 접근이 필요하다고 판단했기 때문이다. 향후에는 이런 부분을 고려한 테스트 케이스의 효율 평가 부분도 접근할 필요성이 있다. 이외에도 효율 높은 테스트 케이스를 선택하기 위한 다양한 연구들이 추가로 필요하다.

참 고 문 헌

[1] B. H. Smith and L. Williams, “An Empirical Evaluation ofthe MuJava Mutation Operators,” in Testing: Academic and Industrial Conference Practice and Research Techniques, pp.193-202, 2007.

[2] Y. Ma, Y. Kwon and J. Offutt, “Inter-Class Mutation Operators for Java”, in Proceedings. 13th International Symposium of Software Reliability Engineering, pp.352-363, 2002.

[3] Y. Ma, M. Harrold, Y. Kwon, “Evaluation of Mutation Testing for Object-Oriented Programs”, in Proceedings of the 28th international conference on Software engineering, pp.869-872, 2006.

[4] M. Chen, X. Qiu and X. Li, “Automatic test case generation for UML activity diagrams”, in Proceedings of the 2006 international workshop on Automation of software test, Shanghai, China, May, 23-23, 2006.

[5] H. C. Kim, S. W. Kang, J. M. Baik and I. Y. Ko, “Test Cases Generation from UML Activity Diagrams”, in Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, pp.556-561, 2007.

[6] S. Kansomkeat and W. Rivepiboon, “Automated-generating test case using UML statechart diagrams, Proceedings of the 2003 annual research conference of the South African institute of computer scientists and information technologists on Enablement through technology”, pp.296-300, September, 17-19, 2003.

[7] P. Samuel, R. Mall and S. Sahoo, “UML Sequence Diagram Based Testing Using Slicing”, in INDICON, 2005 Annual IEEE, pp.176-178, 2005.

[8] Y. S. Ma and J. Offut, “Description of Class Mutation Operators for Java,” http://ise.gmu.edu/~ofut/mujava/mutopsClass.pdf, accessed 7/07/2008.

[9] A. Abdurazik, J. Offutt, and A. Baldini. “A controlled experimental evaluation of test cases generated from UML diagrams”, Technical report, Information and Software Engineering Department, George Mason University, May 2004.

[10] “MuClipse”, http://muclipse.sourceforge.net/index.php, accessed 7/07/2008.

[11] M. Sarma, D. Kundu, R. Mall, “Automatic Test Case Generation from UML Sequence Diagrams”, in Proceedings of the 15th International Conference on Advanced Computing and Communications, pp.60-67, 2007.

[12] Bao-Lin Li, Zhi-shu Li, Li Qing, Yan-Hong Chen, “Test Case automate Generation from UML Sequence diagram and OCL expression”, in Proceedings of the 2007 International Conference on Computational Intelligence and Security, pp. 1048-1052, 2007.

[13] S. W. Kim, J. A. Clark and J. A. Mcdermid, “Investigating the Effectiveness of Object-Oriented Testing Strategies with the Mutation Method”, Investigating the Effectiveness of Object-Oriented Testing Strategies with the Mutation Method, pp.207-225, 2001.

[14] D. Kung, N. Suchak, J. Gao, P. Hsia, Y. Toyoshima and C. Chen, “On Object State Testing”, in Proceedings Computer Software and Applications Conference, pp.222-227, 1994.

[15] R. K. Doong and P. G. Frankl, “The ASTOOT approach to testing object-oriented programs”, ACM Transactions on Software Engineering and Methodology (TOSEM), Vol.3, No.2, pp.101-130, April, 1994.

[16] B. Baudry B. F. Fleurey, J. Jezequel and Y. Le Traon, “Genes and Bacteria for Automatic Test Cases Optimization in the .NET environment”, Software Reliability Engineering, 2002. ISSRE 2002. Proceedings. 13th, pp.195-206, 2002.

[17] T. Yi, F. Wu and C. Gan, “A comparison of metrics for UML class diagrams”, ACM SIGSOFT Software Engineering Notes, Vol.29, No.5, September, 2004.

[18] R. B. France, “A Problem-Oriented Analysis of Basic UML Static Requirements Modeling Concepts” In Proc. of OOPSLA’ 99, pp.57-69, 1999.

[19] A. S. Evans, “Reasoning with UML class diagrams” In Proceedings of the Workshop on Industrial Strength Formal Methods (WIFT’98), IEEE Press, 1998.
[20] C. Nebut, F. Fleurey, Y. Le Traon and J. Jezequel, “Automatic test generation: a use case driven approach”, IEEE Transactions on Software Engineering, Volume 32, Issue 3, pp.140-155, March, 2006.
[21] A. Hartman and K. Nagin, “The AGEDIS tools for model based testing”, In International Symposium on Software Testing and Analysis, pp.129-132, July, 2004.
[22] “COTE: Context and Problem Statement”, http://www.irisa.fr/cote/, accessed 18/09/2008.
[23] E. Gery, D. Harrel and E. Palachi, “Rhapsody: A complete life-cycle model-based development system”, In Proceedings of the Third International Conference on Integrated Formal Methods, pp.1-10, 2002.

이 혁 수
e-mail : hslee4@humaxdigital.com
1999년 동국대학교 전자공학과(학사)
2009년 동국대학교 컴퓨터공학과(석사)
2009년~현 재 휴맥스 품질경영부문 SQE팀 대리
관심분야 : 소프트웨어 테스트, 객체지향 설계, 임베디드 소프트웨어

최 은 만
e-mail : emchoi@dgu.ac.kr
1982년 동국대학교 전산학과(학사)
1985년 한국과학기술원 전산학과(공학석사)
1993년 일리노이 공대 전산학과(공학박사)
1985년~1988년 한국표준과학연구원 연구원
1988년~1989년 데이콤 주임연구원
2002년 카네기멜론대학 소프트웨어공학 과정 연수
2000년, 2007년 콜로라도 주립대 전산학과 방문교수
1993년~현 재 동국대학교 IT학부 컴퓨터공학전공 교수
관심분야 : 객체지향 설계, 소프트웨어 테스트, 프로세스와 메트릭, Program Comprehension, AOP