DOI: 10.3745/KIPSTD.2008.15-D.4.495

AOP를 이용하여 진화된 프로그램의 회귀테스트 기법

이 미 진[†] · 최 은 만[††]

요 약

관점지향 프로그래밍(AOP)은 횡단 관심사까지 모듈화 하여 소프트웨어의 모듈화를 높여주는 새로운 프로그래밍 패러다임이다. 이를 이용하면 레거시 시스템에 손대지 않고 소프트웨어를 확장시킬 수 있다. 관점지향 프로그래밍 자체 혹은 레거시 시스템만의 테스트 기법은 많이 있으나 확장된 프로그램의 테스트 기법에 대해선 많은 연구가 진행되지 않고 있다. 이 논문에서는 관점지향 프로그래밍을 이용하여 소프트웨어를 확장한 경우의 테스트 기법에 대해 관점지향 프로그래밍의 결함 모델에 맞춰 제시한다. 우선 AOP의 반사기능의 객체를 이용하여 교차점 패턴의 부정확한 강도 및 부정확한 애스펙트의 우선순위를 테스트하고, 증명 규칙을 이용하여 기대하는 사후 조건 성립의 실패에 대해 테스트하였다. 또한 set() 교차점을 이용하여 불변 조건 보존의 실패에 대해 테스트하고, 제어흐름 그래프를 이용하여 제어 의존의 부정확한 변형에 대해 확인하는 방법을 제시한다. 실증을 위하여 셋탑박스의 채널 관리 시스템을 구현하여 제시한 각각의 테스트 기법들에 대해 실험하였다.

키워드 : 관점 지향 프로그래밍, 회귀테스트, 소프트웨어 진화

Regression Testing of Software Evolution by AOP

Lee, Mi Jin[†] · Choi, Eun Man[††]

ABSTRACT

Aspect Oriented Programming(AOP) is a relatively new programming paradigm and has properties that other programming paradigms don't have. This new programming paradigm provides new modularization of software systems by cross-cutting concerns. In this paper, we propose a regression test method for program evolution by AOP. By using JoinPoint, we can catch a pointcut-name which makes it possible to test the incorrect pointcut strength fault and the incorrect aspect precedence fault. Through extending proof rules to aspect, we can recognize failures to establish expected postconditions faults. We can also trace variables using set() and get() pointcut and test failures to preserve state invariant fault. Using control flow graph, we can test incorrect changes in control dependencies faults. In order to show the correctness of our proposed method, channel management system is implemented and tested by using proposed methods.

Key Words : Aspect-Oriented Programming, Regression Test, Software Evolution

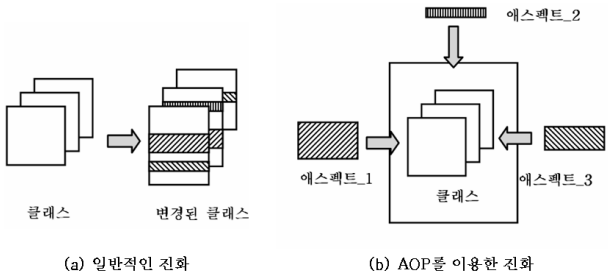
1. 서 론

빠르고 효율적인 소프트웨어 개발을 위해 프로그래밍 기법은 점점 모듈화 되는 방향으로 발전되었다. 현재 많이 사용되고 있는 객체지향 프로그래밍 패러다임은 핵심 관심사의 모듈화를 지원해준다. 이는 핵심 기능을 하나의 모듈로 구현하여 다른 기능을 가진 모듈과 분리함으로써 프로그램을 효율적으로 작성하고 관심사 별로 재사용이 가능하도록 도와준다. 그러나 객체지향 프로그래밍 패러다임만으로는 완벽한 관심사의 분리가 어렵다. 예를 들어 객체지향 프로그래밍에서 소매 애플리케이션에서 구매와 재고관리를 하는 핵심 관심사는 모듈화 시킬 수 있지만, 시스템 전체에 필요한 로깅, 인증 등의 횡단 관심사의 경우 모듈화 시킬 수 없다[6].

Aspect Oriented Programming(AOP)[7]은 이러한 문제를 해결하기 위해 고안된 프로그래밍 패러다임이다. AOP는 여러 핵심 관심사에 걸쳐 나타나는 횡단 관심사를 애스펙트(Aspect)라는 모듈화 단위로 지원하고 있다. 이를 통해 핵심 관심사뿐만 아니라 횡단 관심사까지 모듈화 하여 프로그램의 재사용을 높여준다. 또한 AOP는 별도로 정의된 관심사들이 필요에 의해 언제나 동적으로 플러그인(plug-in) 될 수 있게 해주는 웨이빙(weaving) 메커니즘을 가지고 있다. 이러한 장점으로 인해 AOP를 적용하면 프로그램의 보안과 같은 특정 관점의 테스트를 애스펙트로 정의하여 동적으로 테스트 할 수 있다. 또한 AOP는 일반적인 모듈의 단위 테스트나 컴포넌트의 빌트인 테스트(built-in test) 등에도 적용할 수 있다[1, 2, 10].

한편 횡단 관심사를 독립적으로 처리할 수 있는 AOP의 웨이빙 메커니즘을 잘 이용하면 레거시 시스템을 수정을 가

† 준 회 원 : 동국대학교 대학원 컴퓨터공학과(공학석사)
†† 정 회 원 : 동국대학교 컴퓨터공학과 교수
논문접수 : 2008년 4월 1일
수 정 일 : 1차 2008년 5월 14일
심사완료 : 2008년 5월 26일



(그림 1) AOP를 이용한 진화

하지 않고 확장시키거나 진화시킬 수 있다. (그림 1)에 보는 바와 같이 순수한 확장이나 진화는 소프트웨어를 구성하는 각 요소 안을 수정하거나 추가하지만, AOP는 레거시 부분에 수정을 가하지 않고 독립된 애스펙트를 정의하여 웨이빙에 의해 시스템을 구성할 수 있다. 레거시 시스템과 별도로 독립적으로 개발할 수 있고 별도의 모듈로 정의할 수 있는 애스펙트의 장점을 이용하면 동적인 웨이빙도 가능하다. 따라서 최근에는 실행 중인 시스템의 확장이나 테스트에 대한 연구 사례가 발표되고 있다[3].

이와 같이 AOP의 장점을 이용해 프로그램을 확장한 경우, 확장한 프로그램을 어떻게 테스트해야 할까? (그림 1)의 (a)와 같은 일반적인 확장의 경우 실행경로를 분석하여 추가 또는 변경된 부분이 있는 곳은 다시 새로운 테스트케이스를 만들어 시험하고, 또한 변경이 없더라도 영향 받는 부분을 찾아 이를 다시 테스트 할 수 있다. 이러한 경우의 테스트 방법은 이미 많은 연구가 진행되어 있다. 그러나 (b)와 같이 애스펙트 형태로 확장될 때 레거시 시스템의 테스트 방법에 대한 연구는 아직 미비하다. 이 논문에서는 이처럼 레거시 시스템에 애스펙트 형태로 확장될 때 테스트해야 하는 테스트 범위와 그에 맞는 테스트 방법을 제시하고 있다.

논문의 진행은 다음과 같다. 2장에서는 논문을 위한 배경지식에 해당하는 AOP와 회귀테스트에 대해 간단히 설명하고 있다. 3장에서는 논문의 핵심 주제인 테스트 범위 및 테스트 방법에 대해 소개한다. 각각의 결함 타입에 맞는 테스트 방법을 간단한 예를 통해 보여주고 있다. 4장에서는 3장에서 제시한 테스트 방법을 간단한 시스템을 통해 적용시켜 실험하였다. 5장은 테스트 결과 및 결론을 기술한다.

2. AOP을 이용한 확장과 회귀테스트

관점지향 프로그래밍은 횡단 관심사를 모듈화 하기 위하여 객체지향 프로그래밍과 프로시저 프로그래밍의 기초 위에 이것들의 개념과 구성물을 확장한 것이다. 관점지향 프로그래밍에서도 핵심 관심사는 기존의 선택된 기본 방법론으로 구현한다. 예를 들어, 객체지향 프로그래밍이 기본이라면 핵심 관심사는 클래스로 구현된다. 그러나 횡단 관심사는 관점지향 프로그래밍으로 구현된다.

횡단 관심사를 관리하는데 객체지향 프로그래밍과 관점지향 프로그래밍의 근본적인 차이점은 관점지향 프로그래밍에서는 각 관심사를 구현하는 모듈이 자신에게 적용되는 횡단 관심사를 전혀 모른다는 사실이다. 예를 들어, 비즈니스 로직 모듈은 자신의 메서드가 로깅이 되고 있는지 또는 사용자 인증을 하고 있는지 전혀 모르게 된다. 따라서 각 관심사가 독립적으로 발전해 나갈 수 있으며 이러한 이유로 AOP를 이용한 확장 방법이 제안되었고 많이 사용하고 있다[1, 2, 4]

유지보수 단계에서 오류를 발견하고, 환경을 변화시키고, 기능을 추가하는 등 여러 가지 변경 요인에 의해 원래의 프로그램에 수정을 가하게 된다. 이미 테스트를 마친 프로그램에 수정, 변경을 가한 후 의도하지 않았던 오동작이나 새로운 오류를 일으키는지 확인하기 위한 테스트를 회귀테스트라 하며 여기에는 두 가지 요소가 있다. 첫 번째는 오류를 발견하고 고치고 다시 원래 문제를 일으켰던 것을 테스트 해 보는 것이며 두 번째 요소는 수정된 부분이 다른 부분에 영향을 주지 않는다는 것을 확인하는 것이다. 이 논문에서 말하는 회귀테스트의 의미는 후자의 것으로, 이러한 의미의 테스트는 수정 작업이 성공적으로 이루어졌는지 확인하는 것이 아니라 수정 후 프로그램의 통합이 제대로 이루어졌는지 확인하는 작업이다. AOP로 확장하는 경우, 확장하기 위해 추가한 애스펙트의 테스트는 그렇게 큰 의미를 가지지 않는다. 그보다는 AOP로 확장 이후 확장을 위해 추가된 애스펙트들이 원래의 레거시 시스템과 웨이빙 되어 레거시 시스템에 의도하지 않은 오류가 생성되는지 확인해야 한다.

3. AOP 결함 모델을 기반으로 한 테스트 기법

AOP는 결합점에 교차점과 충고로 프로그램이 확장된다. 이러한 AOP의 특성으로 인해 새로운 결함이 나타나게 된다. 이와 같은 결함에 대해 잘 정의해 놓은 것을 결함 모델이라 한다. 이러한 결함 모델은 프로그래머가 범하기 쉬운 결함에 대해 인지하고 피할 수 있도록 도와준다. 또한 프로그래머가 결함을 피하지 못한 경우라도 결함 모델은 테스터가 결함을 쉽게 발견하고 찾을 수 방법을 제시해준다. 이 논문에서도 AOP를 이용해 진화된 프로그램의 회귀테스트 방법을 제시하기 위해 AOP 결함 모델을 이용하였다.

AOP의 결함 모델은 [5, 11, 16]에서와 같이 현재 여러 정의가 나와 있지만 가장 현실적으로 타당한 [5]에서 제시한 <표 1>과 같은 결함 모델을 따르기로 한다.

교차점과 충고는 각각 그 자체만으로는 의미가 없다. 충고에 교차점을 작성하여 그 교차점에서 선택된 결합점에 충고가 적용되는 것이다. 다시 말하면, 충고는 교차점에서 선택한 모든 결합점에서 실행된다. 이때, 교차점이 올바른 결합점을 선택하고 있지 않을 경우 [14]에서와 같이 잘못된 교차점 패턴 강도를 갖게 된다. 이러한 교차점 패턴의 부정확한 강도 결함을 테스트하기 위해, [8, 9]에선 상태모델을 이용하여 교차점의 패턴 강도 결함을 테스트하고 있다. 그러나 모든 상태 경로에 대해 직접 상태모델을 작성해야 하기 때문에 시간이 많이 소요될 뿐만 아니라 작성하는 과정에서 또 다른 오류가 생성될 수 있다. 이 논문에서는 정확하고

〈표 1〉 결함 모델과 테스트 방법

결함의 종류	설명	알려진 테스트 방법	이 논문에서 제안된 방법
교차점 패턴의 부정확한 강도	결합점 선택을 위한 패턴이 너무 강하거나 약하면 발생하는 오류	- 상태모델의 이용[8, 9]	- 반사기능 객체를 이용한 테스트
부정확한 애스펙트 우선순위	애스펙트의 충고들이 동일한 결합점에 적용되는 경우 우선순위를 프로그램에 정확히 작성하지 않으면 우선순위가 틀리게 적용되는 오류		- 반사기능 객체를 이용한 테스트
기대하는 사후 조건 성립의 실패	횡단 메커니즘에 의해 충고가 삽입된 메서드의 사후 조건은 장담할 수 없게 됨		- 증명 규칙을 이용
불변 조건 보존의 실패	횡단 메커니즘으로 인하여 Invariant가 깨지는 오류	- 상태모델 적용[11, 12] - 데이터흐름 그래프 이용[3]	- Set 교차점 이용
제어 의존의 부정확한 변형	대체충고에 의한 제어흐름의 변형 오류	- 제어흐름 그래프를 이용[17, 18]	- 제어흐름 그래프를 이용

```
1    Class1 {
       …
7      method(int x){
       …
       }
19     method(3);
       …
38   }
```

(a) Class1의 소스코드 위치

```
Aspect1{
  pointcut p1():
    execution(* Class1.method(int);
  around():p1(){
    print("교차점 이름 - " +
      thisJoinPoint.getSourceLocation());
  …
}
```

(b) AspectJ 반사기능 객체의 사용

⇒ Result: 교차점 이름 - Class1.java:19

(그림 2) AspectJ 반사기능의 객체 사용

효율적인 테스트를 위해서 AOP에서 지원하는 반사기능의 객체를 사용하였다.

교차점 패턴 강도를 테스트하기 위해 현재 작성된 교차점이 어떤 결합점들을 선택하고 있는지 알아야 한다. 특정 교차점이 선택해야 하는 결합점들을 미리 작성해본 후, 그 교차점이 실제로 어떤 결합점들을 선택하고 있는지 비교할 수 있다면 교차점 패턴 강도 결함을 확인할 수 있다. (그림 2)는 AspectJ의 반사기능의 객체인 thisJoinPoint의 사용을 보여주고 있다. (그림 2)의 (b)에서 p1()이라는 교차점은 Class1 클래스의 method(int) 메서드의 호출을 결합점으로 가지고 있다. 이때 p1()이 적용되는 충고 내에 thisJoinPoint 반사객체의 getSourceLocation()을 작성하여 확인하면 p1-Class1.java:19와 같은 결과를 확인할 수 있다. 이와 같은 방식으로 p1() 교차점이 실행될 때의 결합점을 찾아 p1()의 교차점명세와 비교해 보면 부정확한 애스펙트 패턴 강도 결함을 찾아볼 수 있다.

결합점에 두 개 이상의 애스펙트가 적용되고 있는지, 그 지점에서 실제 애스펙트가 어떤 순서로 실행되고 있는지 밝혀내기 위한 방법으로 앞서 교차점 패턴 강도 결함을 밝혀내기 위해 사용했던, AOP의 반사기능의 객체를 통해 가져온 데이터를 이용하고 있다. 특정 충고가 수행될 때, 수행되는 충고에서 적용되고 있는 교차점의 이름과 충고가 수행되는 결합점의 위치정보를 충고가 수행되는 순서로 수집한다. 이를 결합점의 위치정보를 기준으로 하여 정렬하면 하나의 결합점에서 적용되는 교차점의 이름이 실행되는 순서대로 나열된다. 이 데이터를 이용하면 특정 결합점에 적용되는 애스펙트 및 애스펙트의 우선순위까지도 한 번에 확인할 수 있다.

```
a(int x, int y)
    sum = x + y;
    print(sum);
```

```
before(int sum) : call(* *print(..)) && args(sum)
        sum = -sum;
```

(그림 3) a() 메서드와 애스펙트

```
{true} : pre-condition
  a(int x, int y) // {x = input1 + input2}
      sum = x + y;
      print(sum); // {output = sum}
{output = input1 + input2} : postcondition
```

(그림 4) a() 메서드와 사후 조건

각 클래스 혹은 메서드는 메서드가 끝날 때 만족되어야 하는 사후 조건(post-condition)을 가지고 있다. 레거시 시스템에서 사후 조건이 만족되었다 하더라도 애스펙트의 횡단 메커니즘에 의해 충고가 삽입된 메서드의 사후 조건은 장담할 수 없게 된다. (그림 3)의 a() 메서드와 메서드에 삽입된 애스펙트를 보면, a() 메서드는 두 개의 변수를 받아 그 두 개의 변수의 합을 구한 후 출력해 주는 메서드이다. 이 메서드의 사후 조건이 입력받은 두 수를 더한 값을 출력해 주는 것이라 하면, a() 메서드 자체만으로는 사후 조건이 만족되지만 그 충고의 삽입으로 인해 print() 메서드가 호출될 때 sum 값을 입력받아 -sum 값을 되돌려 주기 때문에 결국 -sum이 출력되어 사후 조건이 지켜지지 않게 된다.

사후 조건 성립에 대한 결함을 테스트하기 위하여 (그림 4)에 a() 메서드와 같이 증명 규칙을 적용한다. 증명 규칙은 소스 코드를 수학적으로 증명해 나가는 것이다.

애스펙트는 이전, 대체, 이후 충고로 결합점에 삽입된다. 다시 말하면 특정 결합점의 이전이나 이후에 혹은 대체하여 충고에 해당하는 코드가 삽입되는 것이다. 따라서 충고의 종류에 따른 위치에 충고의 내용을 삽입하여 증명 규칙을 적용하면 애스펙트까지 확장한 사후 조건의 테스트가 될 수 있다. (그림 5)의 (a)는 애스펙트의 before() 충고이고 (b)는 a() 메서드 상에서 충고의 내용 삽입이 나타나 있다. 이렇게 메서드에 애스펙트를 직접 삽입하여 증명 규칙을 적용하면 (그림 6)과 같이 된다. 현재의 a() 메서드의 사후 조건은 {output = input1 + input2}이지만, 애스펙트가 삽입된 회색바탕의 코드에 의해 사후 조건에 맞지 않음을 확인할 수 있다.

```
before(int sum) : call(* *.a(..)) && args(sum)
    sum = -sum;
```

(a) 애스펙트의 before() 충고

```
a(int x, int y)
        sum = x + y;
        sum = -sum;
        print(sum);
```

(b) 애스펙트 코드 삽입

(그림 5) 애스펙트 삽입과 레거시 프로그램에 삽입된 코드

{true} : pre-condition
a(int x, int y) {x = input1, y = input2}
sum = x + y; {sum_{old} = input1 + input2}
sum = -sum; {sum_{new} = $-sum_{old}$ = -(input1 + input2)}
print(sum); // {sum_{new} = output}
{output = input1 + input2} : postcondition

(그림 6) 애스펙트 확장된 프로그램에 증명 규칙 적용

```
pointcut trace(object variable) : set(object class.variable) && args(variable);
```

(그림 7) set() 교차점과 args() 매개변수 교차점

불변 조건 보존의 실패 결함을 테스트하기 위해 [8, 9]에서와 같이 전체 프로그램에 모두 상태모델을 적용하기는 어렵다. 그것은 상당히 복잡할 뿐만 아니라 실제프로그램의 실행에는 생각지 못한 변수가 있어 값이 어떻게 변할지 예측하기 어렵기 때문이다. 또 [3]의 데이터흐름 그래프를 통해 데이터흐름을 확인하는 방법은 복잡할 뿐 아니라 아직 실현되지 못하였다. 따라서 이 논문에서는 간단하고 실현 가능한 방법으로 set() 교차점을 이용하여 프로그램이 실제 실행되는 동안의 불변 조건을 검사할 수 있는 방법을 제시하였다.

먼저 불변 조건 보존의 실패 결함을 테스트하기 위해 set() 교차점을 적용하여 인스트루멘트를 작성하였다. set() 교차점 필드 쓰기 교차점으로, 지정한 필드에 쓰기참조가 일어난 경우를 포착한다. (그림 7)은 필드 쓰기 교차점으로, 단순히 쓰기 참조가 일어나는 점만 포착하는 것이 아니라, args() 교차점을 통해 새로 쓰여진 필드의 값을 넘겨받아 보여줄 수 있다.

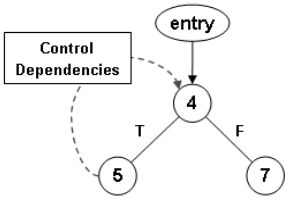
AOP의 충고 중 대체 충고는 결합점에 해당하는 내용 자체를 대체해서 수행되는 것이다. 따라서 이러한 대체 충고를 이용하면 결합점에 새로운 코드가 삽입되어 명령문 사이의 의존 관계를 대신하는 새로운 분기를 만들 수도 있고, 새로운 데이터도 삽입시킬 수 있어 메서드의 제어흐름을 상당히 변경시킬 수 있다.

(그림 8)은 Class1 클래스와 그 내부의 method1() 메서드에 대한 제어흐름 그래프이다. 여기에 (그림 9)의 (a)와 같은 애스펙트를 이용하여 시스템을 진화시킨 경우 (그림 9)의 (b)에서 보는 바와 같이 제어 의존의 변화를 확인할 수 있다. 이와 같은 제어 의존도의 변화가 예상치 못하게 생겨날 경우 찾아내기 어려운 결함이 될 것이다.

제어 의존을 확인하기 위해서는 우선 제어흐름 그래프를 그려봐야 한다. [17, 18]역시 제어 의존의 부정확한 변형을 테스트하기 위해 상태모델을 기반으로 한 애스펙트 흐름그

```
1  Class1 {
2     ...
3     method1(int l) {
4        if(i%4 == 0)
5           f1(i);
6        else
7           f2(i); }
8  }
```

(a) Class1 클래스

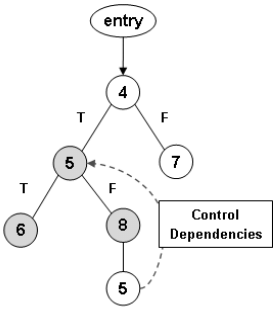


(b) method1() 메서드의 제어 의존도

(그림 8) method1()의 제어흐름 그래프

```
1  Aspect1 {
2     pointcut p1(int i) : call(* Class1.f1(..))
                        && args(i);
3
4     around(int i):p1(i){
5        if(i%2 == 0)
6           print("... ");
7        else
8           proceed(i) ;   }
9  }
```

(a) Aspect1 애스펙트



(b) 애스펙트 삽입 후 제어 의존도

(그림 9) 애스펙트 추가된 제어흐름 그래프

래프를 작성하였는데 이 논문에선 간단한 소스코드를 이용하여 제어흐름 그래프를 애스펙트로 확장하여 작성하고 있다. 애스펙트 확장 후의 제어흐름의 변화를 보기 위해 우선 (그림 8)과 같이 애스펙트가 삽입될 결합점에 해당하는 레거시 시스템의 제어흐름 그래프를 작성해야 한다. 그 후 삽입될 애스펙트의 제어흐름 그래프를 작성하여 (그림 9)에서 보이는 바와 같이 레거시 시스템의 제어흐름 그래프에 애스펙트 제어흐름 그래프를 삽입한다. 그 후 다시 제어 의존도를 살펴보면 애스펙트 추가 후의 제어 의존도를 확인해 볼 수 있다.

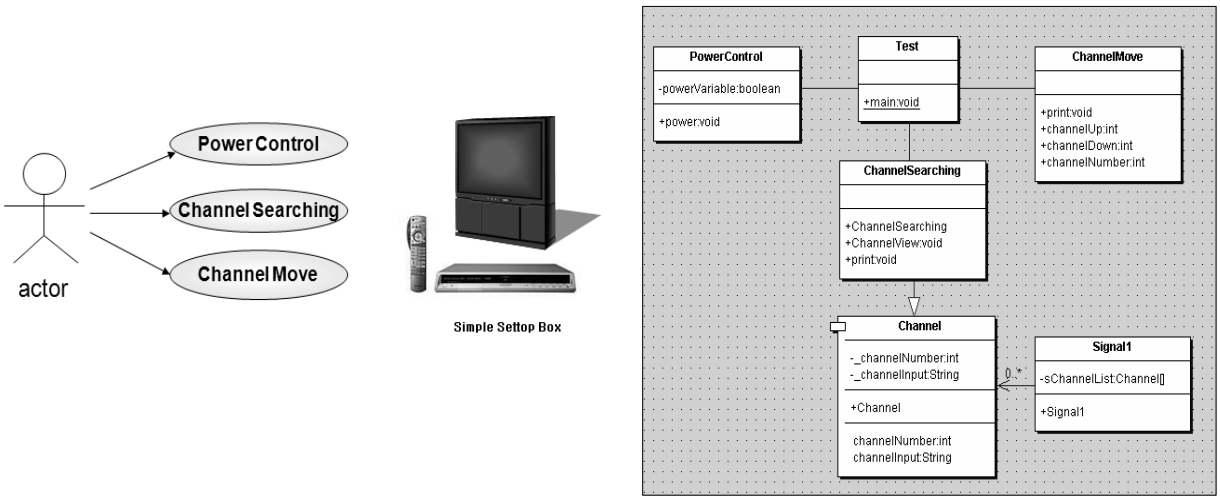
4. AOP를 이용하여 진화된 프로그램의 회귀테스트 실험

이 장에서는 실험을 통해 AOP를 이용하여 진화된 프로그램의 회귀테스트 방법을 보인다. 실험을 위해 간단한 셋탑박스 채널관리시스템을 작성하였다. Java 언어를 이용하여 레거시 시스템을 작성하고 AspectJ를 이용하여 시스템을 확장하였다. 이 논문에서 제시하고 있는 회귀테스트 기법의 현실성을 입증하기 위하여 결함을 주입하고, 테스트를 통해 결함을 확인한다.

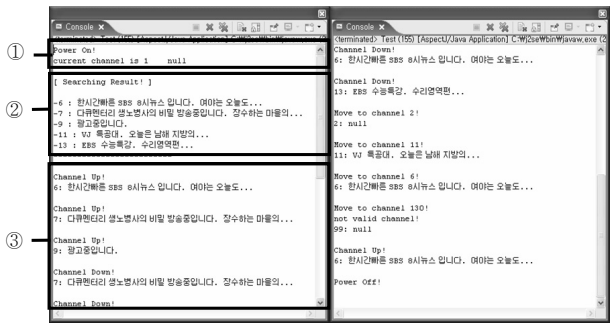
4.1 단계 1: 레거시 시스템 구현

관점지향 프로그래밍을 이용한 확장을 테스트하기 위해 우선 레거시 시스템을 설계하였다.

간단한 셋탑박스 관리 시스템은 (그림 10)과 같이 전원관리와 채널 검색, 채널 이동의 기능을 가지고 있다. 전원관리 기능은 사용자가 전원을 켜고 끌 수 있도록 해준다. 채널 검색 기능을 사용하면 시그널로부터 채널을 가져와 채



(그림 10) 레거시 시스템의 유즈케이스 다이어그램 및 클래스 다이어그램



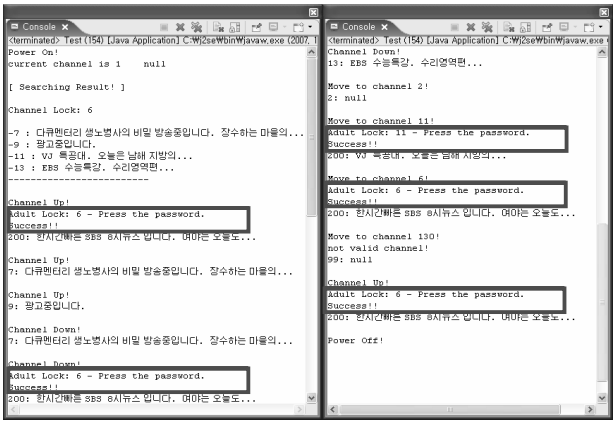
(그림 11) 레거시 시스템의 실행 결과

널을 검색하고, 시그널로부터 채널을 가져오기 위해 채널 검색 기능을 수행하면 현재 시그널에 존재하는 채널 목록을 가져올 수 있다. 이렇게 가져온 채널 목록 안에서 채널의 이동이 가능한데, 채널을 위로, 아래로 그리고 채널 번호로 이동하는 세 가지 기능을 제공한다.

(그림 10)에 나타난 레거시 시스템의 클래스 다이어그램과 같이 Test 클래스에서 PowerControl 클래스의 PowerOn() 메서드를 호출하면 (그림 11)의 ①과 같이 전원이 켜지고 현재 채널이 나타나게 된다. ChannelSearch 클래스를 수행하면 Signal 클래스로부터 신호를 읽어와 채널 목록을 만들고 만든 채널 목록을 print() 메서드를 이용해 (그림 11)의 ②처럼 출력해 준다. ChannelMove의 Channel Up(), ChannelDown(), ChannelNumber() 메서드를 호출하면 각각 채널을 위로, 아래로, 번호에 맞게 수정한 후에 ChannelMove클래스 내의 print() 메서드를 이용해 (그림 11)의 ③과 같이 화면에 채널 번호와 데이터를 출력해 준다.

4.2 단계 2: AOP 확장

Java언어로 작성한 레거시 시스템에 관점지향 프로그래밍 기법을 이용해 시스템을 확장하였다. 확장된 기능 중 AdultLock 기능은 성인채널을 잠그는 역할을 하고, ChannelLock 기능은 유료채널을 잠그는 역할을 한다. AdultLockAspect에선 성인채널 목록을 만들어 성인채널 목록에 있는 채널로 이동한 경우 암호를 입력받아 암호가 맞는 경우 화면을 보여준다. ChannelLockAspect는 유료채널 목록을 만들어 유료채널일 경우 유료메시지가 나타나고 해당 채널을 볼 수 없게 한다. AdultLock과 ChannelLock이 동시에 걸려있는 채널의 경우 항상 ChannelLock이 AdultLock보다 우선해야 한다. 채널 중 6, 11번에 AdultLock 기능을 적용하였고, 6번에 Channel Lock 기능을 적용하였다. (그림 12)는 애스펙트를 확장해서 프로그램을 실행한 경우의 실행결과이다.



(그림 12) AOP를 이용하여 확장된 프로그램의 수행결과

4.3 단계 3: 결함의 발견 및 회귀테스트 기법 적용

앞서 Java언어를 기반으로 한 간단한 셋탑박스 채널 관리 시스템을 구현하고, 레거시 시스템에 AspectJ를 이용해 기능을 확장시켰다. 이처럼 AOP를 이용해 프로그램을 확장한 경우, AOP만의 특징으로 인해 기존의 프로그래밍 메커니즘과는 다른 새로운 결함이 생겨나기 때문에 AOP결함모델에 맞는 회귀테스트가 진행되어야 한다. 이 장에서는 앞서 작성한 간단한 셋탑박스 채널 관리 시스템에서 나타나는 결함을 AOP 결함 모델을 기반으로 하여 인위로 만들어 보고, 3장에 제시한 테스트 기법들을 적용하여 결함을 찾을 수 있음을 보인다.

4.3.1 AOP 반사기능의 객체를 이용한 교차점 패턴 강도 및 애스펙트 우선순위 테스트

실험을 통해 발견되는 교차점 패턴 강도 및 애스펙트 우선순위 결함을 보이고 테스트방법을 적용하여 테스트하였다. 실험방법 및 결과는 다음과 같다.

4.3.1.1 교차점 강도 결함의 발견

(그림 13)은 AdultLockAspect 애스펙트의 일부소스이다. adultChannel() 교차점에서 선택되어야 하는 결합점은 <표 2>에 명시된 바와 같이 ChannelMove 클래스의 print() 메서드가 호출될 때 이다. (그림 13)의 adultChannel() 교차점의 패턴을 보면 ChannelMove 클래스의 print() 메서드가 호출될 때가 결합점으로 취해지고 있는데, 현재시스템의 ChannelMove 클래스의 print() 메서드는 하나밖에 존재하지 않기 때문에 adultChannel() 교차점은 올바른 결합점을 취하고 있다.

channelLockAspect 애스펙트의 lockedChannel() 교차점 역시 <표 2>의 교차점 명세에 나타난 바와 같이 adultLockAspect 애스펙트의 adultChannel() 교차점과 동일한 교차점을 선택하고 있어야 한다. 그러나 lockedChannel() 교차점의 필드 서명을 보면 (그림 13)의 회색바탕의 코드 내용과 같이 call(* *.print())로 작성되어 있다. 만약 현재 시스템에서 print() 메서드가 하나만 존재한다면 현재의 필드 서명은 명세에 맞는 결합점을 선택하고 있을 것이다. 그러나 안타깝게도 현재의 시스템에는 ChannelMove 클래스뿐만 아니라 (그림 15)에 작성되어 있는 ChannelSearching 클래스 또한 동일한 print() 메서드를 가지고 있기 때문에 약한 교차점 강도로 인한 결함이 발생하게 된다.

4.3.1.2 애스펙트 우선순위 결함의 발견

(그림 13)의 AdultLockAspect 애스펙트와 ChannelLockAspect 애스펙트는 모두 (그림 14)의 회색 바탕의 코드인 print() 메서드의 호출을 결합점으로 갖는다. 또 두 애스펙트 모두 결합점에 around() 충고를 적용하고 있다. 따라서 ChannelMove 클래스의 print() 메서드가 호출될 때 AdultLockAspect 애스펙트와 ChannelLockAspect 애스펙트가 모두 적용되어야 한다. 이처럼 동일한 결합점에 하나 이상의 애스펙트가 적용될 때 애스펙트의 우선순위가 정해지게 되는데 따로 프로그램 상에서 애스펙트 우선순위를 명시해 주지 않는 경우 둘 중 어떤 애스펙트가 먼저 실행될지 알 수 없기 때문에 애스펙트 우선순위의 결함이 나타나게 된다.

현재의 시스템에선 4.2에서 명시된 바와 같이 AdultLockAspect 애스펙트보다 ChannelLockAsepct 애스펙트가 항상 먼저 실행되어야 한다. 그러나 (그림 15)의 확장된 프로그램의 수행결과를 보면 ChannelLockAspect 애스펙트가 먼저 실행되는 것을 볼 수 있다.

4.3.1.3 테스트 구현

앞서 발견한 교차점 패턴 강도와 애스펙트 우선순위 결함을 해결하기 위해 충고가 실행될 때의 교차점 정보와 결합점 정보를 수집해야 한다. 3장에서 제시한 바와 같이 AOP의 반사기능의 객체 중 AspectJ에서 지원하고 있는 thisJoinPoint를 이용해 테스트 프로그램을 작성하였다.

확장된 ChannelLockAspect와 AdultLockAspect 애스펙트의 충고 내부에서 현재 교차점의 이름과 thisJoinPoint.getKind() 객체를 이용한 결합점의 종류, thisJoinPoint.getSourceLocation()

```
adultLockAspect : pointcut adultChannel(int nextChannel):call(* ChannelMove.print(..))
channelLockAspect : pointcut lockedChannel(int nextChannel):call(* *.print(..));
```

(그림 13) adultLockAspect와 channelLockAspect의 교차점패턴

〈표 2〉 교차점 명세

교차점 이름	선택되어야 하는 결합점	교차점 이름	선택되어야 하는 결합점
lockedChannel	ChannelMove.java:25	adultLockList	ChannelMove.java:25
	ChannelMove.java:46		ChannelMove.java:46
	ChannelMove.java:56		ChannelMove.java:56

```
class ChannelMove
        print(String data, int nextChannel);
        channelUp(String[] _channelList, int _currentChannel)
                print(_channelList[i], i);  // ChannelMove.java:25
        channelDown(String[] _channelList, int _currentChannel)
                print(_channelList[i], i);  // ChannelMove.java:46
        channelNumber(String[] _channelList, int _currentChannel)
                print(_channelList[i], i);  // ChannelMove.java:56
```

(그림 14) ChannelMove 클래스의 구조

```
class ChannelSearching{
        print(String list, int i);
        ChannelSearching(String[] channelList);
        ChannelView(String[] channelList)
                print(ChannelList[i], i);  // ChannelSearching.java:30  }
```

(그림 15) ChannelSearching 클래스의 구조

PointcutName	JoinpointKind	JoinpointLocation
lockedChannel	method-call	ChannelSearching.java:30
lockedChannel	method-call	ChannelSearching.java:30
lockedChannel	method-call	ChannelSearching.java:30
lockedChannel	method-call	ChannelSearching.java:30
lockedChannel	method-call	ChannelSearching.java:30
adultLockList	method-call	ChannelMove.java:25
lockedChannel	method-call	ChannelMove.java:25
adultLockList	method-call	ChannelMove.java:25
lockedChannel	method-call	ChannelMove.java:25
adultLockList	method-call	ChannelMove.java:25
lockedChannel	method-call	ChannelMove.java:25
adultLockList	method-call	ChannelMove.java:46

lockedChannel	method-call	ChannelMove.java:46
adultLockList	method-call	ChannelMove.java:46
lockedChannel	method-call	ChannelMove.java:46
adultLockList	method-call	ChannelMove.java:46
lockedChannel	method-call	ChannelMove.java:46
adultLockList	method-call	ChannelMove.java:56
lockedChannel	method-call	ChannelMove.java:56
adultLockList	method-call	ChannelMove.java:56
lockedChannel	method-call	ChannelMove.java:56
adultLockList	method-call	ChannelMove.java:56
lockedChannel	method-call	ChannelMove.java:56
adultLockList	method-call	ChannelMove.java:25
lockedChannel	method-call	ChannelMove.java:25

(그림 16) 수집한 교차점과 결합점 정보

PointcutName	JoinpointKind	JoinpointLocation
adultLockList	method-call	ChannelMove.java:25
adultLockList	method-call	ChannelMove.java:46
adultLockList	method-call	ChannelMove.java:56
lockedChannel	method-call	ChannelMove.java:25
lockedChannel	method-call	ChannelMove.java:46
lockedChannel	method-call	ChannelMove.java:56
lockedChannel	method-call	ChannelSearching.java:30

(a) 교차점 이름을 중심으로 정렬

JoinpointLocation	JoinpointKind	PointcutName
ChannelMove.java:25	method-call	adultLockList
ChannelMove.java:25	method-call	lockedChannel
ChannelMove.java:46	method-call	adultLockList
ChannelMove.java:46	method-call	lockedChannel
ChannelMove.java:56	method-call	adultLockList
ChannelMove.java:56	method-call	lockedChannel
ChannelSearching.java:30	method-call	lockedChannel

(b) 결합점을 중심으로 정렬

(그림 17) 교차점 패턴 강도 결함의 테스트 및 애스펙트 우선순위 결함의 테스트

객체를 이용한 결합점 소스코드의 위치를 가져와 데이터베이스에 저장하였다.

(그림 16)은 확장된 애스펙트의 충고부분에 교차점과 결합점 정보를 가져오는 소스를 추가한 후 수행한 결과이다. 그림에서 보이는 것처럼 충고가 실행되는 순서대로 교차점 이름과 결합점의 종류, 결합점의 소스코드 위치가 수집되어 있다. 이렇게 수집된 소스코드를 적절히 정렬하면 교차점패턴 강도결함 및 애스펙트 우선순위 결함을 테스트할 수 있다.

(그림 17)의 (a)테이블을 보면 교차점 이름을 중심으로 결합점들이 나열되어 있는 것을 확인할 수 있다. 이를 교차점 명세와 비교해 보면, <표 2>의 lockedChannel 애스펙트의 교차점은 ChannelSearching.java:30에 해당하는 결합점을 취하지 않기 때문에 이는 약한 강도의 교차점 강도 결함이 나타나고 있음을 볼 수 있다. 또 (그림 17)의 (b)와 같이 결합점을 중심으로 정렬하면 결합점별 교차점이 실행 순서대로 나타나게 된다. 이를 (그림 15)에 정의된 애스펙트 우선순위와 비교해 보면 애스펙트 우선순위결함을 테스트해 볼 수 있다. 현재 애스펙트 우선순위는 항상 channelLockAspect 애스펙트가 adultLockAspect 애스펙트보다 우선시 되어야 하는데 실험 결과에서 확인해 본 바로는 adultLockAspect 애스펙트가 우선되어 실행되고 있기 때문에 애스펙트 우선순위 결함이 발생하고 있음을 확인할 수 있다.

4.3.2 증명 규칙을 이용한 사후 조건 테스트

ChannelNumber() 메서드의 사후 조건은 {0 < nextChannel < 100} 이다. 이 조건이 만족되는지 알아보기 위해 증명 규칙을 작성하였다. (그림 18)은 애스펙트가 삽입된 channelNumber() 메서드의 코드이다. 회색 바탕의 소스들이 애스펙트를 삽입한 것이고, 굵은 글씨로 나타난 것이 각 부분에서의 조건들을 말한다. (그림 18)에서와 같이 사전 조건은 {input > 0}이고, 사

```
{input > 0} : pre-condition
channelNumber()
    if (_currentChannel < 100) // {input < 100}
        nextChannel  = _currentChannel // {nextChannel = _currentChannel}
        [_nextChannel = nextChannel]
        if(lock == true) // {input < 100}
            proceed(_nextChannel)
            [->nextChannel = _nextChannel]
        else
            System.out.println("not valid channel number.")
                proceed(299) {nextChannel = 299}
            [-> nextChannel = 299;]
        print(_channelList[nextChannel], nextChannel)
    else // {input >= 100}
        System.out.println("not valid channel!")
            nextChannel = 99; //{nextChannel = 99}
{ 0 <= nextChannel < 100 } // postcondition
```

(그림 18) 증명 규칙을 적용한 사후 조건 테스트

후 조건은 {0 < nextChannel < 100}이다. 사후 조건을 추적해 나가면 {input > 0} nextChannel < 100 or nextChannel = 299 or nextChannel = 99 {0 <= nextChannel < 99}가 된다. 이때, nextChannel = 299 는 사후 조건 0 <= nextChannel < 99 에 맞지 않기 때문에 사후 조건의 결함이 나타나는 것을 확인할 수 있다.

4.3.3 set() 교차점을 이용한 불변 조건 테스트

현재 시스템에서의 불변 조건중 하나는 nextChannel 이라는 전역변수의 값이 항상 0<=nextChannel<100을 유지해야 하는 것이다. 이를 확인하기 위하여 AOP의 set() 교차점을 이용해 애스펙트 테스트 인스트루멘트를 작성하였다.

(그림 19)는 불변 조건을 테스트하기 위한 테스트 인스트루멘트이다. set() 교차점에 args()를 이용하여 필드쓰기 참

```
aspect InvariantsTestAspec
    pointcut invariants(int nextChannel) : set(int *.nextChannel) && args(nextChannel)
  before(int nextChannel) : tracing(nextChannel)
      int invariantsValue = nextChannel;
      String location = thisJoinPoint.getSourceLocation().toString();
```

(그림 19) 불변 조건 테스트 인스트루멘트

value	location
0	ChannelMove.java:6
2	ChannelMove.java:63
2	ChannelMove.java:9
11	ChannelMove.java:63
299	ChannelMove.java:9
6	ChannelMove.java:63
299	ChannelMove.java:9
0	ChannelMove.java:68
6	ChannelMove.java:29
299	ChannelMove.java:9
0	

(그림 20) 불변 조건 테스트 인스트루멘트 실행 결과

조가 일어난 시점의 값을 가지고 오고 있다. 또한 반사기능 객체를 이용해 필드의 쓰기 참조가 일어난 지점의 위치도 알아본다. 이러한 정보를 데이터베이스에 넣으면 (그림 20)과 같다. 테이블 좌측의 value 부분이 nextChannel에 쓰기 참조가 일어났을 때의 값이고, location이 그 위치를 말한다. 현재의 시스템에선 299란 값이 있는 것으로 보아 불변 조건이 지켜지지 않고 있다.

4.3.4 제어흐름 그래프를 이용한 제어 의존성 테스트

레거시 시스템의 제어 의존성이 올바르다 하더라도 충고의 삽입으로 인해 제어 의존성이 변할 수 있다. 제어 의존성의 변화는 프로그램의 수행 결과상에서 쉽게 드러나지 않기 때문에 발견하기 까다로운 결함 중 하나이다. 따라서 AOP 확장 후 AOP가 적용된 제어흐름 그래프를 통해 제어 의존도를 확인해야 한다.

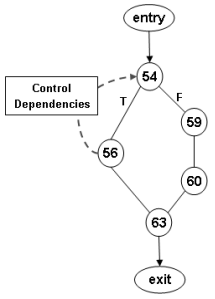
(그림 21)은 이 실험에서 주의 깊게 살펴보아야 할 코드 부분과 그 제어흐름 그래프이다.

(그림 21)의 소스코드를 분석해 보면 입력받은 채널이 100 미만일 경우 print() 메서드가 호출되어 입력받은 채널의 프로그램이 출력되고, 100 이상일 경우는 유효하지 않은 범위의 채널이기 때문에 유효하지 않은 채널이라는 메시지와 시그널을 찾을 수 없다는 메시지가 출력될 것이다. (그림 21)의 제어흐름 그래프에서 보면 제어 의존도가 분명히 드러나게 된다. 56번째 줄의 소스코드는 54번째 줄의 if문이 참인 경우만 수행되어 54번째 줄의 if문에 의존적이다. (그림 22)의 소스코드는 print() 메서드가 호출될 때 호출되는 channelLockAspect 애스펙트이다. channelLockAspect의 around() 충고를 보면 32번째 줄의 if문이 존재한다. 이로 인해 56번째 줄의 제어 의존은 애스펙트의 32번째 줄로 바뀐 것을 볼 수 있다. 이는 입력받은 채널의 값이 300이 넘는지를 확인하고 있다. 레거시 시스템에서의 채널은 100 이상이 되면 안 되지만, 애스펙트를 작성할 당시 충분히 레거시 시스템을 이해하지 못했기 때문에 나타난 결함으로 볼 수 있다.

5. 결론 및 향후과제

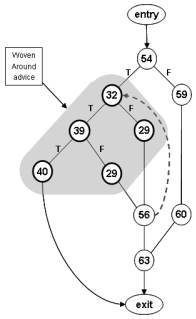
빠르게 변화하는 사용자의 요구에 맞추기 위해, 사용자가 원하는 기능을 빠른 시간에 추가하여 제품을 출시하기 위해, AOP는 기능 추가함에 있어 기존의 소프트웨어를 수정하지 않고 가능하다는 큰 이점을 가지고 있다. 이렇게 AOP를 이

```
51  channelNumber(String[] _channelList, int _currentChannel)
        ...
54      if (_currentChannel < 100)
56          print(_channelList[_currentChannel], _currentChannel);
57      else
59          System.out.println("not valid channel!");
60          _currentChannel = 0;

63      return _currentChannel
```



(그림 21) channelNumber() 메서드 channelNumber() 메서드의 제어흐름 그래프

```
13  around() : lockedChannel(nextChannel)

32      if (nextChannel < 300)
39          if(lock == true)
40              System.out.println("locked channel!");
42          else
43              proceed(_nextChannel);
45      else
48          proceed(299);
```



(그림 22) channlLockAspect 애스펙트의 충고 및 충고 삽입 이후의 제어흐름 그래프

용해 확장한 경우의 회귀테스트를 위해서 AOP의 특징을 잘 이해하고 그에 맞는 회귀테스트 방식을 적용해야 한다.

이 논문에서는 AOP를 이용해 확장된 프로그램의 회귀테스트를 위해서 AOP 결함 모델을 참고로 하여 실험을 설계하였다. 모두 네 가지의 테스트 방법을 제시하여 AOP 결함 모델의 다섯 가지 영역을 커버하고 있다. 첫 번째로 반사기능의 객체를 이용해 교차점과 결합점 정보를 수집하는 기법을 제시하였다. 교차점과 결합점의 정보를 수집하여 교차점 이름으로 수집한 정보를 묶으면 교차점 강도 패턴의 결함을 밝혀낼 수 있었고, 결합점을 중심으로 묶으면 애스펙트 우선순위의 결함을 테스트할 수 있었다. 두 번째 방법은 증명규칙을 이용하여 매개변수의 사후 조건을 테스트하는 것이다. 레거시 시스템에 삽입된 충고의 소스를 추가하여 애스펙트 확장된 프로그램을 증명 규칙을 적용하여 테스트 하였다. 세 번째로 set() 교차점을 이용한 불변 조건의 테스트 방법은 set() 교차점과 args()를 이용해 특정한 필드를 추적하여 변하는 값을 저장하는 테스트 인스트루멘트를 작성하는 것이다. 이렇게 가져온 값이 조건에 맞는지 확인하여 불변 조건을 테스트 할 수 있었다. 네 번째 방법은 제어흐름 그래프를 이용한 방법으로 레거시 시스템의 제어흐름 그래프에 애스펙트 제어흐름 그래프를 삽입하여 애스펙트까지 확장된 제어흐름 그래프를 통해 제어 의존성 결함을 확인해 보았다.

논문을 통해 AOP로 진화된 프로그램의 회귀테스트 방법에 대해 적용해 볼 수 있을 것이다. 향후 교차점 강도 패턴의 결함을 수정함에 있어 [15]를 적용하면 교차점 강도 패턴을 추천해 주는 방안도 연구해 볼 수 있을 것이다. [13]에서와 같이 큰 소프트웨어의 회귀테스트에 mock를 적용한 테스트에 대한 연구 역시 제시한 방법에 적용해 볼 수 있을 것이다. 또한 제시한 방법의 테스트 도구와 자동화된 테스트 방법에 대한 연구도 좋은 연구과제가 될 것이다.

참 고 문 헌

[1] G. Xu, Z. Yang, "A novel approach to unit testing: The Aspect-Oriented way," International Symposium on Future Software Technology 2004, Xian China, October, 2004.

[2] N. Belblida, et. al, "AOP extension for security testing of programs," *Proc. of IEEE Canadian Conference on Electrical and Computer Engineering*, Ottawa, May, 2006.

[3] J. Zhao, "Data-Flow-Based Unit Testing of Aspect-Oriented Programs," In *Proc. 27th Annual IEEE International Computer Software and Applications Conference(COMPSAC'2003)*, Dallas, Texas, December, 2003.

[4] J. Bruel, et al., "Using aspects to develop built-in tests for components," *4th Modeling with UML workshop of 2003 UML conference*, San Francisco, Oct., 2003,

[5] R. T. Alexander, J. M. Bieman, and A. A. Andrews. "Towards the systematic testing of aspect-oriented programs," Technical Report, CS-4-105 Department of Computer Science, Colorado State University, March, 2004.

[6] Ramnivas Laddad, "AspectJ in Action: Practical Aspect-Oriented Programming", Manning Publications, 2003.

[7] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold, "An Overview of AspectJ," In 15th European Conference on Object-Oriented Programming, Budapest, Hungry, 2001.

[8] D. Xu, W. Xu, and K. Nygard, "A State-Based Approach to Testing Aspect-Oriented Programs," *Technical Report, Department of Computer Science North Dakota State University*, September, 2004.

[9] D. Xu, "Test Generation from Aspect-Oriented State Models," Technical Report, NDSU-CS-TR-05-XU02, Sept., 2005.

[10] H. Song, Y. Yin, S. Zheng, "Dynamic aspects weaving in service composition," *Proc. of the 6thInternational Conference on Intelligent Systems Design and Applications(ISDA'06)*, IEEE, 2006.

[11] J. S. Bekken, Roger T. Alexander. "A Candidate Fault Model for AspectJ Pointcuts," *17th International Symposium on Software Reliability Engineering (ISSRE'06)*, 2006.

[12] J. Zhao and M. Rinard, "System Dependence Graph Construction for Aspect-Oriented Programs," MIT LCS Technical Report 891, March, 2003.

[13] Michael Mortensen and Sudipto Ghosh, James M. Bieman, "Testing During Refactoring Adding Aspects to Legacy Systems," *17th International Symposium on Software Reliability Engineering(ISSRE'06)*, 2006.

[14] Otavio Augusto Lazzarini Lemos and Cristina Videira Lopes "Testing Aspect-Oriented Programming Pointcut Descriptors," *International Symposium on Software Testing and Analysis Proceedings of the 2nd workshop on Testing aspect-oriented programs,* 2006.

[15] P. Anbalagan and T. Xie. "Efficient mutant generation for mutation testing of pointcuts in aspect-oriented programs," *In Proc. 2nd Workshop on Mutation Analysis*, November, 2006.

[16] Reza Meimandi Parizi, Abdul Azim Ghani, "A Survey on Aspect-Oriented Testing Approaches," 15th *International Conference on Computational Science and Applications*, IEEE, 2007

[17] W. Xu, D. Xu, V.Goel, and K. Nygard, "ASPECT FLOW GRAPH FOR TESTING ASPECT-ORIENTED PROGRAMS," *In Proceeding of the 8th IASTED International Conference on Software Engineering and Applications(SEA 2004)*, ACTA Press, August, 2004.

[18] W. Xu and D.Xu, "A model-based approach to test generation for aspect-oriented programs," *AOSD 2005 Workshop on Testing Aspect-Oriented programs*, Chicago, March, 2005.

이 미 진

e-mail : alexshel@naver.com
2006년 동국대학교 컴퓨터공학과(학사)
2008년 동국대학교 대학원 컴퓨터공학과 (공학석사)
2008년~현 재 NHN게임서비스 개발팀
관심분야 : AOP, 소프트웨어테스팅 등

최 은 만

e-mail : emchoi@dgu.ac.kr
1982년 동국대학교 전산학과(학사)
1985년 한국과학기술원 전산학과(공학석사)
1993년 일리노이 공대 전산학과(공학박사)
1985년~1988년 한국표준연구소 연구원
1988년~1989년 데이콤 주임연구원
1998~2004년 한국정보과학회 소프트웨어공학연구회 운영위원
2001~2003년 한국정보처리학회 학회지편집위원
2000, 2007년 콜로라도 주립대 전산학과 방문교수
2002년 카네기멜론대학 소프트웨어공학 과정 연수
1993년~현재 동국대학교 컴퓨터공학과 교수
관심분야 : 객체지향 설계, 소프트웨어 테스트, 프로세스와 메트릭, Program Comprehension