韓國 컴퓨터情報學會 論文誌
第13卷 第2號, 2008. 3.

2008-13-2-1-3

효과적인 오류 추적을 위한 수직적 시스템 시험 방법

서 광 익*, 최 은 만**

Vertical System Testing Method For Efficient Error Tracing

Seo Kwang Ik*, Choi Eun Man**

요 약

단위 시험은 모듈의 소스 코드를 면밀히 검토하면서 논리적 오류나 문장 오류 등이 있는지 분석하는 화이트박스 시험이 가능하다. 반면 시스템 수준의 기능 시험은 규모가 크기 때문에 시험 데이터를 입력한 후 출력된 결과가 예상 결과와 같은지 비교하는 블랙박스 시험이 주를 이룬다. 이러한 시스템 시험 단계에서 사용하는 블랙박스 시험은 오류를 발견하더라고 수정을 위해 소스 코드를 추적하기 어려운 문제점이 있다. 뿐만 아니라 시스템 시험 단계에 화이트박스 시험을 사용하는 것은 시험 대상의 추상 수준 달라 쉽지 않다. 이에 본 논문에서는 시스템의 기능처럼 높은 추상 수준을 시험 대상으로 하되 소스 코드 수준까지 화이트박스 스타일로 시험할 수 있는 현실적이고 통합된 시스템 수준의 수직적 시험에 대해 제안한다. 그리고 어떻게 수직적 시험을 적용하는지 UML 명세 모델에서 소스 코드까지 오류를 추적하는 방법을 사례를 통해 설명하고 더불어 오류 추적의 효과성을 보였다.

Abstract

In case of unit testing, White-box test can be used to closely check source code and to analyze logic and statement errors. On the other hand, in case of function testing of system level, Black-box test can be mainly used to compare actual and expected results by inputting test data because the scale of function is large. This Black-Box test in system testing level has problem in tracing errors in source code when we find errors. Moreover applying White-box test is not easy for system testing level because the abstract levels of test target are different. Therefore this paper suggests the vertical test method of a practical and integrated system level which can checks up to source code level using White-box test style although it aims to test the highly abstract level like a system function. In addition, the experiment explains how to apply the vertical test by displaying an example which traces from UML specification model to the source code and also shows efficiency of error trace.

▸ Keyword : 수직적 시험(vertical testing), 기능 시험(function test), 오류추적 가능성(error traceability), UML 시험(UML testing), 시스템 시험(system test), 통합 시험(integration test), 단위 시험(unit test)

• 제1저자 : 서광익 교신저자 : 최은만
• 접수일 : 2008. 1. 22, 심사일 : 2008. 2. 14, 심사완료일 : 2008. 2. 29.
*동국대학교 컴퓨터공학과 박사수료 **동국대학교 컴퓨터공학과 교수
※ 이 논문은 2007년도 동국대학교 연구년 지원에 의하여 이루어졌음

www.kci.go.kr

I. 서 론

소프트웨어 시스템을 개발하는 과정에 여러 가지 다른 레벨과 관점의 모델 및 구현물을 만들게 된다. 그리고 시스템이 결함이 없음을 보이기 위해서는 생성된 모델 및 구현물을 근거로 시험을 한다. 그 예로 UML(Unified Modeling Language)을 이용한 객체지향 개발 방법에서는 요구분석과 설계 작업이 끝나면 사용사례 다이어그램, 순서 다이어그램, 클래스 다이어그램을 산출한다. 그리고 구현 단계에서는 소스 코드를 생성한다. 마지막으로 시스템을 시험하기 위해서는 개발 단계에서 생성한 산출물을 기준으로 시스템의 기능과 동작을 검사한다. 하지만 각 산출물들은 추상 수준이 서로 다를 뿐만 아니라 표현하고자 하는 목적이 다르기 때문에 시험 방법이나 시험 유형이 달라진다. 사용사례 다이어그램을 이용하여 시스템을 시험한다면 사용사례는 하나의 기능 단위를 표현하기 때문에 시스템 수준인 블랙박스 스타일의 기능 시험이 사용될 것이다. 반면 순서 다이어그램이나 클래스 다이어그램을 이용하여 시험을 하면 인터페이스를 통한 모듈 간의 통신이 정확한지 판단하는 블랙박스 스타일의 통합 시험이 효과적이다. 구현 단계에서 산출된 소스 코드를 이용해서 구현한 모듈의 논리적 구조를 면밀히 검사한다면 이는 화이트박스 스타일의 단위 시험이 될 것이다. 이와 같이 산출물과 시험 기법은 밀접한 관계가 있고 산출물의 표현 수준에 따라 시험의 수준도 분리 된다. 하지만 이러한 분리는 단계적 테스트 작업의 유기적인 연관성을 저하시킨다. 실제로 테스트 각 단계와 담당하는 조직에 따라 시험 대상과 목표는 매우 다르다. 그리고 시스템을 통합하고 출시하기 전에 전체적인 시스템을 테스트하는 단계에서는 모델로부터 소스 코드까지 또는 소스 코드로부터 테스트 케이스 또는 모델까지 수직적 추적이 빈번하게 일어난다. 이러한 경우 오류들 사이에 적절한 추적 방법이 결여되어 있다면 기능 시험에서 발견한 오류를 수정하기 위해 소스 코드로 추적하는데 많은 시간과 노력이 필요하게 된다. 따라서 효율적인 오류 수정을 위해서는 각 시험 단계의 작업과 산출물의 유기적인 관계가 추적될 수 있어야 한다.

테스트 작업에 요구분석이나 설계 명세를 근거로 테스트 케이스를 작성하여 모델과 구현이 잘 들어맞는지 확인하는 방법을 많이 사용하고 있다[1]. 즉 시스템의 개발 초기에는 높은 추상성을 가진 사용자의 요구를 찾아내고 시스템을 더욱 깊이 이해해 나가면서 시스템의 구성요소의 골격을 설계한 후 이를 구체적인 프로그래밍 언어로 기술하여 만들어 나간다. 전통적인 테스트 방법은 각 추상 수준에 따라 별도의 목표와 테스트 케이스를 설정하고 각 단계별 테스트 작업에서 결함을 발견하여 단지 개발자들에게 던져 놓는다. 그리고 개발자는 시험 결과 자료에만 의지하여 결함을 해결한다. 하지만 보다 짧은 Time-to-market을 위하여 단계적 테스트 작업들은 서로 유기적으로 결합되어야 한다. 시스템 테스트 단계에서 결함을 발견하였을 때 릴리스를 얼마 앞둔 시점에 그저 블랙박스 테스트 결과를 개발 엔지니어에게 던져놓고 다음 테스트 사이클을 기다리는 것은 매우 위험한 일이다. 또한 개발 당사자들도 소프트웨어가 릴리스 될 시점에 발견된 오류는 매우 신중하여 다른 컴포넌트와의 연관성 및 수정 후의 파급효과를 잘 살펴야 한다. 그러나 단계적 테스트가 각 추상 수준 사이에 단절되어 있어 수직적으로 추적하기가 어렵다. 예를 들면 UML 모델에 근거하여 블랙박스 형태의 테스트를 하는 경우 오류 스팟을 찾아내기 위하여 해당되는 기능 슬라이스의 논리 구조나 알고리즘, 코드 슬라이스를 찾아내기는 쉽지 않다. 더구나 테스트 팀과 개발 팀의 역할이 극명히 나뉘어져 시스템 통합과 테스트 단계에 커뮤니케이션이 원활히 이루어지지 않는 경우 오류 스팟에 대한 추적은 더욱 어렵고 릴리스 시간에 쫓겨 기능을 잘라내는 대수술을 할 수밖에 없게 된다. 이를 위하여 서로 다른 추상 수준 간의 시험에 따라 분리되는 블랙박스 시험과 화이트박스 시험의 한계를 극복하면서 자연스럽게 두 유형의 시험이 연계된 방법이 필요하다.

시스템 수준의 테스트는 주로 명세를 기반으로 시스템의 기능 슬라이스 단위로 이루어진다. 오류를 될 수 있으면 조기에 발견하여 수정 비용을 줄이려고 설계 자체를 테스트 하려는 시도가 있어왔다. 즉 UML로 표현된 설계를 검증하고 테스트하는 기법[1, 2]이나 이를 위한 애니메이션[3] 등이다. 설계 검증이나 시험 방법은 설계 안에 있는 오류를 찾을 수 있다. 그러나 구현 후에 이루어지는 시스템 수준의 테스트는 명세를 기반으로 테스트하지만 구현된 원시코드와 연관을 시키지 않을 수 없다. 블랙박스 테스트 형태를 따라 시스템 단위의 테스트를 하지만 결함이 발생되면 결국 시스템 안으로 들어가 관련된 부분을 찾을 수밖에 없다. 이러한 시험의 단계는 시스템 시험과 통합 시험, 단위 시험의 순서이다. 이러한 순서가 분리되어 있고, 각 단계의 시험을 담당하는 팀마저 분리되어 있다면 많은 시간과 자원이 필요할 것이다. 하지만 기능 슬라이스 단위로 시험을 하되 시스템 수준의 시험과 통합 수준의 시험 그리고 단위 수준 시험을 자연스럽게 연계하면 굳이 각 팀과 업무를 분리하여 관리할 수고를 덜 수 있을 것이다. 또한 블랙박스 시험과 화이트 박스 시험의 경계가 사라져 반드시 두 스타일의 시험을 분리하지 않으면서 상호 보완적이고 각 시험 스타일이 목표하는 결과를 모두 취할 수 있을 것이다.

www.kci.go.kr

이 논문에서 제안한 방법은 시스템 수준의 기능 슬라이스를 블랙박스로 진행하다가 결함이 발견된다면 단위 수준까지 내려갈 수 있고 이벤트의 경로를 추적하여 오류 스팟을 찾을 수 있는 더욱 융통성 있는 시스템 테스트 방법이다. 2장에서는 시스템 테스트와 관련된 연구들에 대하여 고찰해보고 3장과 4장에서는 수직 추적의 개념과 절차를 알아본다. 그리고 5장에서 이 논문에서 제안한 과정으로 실제 테스트 실험한 사례 연구를 설명하고 6장에 결론을 정리하였다.

II. 일반적인 시스템 시험에 대한 고찰

2.1 명세 기반의 시스템 시험

일반적인 시스템 시험은 명세를 기반으로 전반적인 시스템의 기능 및 비기능 요구사항을 시험하는 것을 말한다. 시스템의 명세는 자연어로 기술된 경우보다는 UML 같은 모델 표현 방법을 이용하여 구현에 더욱 잘 매핑될 수 있다. 따라서 UML로 작성된 모델 기반의 시스템 테스트 방법이 많이 제안되었다.

Briand와 Labiche는 UML의 여러 다이어그램을 이용하여 좀 더 자세한 시스템 테스트 방법을 제한하고 있다[4]. 특히 사용사례 다이어그램을 이용하여 사용사례 사이의 의존관계를 파악하고 가능한 이벤트의 경로를 모두 파악하여 이를 구동시키는 테스트 사례를 찾아낸다. 이때 OCL로 기술된 클래스 및 메소드의 명세 정보를 이용한다. 또한 다른 연구로 Abdurazik과 Offutt은 적어도 보다 철저한 시험이 되기 위하여 협력 다이어그램에서 메시지의 경로가 적어도 한 번은 실행되는 방법을 제한하였다[5]. 또한 소프트웨어의 수정을 상태 다이어그램을 이용하여 표현하고 수정된 코드를 시험하기 위한 테스트 케이스를 생성하는 기법도 제안하였다[2]. Briand와 그 동료들은 위의 방법을 더욱 개선하여 메시지 호출과 이벤트, 액션의 다양한 타입들까지도 시험할 수 있는 방법으로 발전시키고 있다.

이제까지의 연구에서 접근하는 방법은 명세기반의 테스트 케이스를 찾되 커버리지를 높이는 방법이나 AI를 동원한 방법[6] 일변도이다. 시스템을 철저히 시험하려면 커버리지를 높이는 것과 함께 명세기반의 테스트 케이스를 찾고 이들을 이용하여 시험하였을 때 결함이 있는 부분의 구체적인 추적이 뒷받침되어야 한다. 즉 시스템이 릴리스 될 시점에 테스트 엔지니어와 개발자 사이에 분리된 참조 모델을 사용하여 서로의 의사소통을 해치는 일이 없이 시스템의 여러 추상 레벨을 오르내리며 추적할 수 있는 방법이 필요하다.

2.2 전통적 V 모델에서의 시험

전통적인 소프트웨어 개발 프로세스인 폭포형 모델에 검증과 테스트 작업을 강조한 V 모델이 있다. 그림 1의 왼쪽은 소프트웨어 개발 절차를 보여주고 있으며 오른쪽이 각 개발 단계에서 산출된 결과를 시험하고 확인하는 단계가 매칭되어 있다. 순수한 V 모델에서는 단계적 테스트가 이상적으로 이루어지고 각 단계에서 발견된 오류들이 다른 시험 작업과 상관이 없는 것처럼 나타나 있다. 그러나 실제 작업에서는 각 단계별 테스트가 순차적으로 이루어지다가 결함이나 수정이 이루어지면 그 수준의 스팟에서 다른 수준의 스팟으로 추적이 필요하고 다른 수준의 테스트가 필요하다.

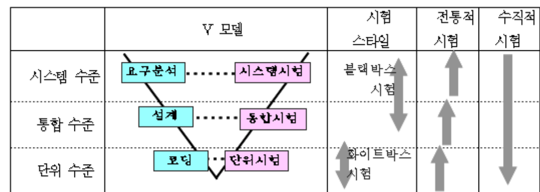

그림 1. 시험 수준별 상관관계
Fig 1. Testing Level Interrelation

예를 들어 시스템 수준의 시험을 하다가 결함을 발견하면 결함이 발견될 수 있는 계통, 즉 그 테스트에 의하여 실행된 부분을 찾아야 하고 그 스팟을 찾기 위하여 그 주위의 통합이 잘 되었는지 다시 시험해 볼 수 있다. 통합 시험 수준으로 내려가 자세히 시험한 후 의심 가는 부분이 있다면 각 모듈 단위의 시험을 더욱 자세한 화이트 박스 방법을 써서 점검할 수 있어야 한다.

이러한 철저한 테스트가 불가능 한 이유는 소프트웨어 규모가 커지면서 단위 시험이 철저히 분업화되고 분업화 된 팀 사이에 커뮤니케이션을 도와 줄 수 있는 모델이나 추적 가능성이 제공되지 않기 때문이다. 따라서 본 논문에서 제시하고 있는 수직적 시험이 가능하려면 모델과 시험 사례간의 완전한 추적 방법이 필요하다. 요구분석에서 설계 모델로, 모델에서 테스트 케이스를 통하여 구현 코드로 추적할 수 있다면 시스템 테스트 후에 어떤 부분에 결함이 있는지 그리고 어디를 고쳐야 하는지 알 수 있게 된다.

이 논문에서는 UML 중심으로 표현된 모델에서 최대한의 커버리지를 추구하기 위한 추적가능성을 제시한다. 명세 중심의 시스템 테스트의 여러 방법 중에서 어떤 것, 또는 어떤 조

www.kci.go.kr

합이 시스템의 검증을 철저하게 커버하는지 연구하였다[9]. 즉 사용사례, 협동 다이어그램, Object-Z, OCL(Object Constraint Language), 확장된 사용사례 등 다섯 가지 방법에 대하여 비교 실험하였다. 그 결과 확장 사용사례와 OCL를 병행하여 사용하는 것이 시스템 내부의 구현을 잘 드러내고 따라서 더욱 좋은 추적 링크를 만들 수 있다. 확장 사용사례는 시스템 내부에서 발생하는 논리적인 프로그램의 흐름을 기능단위로 시험하는 반면 OCL은 특정 시나리오에 관계없이 클래스 간의 관계 또는 속성과 메소드, 메소드 간의 인터페이스 등과 같은 관계들을 중심으로 커버한다. 다시 말하면 확장 사용사례는 시스템의 기능을 수직적으로 자른 슬라이스가 되며 OCL은 이들 슬라이스가 구현되어 조금씩 묻혀진 클래스나 메소드의 내용이 엮여지기 때문에 마치 날줄과 씨줄로 시험 대상이 되는 커버리지를 높인다.

III. 수직시험을 위한 기능 슬라이스

이 논문에서 재안하는 수직 추적에 의한 시스템 시험은 두 가지의 중요한 특징이 있다. 첫째는 단위 시험에서부터 통합 시험에 이르는 과정을 수시로 추적하면서 다시 할 수 있고 추상 수준을 넘나들 수 있다는 점이다. 또 다른 특징은 시스템을 수직적 기능 슬라이스로 분리하고 분리된 영역을 시험하되 문제가 있는 부분은 점차 원시코드 수준의 스캔을 하여 결함이 있는 부분을 집어 낸다는 관점에서 수직적 엄격한 시험이다.

소프트웨어 각 단위의 구현이 끝난 후 통합이 잘되면 QA 팀이 주도하는 테스트 단계로 진입하게 된다. 이 단계의 테스트는 명세를 근거로 사용자의 요구가 잘 반영되었는지 테스트한다. 이 단계에 화이트 박스와 같은 방법으로 소프트웨어 내부의 모든 코드를 분석하고 테스트하기란 쉽지 않다. 따라서 블랙박스 시험 방법을 이용하여 시스템이 사용자에게 제공해야 하는 서비스와 성능 위주의 테스트가 이루어진다.

사용사례는 어떻게 엔드 유저 또는 시스템 간의 상호 작용을 하는지 명확하게 정의하는데 사용된다. 즉, 시스템의 행위를 시스템 외부의 사용자 관점에서 모델링하고 규정한다. 반면 사용사례 슬라이스는 사용사례 인스턴스의 조각이라고 할 수 있다. 사용사례 하나에는 무수한 이벤트의 경로가 나오는데 이 중에 테스트하기 위하여 잘라낸 실행 경로를 말한다. 따라서 각 사용 사례 슬라이스는 디자인 모델로써의 사용 사례 실현에 대한 구체적인 관점을 보여줄 수 있다.

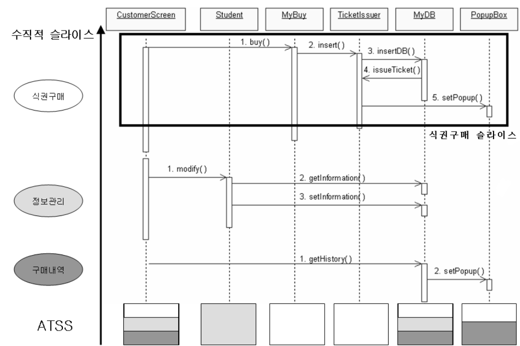

그림 2. ATSS의 기능 슬라이스
Fig 2 . Function Slice of ATSS

그림 2는 식권 자동 발매 시스템에 대한 사용사례와 해당 슬라이스를 보여주고 있다. 고객은 '식권구매'와 고객에 대한 '정보관리' 그리고 '구매내역'을 식권 자동 발매 시스템을 통해 확인한다. 따라서 각 기능에 해당하는 사용사례가 있고 각 사용사례의 실현을 위해 참여하고 있는 클래스와 메소드의 집합인 사용사례 슬라이스를 보여주고 있다. CustomerScreen, MyBuy, TicketIssuer, MyDB, PopupBox의 집합은 '식권구매'에 대한 슬라이스이다. 또한 '정보관리'에 대한 슬라이스는 CustomerScreen, Student, MyDB의 집합이고, '구매내역' 슬라이스는 CustomerScreen, MYDB, PopupBox의 집합니다.

시스템 테스트는 사용사례 단위의 기능 슬라이스를 기초로 분리하여 이루어진다. 최근 시스템의 명세는 UML로 표준화되어 가고 있으며 사용사례 다이어그램에 분리할 수 있는 기능 슬라이스가 잘 나타나 있다. 예들 들어 식권 자동 발매 시스템을 기능 단위로 쪼개본다면 그림 2에 표현된 것처럼 된다.

그림 2에 나타낸 것처럼 시스템은 여러 기능이 모여 이루어진다. 엔드유저에게 보이는 하나의 기능은 여러 개의 클래스가 협력하여 메시지를 호출함으로 구현된다. 예를 들어 식권구매 하는 기능은 서로 다른 클래스, CustomerScreen, Student, MyBuy, TicketIssuer 등이 협력하여 메시지를 정확히 주고받음으로써 그 기능을 제공할 수 있다.

시스템을 구현한 이후에 테스트하는 과정을 그림 2에 비추어 다시 생각해 보자. 먼저 각 단위 모듈을 구현하는 과정에서 그림 2의 맨 하단의 박스로 표현된 클래스들을 시험한다. 이 때 시험하는 것은 클래스 안의 메소드에 구현된 알고리즘의 각 경로들, 개별 메소드들의 호출 결과, 메소드 안의 불변 조건 등이다. 다음 통합 테스트는 구현된 클래스 및 모듈들을 등록하고 이를 통합하여 빌드하는 과정에 새로 통합하는 모듈들이 잘 인터페이스 되는지 시험한다. 시스템 단위의 시험 단

www.kci.go.kr

계에는 기능 슬라이스와 그 밖의 성능 및 비기능적 요구 관점에서 테스트한다. 즉 그림 2의 왼편에 있는 사용사례 슬라이스별로 테스트 케이스를 만들고 실제 이를 하단에 있는 클래스 구현에 주입하여 실행 결과를 검토하는 것이다.

여기에서 제일 큰 애로 사항이 왼쪽의 수직적 슬라이스와 맨 하단의 코드와의 매핑이 어려운 점이다. 즉 수직적 슬라이스를 시험하기 위한 블랙박스 시험과 코드를 시험하는 화이트박스 시험의 연계 방법이 제공되지 않으면 결함의 현상이나 모델 상의 위치 정도만 파악될 수 있다. 예를 들어 단위 시험에서 발견하지 못한 결함이 시스템 시험까지 흘러와 어떤 기능 슬라이스의 시험에서 발견되었다고 하자. 해당되는 스팟을 수직으로 내려가 찾아 다시 단위시험을 하든지 아니면 경로를 추적하여 결함의 위치를 찾고 싶을 것이다. 시스템 시험에서 발견된 결함 때문에 다시 단위 테스트 단계 또는 통합 시험을 다시 하는 것은 비효율적이다.

모델과 구현의 매핑이 어려운 것은 아니나 규모가 커지면 일일이 파악하는 것이 불편하고 특히 하나의 사용 사례 안에서도 여러 작은 단위의 테스트 사례가 나올 수 있어 테스트 슬라이스 단위의 매핑과 추적이 필요하다. 즉 그림 2에서 완성된 클래스 안에 여러 기능들이 복합적으로 내재되어 있는데 어떤 부분이 어떤 테스트 슬라이스에 의하여 시험되었는지 스캔할 수 있다면 테스트 및 유지보수 작업에 효과적일 것이다.

Ⅳ. 수직적 시스템 시험 절차

일반적인 시스템 테스트와 이 논문에서 제시하는 테스트 프로세스는 다음과 같은 큰 차이가 있다. 먼저 순수한 V 모델에서는 단위 테스트부터 통합 테스트, 시스템 테스트로 순차적으로 진행된다. 하지만 수직적 시스템 시험은 시스템 시험에서 다시 단위 시험으로 진행되는 방법을 제공한다. 수직적 시스템 테스트 단계에 들어가면 시스템 수준의 테스트 사례를 자세히 만들고 이를 이용하여 시험한 후 결함을 보일 때 결함을 보인 테스트 사례만을 더욱 세세한 메시지-메소드 경로를 만들어 시험한다. 수직적 시스템 테스트 프로세서의 개략적인 과정이 그림 3에 있다.

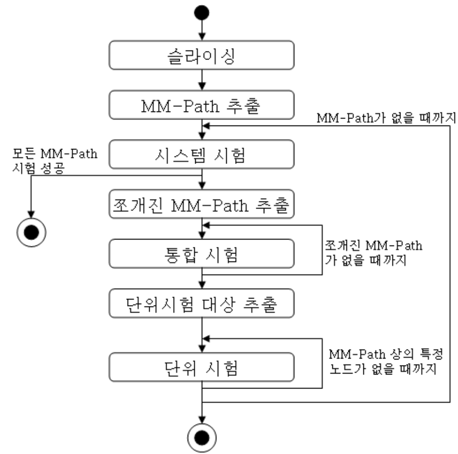

그림 3. 수직 시스템 시험 절차
Fig 3. Vertical System Test Process

먼저 기능 단위로 슬라이싱을 한 후 슬라이싱 된 기능에 대한 MM-Path 를 추출한다. 그리고 시스템 시험을 수행한

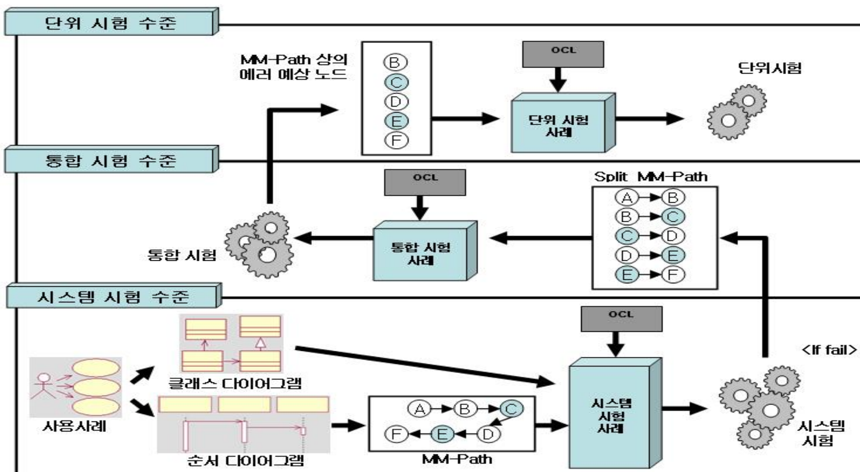

그림 4. 수준별 시험 절차와 필요한 정보
Fig 4. Test Process and Required Information in abstract level

www.kci.go.kr

다. 이 단계에서 모든 MM-Path에 오류가 없으면 시험 대상의 기능에 오류가 없다고 할 수 있다. 그러나 만약 오류가 발견되면 더 작은 단위로 이루어진 MM-Path를 분할해 가면서 통합 시험을 한다. 이 과정에서 오류가 있는 단일 노드를 찾는 것이다. 이러한 과정을 좀 더 자세히 보면 그림 4와 같다. 그림 4는 각 시험 수준 별로 시험 사례를 작성하는데 필요한 요소들과 오류 지점을 찾기까지의 작업들이 나와 있다.

융통성 있는 수직적 시험은 시스템 시험 수준의 기능 시험으로부터 시작해서 단위 시험으로 진행한다. 즉 잠재된 오류 지역을 탐색하면서 거꾸로 오류 발생 지점을 추적하는 것이다. 그림 4와 관련하여 구체적인 시험 절차를 4.1절에서부터 자세히 설명하였다.

4.1 시스템 시험

단계 1: 슬라이싱 된 클래스 그룹들의 메시지와 이벤트를 분석하여 메소드 메시지 경로를 추출한다.

기능 시험에 대한 시험 사례를 만들기 위해서 요구분석 및 설계 단계에서 명세한 사용 사례 단위로 클래스 다이어그램과 순서 다이어그램의 메소드 간의 호출 관계로 메소드 메시지 경로를 생성한다. 메소드 메시지 경로는 사용사례를 실현하기 위한 클래스의 메시지에 의해 연결된 연속된 메소드의 경로이다. 따라서 하나의 시험 사례를 구성하기 위해 메소드 메시지 경로의 클래스 중 처음으로 구동되는 클래스와 맨 마지막에 호출되는 클래스를 기준으로 기능 단위의 시험 대상을 작성한다. 기능 한 개는 여러 개의 MM-Path로 이루어질 수 있어 하나의 기능을 시험하기 위해서는 여러 개의 MM-Path를 모두 시험해야 한다. 그림 4에서는 수직적 시스템 시험을 설명하기 위해 예로 하나의 MM-Path를 (A, B, C, D, E, F)로 가정하였다.

단계 2: MM-Path 중 처음과 마지막으로 호출되는 메소드의 조건을 기술하고 있는 OCL로부터 시스템 수준의 시험 사례를 추출하고 시험을 수행한다.

일반적으로 요구분석 단계에서는 시나리오를 작성한다. 만약 요구분석 단계에서 작성된 시나리오가 없다면 시험 데이터를 추출하기 위한 현실적인 시험 시나리오를 작성하게 된다. 이들 시나리오를 기준으로 사용사례의 입출력 및 제약사항과 각 클래스에 대한 다중도 및 제약사항등을 OCL로 정의할 수 있다. 클래스에 대한 OCL 선후 제약조건(pre/post-condition)은 메소드가 호출되기 전이나 호출 된 후 클래스가 포함하고 있는 속성들의 값들을 정의할 때 사용된다. 따라서 시험 데이터를 입력하여 시험을 수행할 때의 클래스 속성과 실행한 후의 클래스 속성을 정의하고 있는 OCL은 시험 사례를 추출하는 자료가 된다. 이러한 OCL을 기준으로 MM-Path 중 처음과 마지막으로 호출되는 메소드와 관련 변수들을 대상으로 시험 데이터와 유효한 결과 값을 결정한다. 이 때 OCL의 선 제약 조건으로부터 가장 먼저 호출 된 메소드와 관련 변수에 대해 입력할 시험 자료를 생성한다. 그리고 시험을 수행 한 후 사용사례를 실현하고 있는 클래스 중 가장 나중에 호출된 클래스의 또는 기타 속성들이 후 제약 조건을 준수하고 있는지 시험한다. 이것은 메소드 메시지 경로의 맨 처음과 마지막 노드의 입력과 출력 데이터를 이용한 블랙박스 스타일의 시험이 된다. 만약 그림 2의 MyBuy.buy()가 실행될 때 입력해야 하는 값의 범위를 OCL로 정의한다면 그 값은 '식권구매'의 기능을 시험하기 위한 시험 자료 값을 사용할 수 있다. 또한 PopupBox.setPopup()의 반환해야 하는 값에 대해 정의하고 있다면 그 값은 '식권구매'의 실행 결과를 비교할 수 있는 예상결과로 사용할 수 있다.

4.2 통합 시험

단계 3: 호출과 피호출 관계인 쪼개진 메소드 메시지 경로를 시험한다.

만약 출력 값이 예상 결과와 다르면 메소드 메시지 경로 중 어떤 노드에 오류가 발생한 것이므로 오류 발생 가능 구역을 탐색해야 한다. 오류를 탐색하는 방법은 그림 4의 통합 시험 단계와 같이 두 개의 노드를 하나의 쪼개진 MM-Path로 구성하고 쪼개진 MM-Path를 대상으로 통합 시험을 한다. 그림 4의 시스템 시험 수준의 MM-Path에는 노드 (A, B, C, D, E, F)가 있다. 그리고 노두 C와 E에 오류가 있다고 가정하면 (A, B, C, D, E, F)를 대상으로 한 시스템 시험은 실패를 할 것이다. 이 때 (A, B), (B, C), (C, D), (D, E), (E, F)와 같이 호출 관계에 있는 노드를 하나의 짝을 만든다. 이것은 오류 발생 가능 노드를 포함하고 있는 쪼개진 메소드 메시지 경로가 될 것이고 통합 시험 수준에서는 이를 대상으로 통합 시험을 수행한다. 이 과정에서 노드 C와 E에 오류가 있기 때문에 쪼개진 MM-Path 중 (B, C)와 (C, D) 그리고 (D, E)와 (E, F)의 시험 결과는 실패일 것이다. 이 과정에서 호출 관계에 있는 노드 간의 인터페이스의 프로토콜이 잘 맞는지 시험할 수 있다. 그리고 동시에 노드 C와 E에 오류가 있다는 것을 찾아내고 다음 단계인 단위 시험으로 진입한다.

4.3 단위 시험

단계 4: 오류를 포함하고 있는 노드(메소드)의 연산 알고리즘 또는 관련 변수들을 시험한다.

단위 시험에서는 통합 시험에서 찾아낸 오류를 포함하고 있는 노드를 중심으로 시험을 한다. 때 단위 시험의 대상은

www.kci.go.kr

시스템의 종류와 크기에 따라 달라 하나의 함수나 클래스 또는 프로시져가 될 수 있지만 본 연구에서는 MM-Path를 이루고 있는 노드이기 때문에 클래스의 메소드가 그 대상이다. 따라서 단위 시험에서는 잠재 오류 노드(메소드)를 중심으로 시험한다. 여기에서는 메소드의 논리적인 연산과 연산에 참여하는 변수들을 검사하게 된다. 이러한 단위 시험의 연구 결과는 다양하다. 그 중 논리 구조를 검사하고 데이터를 분석하여 구조적인 결함이나 데이터 구조, 데이터 선언과 관련한 검사 방법이 있다. 이와 같이 통합 시험에서 걸러진 노드에 대해 철저한 시험을 하여 원시 코드 중 문제가 있는 코드 라인까지 제시하게 된다.

표 1. 수직적 시스템 시험 참여 모델
Table 1. Participating Model in Vertical System Test

작업 순서		시험 대상	참여 모델			
			사용사례 다이어그램	클래스다이어그램	순서 다이어 그램	OCL
1	기능 슬라이싱		○	○		
2	MM-Path 추출			○	○	
3	시스템 시험	MM-Path				○
4	쪼개진 MM-Path 추출			○	○	
5	통합 시험	쪼개진 MM-Path				○
6	단위시험 대상 추출			○		
7	단위 시험	MM-Path 상의 특정 노드				○

융통성 있는 시스템 시험을 수행하는 작업 순서에 따라 참여하는 모델을 표 1에 정리하였다. 기능 슬라이싱을 위해서는 사용사례 다이어그램과 클래스 다이어그램이 필요하다. 그리고 최초의 MM-Path를 구하기 위해서는 클래스 다이어그램 순서 다이어그램이 준비되어야 한다. 이와 같이 수직적 시스템 시험을 위해서는 V 모델에서 제시하고 있는 것과 같이 개발 단계의 요구분석이나 설계 작업의 산출물을 사용한다.

V. 사례 연구

5.1 수직적 시험 사례

'식권구매' 시험사례는 사용사례에 관한 제약사항과 메소드 메시지 경로에 참여하고 있는 클래스 속성들의 제약사항들로 구성할 수 있다. 그림 5는 이러한 제약사항들은 OCL로 표현하였다. 사용사례에 관한 OCL의 제약조건들은 기능 수준의 시험 기준이 된다. 그리고 클래스에 관한 OCL은 통합 수준의 시험 조건이 된다. 이들을 통해 시험 사례를 작성할 수 있다.

그림 2에서 사용사례와 클래스 다이어그램 및 순차 다이어그램을 사용하여 슬라이싱 하는 과정을 보였다. 그림 2와 같이 ATSS 기능은 크게 '식권구매' 기능과 구매자의 '정보관리' 그리고 '구매내역'을 조회할 수 있는 기능이 있다. 그 중 가장 핵심이 되는 기능이 '식권구매' 기능이기 때문에 학번으로 로그인하여 식권을 구매하는 '식권구매' 기능을 중심으로 시험 사례를 설계하고 실험한다. 표 2는 '식권구매' 기능 시험 사례의 한 예이다. 시험 데이터의 조건은 일단 유효한 로그인 데이터인지 확인하는 시험 데이터 변수 loginID가 있다. 그리고 여러 기능 중 '식권구매' 기능을 선택하면 JComboBox에서 TransCode가 값 2를 저장해야 한다. 또한 식권 구매가 가능한 기간이 당일로부터 이틀 뒤까지 총 3일이 유효하므로 ccDate가 0과 2 사이의 값만이 유효하다. 마지막으로 시험 결과는 PopupBox.setPopup()이 반환해야 하는 결과 값을 검사하여 평가한다. 그림 5는 이와 같은 기능의 제약 조건을

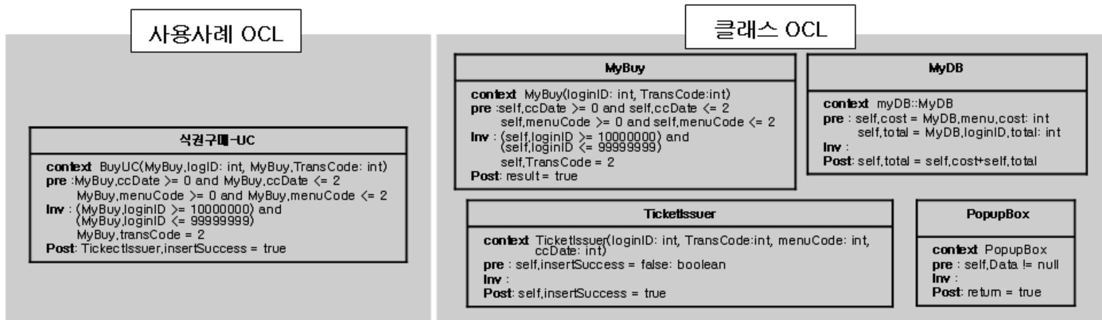

그림 5 OCL 표현
Fig 5. OCL Presentation

www.kci.go.kr

OCL로 표현한 결과이고 이를 적용하여 시험 사례를 생성한 것이 표 2이다. OCL에 표현된 불변식(invariant)과 선후조건(pre/post-condition)에 의해 시험 입력 데이터와 예상 출력 데이터를 생성할 수 있다. OCL의 선 제약 조건(precondition)과 불변식(invariant)의 조건들이 입력 데이터의 조합으로 쓰일 수 있다. 그리고 후 제약 조건(postcondition)이 시험사례의 예상 출력 값이 될 수 있다. 예상 출력 값(Oracle)과 실제 출력 값을 비교함으로 시험의 성공과 실패를 자동으로 결정할 수 있다. 이와 같은 방법으로 작성한 통합 시험 수준의 시험사례를 작성할 수 있다.

만약 BuyingTicket의 시험 결과 새로운 거래가 실패해 데이터베이스의 반영 유무를 알리는 insertSuccess의 속성이 만약 false라면 이 기능은 오류가 있음을 의미한다. 따라서 시험 엔지니어는 더 자세한 시험을 위해 낮은 추상 수준의 시험 사례를 작성해 오류 발생 지점을 파악해야 한다. 이를 위해 다시 각 클래스에 대해 OCL로 표현한 불변식(invariant)과 선후조건(pre/post-condition)를 이용하여 MM-Path에 참여하고 있는 클래스의 상호 작용 부분을 검사한다. 그리고 그 과정에서 오류가 발견되면 발견된 부분을 중심으로 MM-Path를 쪼개 더 자세한 시험을 한다.

표 2. 식권구매 기능 시험사례
Table 2. Buying Ticket Function Testcase

TC#	시험 수준	입력 데이터	소속	기대 결과	소속	시험 결과
UC-1	시스템	10000000<=loginID<=99999999	MyBuy	insertSuccess = true	TicketIssuer	
		TransCode = 2	MyBuy			
		0 <= ccDate <= 2	MyBuy			

즉, 슬라이싱된 전체 MM-Path에서 오류가 발견되었다면, 그 MM-Path를 구성하고 있는 클래스 간의 메소드에 대한 입출력 부분을 검사해서 기대 결과와 다른 지점을 파악해 더욱 자세한 시험을 한다. 만약 TicketIssuer의 issueTicket()의 실행 결과에 오류가 있다면 MyDB.insertDB()와 Ticket.issueTicket()의 상호 작용으로 인한 입출력 부분에 대한 검사를 통해 그 부분에 문제가 있음을 알 수 있다. 그렇다면 그림 6과 같이 BuyingTicket의 전체 MM-Path 중 오류가 잠재되어 있는 MyDB.insertDB()와 Ticket.issueTicket()의 상호 작용만을 잘게 쪼개진 MM-Path를 추출하고 그 부분에 대한 더 자세한 시험을 한다. 표 3은 이러한 수-직적 추적을 통해 잘게 쪼개진 MM-Path에 대한 시험 사례를 보이고 있다.

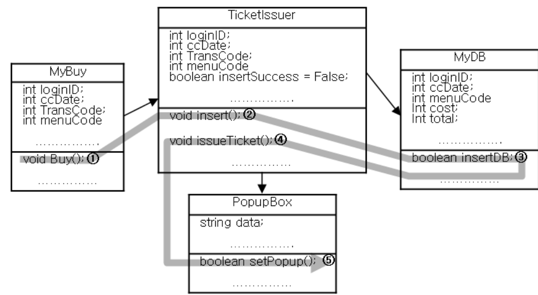

그림 6. 식권구매 슬라이스의 MM-Path
Fig 6. MM-Path of Buying Ticket Slice

표 3. 수직 추적성을 반영한 시험사례
Table 3. Testcase Reflecting Vertical Traceability

TC#	시험 수준	입력 데이터	소속	기대 결과	소속	시험결과
TC-1-4	통합	1000000<=loginID<=99999999	My Buy	insertSuccess = true	Ticket Issuer	
		insertSuccess = false	Ticket Issuer	record = string	MyDB	

만약 개발자의 실수로 insertSuccess의 값을 true로 하는 원시 코드를 누락했거나 잘못 작성했다고 하면 기대 결과(Expected Output)과 다른 결과를 낳게 된다. 표 3은 그림 6의 BuyingTicket 시험 사례에 해당하는 MM-Path 상에 있는 메소드 ticketIssue() 와 setPopup() 상호 작용에 대한 시험 사례를 작성한 것이다. MyBuy의 멤버 변수인 loginID와 insertSuccess의 유효 값을 입력 자료로 사용하면 그 결과 TiketIssuer는 true로 그리고 MyDB의 record에 해당 거래에 대한 기록이 저장되어야 한다. 그런데 TicketIssuer의 insertSuccess가 여전히 false라는 것은 거래가 정확히 반영되지 않았거나 아니면 거래가 실패했다는 것을 말한다. 따라서 전체 MM-Path에서 잘게 쪼개진 MM-Path 그리고 잘게 쪼개진 MM-Path를 구성하고 있는 해당 요소를 자세히 시험함으로써 수직적이면서 엄격한 시험을 통해 오류가 잠재되어 있는 원시 코드에 접근할 수 있다.

5.2 시험 결과

엄격한 방법의 효과를 확인하기 위해 표 4에서 제안한 여섯 가지의 UML 기반 시험을 비교했다. 먼저 시험 대상 시스템은 식권 자동 발매 시스템에 다양한 타입의 에러를 삽입했다. 시험 대상 범위는 식권 자동 발매 시스템의 '식권구매'와

www.kci.go.kr

‘개인정보관리’ 그리고 ‘거래조회’의 세 가지 사용 사례를 대상으로 한다. 각 사용 사례는 다섯 개에서 일곱 개 정도의 이벤트를 가지고 있고 각 이벤트는 MM-Path을 구성하는 요소가 된다. 표 4는 ‘식권구매’에 대한 간단한 사용사례 명세이다.

표 4. 식권구매 사용사례 슬라이스
Table 4. Buying Ticket Usecase Slice

사용사례 : 식권구매	
이벤트	1. 로그인하기 위해 ID와 비밀번호를 입력한다. 2. 날짜를 선택한다. 3. 메뉴 목록을 검색한다. 4. 메뉴를 선택한다. 5. 돈을 지불한다. 6. 영수증을 발급받는다. 7. 로그아웃한다.
제한조건	예약 가능 날짜는 현재로부터 3일 이내만 가능하다.

표 5는 여러 시험 기법들을 시행한 후 오류 발견과 추적에 대한 정보를 정리했다. 이 결과로부터 UC/SQ/CL의 조합으로 된 기법이 다른 기법들 보다 오류 발견 정도와 에러의 위치를 파악하는데 필요한 정보가 더 많음을 알 수 있다. 그 이유는 시스템을 시험하는 동안 UML 다이어그램과 원시 코드 사이에서 더 자세한 운행이 가능하기 때문이다. 또한 융통성 있는 수직 시스템 시험에서 제공하는 추적을 위한 링크 정보에 의해 오류를 발생시키고 있는 이벤트나 기능들을 더 자세히 관찰하고 들여다 볼 수 있기 때문이다.

표 5에서 통합 수준의 음영으로 표시된 영역은 시험을 통해 오류를 발견했지만 이들 오류를 추적할 수 있는 정보가 없음을 알 수 있다. 일반적으로 통합 시험에서는 블랙박스 시험 유형이 쓰이기 때문에 소스 코드까지 오류를 추적할 수 있는 정보가 부족하다. 예를 들어 사용 사례만을 사용한 시험 기법의 경우 시험을 통해 시스템이 결함을 가지고 있음을 알아냈지만 개발자에게 그 결함을 수정할 수 있는 정보를 주지 못하고 있음을 알 수 있다. 개발자는 단지 경험에 의해 오류가 내재되어 있는 위치를 추측하고 가정할 뿐이다.

표 5에서 에러를 찾기 위한 정보와 발견한 에러 간의 차이는 단위 수준이 통합 수준 보다 더 크다는 것을 알 수 있다. 특히 이러한 관계는 단순 시험 기법(Pure testing)의 SQ와 UC/SQ의 기법을 보면 더욱 확연히 드러난다. 이러한 이유는 시스템 또는 통합 시험 사례는 통합 또는 단위 시험의 오류 발견을 위한 정보가 있지만 오류를 발견한 이후에 시험 사례 간의 관련성 또는 시험 사례와 오류 위치와의 연관성을 보여주지 못하고 있기 때문이다. SQ와 UC/SQ는 통합 시험 레벨에서는 오류를 검출할 뿐만 아니라 추적성을 지원하고 있지만 단위 수준의 추적 정보는 제공하지 못하고 있다.

표 5. 각 시험 기법의 에러 검출
Table 5. Error Detection of Test Methods

추상 수준	삽입된 에러 유형	단순 시스템 시험			융통성 있는 수직 시스템 시험
		UC	SQ	UC/SQ	UC/SQ/CL
시스템 수준	기능정확성 오류	가능	-	가능	가능
	기능적합성 오류	가능	-	가능	가능
	기능완전성 오류	가능	-	가능	가능
통합 수준	메시지 패싱 오류(4)	-	4	4	4
	메소드 파라미터 타입 오류(5)	-	5	5	5
	메소드 리턴 타입 오류(5)	-	5	5	5
단위 수준	메소드 알고리즘 오류(3)	-	-	-	3
	멤버 데이터 타입 오류(5)	-	-	-	5
	멤버 데이터 미싱 오류(5)	-	-	-	5
	멤버 데이터 범위 오류(1)	-	-	-	1

(error location information against source code/detected errors)

표 5에서 오류 위치를 추적하기 위한 정보와 검출된 오류의 개수의 차이가 높은 수준보다는 낮은 수준에서 더 많다는 것을 확인했다. 이것은 블랙박스 시험 유형의 단점을 보이고 있는 부분이다. 블랙박스 시험 유형은 개발자나 시험 엔지니어에게 경제적인 시험 방법을 제공하지만 오류 발견 이후 소스 코드까지 오류를 추적하는 것이 쉽지 않음을 의미한다. 만약 시험 기법들이 모든 추상 수준에 대해 충분한 시험 정보를 제공한다면 이것은 곧 추적가능성을 지원하는 것이고 잠재되어 있는 오류 지점을 자세히 들여다 볼 수 있는 수단을 제공하는 것이다. 대부분의 UML 기반 시험 기법들은 사용 사례 또는 모델 추상 수준 정도로 정보들이 제한되어 있다. 이것은 오류 추적을 어렵게 만드는 원인 중에 하나가 된다. 이러한 문제를 해결하기 위해 본 논문에서 제안한 시험 기법은 시험 사례를 생성하는 동안 각 추상 수준에 대한 시험 정보들을 추출하기 때문에 추적가능성과 줌인 기능을 제공하고 있다.

www.kci.go.kr

표 6. 시험 기법 비교 평가
Table 6. Comparison of Test Methods

추상 수준	단순 시스템 시험			융통성 있는 수직 시스템 시험
	UC	SQ	UC/SQ	UC/SQ/CL
시스템 수준	○	X	○	○
통합 수준	X	○	○	○
단위 수준	X	X	X	○

표 5의 실험 내용을 근거로 표 6에 실험 결과를 간단히 정리했다. 단순 시스템 시험 기법과 본 논문에서 제안한 융통성 있는 수직 시스템 시험이 찾아낼 수 있는 오류의 추상 수준이 각각 다름을 알 수 있다. 수직 시험은 시스템 수준부터 단위 수준까지 모두를 커버하고 있지만 단순 UC는 시스템 수준만 가능하고 SQ는 통합 수준만 가능하다. 그리고 UC/SQ는 오류 접근 수준이 시스템 수준과 통합 수준까지만 가능함을 실험으로 알 수 있다.

Ⅵ. 결론

본 논문은 시스템 시험에서 추적가능성을 지원하기 위한 구체적인 절차에 대해 설명했다. 예시한 추적가능성 링크의 표현과 오류 검출 및 위치 파악에 대한 사례 연구를 통해 제안한 기법이 실행 가능함을 보였다. 이러한 시험 방법의 실현은 그림 1에서 설명한 것과 같이 기존의 상향식 시험 방법과는 다른 관점에서 접근하고 있다. 이를 수직적 시험이라고 명명했으며 그 결과 기능 수준에서 소스 코드 수준으로 오류를 파악할 수 있는 방법을 제공했다. 추적가능성 링크는 사례 연구에서 사용된 예제보다 훨씬 더 클 것이다. 하지만 자동화 도구를 이용해 링크 정보들을 번역하여 이러한 문제는 해결할 수 있다.

에러 검출에 대한 효율성과 관련해서 추적가능성 링크에 대한 UC/SQ/CL의 조합이 가장 높았음을 사례 연구에서 보았다. 그러나 일반적으로 추적을 위한 최적화된 링크의 조합을 선택하기 위해서는 삽입된 오류의 타입들을 확장해야 할 것이다. 추적가능성 정보를 기반으로 한 시스템 시험의 개념은 다양한 도구들의 지원을 필요로 한다. 그리고 이러한 정보를 지속적으로 일관성 있게 사용할 수 있는 방법이 필요할 것이다.

참고문헌

[1] L. Briand and Y. Labiche, "A UML-based approach to system testing", Proc. 4th International Conf. on UML – The Unified Modeling Language, Modeling Language, Concepts, and Tools, Toronto, CA, 2001, LNCS 2185, Springer, 2001, pp. 194-208

[2] J. Offutt, A. Abdurazik, "Generating tests from UML specifications", Proc. 2nd International Conf. on UML, 1999, pp.416-429.

[3] Dinh-Trong, S. Ghosh, R. B. France, M. Hamilton, and B. Wilkins, "UMLAnT: An Eclipse Plugin for Animating and Testing UML Designs", Eclipse Technology Exchange Workshop in OOPSLA, San Diego, 2005, pp.?

[4] L. Brian, Y. Labiche, "A UML-based approach to system testing", Software and System Modeling, 1(1), 2002, pp.10-42.

[5] A. Abdurazik, J. Offutt, "Using UML collaboration diagrams for static checking and test generation", International Conf. on UML, 2000, pp.383-395.

[6] M. Scheetz, A. von Mayrhauser, R. France, E. Dahlman, and A. E. Howe, "Generating test cases from an OO model with an AI planning system", Proc. 10th International Symposium on Software Reliability Engineering, Boca Raton, Florida, 1999, pp.250-259

[7] O. Gotel and A. W. Finkelstein, "An analysis of the requirements traceability problem", Proc. of the International Conf. on Requirements Engineering, Colorado Springs, CO, 1994, pp.94-10.

[8] B. Ramesh, "Factors influencing requirements traceability in practice", Communications of the ACM, 41(12), 1998, pp.34-34.

[9] K. Seo and E. M. Choi, "Comparison of five black-box testing methods for object-oriented software", Proc. 4th ACIS International Conference on Software Engineering Research, Management & Applications, Seattle, WA, 2006, pp.213-222.

www.kci.go.kr

〔10〕 A. Andrews, R. N. Francs, S. Ghosh, and G. Craig, "Test Adequeacy Criteria for UML Design Models", Journal of Software Testing, Verification and Reliability, 13(2), 2003, pp.95-127..

〔11〕 E. Dustine, Effectivve Software Testing: 50 specific ways to improve your testing, Addison-Wesley, 2003.

〔12〕 J. Hartmann, C. Imoberdorf and M. Meisinger, "UML-Based integration testing", Proc, ACM SIGSOFT International Sysmposium on Software Reliability Engineering, Florida, 1999, pp.250-259.

〔13〕 P. C. Jorgensen and C. Erickson, "Object-Oriented Integration Testing", Communications of the ACM, 37(9), 1994, pp.30-37.

저 자 소 개

서 광 익

2004년 8월 : 동국대학교 컴퓨터공학석사

2006년 8월 : 동국대학교 컴퓨터공학 박사 수료

관심분야 : 소프트웨어 테스트, 소프트웨어 품질, 프로세스

최 은 만

1982년 동국대학교 전산학과 졸업(학사)

1985년 한국과학기술원 전산학과(공학석사)

1993년 일리노이 공대 전산학과(공학박사)

1985년~1988년 한국표준연구소 연구원

1988년~1989년 데이콤 주임연구원

2000년, 2007년 콜로라도 주립대 전산학과 방문교수

1993년~현재 동국대학교 컴퓨터공학과 교수

관심분야: 객체지향 설계, 소프트웨어 테스트, 프로세스와 메트릭, Program Comprehension, AOP

www.kci.go.kr