수직 추적가능성을 제공하는 엄격한 시스템 테스트

서 광 익[†] · 최 은 만[††]

요 약

추적이란 개념은 모델 중심의 개발에서만이 아니라 테스트를 위하여 매우 중요한 요소이다. 수직적인 추적은 모델로부터 테스트와 디버깅 단계에 이르기까지 시스템을 잘 관리할 수 있게 한다. 또한 테스트 단계에서 발견한 결함에 대한 오류를 추적할 때 발생하는 추상 수준의 오르내림을 추적가능성으로 극복하게 한다. 이 논문에서는 시스템 테스트 수준과 통합 테스트 수준에서 UML을 이용하여 수직적으로 연결하여 더욱 엄격한 테스트가 되는 방법을 제안한다. 실험을 통하여 추적이란 개념이 어떻게 작동하며 오류 부분을 찾아내는지 얼마나 효과적인지 보이며 모델에서 코드까지의 구체적인 사례를 이용하여 방법을 소개한다.

키워드 : 시스템 테스트, 추적가능성, UML 테스트, 통합 테스트, 오류 추적가능성

Rigorous System Testing by Supporting Vertical Traceability

Seo Kwang Ik[†] · Choi Eun Man[††]

ABSTRACT

Traceability has been held as an important factor in testing activities as well as model driven development. Vertical traceability affords us opportunities to improve manageability from models and test cases to code in testing and debugging phase. Traceability also makes overcome to difficulties of going up-and-down abstraction level to find out error spot of faults discovered by testing This paper represents a vertical test method which connects a system test level and an integration test level in a test stage by using UML. Experiment of how traceability works and how effective focus on error spots has been included using concrete examples of tracing from models to the code.

Key Words : System Test, Traceability, Uml Test, Integration Test, Error Traceability

1. 서 론

작은 실수에 의하여 유입되는 오류 때문에 테스트와 유지보수에 막대한 비용을 지불하는 경우를 볼 수 있다. 많은 비용의 근원을 따져보면 문제 자체의 복잡성보다는 여러 가지 다른 레벨과 관점의 모델 및 구현물 사이에 적절한 추적 방법이 제공되지 않아 단계적 테스트 작업들과 디버깅이 유기적으로 결합되어 있지 않기 때문이다. 테스트 각 단계와 담당하는 조직에 따라 시험 대상과 목표는 매우 다르다. 그러나 시스템을 통합하고 출시하기 전에 전체적인 시스템을 테스트하는 단계에서는 모델로부터 코딩까지 또는 코딩으로부터 테스트 케이스 또는 모델까지 수직적 슬라이스의 추적이 빈번하게 일어난다.

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 단계별 각 테스트 작업에서 결함을 발견하여 단지 QA 팀이나 개발자들에게 던져 놓는데 까지를 커버한다.

하지만 보다 짧은 Time-to-market을 위하여 단계적 테스트 작업들은 서로 유기적으로 결합되어야 하며 이를 위하여 추적가능성이 제공되어야 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 찾

† 준 회 원 : 동국대학교 컴퓨터공학과 박사수료
†† 정 회 원 : 동국대학교 컴퓨터공학과 교수(교신저자)
논문접수 : 2007년 5월 1일, 심사완료 : 2007년 7월 21일

아내기는 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.

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 자체의 검증이나 시험은 이러한 방법으로 설계의 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.

완성된 소프트웨어를 모델만을 기반으로 하는 시험에는 한계가 있다. 모델에 표현된 추상 수준은 한계가 있으며 시험하려는 시스템 내부의 자세한 추적과 확인이 필요하다. 마치 자동차에 결함이 있는지 시험하는 경우 설계 도면이나 자동차의 기능 단위를 근거로 시험하고 그 현상만으로 테스트하기는 어렵다. 내부가 복잡해지고 전자적 부품이 많아지면서 요즘은 스캐너를 이용하여 시험 중 내부 상태를 찍어보고 결함이 발견된 부분을 쉽게 추적한다. 소프트웨어를 시험하는데도 유사한 접근 방법이 필요하다는 것이 이 논문의 가설이다.

이 논문에서 제안한 방법은 시스템 테스트를 블랙박스로 진행하다가 결함이 발견된다면 단위 수준 또는 구현 언어의 구문 수준까지 내려갈 수 있고 기능 슬라이스 별로 이벤트의 경로를 추적하여 오류 스팟을 찾을 수 있는 더욱 엄격한 시스템 테스트 방법이다. 2장에서는 시스템 테스트와 관련된 연구들과 이 논문에서 도입된 추적가능성에 대하여 고찰해보고 3장과 4장에서는 수직 추적의 개념과 절차를 알아본다. 그리고 5장에서 결과물 간의 추적가능성 구현에 대해 설명하고 이 논문에서 제안한 과정으로 실제 테스트 실험한 사례 연구를 6장에 기술하였다.

2. 관련 연구

2.1 명세 기반의 시스템 테스트

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 릴리스될 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 추적가능성

추적가능성이란 소프트웨어 개발 결과물에 포함된 내용을 명세서를 처음 작성할 때부터 사용에 이르기까지 추적할 수 있음을 뜻한다. 요구분석에 포함된 특정 기능이 있을 때 그 부분을 담당하는 설계, 또한 그 부분에 대한 구현 코드, 그 기능을 시험하는 테스트 케이스 등이 결과물의 내용이다. 추적가능성은 여러 목적으로 사용될 수 있다. 모델에 포함된 내용이 모두 구현되었는지 나타내기 위한 커버리지 체크가 하나의 목적이 될 수 있고 모델의 변경을 관리하기 위하여 변경된 모델이 코드의 어떤 부분에 해당되는지, 또한 모델 변경의 영향이 코드의 어떤 부분에 영향을 주는지를 파악하기 위하여 추적 링크를 사용한다. 하지만 이제까지의 연구 발표된 문헌들을 보면 대부분 요구의 추적이나 변경 효과분석에 초점을 두고 있다[7, 8].

이 논문에서 제시하는 엄격한 시스템 테스트는 완전한 추적가능성이 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 논문에서 제시하는 추적가능성은 UML 중심으로 표현된 모델에서 최대한의 커버리지를 추구한다. 명세 중심의 시스템 테스트의 여러 방법 중에서 어떤 것, 또는 어떤 조합이 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.

3. 수직 추적에 의한 시스템 테스트

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.

3.1 추상화 수준과 V 모델

전통적인 소프트웨어 개발 프로세스인 폭포형 모델에 검증과 테스트 작업을 강조한 V 모델이 있다. (그림 1)의 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.

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 자세한 시험 후 의심 가는 부분이 있다면 각 모듈 단위의 시험을 더욱 자세한 화이트 박스 방법을 써서 할 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.

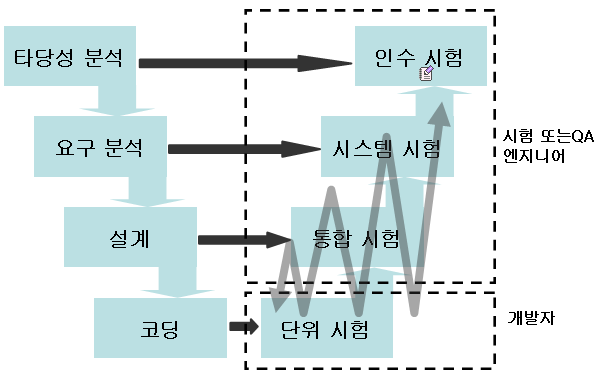
3.2 수직 시험을 위한 슬라이스

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.

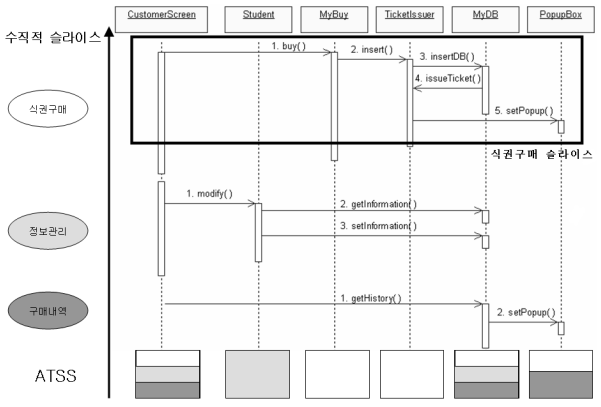
사용사례는 어떻게 엔드 유저 또는 시스템 간의 상호 작용을 하는지 명확하게 정의하는데 사용된다. 즉, 시스템의 행위를 시스템 외부의 사용자 관점에서 모델링하고 규정한다. 반면 사용사례 슬라이스는 사용사례 인스턴스의 조각이라고 할 수 있다. 사용사례 하나에는 무수한 이벤트의 경로가 나오는데 이 중에 테스트하기 위하여 잘라낸 실행 경로를 말한다. 따라서 각 사용 사례 슬라이스는 디자인 모델로써의 사용 사례 실현에 대한 구체적인 관점을 보여줄 수 있다.

(그림 2)는 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 표준화 되어 가고 있으며 사용사례 다이어그램에 분리할 수



(그림 1) 수직 추적가능성이 있는 V 모델



(그림 2) ATSS의 기능 슬라이스

있는 기능 슬라이스가 잘 나타나 있다. 예를 들어 식권 자동 발매 시스템을 기능 단위로 쪼개본다면 (그림 2)에 표현된 것처럼 된다.

(그림 2)에 나타낸 것처럼 시스템은 여러 기능이 모여 이루어진다. 엔드유저에게 보이는 하나의 기능은 여러 개의 클래스가 협력하여 메시지를 호출함으로 구현된다. 예를 들어 호텔에 체크인 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 2)에 비추어 다시 생각해 보자. 먼저 각 단위 모듈을 구현하는 과정에서 (그림 2)의 맨 하단의 박스로 표현된 클래스들을 시험한다. 이 때 시험하는 것은 클래스 안의 메소드에 구현된 알고리즘의 각 경로들, 개별 메소드들의 호출 결과, 메소드 안의 불변조건 등이다. 다음 통합 테스트는 구현된 클래스 및 모듈들을 등록하고 이를 통합하여 빌드하는 과정에 새로 통합하는 모듈들이 잘 인터페이스 되는지 시험한다. 시스템 단위의 시험 단계에는 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.

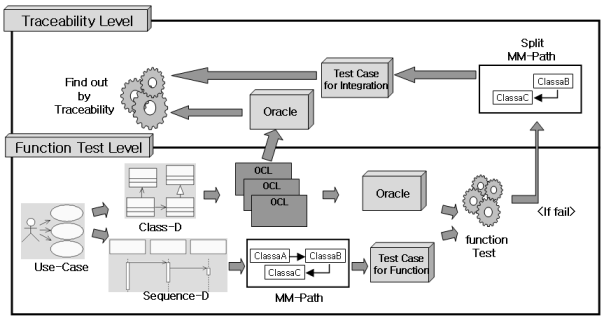
여기에서 제일 큰 애로 사항이 왼쪽의 수직적 슬라이스와 맨 하단의 코드와의 매핑이 어려운 점이다. 시험 대상인 슬라이스와 코드의 관련 부분에 대한 추적가능성이 제공되지 않으면 결함의 현상이나 모델 상의 위치 정도만 파악될 수 있다.

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 2)에서 완성된 클래스 안에 여러 기능들이 복합적으로 내재되어 있는데 어떤 부분이 어떤 테스트 슬라이스에 의하여 시험되었는지 스캔할 수 있다면 테스트 및 유지보수 작업에 효과적일 것이다.

4. 엄격한 시스템 테스트 프로세스

일반적인 시스템 테스트와 이 논문에서 제시하는 테스트 프로세스는 다음과 같은 큰 차이가 있다. 먼저 V 모델에서의 단위 테스트 및 통합 테스트는 순차적으로 진행된다. 다음에 엄격한 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 시험하고 이 단계에 추적가능성이 제공된다는 점이다.

(그림 3)의 엄격한 시스템 테스트 과정은 크게 두 가지로 나눈다. 아래 부분은 시스템 테스트를 처음으로 할 때 엄격한 테스트 케이스를 만들고 시험하는 과정을 나타낸 것이다. 위 부분은 엄격한 테스트 케이스를 통과하지 못하고 결함 현



(그림 3) 엄격한 시스템 테스트 프로세스

상을 보인 경우 더욱 분할된 MM-path를 만들어 시험한다.

시스템 시험 단계의 테스트 사례를 만들기 위하여 분석 및 설계 단계에서 명세한 사용사례 및 클래스 다이어그램, 순차 다이어그램을 이용한다. 자세한 방법은 [1]에 표현된 것처럼 순차 다이어그램에 나오는 여러 경로를 분석한 후 [10]에 제시된 커버리지 기준에 만족하는 테스트 케이스들을 구한다. 테스트 경로들을 구동시키는 테스트 입력 값과 예상 결과는 OCL로 표현된 전제조건(pre-condition)과 결과조건(post-condition)을 이용하여 구할 수 있다.

테스트 사례와 관련된 클래스 중 가장 먼저 호출된 클래스에 입력할 자료의 조건을 택하고 더불어 테스트 사례의 맨 끝에 위치한 클래스의 결과 조건 및 불변 조건을 OCL로 표현된 설계에서 찾아낸다. 예를 들어 '식권구매' 기능을 테스트 한다고 할 때 MyBuy.buy() 함수의 전제조건을 찾아보면 테스트 입력 값을 구할 수 있고 PopupBox 클래스에서 setPopup() 메소드가 호출된 후의 결과조건과 불변조건을 찾으면 그것이 예상 결과이다.

한편 설계 단계에서 자세한 제약사항들을 OCL로 정의하지 않은 경우 위에서 말한 테스트 오러클을 발견하기 어렵다. 이러한 경우에 자세히 구현된 코드를 추적할 수 있다면 테스트 예상 결과를 구하기 용이하다. 설계 명세 자체를 테스트하는 경우 OCL로 정의되어 있지 않다면 심볼릭 실행(symbolic execution)을 이용할 수 있으나 구현 후에는 이러한 정보들이 원시코드에 잘 나와 있기 때문이다.

엄격한 시스템 테스트 사례가 성공적으로 수행되었다면 그 부분의 시스템 슬라이스는 확인된 것이다. 결함이 발견된 경우 더 자세한 테스트에 들어간다. 여기서 MM-path를 이용하는데 이는 사용사례 안에 포함된 이벤트 호출의 연속이다. 하나의 사용사례 안에는 여러 개의 테스트 경로가 존재할 수 있는데 하나의 테스트 케이스를 이루는 작은 메시지 호출의 집합이 MM-path이다.

엄격한 시험을 수행하는 과정에서 참여하는 모델과 시험 영역을 정리하면 <표 1>과 같다. 각 추상 수준별로 커버리지와 참여 모델이 표현되어 있다. 시스템 시험 단계에서는 사용사례 슬라이스를 중심으로 시험 사례를 작성한다. 그리고 MM-Path의 맨 처음과 마지막 노드의 입력과 출력 데이터를 이용한 블랙박스 스타일의 시험을 수행한다. 만약 출력 값이 예상 결과와 틀리다면 MM-Path 중 어떤 노드에

〈표 1〉 엄격한 시험 참여 요소

추상 수준	시스템	통합	단위
커버리지	MM-Path	쪼개진 MM-Path	MM-Path 상의 특정 노드
모델	사용사례 슬라이스	순서 다이어그램, 클래스 다이어그램	클래스 다이어그램

오류가 발생한 것이므로 전체 MM-Path가 아닌 오류 발생 가능 구역을 탐색한 후 오류가 발생한 노드를 중심으로 이와 연관이 있는 노드들 간의 통합 시험을 실행한다. 즉 오류 발생 지역을 포함하고 있는 쪼개진 MM-Path가 될 것이다. 이 과정에서 오류 발생 노드가 확인되면 해당 노드에 대한 단위 시험으로 진행한다.

5. 추적가능성

소프트웨어를 구성하는 여러 결과물 간에 추적가능성을 제공하려면 여러 결과물의 구조를 분석하고 어떤 참조가 필요한지 알아내야 한다. 요구분석으로부터 설계 및 원시코드, 테스트 케이스에 이르는 여러 결과물들은 각각의 구조와 의미를 가지고 있으나 의미적 연관성은 잘 드러나 있지 않다. 대부분 엔지니어의 훈련에 의하여 설계 부분을 짐작하고 경험에 의하여 코드 위치를 찾아낼 뿐이다.

이 논문에서는 테스트에 유용한 추적 링크를 찾아내고 이를 각 결과물에 표시하여 연관성을 제공한다. 먼저 여러 추상 계층의 분석 명세들 사이에 연관성을 분석하고 참조 방법을 확보하기 위하여 링크를 제공하였다.

이 논문에서 제시하는 엄격한 시스템 테스트 방법에서는 요구분석 모델에서 객체 모델, 객체모델에서 원시코드로의 추적이 가능하며 뿐만 아니라 테스트 사례에서 해당되는 요구 모델, 객체모델, 커버되는 원시코드로의 추적이 가능하다.

추적가능성을 제공하기 위한 링크의 설정은 추성 레벨에 따라 다음과 같은 다른 방법으로 가능하다.

- 원시코드의 링크 - javadoc에 의하여 처리되는 태그를 달아서 처리한다. 예를 들어 {@link ReservationUML. CustomerStaff}과 같이 표시하여 객체 모델을 추적할 수 있게 한다.
- UML 모델에서의 링크 - UML 모델의 각 요소들, 예를 들면 사용사례, 클래스, 상태, 인퍼페이스 등은 확장 방법의 하나인 태그를 이용하여 링크 표시를 한다. 예를 들어 {implementedBy = java.appl.hotel.ReservationAppl. java}라고 객체 안의 클래스에 표시하면 이것이 원시코드로의 추적을 가능하게 한다.
- 테스트 케이스와의 링크 - 테스트 케이스는 일반적으로 (그림 4)와 같은 테이블 형태를 취하는 문서파일이다. 따라서 하이퍼텍스트 링크를 사용하여 추적가능성을 제공한다.

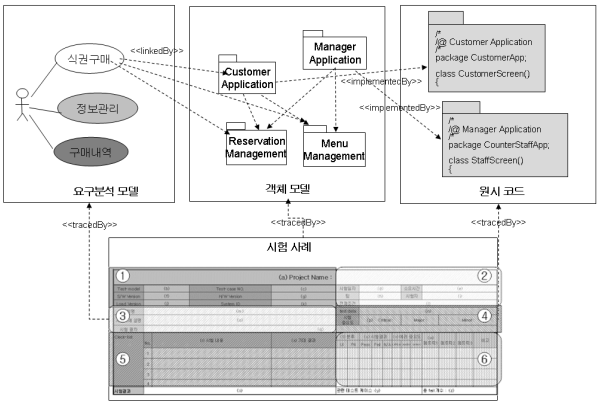
복잡한 소프트웨어 시스템에는 분석 및 설계에 많은 요소가 존재하며 또한 원시코드와 테스트 케이스도 방대하다. 따라서 추적가능성을 제공하기 위한 링크도 상당한 분량이 될 수 있다. 따라서 도구를 제공이 필요하며 이에 따라 소프트웨어가 변경될 때 추적 링크의 변경도 용이해진다.

추적성을 도구로 제공하는 방법은 두 가지가 있다 리파지토리에 링크를 제공하는 방법과 각 도구나 문서에 추적 링크를 구현하는 것이다. 전자의 경우 여러 CASE 도구들을 XML로 쓴 리파지토리에 연결할 필요가 있다.

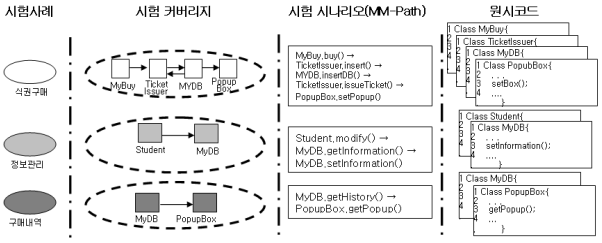
이러한 환경을 구축하기 위해서는 시험 사례와 UML 명세 그리고 원시 코드를 추적할 수 있는 정보가 필요하다. (그림 2)에서 자동 식권 발매 시스템의 사용사례에 대한 슬라이스 결과를 보였다. '식권구매'와 '정보관리' 그리고 '구매내역' 기능은 세 개의 사용사례를 말한다. 그리고 세 개의 클래스에 소속된 메소드들이 서로 연합하여 각 사용사례를 실현하게 된다.

(그림 2)의 사용사례는 곧 하나의 시험 사례 단위로 전환할 수 있다. (그림 5)는 세 개의 사용사례에서 세 개의 시험 사례로 각각 전환된 것을 보이고 있다. 그리고 시험 사례와 관련된 시험 범위와 시험 시나리오 그리고 원시 코드와의 관계를 보이고 있다. 각 시험 사례는 사용사례를 실현하기 위한 클래스들로 구성된 시험 범위를 가지게 된다.

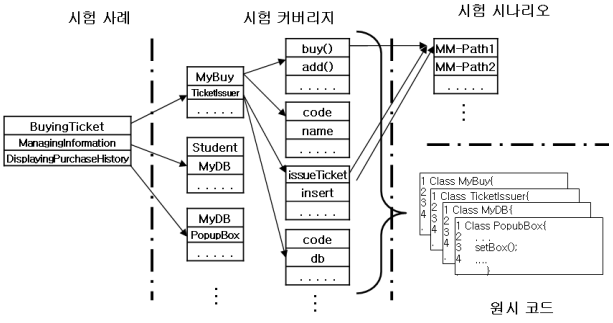
식권을 구매에 대한 시험을 위해서는 BuyingTicket 클래스와 TicketIssuer, MyDB, PopupBox 클래스가 시험 범위에 속한다는 것을 알 수 있다. (그림 5)에서 사용사례를 구현하기 위한 메소드 메시지 경로는 시험을 실행하는 시험 시나리오가 된다. 다시 말해 식권을 구매에 대한 기능을 시험하기 위한 BuyingTicket 시험 사례의 MM-Path는 MyBuy.buy() → TicketIssuer.insert() → MyDB.inertDB() → TicketIssuer.issueTicket() → PopupBox.setPopup() 이 된다. 그리고 시험 시나리오를



(그림 4) 추적 링크



(그림 5) 시험 사례로부터 원시 코드까지의 추적



(그림 6) 추적 정보 계층

실행하면서 오류를 검사하게 된다. 오류가 발견되면 메소드 메시지 경로의 클래스와 메소드 그리고 속성을 원시 코드에서 찾을 수 있다. 이러한 시험 환경을 구축하기 위해서는 시험 사례와 시험 범위에 속한 클래스 그리고 시험 시나리오에서 호출되는 메소드와 속성 뿐 아니라 이들이 삽입된 원시 코드간의 추적이 가능하도록 추적 링크 정보가 필요하다.

시험 사례에서 원시 코드까지 추적하기 위해서는 (그림 5)에 표현된 요소들이 모두 추적 링크 정보로 표현이 되어야 한다. 시험 사례와 시험 범위 그리고 시나리오와 원시 코드의 관계를 살펴보면 그 구조가 트리 형태의 계층 구조를 갖고 있음을 알 수 있다. 하나의 시험 사례는 그와 관련된 여러 개의 클래스를 갖고 있고 각 클래스는 또한 여러 개의 메소드와 속성을 갖고 있다. 그리고 클래스와 메소드의 조합으로 시나리오를 구성할 수 있다. 속성은 OCL을 이용한 시험 대상이 된다. 그리고 클래스와 메소드 그리고 속성은 모두 원시 코드에서 포함되어 있다. (그림 6)은 이러한 메타 정보의 계층을 표현하고 있다.

6. 사례 연구

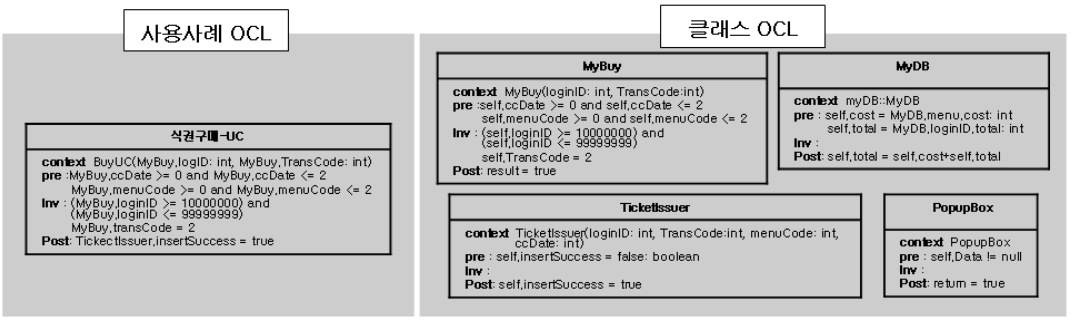
본 절에서는 식권 자동 발매 시스템(ATSS: Automatic Ticket Sales System)을 이용해 수직적 시험을 수행하는 과정과 시험 결과 도출된 시험 사례를 구한다.

6.1 수직적 시험 사례

'식권구매' 시험사례는 사용사례에 관한 제약사항과 메소드 메시지 경로에 참여하고 있는 클래스 속성들의 제약사항들로 구성할 수 있다. (그림 7)은 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 2)에서 사용사례와 클래스 다이어그램 그리고 순차 다이어그램을 사용하여 슬라이싱 하는 과정을 보였다. 슬라이싱 결과물 중 만약 '식권구매' 기능을 테스트 한다고 할 때 MyBuy.buy() 함수의 전제조건을 찾아보면 테스트 입력 값을 구할 수 있다. 그렇다면 (그림 2)에서 BuyingTicket을 시험하기 위해 CustomerScreen을 이용해 테스트 입력 값을 MyBuy.buy()로 입력하게 된다. 그리고 PopupBox.setPopup()이 반환해야 하는 결과 값을 검사해 시험 결과를 평가한다. 시험을 위해 추출한 (그림 7)의 OCL을 이용하여 '식권구매' 기능시험에 대한 시험사례를 생성하면 <표 2>와 같다. 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.

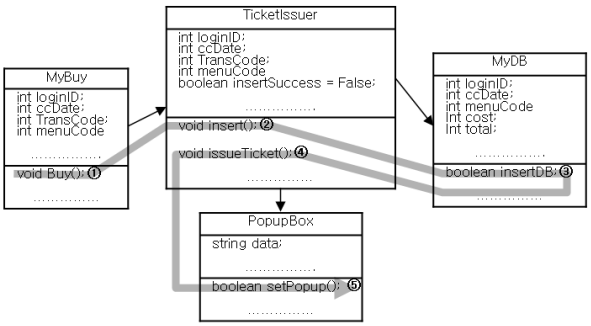
만약 BuyingTicket의 시험 결과 새로운 거래가 실패해 데이터베이스의 반영 유무를 알리는 insertSuccess의 속성이 만약 false라면 이 기능은 오류가 있음을 의미한다. 따라서 시험 엔지니어는 더 자세한 시험을 위해 낮은 추상 수준의 시험 사례를 작성해 오류 발생 지점을 파악해야 한다. 이를 위해 다시 각 클래스에 대해 OCL로 표현한 불변식(invariant)



(그림 7) 제약사항의 OCL 표현

〈표 2〉 식권구매 기능 시험사례

TC#	시험 수준	입력 데이터	소속	기대 결과	소속	시험 결과
UC-1	시스템	1000000<=loginID<=99999999	MyBuy	insertSuccess = true	TicketIssuer	
		TransCode = 2	MyBuy			
		0 <= ccDate <= 2	MyBuy			
		0 <= menuCode <= 2	MyBuy			



(그림 8) 식권구매 슬라이스의 MM-Path

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.

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 8)과 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 3>은 이러한 수직적 추적을 통해 잘게 쪼개진 MM-Path에 대한 시험 사례를 보이고 있다.

만약 개발자의 실수로 insertSuccess의 값을 true로 하는 원시 코드를 누락했거나 잘못 작성했다고 하면 기대 결과(Expected Output)과 다른 결과를 낳게 된다. <표 3>은 (그림 8)의 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.

6.2 시험 결과 비교

<표 4>는 추적을 위한 정보의 조합이 어떤 시험 단계를 커버하고 있는지 보이고 있다. 소프트웨어 시험을 위해 다양한 UML 기법들이 사용되었다. 그 중 추적성을 제공하고 있는 최적의 UML 다이어그램의 조합이 어떤 것인지 확인하기 위해 다양한 UML 기반 시험들의 방법을 비교하였다. 표 1에서 본 논문에서 제안한 시험 기법을 포함한 엄격한 시험 방법이 다른 방법들 보다 상대적으로 더 넓은 추상 커버리지와 시험 사례 사이의 다양한 추적 링크 정보가 있다는 것을 알수 있다. 'Linkage process or mechanism'은 각 추상 수준의 연관성 있는 시험 데이터들 간의 접근 방법이 있는가에 대한 항목이다. 시험 사례가 있지만 이들 사이에 있는 연관성을 정의하지 못했다면 추적성이 없다고 할 수 있다. 시험 사례가 있는 것만으로는 추적이 어렵고, 따라서 시험 사례와 그에 해당한 추적 방법이 있어야 한다. 예를 들어 SQ/ST와 같은 경우 SQ는 ST를 이용한 시험 사례를 만들기 위한 전체 시나리오의 범위를 제시하는 정도로만 쓰이고 있습니다. 따라서 SQ가 시험 사례를 구축하는 과정에 쓰였다 하더라고 시험 데이터들 간의 연관성에 대한 정보를 갖고 있다고 보기 어렵기 때문이다.

추적가능성을 제공하는 정보의 합리적인 조합과 본 논문에서 제안한 추적성 지원을 통한 엄격한 방법의 효과를 확인하기 위해 <표 4>에서 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 '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

〈표 3〉 수직 추적성을 반영한 시험사례

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

〈표 4〉 시험 단계별 시험 기법 추적 정보 커버리지

		UC	SQ	UC/SQ	SQ/ST	SQ/CL	UC/SQ/ST	UC/SQ/CL
시험 수준l	시험	○	○	○	○		○	○
	통합		○	○	○	○	○	○
	단위					○	○	○
추적 링크 정보	S-I						○	○
	I-U						○	○
	U-S/C							○

〈표 5〉 식권구매 사용사례 슬라이스

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.

5>는 '식권구매'에 대한 간단한 사용사례 명세이다.

(UC: use case, SQ: sequence, UC/SQ: UC and SQ, SQ/CL: SQ and class, SQ/ST: SQ and state chart, UC/SQ/ST: UC,SQ, and ST, UC/SQ/CL: UC, SQ, and CL, S-I: System and Integration, I-U: Integration and Unit, U-S/C: Unit and Source Code)

<표 6>은 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 엄격한 시험에서 제공하는 추적을 위한 링크 정보에 의해 오류를 발생시키고 있는 이벤트나 기능들을 더 자세히 관찰하고 들여다 볼 수 있기 때문이다.

(error location information against source code/detected errors)

<표 6>에서 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.

SQ/ST와 SQ/CL 그리고 UC/SQ/ST 경우 역시 통합 수준 시험에서 오류를 추적할 수 있는 정보가 없음을 알 수 있다. 이들 시험 기법에서 사용되는 순서 다이어그램(SQ)는 단지 상태도(ST)나 클래스 다이어그램(CL)을 이용해서 시험사례를 작성하기 위한 시나리오, 즉 시험의 범위를 작성하기 위해 사용되고 있고, 순서 다이어그램이 제공해 주는 정보들을 작성된 시험 사례가 포함하지 않고 있다. 따라서 상태도나 클래스 다이어그램이 주로 적용되는 수준인 단위 시험에 대한 오류를 추적할 수 있는 정보가 있지만 통합 수

〈표 6〉 각 시험 기법의 에러 검출

추상 수준	삽입된 에러 유형	단순 시스템 시험					엄격한 시험	
		UC	SQ	UC/SQ	SQ/ST	SQ/CL	UC/SQ/ST	UC/SQ/CL
통합 수준	메시지 패싱 에러(4)	0/4	4/4	4/4	0/4	0/4	0/4	4/4
	메소드 파라미터 타입 에러 (5)	0/5	5/5	5/5	0/5	0/5	0/5	5/5
	메소드 리턴 타입 에러(5)	0/5	5/5	5/5	0/5	0/5	0/5	5/5
단위 수준	메소드 알고리즘 에러(3)	0/3	0/3	0/3	3/3	0/3	3/3	3/3
	멤버 데이터 타입 에러 (5)	0/5	0/5	0/5	0/5	0/5	0/5	5/5
	멤버 데이터 미싱 에러(5)	0/5	0/5	0/5	0/5	0/5	0/5	5/5
	멤버 데이터 범위 에러 (1)	0/1	0/1	0/1	1/1	0/1	1/1	1/1

준에서의 추적 정보는 부족하다는 것을 알 수 있다. 이는 다양한 수준의 다이어그램이 시험 사례를 작성하는데 사용이 되었지만 이들이 제공하고 있는 고유의 정보가 시험 사례에 적용이 되지 않았다면 오류에 대한 추적이 어려움을 알 수 있다.

<표 6>에서 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 사례간의 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. UC/SQ/ST의 경우 메소드 알고리즘 에러와 멤버 데이터 범위 에러에 시험 사례에서 상태도로부터 원시 코드까지의 추적이 가능한 정보를 제공하고 있음을 알 수 있다. 상태도는 외부 이벤트 또는 내부 메소드에 대한 특정 상태를 유지하고 있는 클래스의 동적인 면을 검사하도록 지원한다. 또한 알고리즘에 의해 변경되는 클래스 속성들의 변이를 정의하고 어떤 시점에서의 속성 상태들을 검사할 수 있다. 하지만 누락한 속성이나 타입에 대한 정보를 지원하는 방법은 부족하다.

위 표에서 오류 위치를 추적하기 위한 정보와 검출된 오류의 개수의 차이가 높은 수준보다는 낮은 수준에서 더 많다는 것을 확인했다. 이것은 블랙 박스 시험 유형의 단점을 보이고 있는 부분이다. 블랙 박스 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. 본 논문에서 제안한 시험 기법은 시험 사례를 생성하는 동안 각 추상 수준에 대한 시험 정보들을 추출하기 때문에 추적가능성과 줌인 기능을 제공한다.

7. 결 론

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. 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.

참 고 문 헌

[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, pp. 194-208, 2001.

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

[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.

[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*, pp.383-395, 2000.

[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, pp.250-259, 1999.

[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, pp.94-10, 1994.

[8] B. Ramesh, “Factors influencing requirements traceability in practice,” *Communications of the ACM*, 41(12), pp.34-4, 1998.

[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, pp.213-22, 2006.

[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), pp.95-127, 2003.

[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, pp.250-259, 1999.

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

[14] Pilskalns, A. Andrews, R. France, and S. Ghosh, "Rigorous Testing by Merging Structural and Behavioral UML Representations," Sixth International Conference on the Unified Modeling Language, San Francisco, *LNCS 2863, Springer*, pp.234-248, 2003.

[15] P. Frohlich and J. Link, "Automated Test Case Generation from Dynamic Models," *14th European Conf. OOP(ECOOP' 2000), LNCS 1850, Springer*, pp. 472-492, 2000.

[16] P. Graubman and E. Rudolph, "Testing of UML models with Emphasis on Speial UML Language Features like Sequence Diagrams," *Conf. Unified Modeling Language(UML' 2000), LNCS 1939, Springer*, pp.32-46, 2000.

[17] M. d. M. Gallard, P. Merino, E. Pimentelis, "Debugging UML Designs with Model Checking," *Journal of Object Technology*, 1(2), PP. 101-117, 2002.

[18] A. Egyed, "A scenario-driven approach to traceability," International Conference on Software Engineering archive Proc. the 23rd International Conference on Software Engineering table of contents, Canada, pp.123-132, 2001.

[19] A. Egyed, "SUPPORTING SOFTWARE UNDERSTANDING WITH AUTOMATED REQUIREMENTS TRACEABILITY," *International Journal of Software Engineering and Knowledge Engineering, 15(5)*, pp.783-810, 2005.

[20] Fasolino, A.R and Visaggoi, G, "Improving software comprehension through an automated dependency tracer," Fasolino, A.R.; Visaggio, G. Proc. Seventh International Workshop, Program Comprehension, pp.58-65, 1999.

서 광 익

e-mail : bradseo@dongguk.edu
2002년 동국대학교 컴퓨터공학과(학사)
2004년 동국대학교 대학원 컴퓨터공학과 (공학석사)
2006년 동국대학교 대학원 컴퓨터공학과 (박사수료)
관심분야 : 소프트웨어 테스트, 소프트웨어 품질, 프로세스

최 은 만

e-mail : emchi@dgu.ac.kr
1982년 동국대학교 전산학과(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~현 재 동국대학교 컴퓨터공학과 교수
관심분야 : 객체지향 설계, 소프트웨어 테스트, 프로세스와 메트릭, Program Comprehension, AOP