研究論文

소프트웨어 시험 전략과 신뢰도 모델적용 연구

문숙경
목원대학교 정보통계학과

A Study of the Software Testing Methods and fitness of the Reliability Models

Soog Kyung Moon
Department of Information and Statistics, Mokwon University

Keywords : Software Reliability Model, Software Testing, Fitness, Fault, Failure, Testing Methods, Model Assumption

ABSTRACT

Software testing during development and operation should exercise to obtain the desired software quality and leave failure data set. So far, many software reliability models are classified and can be used to measure a software reliability only based on its failure history. But, in practice, developers or testers of software systems must decide which existing software reliability model can be fitted. In this paper, we will show that an appropriate reliability model can be selected by considering relations between characteristics of each testing environment and models' assumptions. Several methods of software testing are presented and discussed. Also, unit test, integrated test, function test and system test that are sequentially exercised during development will be introduced.

1. 서론

성공적인 소프트웨어는 그 소프트웨어에 에러들이 없음을 나타내는 것이다. 소프트웨어 시험이란 에러를 발견할 의도로 그 소프트웨어를 실행시키는 과정이며, 성공적인 시험은 아직 발견하지 못한 에러를 찾아내는 것이다. 그러므로 시험의 목표는 최소한의 시간과 노력으로 다른 부류의 에러를 체계적으로 검출하려는 방법들을 설계하는 것이다. 실제로 소프트웨어 개발에 종사하는 여러 기관들은 개발된 전체 프로젝트 노력의 40% 정도를 시험에 소모한다고 보고하고 있다. 따라서 소프트웨어 시험의 중요성, 그리고 소프트웨어의 품질과 관련되어 그것이 가지는 함축성은 아무리 강조해도 지나치지 않을

것이다. 소프트웨어 시험의 중요성을 인식하고 지난 수십 년 동안 이 분야에서 많은 연구 활동들이 이루어져왔다. 주요한 연구 활동들은 기초이론 정립, 시험 방법론, 시험 도구 개발 등의 분야들이다.

이처럼 소프트웨어 시험의 궁극적인 목적은 에러를 제거하여 품질의 향상을 도모하는데 있지만, 제한된 시간 내에 모든 에러를 제거한다는 것은 사실상 불가능하다는 것이다. 대부분의 소프트웨어 신뢰도 모델에서는 프로그램에 잔존하는 에러 수를 예측하여 시험의 중단 시점을 결정할 수 있도록 하여준다. 시험 결과 얻어지는 에러 및 수정되어지는 에러 정보를 이용하여 프로그램 내에 잔존하는 에러 수를 예측하는 신뢰도 모델과 시험과는 불가분의 관계가 있는 것이다. 이처럼 지금까지 제안된 많은 소프트웨어 신뢰도 모델들은 나름대로 가정을 세우고 그 틀 속에서 복잡한 수학적 공식과 확률 이론들을 함축하고 있다. 하지만 무엇보다 중요한 문제는 프로그램 개발자나 시험 담당자들이 실제 그들 시험 환경과 가장 잘 부합되는 신뢰도 모델을 적용하기란 그리 쉬운 일은 아닐 것이다. 모델을 잘못 적용할 경우 신뢰도 결과치는 많은 오차를 수반하게되리라는 것은 자명한 일이기 때문이다. 그래서 각각의 소프트웨어 개발 및 시험 환경에 가장 부합되는 모델을 찾아 적용하여야만 정확한 예측이 가능한 것이다. 본 고의 목적은 실질적으로 각각의 소프트웨어 개발 현장이나 운용 현장에서 시험 결과 얻어지는 시험자료를 이용하여 소프트웨어 신뢰도를 측정하는 작업에 도움을 주기 위함이다

소프트웨어의 여러 가지 시험 기법 및 방법들을 2장에서 소개하고, 3장에서는 소프트웨어 개발 단계별 시험 종류 및 시험 전략들을 기술하고, 소프트웨어 신뢰도 모델에서 사용되어지는 가정들을 열거한 후 각각의 시험 환경과 모델과의 적합성 내용을 4장에서 논하고자 한다.

2. 시험 방식

소프트웨어 시험을 위해 다양한 시험 기법들이 개발되어 왔으며 이들은 개발자나 시험 전담반들에게 시험에 있어 체계적인 접근 방법을 제공해왔다. 더욱이 이들 방법들은 시험 중 혹은 완료 시기의 지침뿐 아니라, 에러 검출 능력을 증대시켜준다. 임의의 소프트웨어 제품은 시험 단계에 따라 black-box 시험이나 white-box 시험 방식 중 하나를 선택하여 시험한다. 소프트웨어를 내부 내용들이 보이는 white-box로, 또한 내부 내용을 볼 수 없는 black-box로 각각 간주한다고 해서 붙여진 이름으로서, 소프트웨어 내부 동작과 흐름을 알 수 있을 때는 white-box 시험 기법을, 그러하지 못할 때는 black-box 시험 기법이 주로 사용되어 진다. 본 고는 이들 각각에 대하여 개괄적으로 설명하며 각 시험 기법의 사례들은 참고 문헌 [1], [2]를 참조해 주기 바란다.

2.1 White-Box 시험

소프트웨어의 내부 동작을 사전에 알고 있을 경우 사용되어지는 기법으로서, 내부 동작이 명세서에 따라 수행되며, 모든 내부 흐름들이 적절히 실행되는 가를 보증하기 위해 실시되어진다. 즉, white-box 시험은 소프트웨어 내의 논리적 흐름 구조를 이용하여 시험하는 기법으로 다음과 같은 기본적인 시험

원칙들이 있으며, 여러 가지 종류의 시험 방법들이 연구되어져 왔다.

- 소프트웨어 내에 존재하는 모든 실행 가능한 개별적인 경로는 최소한 한번은 검증된다
- 모든 조건문에서 true측과 false측을 검토한다.
- 모든 loop은 최소한 경계값들에 대하여 시험된다.
- 소프트웨어 각각의 모듈내의 자료 구조를 검증한다.

2.1.1 실행문 시험(statement testing)

소프트웨어 내의 모든 실행문들이 최소한 한번은 실행되어야 한다.

2.1.2 분지 시험(branch testing)

분지 시험은 각 분지에서 true나 false 결과가 최소한 한번 수행되도록 시험하는 것이다.

2.1.3 경로 시험 (path testing)

Statement 시험이나 분지 시험은 간과될 수 있는 에러들이 존재한다. 이를 위한 해결책으로 모듈내의 모든 경로들이 수행되도록 설계하는 경로 시험이 고안되었다. 그러나 loop들을 포함하는 모듈에는 통상 무한한 경로들이 존재하므로 이 기법은 실제로 비현실적이다. 더구나 일부 경로 중에는 실행시킬 수 없는 경로들이 발생하게 된다. 여기서 실행시킬 수 없는 경로란 그 경로를 수행시킬 수 있는 입력 데이터가 존재하지 않음을 의미한다. 이러한 비현실성으로 인해 현재는 경로 시험보다는 약하지만 분지보다 강력한 기법들이 소개되고 있다. 이들은 주로 소프트웨어 제어 흐름을 분석하고 이를 근거로 하는 시험 방법들이다. 대표적인 기법들은 domain 시험, symbolic 시험, 분할 분석 방법(partition analysis method)등을 들 수 있다.

2.1.4 데이터 흐름 시험(data flow testing)

분지 시험과 경로 시험 사이의 간격을 좁히기 위해 제어 흐름이 아닌 데이터의 흐름을 분석하고 이들을 시험 기법으로 적용하려는 새로운 방향의 연구 활동들이 진행되고 있다. 이 기법은 데이터의 흐름에 따라 형성되는 경로들을 토대로 시험하는 것이며 제어 흐름에 기본을 두는 시험 기법보다 경로 시험 기법에 더 가깝다고 볼 수 있다.

2.2. Black-Box 시험

black-box 시험은 소프트웨어의 내부 흐름이나 내부 구조를 알 수 없을 경우에 주로 사용되는 기법으로서, 소프트웨어 내에 구현되어져야 할 모든 기능들을 기능 명세서에 수록된 대로 실행되어지는 지를 검증하는 시험 기법이다. 참고로 다음과 같은 몇 가지 기능 시험들의 예를 들 수 있다.

- 부정확한 기능이나 누락된 기능
- 인터페이스 상의 일관성 검증
- 성능 상의 에러 검증
- 초기화나 종료 시에 발생되는 에러 검증

기능들을 충분하게 시험할 수 있는 입력 데이터를 제공하는 기법이 주로 사용되어진다. 즉, 소프트웨어 내의 모든 에러를 발견하기 위해서는 입력 영역내의 모든 값들을 시험하여야 되는데 실질적으로 그 방법은 무한번의 시험을 요구하기 때문에 불가능하다. 그러므로 접근 방법은 입력 영역을 여러 개

의 부 영역으로 분할하여 그 부 영역의 대표 값이 각각의 부 영역 내에서 발생하는 많은 에러를 발견할 수 있다는 가정아래 영역에 대한 분할 방법과 분할된 영역 내에서 대표 값을 추론하는 방향으로 연구되어져 왔다.

2.2.1 등가 분할(equivalent valueanalysis)

부 영역을 나눌 때 대등한 값들을 가지는 입력 혹은 출력 부분들을 묶어 하나의 부 영역으로 그리고 이렇게 나눠진 부 영역별에서 대표 값으로 시험을 실시하는 방법이다. 등가 분할 기법은 임의로 시험을 선택할 수 있다는 점에서 선호되지만 아주 민감한 시험들은 간과할 수 있다는 약점이 있다.

2.2.2 경계치 분석(boundary value analysis)

분명하게 분석된 것은 아니지만, 경험적으로 상당수의 에러들이 입력 영역의 경계 조건 근방에서 발생된다. 경계 조건이란 입력 등가 영역과 출력 등가 영역의 경계 상, 위, 아래에 형성되는 상황이다. 경계 치에 대한 시험은 등가 분할 기법과 유사하나 대표 값을 선택하는데 있어서 경계 치에 가장 가까이 있는 데이터를 대표 값으로 선택하는 것이 에러를 발견하는데 더욱 민감하다는 경험에 입각하여 제시된 시험 기법이다. 따라서 경계치 분석 기법은 등가 분할 기법을 보충하는 시험 기법으로 이용될 수 있다.

경계치 분석이 정확히 실행만 된다면 가장 효율적인 시험 기법이 될 수 있으나 사실상 경계 조건이란 매우 미묘한 것일 수 있기 때문에 많은 고려를 해야만 한다.

2.2.3 원인-결과 그래프 기법(cause-effect graphing)

원인-결과 그래프는 매우 민감한 부분들을 체계적인 방식으로 시험 방법들을 추출해 내는 기법이며, 특히, 명세서들에 존재하는 애매함이나 불완전함을 발견할 수 있는 좋은 기법이다. 입력 조건들을 원인으로 간주하고 각 입력 조건들에 대해 실행된 결과 혹은 행위들을 결과로 표현되는데, 각각의 논리적인 조건과 대응하는 행위를 그래프로나 표 등으로 정밀하게 표현되는 시험 기법이다. 이 기법은 다음의 네 단계를 따른다.

- 원인들(입력 조건들)과 결과들(행위들)이 각 모듈에 대해서 작성되며, 각각에 대해 하나의 식별자가 배정된다.
- 원인-결과 그래프가 작성된다
- 그래프를 decision 테이블로 변환시킨다.
- decision 테이블의 각 rule들을 시험으로 변환시킨다.

3. 소프트웨어 시험 전략

절차적인 측면에서 볼 때 시험은 순서적으로 처리되는 네 개의 단계들로 생각할 수 있다. 크게 유니트 시험, 통합 시험, 기능 시험 그리고 시스템 시험으로 나뉜다. 초기에, 시험은 원시 코드로 구현된 소프트웨어의 각 모듈이 유니트로서 절절하게 기능을 수행하는 지를 보장하는 것에 중점을 두며 이러한 시험을 유니트 시험이라고 부른다. 모듈의 제어 구조상에 존재하는 특정한 경로들을 실행시키는 white-box 시험 기법을 많이 이용한다. 다음에, 유니트 시험이 완료된 모듈들은 완전한 소프트웨어 패키지를 생성하기 위해 통합된다. 통합된 프로그램에 대해 실시하는 통합 시험은 각각에 대한 검증뿐 아니라 합침으로 인해 프로그램이 제대로 잘 구성되었는지에 대해서도 함께 시험된다.

black-box 시험 설계 기법들이 이 기간 동안 선호된다. 그리고 정의 단계에서 설정된 기능 기준이 시험되어야 하는데 이러한 기능 위주의 시험을 기능 시험이라 한다. 기능 시험은 소프트웨어가 모든 기능이나 성능적인 요구 사항들을 충족시키는 지를 시험하는 마지막 보증 단계이다. 통상 black-box 시험 기법들만 기능 시험동안 이용된다. 마지막 시험 단계인 시스템 시험은 소프트웨어 공학의 영역을 벗어나 광의의 컴퓨터 시스템 공학 범주로 여겨진다. 일단 기능 시험이 끝난 소프트웨어는 다른 시스템 요소들(하드웨어, 사용자, 데이터베이스 등)과 결합된다. 시스템 시험은 모든 요소들이 적절하게 조화되어 전체 시스템 기능이나 성능이 달성되는 가를 검증한다.

3.1 유니트 시험

유니트 시험은 소프트웨어 설계의 가장 작은 단위인 모듈에 대하여 검증을 수행하는 단계이다. 실제 수행된 결과가 예상한 결과와 일치되는 가를 확인할 수 있는 기술이 요구된다. 유니트 시험시 사용하는 일반적인 시험 설계 기법은 2장에서 설명한 white-box 시험 기법들이 적용된다. 유니트 시험 과정은 모듈에 대한 명세서로부터 시험을 추출하고 예상 가능한 동작의 확인, 모듈의 실제 수행과 동작 과정의 탐지, 마지막으로 예상 결과와 실제 결과를 분석하는 단계로 이루어진다. 그러나 모듈은 타 모듈과 밀접한 관련성을 가지고 있기 때문에 유니트 시험을 하기 위해서 driver와 stub같은 시험 환경이 필요하다. 여기서 driver는 시험을 받아들여 모듈에 넘겨주고 관련된 결과를 출력하는 역할을 하며 stub는 최소한의 데이터 처리 기능과 인터페이스가 제대로 되었는가를 확인하는 기능을 한다. 유니트 시험에서 driver와 stub는 실제 완전한 기능을 갖고 있는 모듈이 아니고 단지 유니트 시험을 수행하는데 필요한 소프트웨어이므로 보다 세련된 stub의 요구는 많은 부담이 따른다. 따라서 여러 모듈과 관련성을 갖고 있는 모듈에 대한 구체적인 시험은 통합 시험에서 수행된다.

3.2 통합 시험

통합 시험은 유니트 시험이 끝난 모듈들을 설계 원칙에 따라 최종 소프트웨어 구조로 통합하면서 각 모듈간에 관계되는 인터페이스를 시험하는 단계이다. 점증적인 통합 방법에는 top-down 통합화 방법과 bottom-up 통합화 방법이 있다.

3.2.1 Top-Down 통합화

top-down 통합화는 주 모듈에서 제어 흐름 순서에 입각하여 하향식으로 모듈들을 통합하며 top-down 통합 시험은 시험과정 초기에 주요한 제어점이나 조건들을 검증함으로써 소프트웨어를 계층적으로 구조화하는데 필요한 결정사항이 잘못된 경우 초기에 확인되므로 후에 발생되는 심각한 에러를 방지할 수 있다는 장점이 있다. 그러나 top-down 통합 시험의 문제점은 상위 단계의 모듈을 충분하게 시험하기 위해서는 계층상 하위 단계 모듈의 처리가 요구되는데 아직 개발이 덜 된 하위 단계 모듈을 대신할 가상적 모듈인 stub에서 중요한 데이터를 제공할 수 없다는데 있다. 이에 대해 시험자는 아래 열거한 사항중 한 가지를 선택할 수 있다.

- 실제 모듈의 기능을 묘사할 수 있는 stub를 개발한다.
- stub들이 실제의 모듈로 대치될 때까지 이에 관련된 시험을 연기한다.
- 소프트웨어 통합을 프로그램 구조의 계층상 가장 낮은 단계에서 출발한다.

3.2.2 Bottom-Up 통합 시험

프로그램 구조의 계층상 가장 하위의 모듈에서 통합을 시작한다. 따라서 하위 단계에서의 프로세싱이 항상 가능하므로 가상의 모듈인 stub가 필요 없다. bottom-up 통합 시험 과정은 다음과 같은 절차로 진행된다.

- 가장 하위의 모듈들은 소프트웨어의 특별한 기능을 수행할 수 있는 cluster (build)로 묶는다.
- 시험의 입력과 결과를 조정할 수 있는 상위 개념의 모듈인 driver를 만든다.
- cluster를 시험한다.
- driver를 제거하고 cluster들은 보다 위의 level의 모듈들과 묶는다.

3.3 기능 시험

통합 시험에서 인터페이스에 관련된 에러들이 검출되고 그러한 에러들이 수정된 후 소프트웨어는 완전히 패키지화 된다. 그런 후에 기능 시험 단계가 시작될 수 있다. 기능 시험은 때로 validation test(확인 시험)로도 불리며 이 두 용어는 서로 혼동 없이 사용된다. 기능 시험이란 여러 가지로 정의될 수 있으나 여기서는 소프트웨어의 구현된 기능들이 사용자의 적절한 기대치에 부합되는가를 확인하는 과정으로 정의하기로 한다. 적절한 기대치들은 그 소프트웨어에 대해 사용자가 볼 수 있는 특성들을 기술한 문서인 소프트웨어 요구사항 명세서에 정의된다. 소프트웨어 기능은 요구사항들과의 일치성을 확인하기 위해 등가 분할, 경계치 분석, 원인-결과 그래프 등 일련의 black-box 시험 기법이 이용된다. 시험은 주로 모든 기능적인 요구사항의 만족 여부, 모든 성능 요구사항의 만족 여부, 문서화의 정확성 기타 다른 요구 사항(이식성, 호환성, 에러 복구, 유지 보수)의 충족 여부 등에 중점을 두고 시행되어진다.

3.3.1 알파-베타 시험

소프트웨어 개발자는 사용자가 소프트웨어를 실질적으로 어떻게 사용할 것인가를 예측하기가 어렵다. 수행문을 잘못 사용할 수도 있고, 데이터를 잘못 조합하여 사용할 수도 있으며, 시험자에게는 올바르게 보여지는 결과가 현장에서의 사용자에게는 이해할 수 없는 결과로 보여질 수 있기 때문이다. 대다수의 소프트웨어 제품 생산업자들은 사용자들만이 찾아낼 수 있다고 여겨지는 에러를 검출하기 위해 알파-베타 시험을 이용한다. 개발자가 있는 장소에서 통제된 환경 하에 사용자에 의해 수행되는 것을 알파 시험이라 하며 이 과정에서 발생하는 에러와 이용 상 문제점을 개발자는 기록하여 소프트웨어를 수정할 수 있는 자료로 삼는다. 한편, 베타 시험은 그 소프트웨어의 사용자에 의해 사용자가 있는 장소에서 개발자가 제어할 수 없는 환경 하에 그 소프트웨어를 실제로 사용하는 시험이다. 사용자는 베타 시험동안 발생된 모든 문제를 기록하고 이들을 개발자에게 주기적으로 통보한다. 베타 시험에서 보고된 모든 문제를 토대로 개발자는 그 소프트웨어를 수정하고 이를 제품화하여 전체 사용자 측면에서 release할 준비를 한다.

3.4. 시스템 시험

다른 시스템 요소(하드웨어, 사용자, 데이터베이스 등)과 결합된 소프트웨어가 모든 요소와 적절하게 조화되어 전체 시스템 차원에서의 기능이나 성능이 달성되는 가를 검증하는 시험이다. 시스템 시험은 흔히 잘못 이해되기도 하며 가장 어려운 시험 과정이다. 시스템 시험에서의 시스템이란 소프트웨어와 하드웨어 그리고 관련 정보(information) 등을 합친 최종 산물(object)을 지칭한다. 그래서 주로 시스템의 성능(performance)이나 용량(capacity) 혹은 보안(security) 시험 같은 시험을 수행한다.

시스템 테스트의 시험 활동들은 다음과 같이 분류된다.

3.4.1 성능 시험

성능 시험은 어떤 작업 부담이나 배치 조건하에서 소프트웨어들의 특정한 run-time 성능이나 효율에 중점을 두고 수행된다. 성능 시험은 흔히 부하 시험과 결합하여 수행되며, 프로세서 cycles와 같은 자원의 이용 정도를 측정하고, 수행 주기의 제어, 발생되는 event(인터럽트)의 logging 상태 등을 측정한다. 나아가 시스템의 악화나 가능성 있는 시스템 고장을 유도하는 상황을 찾아낼 수 있다. 예를 들어 전전자 교환기의 경우 다음과 같은 사항들이 시스템 성능으로 고려될 수 있을 것이다.

- 평균 dial tone 지연 정도
- 과다 집중 문제의 존재 여부
- dead lock의 가능성 여부

3.4.2 신뢰도 시험

시스템 시험 동안 그 시스템의 신뢰도에 대한 계량치가 계산되어야 하는데 예를 들어 교환기의 경우는 다음과 같은 요인들을 고려하여야 한다.

- mean down time
- mean time to failure
- mean time to recover

3.4.3 용량 시험

용량 시험은 방대한 양의 데이터를 소프트웨어가 처리할 수 있는 지를 시험하는 것이다. 즉, 용량 시험은 객관적으로 명시된 데이터의 양을 소프트웨어가 처리할 수 있는 가를 검증하는 과정이다. 예를 들면 교환기가 1000 호들을 처리하도록 설계되었다면 시험 목적이 만족되는 가를 검증하기 위해 1000 호들(심지어 1001 호들)을 발생시켜 본다.

3.4.4 부하 시험

부하 시험은 과부하 상태 혹은 비정상적 상태에서 소프트웨어의 동작 여부를 시험하게 된다. 과부하 상태란 짧은 시간 주기에서 방대한 양의 데이터를 가한다는 것을 의미한다. 예를 들면 교환기는 짧은 시간 주기 내에서 동시에 대량의 호들을 소프트웨어에 전달시켜 보거나 평균적으로 한두 번 발생하는 인터럽트를 초당 10회 이상 발생시켜 봄으로서 부하 시험이 가능하다.

3.4.5 보안 시험

보안 시험이란 불법적인 침입으로부터 시스템이 자신을 방어하는 보호체계를 검증하는 것을 말한다. 보안 시험을 수행하는 시험자는 시스템에 침입하는 역할을 수행하여야 한다. 즉, 암호를 얻고자 시도하거나 시스템의 방어책을 허물어뜨릴 시도도 하며, 시스템을 조작하여 다른 사용자에게 서비스를 거

부하도록 만들어 보기도 하며, 또 시스템 에러를 발생시킬 의도로 복구 도중에 침입을 시도하기도 하고, 시스템 엔트리의 키를 찾기 위해 정보를 진열시켜 보는 등의 시험을 수행한다.

3.4.6 배치 시험

교환 시스템과 같은 소프트웨어는 테이프, I/O 장비, line, trunk 같은 다양한 하드웨어 배치를 지원한다. 흔히 이들 장비의 가능한 배치들이 너무 많아서 각각에 대해 시험하는 것이 어렵지만 최소한 각 유형별 하드웨어 장비와 그들의 최소 또는 최대 배치 하에서 소프트웨어는 시험되어야 한다.

3.4.7 호환성 시험

개발된 많은 소프트웨어는 전혀 새로운 것이 아니며 그들은 존재하는 시스템과 호환성을 가질 수 있어야 하는 목적을 가질 수 있다. 예를 들어 새로운 교환기는 기능들이 부족한 몇몇 교환기의 대치품일 수 있으며 이러한 교환기들과 상호 호환적이어야 한다.

3.4.8 복구 시험

복구 시험은 어떠한 fault(소프트웨어 에러, 하드웨어 고장, 데이터 에러)가 발생했을 때 이에 대해 복구할 수 있는 기능과 주어진 시간 내에 처리 능력이 회복될 수 있어야 한다. 어떤 경우에는 시스템이 fault-tolerant해야 한다. 즉, 한 fault의 발생으로 인해 시스템의 전 기능들이 중단되는 사례가 발생해서는 안 된다. 따라서 복구 시험은 여러 가지 방식으로 소프트웨어의 고장을 강제로 발생시키고 이에 대한 복구가 적절하게 수행되는지를 검증하는 시험이다. 예를 들면 소프트웨어 에러들로부터 시스템이 복구되는 가를 시험하기 위해 의도적으로 에러들이 주입될 수 있다. 하드웨어 고장은 모사될 수 있으며 데이터 에러들은 시스템의 재시동을 분석하기 위해 의도적으로 발생되거나 모사될 수 있다.

만약 복구가 시스템 자체에서 자동적으로 수행되면 재 초기화, 검침 체계, 자료 복구와 재시동 등이 정확하게 수행되는 지를 시험하고, 복구가 사용자에 의해 수동적으로 수행되는 시스템의 경우에는 수리 작업이 주어진 시간 내에 수행될 수 있는 지를 시험한다.

3.4.9 기억장소 시험

경우에 따라 소프트웨어는 기억장소 특성을 가질 수 있다. 이와 같은 시스템은 많은 양의 주/보조 기억 장치를 이용한다. 기억장소의 다른 제약 조건하에서 소프트웨어는 시험되어야 한다.

4. 신뢰도 모델에서의 가정들

모델을 사용하기 위해선 기본적인 가정들이 필요하다. 때론 이런 가정들이 실제 상황과 거리감이 있는 것들이 꽤 많다. 그래서 이런 가정들을 현실에 가깝도록 다듬어 새로운 모델이 만들어지곤 한다. 기존 신뢰도 모델들에서 주로 사용되어진 가정들은 다음과 같다.

첫 번째 가정으로, 모든 에러가 같은 정도의 중요도를 가진다고 본다. 즉, 시스템에 치명적인 고장을 일으키는 에러나 한 두 개의 철자가 잘못 쓰여진 에러도 같게 취급을 한다. 그리고 두 번째 가정으로서, 모든 에러는 발견되는 즉시 수정되어진다고 본다. 수정되어지지 않으면 프로그램을 실행시키지 않는다고 본다. 그리고 에러 수정 외엔 프로그램

을 변경하지 않는다고 가정한다. 또한 중요한 가정으로서, 발견된 에러를 수정할 시 새로운 에러들을 만들지 않는다고 가정한다. 이 가정은 현실적으로 볼 때, 대체로 새로운 에러들이 충분히 만들어질 수 있으므로 많은 학자들에 의해 수정되어졌다. 대표적으로 Miyamoto[5]는 그의 논문에서 새로운 에러들이 만들어질 수 있다고 가정하고, 고장 발견률 $z(t)$를 최종 시험기간에 프로그램에 잔존하는 에러 수에 비례한다고 하였다. 그리고 소프트웨어를 시험하기 시작할 때, 소프트웨어 내에 N개의 결함이 존재하고, 이들 각각의 에러들은 서로 독립적이라는 사실이다.

앞에서처럼 일련의 시험 기간에서 행해지는 시험들은 각각 저마다의 특징과 성격을 가지고 있다. 그러므로 시험 순서의 랜덤화 여부, 고장 발생 형태, 에러에 대한 수정이 즉시 이루어질 수 있는지의 여부 및 에러 수정 시 또 다른 에러 유발 유무 등이 시험의 각 단계마다 다를 수 있다. 그러므로 시험의 각 단계마다 적용되어질 수 있는 신뢰도 모델이 다를 수 있는 것이다.

4.1. 시험의 랜덤화

대부분의 많은 모델에서 고장을 일으키는 에러 간에는 서로 독립이라는 사실을 가정하고 있다. 즉, 발견된 에러 간에는 서로 독립이라는 사실이다. 이는 시험의 랜덤화와 깊은 연관성이 있는 것으로 시험 순서들이 거의 정해진 틀에 의해 이루어지는 경우엔 랜덤화로 볼 수 없을 것이다. 예컨데, 유니트 시험, 통합 시험, 기능 시험 등과 같은 시험들에서는 시험의 랜덤화 보다는 틀에 짜여진 순서에 의해 시험이 주로 실시되므로 발견된 에러 간의 독립성 가정은 이런 시험 종류에는 위배되는 경우가 대부분일 것이다. 특히, 유니트 시험 같은 white-box 시험 기법이 주로 이용되는 경우엔, 발견되는 에러 간에 연관성이 존재하는 경우가 오히려 많다.

4.2 각 에러의 고장 발생 확률

신뢰도 모델에서 많이 가정하는 또 다른 사실은 소프트웨어 내에 존재하고 있는 각각의 에러가 발견될 확률과 고장을 발생시킬 확률은 같다고 본다는 것이다. 또한 발생되는 고장의 종류도 시스템에 미치는 영향에 따라 치명적인, 보통, 가벼운 것으로 구분되어지는 것이 더 현실적이나 모두 같다고 간주하는 것이 대부분의 모델들의 가정이다. 이 또한 유니트나 통합 시험 같은 시험 단계에선 적합하지만 기능 시험, 시스템 시험 같은 시험 단계에선 고장 유형이 점점 복잡하고 복합적인 양상을 띤다. 하드웨어와 결합하여 시스템 차원에서 시험을 실시하는 시스템 시험 단계에서는 고장에 대한 원인인 에러가 소프트웨어인지 하드웨어인지 혹은 인터페이스 부분인지 등이 명확히 밝혀지지 못하는 경우도 발생한다. 기능, 혹은 시스템 시험 같은 단계에서는 단순히 발생하는 에러를 헤아리는 모델을 적용시키는 것은 재고해 볼 일일 것이다.

4.3 에러에 대한 수정 시간과 새로운 에러 발생 유무

대부분의 신뢰도 모델에서는 에러 발생 시 수정하는데 걸리는 시간을 무시하거나 즉시 수정이 이루어진다고 가정하고 있다. 또한 수정 시에 새로운 에러들이 발생하지 않는다

라는 가정을 하고 있다. 물론 이러한 기존의 가정들에 반하는 여러 종류의 새로운 모델들을 탄생시켰다. 예컨데, Goel and Okumoto의 imperfect debugging model은 수정 시에 새로운 에러가 만들어질 수 있다는 가정을 첨가한 모델로서, 그리고 Schneidewind는 에러 발생 시 수정하는데 걸리는 시간까지 고려한 모델을 제안하였다. 주로 단순한 그리고 수정이 용이한 에러들이 발견되는 유니트나 통합 시험 단계에서는 위의 기존 가정들이 상당한 정도 만족하는 편이나 대체로 복잡한 양상을 띠는 기능, 시스템 시험 단계에서의 에러들은 즉시 수정이 어렵고 나아가 수정 시 새로운 에러가 생성될 수 있는 여지가 많다.

4. 결론

본고는 시험 방법론의 개념에 중점을 두고 시험의 설계 기법들과 시험 전략에 대해 개략적으로 서술했다. 시험의 설계 기법의 주요한 목적은 소프트웨어의 에러들을 가능한 한 많이 검출하기 위한 것이다. 이러한 목적을 성취하기 위해 white-box 시험은 소프트웨어의 제어 구조에 중점을 두며 통상 작은 소프트웨어 요소(모듈, 작은 모듈군)에 적합하다. 반면, black-box 시험은 소프트웨어의 내부 동작에 관계없이 기능적인 요구사항들을 가능하게 되며 대규모 소프트웨어 시험에 적합하다.

앞서 언급된 바와 같이 소프트웨어 시험의 목적은 에러를 검출하는 과정이다. 양질의 소프트웨어에 대한 요구는 더욱더 체계적인 시험을 요구하게 된다. 그러한 목적을 충족시키기 위해 유니트 시험, 통합 시험, 기능 시험과 시스템 시험이라는 점진적 시험 단계들로 구분하였고, 각각의 시험 단계는 시험 설계 기법과 적절히 결합되어 체계적인 시험을 수행하게 된다. 이처럼 일련의 시험 기간에서 행해지는 시험은 각각 저마다의 특징과 성격을 가지고 있다.

기존의 신뢰도 모델에서 주로 많이 사용되어진 가정들, 즉, 시험 순서의 랜덤화 여부, 고장 발생 형태, 에러에 대한 수정이 즉시 이루어질 수 있는지의 여부 및 에러 수정 시 또 다른 에러 유발 유무 등이 시험의 각 단계마다 다를 수 있다. 유니트, 통합, 기능 시험까지 정도는 시험이 대체로 정해진 순서에 의해 실시되므로 시험 순서의 랜덤화라는 가정은 대체로 맞지 않는 편이다. 그리고 나머지 가정들은 오히려 기능이나 시스템 시험 단계에서 부적절한 가정이었다. 이처럼 시험의 제 단계 중에서 중요하고도 어려운 시험인 기능 시험이나 시스템 시험 단계에서는 특히, 단계의 성격에 어울릴 수 있는 가정들을 사용하는 신뢰도 모델들을 개발하여야할 것이다. 본 고의 취지는 구체적인 수학적 모델을 제시하는 것 대신에 각 시험 단계별로 실제 시험 환경을 살펴보고 각 시험 환경별 사용 가능한 모델을 적용하기를 원하는, 실제적인 것을 추구하자는 것이다. 향후 각 시험 단계별로 적용 가능한 구체적인 모델을 개발하는데 미력하나마 도움이 되길 바란다.

참고문헌

[1] Myers, G., The Art of Software Testing, Wiley, 1979.

[2] Pressman, R. S. Software Engineering, A Practitioner's Approch, McGraw-Hill

Inc., 1987.

[3] Miller, E. and W. E. Howden, Software Testing and Validation Techniques, 2nd ed., IEEE Computer Society Press, 1981.

[4] B. Chandrasekaran and S. Radicchi, Computer Program Testing, North Holland, 1981.

[5] McCabe, T., "A Software Complexcity Measure," IEEE Trans. on Software Eng., Vol.2, Dec., 1976, pp. 308-320.

[6] L. J. White and E. I. cohen, "A Domain Strategy for Computer Program Testing", IEEE Trans. on Software Eng., SE-6, 3, May 1980, pp. 236-246.

[7] D.J. Richardson, L.A. Clarke, and D. L. Bennett, "A Partition Analysis Method to Increase Program Reliability," Fifth International conference on Software Eng., March, 1981, pp. 244-253

[8] W. E. Howden, "Reliability of the Path Analysis Testing Strategy", IEEE Trans. on Software Eng., SE-2, Vol. 3, Sept., 1976, pp. 208-215.

[9] Janusz W. Laski and Bogdan Korel, "A Data Flow Oriented Program Testing Strategy," IEEE Trans. on Software Eng., SE-9,Vol.3, May 1983, pp. 347-354.

[10] Simeon C. Ntafos, "On Required Element Testing," IEEE Trans. on Software Eng., SE-10 Vol. 6, Nov. 1984, pp. 795-803.

[11] S. Rapps and E.J. Wevuker, "Selecting Software Test Data Using Data Flow Information," IEEE Trans. on Software Eng., SE-11, Vol.4, April 1985, pp. 367-375.

[12] Panzl, D., "Automatic Software Test Driver," IEEE Computer, 11(4), 1978, pp. 44-50.

[13] Bell Labs, System Test Tools- 5ESS System Test Methodology(Ⅱ), 1985.