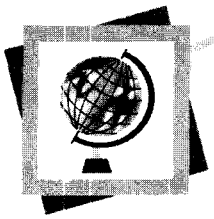
| 특집 01 |



# 소프트웨어 시험 기술 현황

김상운 · 마유승 · 신종민 · 이숙희 · 권용래
(한국과학기술원)

목 차

1. 서 론
2. 시험 충분성 기준(Test Adequacy Criteria)
3. 충분성 기준의 원칙
4. 시험 케이스 설계(Test Case Design)
5. 생명 주기에 따른 시험 기술

## 1. 서 론

소프트웨어의 응용 분야가 꾸준히 확장되고 소프트웨어의 결함으로 인하여 막대한 경제적 손해가 유발되거나 인명 손실로 이어지는 사례가 빈번히 보고되고 있다. 특히 소프트웨어가 실시간 임베디드 시스템 등의 중요한 요소로 자리잡고 시스템 전체의 평판이 소프트웨어의 정상적인 작동에 좌우되는 사례가 증가함에 따라 소프트웨어의 품질에 대한 요구가 현저히 높아가고 있으며 머지 않아 소프트웨어 품질이 소프트웨어 산업에서 결정적인 성공기준이 될 것으로 예상되고 있다[1]. 조사에 따르면 시험은 소프트웨어 개발 비용의 50퍼센트 이상을 점유한다고 하는데 이 비율은 critical 소프트웨어의 경우에는 더욱 높아질 것으로 예상된다.

소프트웨어 품질을 원하는 수준으로 유지하려면 소프트웨어 제작 과정에서 결함이 소프트웨어에 유입되지 않도록 노력하는 것이 중요하고 이런 노력 끝에 완성된 소프트웨어를 실행시켜 보고 만일 결함이 있으면 이것을 제거해야 할 것이다. 소프트웨어의 결함을 사전에 예방하기 위하여 최신 소프트웨어 개발 기법을 도입 적용하고 독립적인 확인 및 검증기법(Validation & Verification)과 같은 정적 분석에 기반을 둔 기법이 활용되고 있다. 또한 일반적인 공학 분야에서 널리 사용되고 있는 품질관리기법을 도입하여 나름대로의 성과를 거두고 있다. 그러나 일단 소프트웨어에 유입된 결함을 제거하는 데에는 소프트웨어 시험이 거의 유일한 수단이다. 그뿐 아니라 소프트웨어 시험은 구현된 소프트웨어를 제어 가능하고 실제 운용 환경에 가까운 상황하에서 실행시켜 정상적 작동 여부를 확인해 볼 수 있다는 점에서 다른 어떠한 품질확인 방법이 가질 수 없는 장점을 보유하고 있다.

본 논문에서는 소프트웨어 시험의 기본적인 충분성 기준에 대해 소개하고, 초기 시험 기법부터 근래 적용되는 다양한 분야의 시험 기법을 정리하여 소개하며, 향후 개선 방향에 대해 제안한다.

소프트웨어 시험은 구현된 소프트웨어를 입력 데이터 즉, 시험 케이스를 가지고 실행시킨 다음에 출력으로 얻어지는 결과가 예상했던 결과와 일치하는가를 살펴 보는 기법이다. 만일 예상대로의 출력이 얻어지면 소프트웨어의 구현이 명세서(specification)의 내용과 일치한다는 결론을 얻을 수 있으나 일치하지 않을 경우에는 원인을 찾아내 코드를 수정하여 원인을 제거해야 한다. 소프트웨어 시험의 일차적인 목표는 결함의 존재여부를 확인하는 것이지만 시험의 궁극적인 목표는 코드가 요구되는 품질 수준을 갖추었는지를 확인하는 것이다. 따라서 시험의 궁극적인 목표를 성취하려면 품질 요소를 시험 케이스와 관련을 지워 보아야 한다. 예컨대 코드의 정확성을 판단하려면 코드의 가능한 모든 실행 경로에 대하여 코드가 옳은 결과, 즉 명세서와 일치하는 결과를 출력함을 보여야 한다. 따라서 시험에서는 많은 수의 시험 케이스를 실행시켜보고 실행 결과를 분석한 다음에 최종적인 결론을 도출하게 된다.

구현된 소프트웨어를 실행시킬 때 사용하는 시험 케이스는 보통 명세서로부터 얻기도 하고 코드 자체를 바탕으로 얻기도 한다. 경우에 따라서는 시험 경험이나 흔히 검출되는 결함의 유형에서 힌트를 얻어 도출하기도 한다. 시험을 통하여 살펴 보고자 하는 코드의 부분을 실행시킬 수 있는 적절한 입력값을 준비해야 하는 작업과 출력을 비교할 수 있는 객관적인 수단 즉, 오라클을 준비하는 작업은 모두 명세나 코드를 충분히 이해해야 가능한 일이기 때문에 기술적으로 어려운 작업이다. 그러나 이와 같이 준비한 시험 케이스가 1) 시험 대상인 소프트웨어에 대해 요구되는 품질 수준을 판정하는 데 충분한지(Test Adequacy Criteria), 2) 살펴보고자 하는 소프트웨어의 모든 측면을 실행시켜 볼 수 있는 것인지(Test Coverage Criteria), 3) 준비한 시험 케이스를 모두 실행시키고 원하는 시험 결과를 얻었을 때 시험을 종결시킬 수 있는지(Test Completion Criteria)를 판단할 수 있어야 한다.

코드에 기반을 둔 소프트웨어 시험 기술은 상당한 발전을 거두어, 이제는 일단 요구되는 품질 수준이 정의되면 자원이 허용하는 범위에서 위에 서술한 시험 조건을 충족시킬 수 있는 시험 케이스를 산출해낼 수 있다고 볼 수 있다. 그리고 거의 대부분의 작업이 자동화 도구의 뒷받침을 받을 수 있다. 그러나 명세서에 기반을 둔 시험 기법은 소프트웨어 명세 기술의 수준에 좌우되기 때문에 아직도 실용 가능한 기법이 많지 않은 실정이다. Harrold는 컴포넌트 기반 시스템의 시험 기법, 설계, 요구명세서, 구조 명세서 등의 구현 단계 이전에 산출되는 문서를 바탕으로 하는 시험 기법 그리고 진화하는 소프트웨어 시험 기법을 당면한 연구 과제로 꼽고 있다[2].

이 논문의 2절에서는 각종 시험 기준을 중심으로 소프트웨어 시험의 이론적 기반을 서술한다. 3절에서는 2절의 시험 기준을 기초로 하고 명세서에 기반을 둔 시험 케이스 생성 방법과 코드에 기반을 둔 시험 케이스 생성 방법을 상세히 설명한다. 아울러 결함(fault)에 기반을 둔 시험 케이스 설계 기법을 뮤테이션 시험을 중심으로 소개한다. 4절에서는 소프트웨어 생명주기에 따라 단위 시험 기법, 통합시험 기법과 회귀 시험 기법을 소개한다. 다양한 소프트웨어 유형이 여러 분야에서 응용됨에 따라 이러한 유형의 소프트웨어를 효과적으로 시험하기 위한 연구도 활발히 진행되고 있다. 객체 지향 프로그램, 실시간 병렬 프로그램, 임베디드 소프트웨어, 웹 응용 프로그램 시험 등이 그것이다. 본문에서는 한정된 지면 관계로 이에 관련한 연구 결과를 일일이 소개할 수 없지만 객체 지향 프로그램과 컴포넌트 프로그램의 시험에 관해서는 순차적 시험 기법을 직접 적용할 수 있는 부분과 객체지향 프로그램 시험을 위해 특별히 연구된 결과를 구분하여 소개한다.

## 2. 시험 충분성 기준(Test Adequacy Criteria)

소프트웨어 시험에 있어서 시험 대상이 되는 프로그램에 대하여 얼마나 "충분한" 시험이 수행되었는지에 대한 의문이 발생할 수 있다. 이러한 질문에 답하기 위해서는 소프트웨어 시험에 있어서도 시험의 '질'에 대한 측정이 이루어져야 한다. 일반적으로 소프트웨어 시험의 품질을 측정하기 위해서는 시험에서 입력값으로 사용되는 데이터의 충분성(Adequacy)을 살펴보게 되는데, 이 때 시험 데이터 집합이 주어진 프로그램이나 명세에 대해서 충분성 여부를 논할 기준이 필요하게 된다.

시험 충분성 기준이란 소프트웨어가 얼마나 충분하게 시험되었는가를 판별하는 기준으로써 시험 완료 기준 또는 시험 데이터 선정 기준이라고도 한다. 소프트웨어 시험은 미리 설정한 시험 충분성 기준을 근거로 해당 프로그램 시험을 위해서는 어떠한 데이터를 선별해야 하며 얼마만큼의 시험을 수행해야 하는지 결정하게 된다. [3]에 의하면 이러한 시험 충분성 기준은 신뢰할 만해서(reliable) 오류를 발견하는 능력에 일관성이 있는 시험 데이터를 선별할 수 있고, 타당해서(validity) 프로그램 내에 존재하는 각각의 오류를 찾아낼 수 있는, 즉 의미 있는 시험 데이터 선별이 가능해야 한다.

## 3. 충분성 기준의 원칙

Weyuker[4]는 프로그램 기반 시험에 사용되는 시험 충분성 기준을 평가하기 위한 여러 원칙을 제시했다. 즉, 충분성 기준이라고 제안되는 것들이 적절한 기준이 되기 위해서는 어떠한 조건들을 만족해야 하는지를 판별할 수 있도록 여러 척도들을 제안하였다. 다음 8가지는 Weyuker가 처음으로 제안하였던 척도들이다.

- 적용가능성(Applicability)
  모든 프로그램에는 유한개의 적절한 시험 집합이 존재한다.
- 소모적이지 않은 적절성(Non-exhaustive Applicability)
  프로그램 전체를 철두철미하게 시험하지 않고도 프로그램을 충분히 시험할 수 있는 시험 집합이 존재한다.
- 단조성(Monotonicity)
  한 시험 데이터 집합이 어느 프로그램에 적합할 경우 해당 시험 데이터 집합의 부분 집합도 적합성을 가진다.
- 공집합의 부적합성(Inadequate Empty Set)
  공집합은 어떠한 프로그램의 시험에도 충분하지 않다.
- 비확장성(Anti-extensionality)
  의미가 같은 두 개의 프로그램이 있을 경우 한 시험 데이터가 한 프로그램의 시험에 충분해도 다른 하나의 프로그램 시험에 반드시 충분한 것은 아니다.
- 일반적 복수 변화성(General Multiple Change)
  구조상으로 유사한 두 개의 프로그램이 있을 때, 한 프로그램에 충분한 시험 데이터가 다른 프로그램의 시험에 반드시 충분한 것은 아니다.
- 비분해성(Anti-decomposition)
  한 프로그램이 다른 프로그램의 부분 집합일 경우 상위 프로그램에 적합한 시험 데이터 집합이 하위 프로그램에서는 반드시 적합하지는 않다.
- 비결합성(Anti-composition)
  독립적인 두 프로그램이 있을 경우, 한 프로그램에 적합한 시험 데이터 집합은 두 프로그램이 결합된 프로그램에 반드시 적합하지는 않다.

이후 Weyuker는 이전의 척도에 세 가지를 추가하여 척도를 보안한다[5]. 아래의 세 가지

척도들은 기존척도에서 해결하지 못한, 적절치 못한 시험 기준을 제거하는 데 목적을 두고 있다.

- 재명명성(Renaming)
  한 프로그램에 적합한 시험 데이터 집합은 이름만 바뀐 다른 프로그램에 적합하다.
- 복잡성(Complexity)
  N개의 시험 데이터가 있어야 적합하게 시험이 가능한 프로그램은 N-1개의 시험 데이터로는 적합한 시험을 할 수가 없다.
- 문장 커버(Statement Coverage)
  시험 데이터 집합은 프로그램에 존재하는 모든 문장을 시험 수행할 수 있어야 한다.

문장 충분성 기준이나 분기 충분성 기준의 경우는 위의 척도들 중 다섯 가지만 만족시키는 반면, 뮤테이션 충분성 기준의 경우는 위에 있는 척도들을 모두 만족 시킨다. Weyuker의 척도들은 시험 기준을 평가하는데 일관성이 부족하며, 충분히 형식화되어 있지 못하여서 프로그램 기반 시험 기준을 평가하는 척도로써의 적합하지 못하다는 이의 제기도 있었으나[6], 지금까지도 새로운 시험 충분성 기준을 개발하는 경우 이 척도들을 근거로 새로운 기준의 적절성 여부를 판단한다.

## 4. 시험 케이스 설계(Test Case Design)

### 4.1 분할시험(Partition Testing)와 임의적 시험(Random Testing)

임의적 시험은 시험 데이터를 얻기 위한 가장 간단한 방법으로, 프로그램의 입력 도메인으로부터 임의로 생성한다. 따라서 임의적 시험은 프로그램이나 시험 기법에 대한 지식이 없이도 시험 데이터를 얻을 수 있으나 시험 기준 (Test Criteria)을 만족시키기 어려운 단점이 있다.

분할 시험은 입력 도메인을 여러 서브도메인들로 분할한 후에 각 서브도메인에 대하여 서브도메인을 대표할 수 있는 하나의 시험 데이터를 사용하는 방법이다. 이때 하나의 서브도메인은 동일한 행위를 보이는 입력들로 이루어진 동치 클래스로 가정한다. 분할 시험은 프로그램의 특성에 따른 다양한 시험 기준을 만족시키는 시험을 수행할 수 있으며[7] 대부분의 시험 기법들은 분할 시험을 기반으로 연구되어 왔다.

일반적으로 분할 시험은 시험 기준에 따른 체계적인 시험을 가능케 하기에 임의적 시험보다 우월하다고 알려져 왔다. 그러나, 시험의 본질적인 목적인 오류(fault)를 찾는 능력에 대한 두 시험 기법의 비교 연구는 지난 20여년간 계속해서 이루어지고 있다. [8]에서는 프로그램을 동치 클래스들로 구분하는 것은 이론적으로 가능한 일이며, 시험 기법을 통해 생성된 서브도메인의 값들은 항상성을 유지하지 못함을 주장하였다. 특히 생성된 서브도메인들에 프로그램의 오류가 고루 분포되어 있을 경우 임의적 시험과 분할 시험의 오류 확인 능력은 거의 동일하다는 실험 결과를 제시하고 있다.

한편, [9], [10]은 서브도메인에 있는 오류의 수나 서브도메인의 크기 비율을 고려한 분할 시험은 임의적 시험보다 나쁘지는 않다는 결과를 제시하고 있다. 프로그램의 오류가 하나의 서브도메인에 집중되어 있는 경우에 임의적 시험보다 효과적이라고 주장한다. 그러나, [11]의 연구에서는 서브도메인들 간의 크기 또는 중요도가 극단적으로 불균형을 이룰 경우에는 두 시험 방법은 비슷한 결과를 보인다고 밝히고 있다.

이론적으로는 분할 시험이 보다 체계적으로 프로그램을 시험할 수 있기 때문에 동일한 개수의 데이터만을 이용할 수 있을 경우에는 임의적 시험보다 뛰어나다. 하지만, 저렴한 수행비용으로 많은 시험 데이터를 사용하는 경우에는 분할 시험과 비슷한 결과를 보인다.

## 4.2 명세에 기반한 시험 케이스 설계

명세에서 명시하는 바를 분석하고, 주어진 입력값과 출력 값의 관계를 추출하며, 추출된 정보를 기반으로 시험 케이스를 생성하는 방법을 명세 기반 시험 케이스 생성 방법이라 한다. 흔히 블랙박스(Black-Box), 기능성(functional) 시험이라고도 불리며, 특히 대규모의 단위 시험이나 시스템 시험을 수행하는 경우 유용하게 사용된다. 다음은 널리 알려진 몇 개의 명세 기반 시험 기법을 소개한다.

■ 경계값 시험(Boundary Value Test)

앞서 설명한 분할 시험에서 동치 클래스로 구분하는 방법의 한가지 휴리스틱으로 제안된 경계값 시험 기법은 입출력 변수들의 범위를 구하여 범위의 양측 경계값, 경계에 근접한 값, 그리고 정상값, 총 5개의 동치 영역으로 구분한다. 경계에 근접한 값들을 경계값 이전과 이후로 더 분리하는 강건 경계값 시험(Robust Boundary Value Test)도 이에 해당한다.

■ 결정 테이블 시험(Decision Table Test)

하드웨어 시스템의 시험을 위해 제안되었던 결합적 시험 기법 (Combinational Test)이 소프트웨어 시험에 적용된 사례다[3]. 프로그램의 논리적인 관계를 위주로 시험한다. 입력 변수들의 조건(Condition)들을 나열하고, 이에 반응하는 행위(Action)를 규칙(Rule)으로 표기하여 명세에서 기술하고자 하는 모든 입력 또는 출력에 대한 입력 공간을 올바르게 구분케 한다.

대표적인 2가지 명세 기반 시험들은 시험자가 직접 명세를 분석하고, 시험에 필요한 정보를 추출해야 하는 고비용의 작업을 필요로 한다. 따라서 좀 더 자동화된 시험을 위해 정형적인 명세 기법을 사용하게 된다. 수준별 정도가 다르지만, 정형 명세를 사용함으로 시험 케이스 생성까지의 단계의 대부분을 자동화시킬 수 있다. 하지만, 자동화의 정도는 명세 기법 자체가 아니라 시험자가 기술하는 명세의 수준에 따라 달라진다는 것을 주지해야 한다.

## 4.3 코드에 기반한 시험 케이스 설계

### 4.3.1 제어 흐름 충분성 기준(Control Flow Adequacy Criteria)

제어 흐름 시험은 시스템의 논리적 결정을 검사한다[12]. 제어 흐름 그래프를 통해 시험하고자 하는 프로그램의 모든 문장을 그래프의 노드(Node)에 관계 짓고, 문장 간의 제어흐름을 에지(Edge)로 표현한다. 그래프 완성 후 아래의 시험 기준에 따라 시험 케이스를 설계한다.

■ Statement 커버리지

그래프 상에 존재하는 모든 노드를 적어도 한번 수행할 수 있도록 시험 케이스를 설계한다. 이 기준은 IEEE 표준으로 권장되고 있지만, 시험하려는 시스템의 디자인과 코드의 분기 구조를 고려하지 않는 단점이 있다.

■ Branch 커버리지

그래프 상에 존재하는 모든 분기를 최소한 한번 시험하도록 시험 케이스를 설계한다. 이 기준은 코드 기반 시험의 최소 요건으로 인정되고 있지만, 결정을 이루고 있는 조건들 간의 의존성이나 결정의 복잡도는 고려하지 않는다.

■ Path 커버리지

제어흐름 그래프 상에 존재하는 모든 경로를 최소한 한번 수행하도록 설계한다. 제어 흐름 시험 기준 중 가장 완벽한 기준이지만, 한정된 자원 때문에 현실적으로 성취하기 어렵다.

■ Full Predicate 커버리지

한 결정 내의 모든 조건들이 가질 수 있는 모든 결과 값들을 적어도 한번씩 검사할 수 있도록 시험 케이스를 설계한다. 각각의 결정과

결정을 이루는 조건과의 의존성에 대해서는 고려하지 않는다.

■ Modified Condition/Decision Coverage

모든 조건자 커버리지를 만족시키면서 결정과 그 결정을 이루는 조건들간의 의존성을 시험할 수 있도록 시험 케이스를 설계한다.

위에서 설명한 시험 기준을 정의하는 중요한 목적은 시험의 효율성을 높이면서 시험 케이스들의 수를 최소화하는 것이다. 따라서 각 기준의 효율성을 비교하는 것은 중요하다. [13]에서는 제어 흐름 시험 기준과 자료 흐름 시험 기준의 효율성을 실험을 통하여 비교하였다.

#### 4.3.2 자료 흐름 충분성 기준(Data Flow Adequacy Criteria)

자료 흐름 시험은 시험 케이스를 선정할 때 프로그램 내 자료 흐름 사이의 관계를 이용한다. 즉, 프로그램 내의 모든 자료들에 대하여 각 자료의 선언과 사용 관계를 살펴볼 수 있는 시험 케이스를 요구한다. 중요한 자료 흐름 시험 기준에는 다음과 같은 것들이 있다. 그 외의 자료 흐름 시험 기준은 [15]에서 찾아볼 수 있다.

■ All-du Paths 커버리지

모든 변수들에 대해 값이 정의되고 사용되기까지의 모든 경로(definition-use path)를 시험할 수 있는 시험 케이스를 요구한다. 이 기법은 가장 강력한 자료 흐름 시험 기법이다.

■ All-Uses 커버리지

모든 변수들에 대해 값이 정의되고 사용되기까지의 모든 경로들 중 값이 한번만 정의된 경로들(def-clear path)을 적어도 한번 검사할 수 있는 시험 케이스를 요구한다.

■ All-Definitions 커버리지

모든 변수들에 대해서, 한 변수에 값이 정의될 때마다 그 정의된 값이 조건 사용 또는 계산 사용으로 사용되는 경로를 적어도 한번 검사할 수 있는 시험 케이스를 요구한다.

■ All-Predicate-Uses 커버리지

모든 변수들에 대해 각각의 변수들의 정의된 값들이 비교 연산에 사용될 때(predicate use) 될 때까지의 경로들을 적어도 한번씩 시험할 수 있도록 시험 케이스를 설계한다.

■ All-Computational Uses 커버리지

모든 조건-사용 커버리지와 비슷하게 모든 계산-사용만 검사한다.

### 4.4 오류기반 시험 기법(Fault-Based Test)

오류 기반 시험 기법이란 지정된 형태의 오류가 프로그램 내에 존재하는지 살펴보는 시험기법이다. 프로그램의 신뢰도를 추정하기 위해 제안된 에러 심기 (error-seeding)[16]기법을 바탕으로 한다. 프로그램의 신뢰도는 전체 삽입된 오류 중 발견된 오류의 비율로 신뢰도를 측정하였다.

뮤테이션 시험[17]은 오류기반 시험의 가장 대표적인 시험 기법으로 시험데이터의 효용성을 측정하기 위해 제안된 기법이다. 시험 데이터의 효율성이란 시험 데이터가 오류(fault)를 찾아내는 능력으로, 주어진 프로그램에 대해 고의적으로 오류가 삽입된 뮤턴트(mutant)와 원 프로그램(original program)에서 다른 결과를 내는 것을 효율적인 시험 데이터로 판단한다.

뮤테이션 연산자의 정의는 뮤테이션 시험에서 기본이 되는 연구 중의 하나이다. 뮤테이션 시험에서 오류의 삽입은 뮤테이션 연산자(mutation operator)들에 의해 이루어진다. 오류 기반 시험은 실제 자주 발생하고 검출이 쉽지 않은 오류를 대상으로 했을 때 효율적이다. 따라서 뮤테이션 연산자의 선택 및 정의는 매우 중요하다. 뮤테이션 연산자는 프로그램의 코드를 직접 변형시킴으로써 오류를 삽입하므로 프로그램 언어에 따라 연산자 집합들이 정의된다. 대다수의

연구들이 순차프로그램 언어에 대해 연구가 되어왔는데, C 언어와 Fortran 언어를 위해 정의된 연산자[18]가 대표적이다. 현재까지, 객체지향 언어에 대해서는 Java 언어에 대해서만 정의되었는데[19], 객체지향 언어의 특징인 상속, 다형성 등의 특성을 고려하고 있다.

일반적으로 뮤테이션 시험 기법은 거대한 수의 오류 프로그램들을 만들어낸다. 따라서 대량의 뮤턴트의 생성과 수행을 하는 데 많은 비용이 들어간다. 이를 해결하기 위해 수많은 비용 감소 기법이 제안되었다. 비용감소 기법은 크게 "Do fewer", "Do faster", "Do smarter"의 세가지 범주로 나뉜다. "Do fewer" 기법으로는 시험의 효율성을 유지하며 일부의 뮤턴트만을 사용하는 샘플링 기법[20]과 선택된 몇몇의 뮤테이션 연산자에 의한 뮤턴트를 사용하는 선택적 뮤테이션 (Selective Mutation) 기법[21]이 있다. "Do smarter" 기법은 시험 수행을 여러 기계에 분산시켜 수행시킴으로써 비용을 줄인다. 하드웨어를 이용하는 방법[22]이 이에 해당한다. "Do faster" 기법은 뮤턴트들의 생성 혹은 수행 시간을 가능한 최대로 단축시키려는 기법이다. 컴파일러가 통합된 뮤테이션 시험 기법[23]과, 여러 뮤턴트를 하나의 뮤턴트로 수집하여 뮤턴트 들의 컴파일 시간을 줄이는 MSG 기법[24]이 이에 속한다. 최근 Java 프로그램을 대상으로 바이트 코드에 직접 오류를 삽입하여 컴파일 시간을 없애는 방법[25]이 개발되었다.

### 4.5 상태기반 시험(State-based Test)

전통적인 절차적 프로그램에서 그 프로그램의 행위는 일반적으로 프로그램 내의 수행 경로에 의해서 구분되며, 이러한 수행 경로는 프로그램의 입력값에 의해서 결정된다. 이러한 프로그램의 행위는 프로그램의 입력값에 의해서 행위가 결정되기 때문에, 프로그램의 입력값을 시험데이터로 생성하여 이용한다. 그러나, 리액티브 소프트웨어 (Reactive Software)나 객체 지향 프로그램의 객체같은 경우에는 입력값만으로 프로그램의 행위를 결정지을 수 없으며, 프로그램의 상태에 따라 입력이 어떤 상태에서 입력값을 받았는지에 따라서 다른 행위를 보인다. 이와 같은 프로그램의 명세나 프로그램의 모델은 유한 상태 기계(FSM)와 같은 상태 전이 다이어그램(state transition diagram)으로 주로 표현되며, 프로그램의 행위는 프로그램의 근원 상태 (Source State)와 입력값의 쌍에 대해서 프로그램이 도달하는 목적 상태(Target State)와 상태 전이 시의 출력값으로 정의된다.

상태기반 시험 (State Based Test)은 상태를 가지는 프로그램을 대상으로 하는 시험 기법으로, 입력값과 상태를 고려하여 시험 데이터를 생성한다. 시험 데이터는 근원 상태와 입력값의 쌍으로 구성되며, 이를 이용하여 근원 상태의 입력값에 대해서 잘못된 결과값을 출력하는 출력 오류와 잘못된 목적 상태에 도달하는 전이 오류 (Transfer Fault)를 확인하는 것을 목적으로 한다[7][8].

상태 기반 시험을 실행하는 데 있어 고려사항은 구현된 프로그램에서 생성된 시험 데이터 수행을 위한 근원 상태를 고정하는 것이 어렵다는 것이다. 일반적인 상태 기반 시험기법에서는 시험 데이터로 근원 상태를 생성하는 대신에 시작 상태에서 근원 상태에 도달할 수 있는 입력값의 시퀀스를 생성한다. 이를 위해 상태 전이 다이어그램에 다양한 커버리지 기준을 적용한다. [26]에서는 제어 중심의 커버리지 기준으로 상태 커버리지 기준, 전이 커버리지 기준, 경로 커버리지 기준 등을 제안하고 있으며, [27]에서는 데이터를 중심으로 하는 D-U 커버리지 기준을 제안하고 있다.

현재의 상태 기반 시험에 대한 연구들은 확장된 유한 상태 기계(Extended FSM)로부터 어

떻게 시험 데이터를 생성하는 방법에 관심을 가지고 있다. 현재에 리액티브 소프트웨어의 명세 언어로 널리 사용되고 있는 Statecharts와 같은 명세 언어들은 FSM의 표현적 한계로 인하여, 병행성 및 계층적 구조와 같은 다양한 확장된 특성을 사용하고 있다. 이러한 확장된 특성들은 프로그램의 행위를 기술하는 것을 용이하게 한다. 그러나, 이러한 확장된 특성들로 인하여 기존의 커버리지 기준을 명세나 시스템의 모델에 바로 적용하기가 어려운 문제가 발생한다. 이러한 연구들의 기본적인 아이디어는 확장된 유한 상태 기계로 기술된 프로그램의 행위에 대해서 동일한 행위를 기술하는 확장된 특성이 없는 유한 상태 기계로 변환한 후에 기존의 커버리지 기준을 적용한다는 것이다[28]. 이때, 확장된 특성으로 기술된 행위를 어떻게 변환할 것인가 하는 문제는 확장된 특성이 가지는 특징과 그것으로 기술된 행위의 특징에 따라서 결정되며, 이를 결정하는 것이 상태 기반 시험에 대한 현재의 연구에서 이루어지고 있다.

## 4.6 객체 지향 기반 시험(Object-Oriented Test)

객체지향 프로그램은 순차프로그램과 구조적, 기능적 측면에서 차이가 있어서 순차 프로그램을 대상으로 하는 기존의 시험기법을 객체지향 프로그램에 맞게 변용하거나, 추가적인 시험 기법들이 연구되었다. 우선 순차 프로그램과 객체지향 프로그램들은 시험 수준에서 차이가 있다. 객체지향 프로그램의 시험에서는 크게 알고리즘 수준, 클래스 수준, 클러스터 수준, 시스템 수준의 시험 단계[29]로 구분될 수 있다. 이 중 알고리즘 수준과, 시스템 수준의 시험은 순차 프로그램에서의 단위 수준, 시스템 수준의 시험과 비슷하므로 기존의 기법을 그대로 사용한다. 클래스 수준, 클러스터 수준 시험에 대해서는 객체지향적 특성들이 반영이 되어 많은 연구가 진행되어 왔는데, 대표적인 시험기법으로는 데이터 흐름 시험 기법으로, 많은 연구들이 수행되어 왔다. 추가적인 특성으로는 상태 의존적 행위, 캡슐화(encapsulation), 상속(inheritance), 다형성(polymorphism), 동적 바인딩(dynamic binding) 등을 고려하여 시험한다.

■ 상태 의존적 행위, 캡슐화

객체의 상태는 메소드의 입력과 출력의 일부분으로, 메소드의 수행으로 인해 객체의 상태가 변해간다. 따라서 객체가 올바른 상태값을 가지고 있는지, 메소드 수행으로 인한 객체 상태간의 전이가 제대로 되는지를 조사해야 한다. 일반적으로 객체 상태는 상태 모델(state model)로 기술되므로, 상태기반 시험 기법들을 한 개의 클래스를 대상으로 하여 객체지향 프로그램에 맞게 적용시키는 기법[30]이나 상태모델의 확장[31]에 관한 연구가 이루어졌다.

■ 상속, 다형성, 동적 바인딩

상속은 객체지향 프로그램 개발 및 시험에서 재사용이라는 장점을 제공한다. 상속은 특히 회귀 시험에 유용하게 이용될 수 있는데, 상속을 이용하여 시험데이터의 재사용을 최적화하기 위한 많은 연구들[32]이 수행되었다. 상속은 다형성, 동적 바인딩 개념과 같이 작용하여 새로운 타입의 오류들[33]을 만들어낸다. 이러한 오류들을 검출하기 위한 기법들[19]은 다형성에 의해 변할 수 있는 클래스의 모든 가능한 형태를 시험해 보도록 권장한다.

## 4.7 컴포넌트 시험(Component Testing)

컴포넌트 기반 소프트웨어는 개발 형태나 시험 단계의 유사성으로 보아 기존의 객체지향 소

프트웨어의 시험과 유사하게 시험 기법들이 접근가능 하지만, 단위 컴포넌트를 개발하는 컴포넌트 개발자와 개발된 단위 컴포넌트를 조립하여 소프트웨어를 구현하는 컴포넌트 사용자가 분리되어 있다는 점, 그리고 개발자와 사용자 간에는 구현된 컴포넌트와 함께 최소한의 정보가 제공된다는 점으로 인해 기존의 시험 기법들을 쉽게 적용시킬 수 없다.

컴포넌트 개발자 측면에서는 개발과정에 있어 생성되는 문서나 시험 관련 정보의 이용이 가능하여 기존에 제안된 여러 시험 기법을 적용할 수 있다. 반면에 컴포넌트 사용자들은 구현된 컴포넌트와 간략한 명세를 제공받게 되어 시험에 필요한 충분한 정보를 구할 수 없으므로, 한정된 정보를 이용하는 시험 기법들이 제안된다. Voas는 개발자와 사용자가 아닌 독립적인 제3자에 의해서 시험을 수행하고, 수행된 결과를 기반으로 품질을 측정하자는 구조를 제안 한다[34]. 이러한 구조에서는 컴포넌트 개발자는 독립적인 시험 기관에서 사용하는 시험 기법을 적용가능할 만큼의 정보를 제공하고, 컴포넌트 사용자는 시험 기관을 신용하여, 시험한 결과를 토대로 필요한 품질의 컴포넌트를 선택할 수 있게 된다.

필요한 정보의 부족을 해소하기 위한 방법 중 하나로 Wang이 제안한 Built-In Test(BIT)가 있다[35]. BIT는 컴포넌트 사용자에게 전달되는 컴포넌트 구현물에 개발 시 사용한 시험 케이스나 시험 케이스를 생성할 수 있는 함수를 추가하는 기법이다. 그래서 컴포넌트를 획득한 후 컴포넌트에서 제공되는 시험 케이스를 취득하여 자기 상황에 맞게 시험할 수 있고, 또는 필요한 시험 케이스를 생성할 수도 있게 된다. Self-testing[36]은 사전에 정의된 메소드 호출 기능을 추가 구현하는 기법으로 BIT와 유사하다. 이러한 기법은 컴포넌트 개발자에게 최대한의 협조를 제공받아야 한다는 단점이 있다. 이를 보완한 [37]에서는 구현된 컴포넌트를 넘겨받아 외부에 시험 관련 기능을 추가한 래퍼(Wrapper)를 통해 호출을 해준다. 따라서 소스코드의 제공 없이도 Self-Testing의 기능을 수행하도록 접근하였다.

## 5. 생명 주기에 따른 시험 기술

### 5.1 단위시험(Unit Testing)

단위시험은 생명 주기에 있어서 가장 하위 단계의 단위를 구현하고 나서 수행하는 시험으로 보통은 메소드를 최소 단위로 보지만, 객체지향 소프트웨어인 경우 객체가 하나의 단위가 되며, 컴포넌트 소프트웨어의 경우 하나의 컴포넌트가 시험 단위가 된다. 따라서 단위 간의 관계를 보려는 시험 기법들을 제외한 기존에 제안된 대부분의 시험 기법들이 단위시험에 해당한다고 할 수 있다.

단위시험에서는 단위 내에 존재하는 논리적인 오류나, 구현상의 오류, 그리고 잘못된 계산 등을 찾기 위해 시험 케이스를 설계한다. 구현된 코드가 존재하므로, 명세기반 시험과 코드기반 시험 방법 모두를 사용할 수 있으며, 단위시험을 통해서 오류를 수정하고 나서 그 다음 단계의 시험을 수행할 수 있다. 단위시험의 필요성을 이해시켜야 하고, 체계적으로 시험을 계획하며, 빈번한 작업을 손쉽게 수행 시킬 수 없다는 점들이 장애요인이었으나 자동화된 단위 시험 환경[39]이 제안되며 점차 나아지고 있다.

### 5.2 통합시험(Integration Testing)

통합시험은 프로그램 단위들의 집합인 프로그램 모듈이 제대로 동작하는지를 살펴보는 것으로서 시스템 통합 이전에 수행되는데, 통합시험에서는 단위시험에서 이미 시험된 모듈들을 통합하여 시험하므로 단위시험의 내용과 중복

되지 않도록 하는 것이 중요하다. 특히 객체지향 프로그램의 통합 시험은 순차적 프로그램의 통합시험과는 특성이 다르다. 순차 프로그램에서는 모듈 간의 호출 관계에 기반하여 시험을 수행한다. 하지만 객체지향 프로그램에서는 객체의 상태에 따라 동일한 메소드에 대해서도 수행 결과가 다르며, 모듈 간의 관계가 호출관계 외에도 상속, 다형성 등의 특성에 의해 다양한 관계 존재하므로 이들을 고려하여 통합 시험을 수행해야 한다.

객체지향 프로그램을 대상으로 통합 기법들은 주로 클래스의 시험 순서에 관심을 가진다[32]. 데이터 흐름 기반 기법을 이용 또는 확장하여 객체지향 프로그램의 통합 시험에 이용하는 연구들[15]도 수행되었다. 하지만 데이터 흐름 기법의 적용은 객체지향 특화된 특성들을 제대로 다루는 데 어려움이 있다.

### 5.3 회귀시험(Regression Testing)

회귀 시험은 프로그램에 수정이나 확장이 가해진 후에도 제대로 동작하는지 살펴보는 시험 기법으로 수정작업이 제대로 이루어졌는지, 그리고 수정된 부분이 프로그램의 다른 부분에 영향을 주지 않는다는 것을 확인하는 것이 목적이다.

회귀 시험에서 중요하게 고려되는 문제 중 하나는 이전 버전에서 사용됐던 시험 데이터 중에서 새로운 버전의 프로그램 시험에 적합한 부분을 뽑아내는 방법이다. 가장 간단한 방법인 "Retest-all" 기법은 예전 버전의 프로그램을 시험하는 데 사용했던 시험 데이터를 모두 재사용하는 방법인데, 비용이 고가인 단점이 있다. 선택적 회귀 시험(Selective Regression Test) 기법은 이미 사용했던 시험 데이터의 일부분만을 이용하는 방법이다. 초기에는 최소화 기법(Minimization)을 이용하여 프로그램 코드에서 변경된 부분을 커버할 수 있는 최소한의 시험 데이터를 뽑거나[40], 데이터 흐름을 커버하는 회귀 시험 기법[41]이 개발되었다. 하지만 이 기법들은 이전 버전의 시험 데이터 중 새 버전의 오류를 검출할 수 있는 모든 시험 데이터를 뽑아내지 못하고 변경된 코드 부분이나 데이터 흐름만을 커버하였다. 안전한 회귀 시험 선택기법 연구[42]는 앞선 기법들의 단점을 해결하여 기존의 시험 데이터로부터 새로운 시험 데이터를 모두 뽑아내는 기법들로, 불필요한 시험데이터들을 얼마나 뽑아내는지에 대한 정확도(precision), 수행의 효율성, 적용 대상의 규모(granularity)에 따라 기법들 간에 차이가 있다.

1990년대 후반부터 활발한 연구가 진행된 시험 케이스 우선선택 (Prioritization) 기법[43]은 시험 데이터에 우선순위를 두어서 우선순위가 높은 시험 데이터를 먼저 수행시키는 방법이다. 이 기법은 똑 같은 시험 데이터를 이용하더라도 시험 수행 과정에서 프로그램의 오류가 더 빨리 검출되게 한다. 최근에는 회귀시험에 대한 많은 연구들이 객체지향 프로그램을 대상으로 이루어졌다. [44]기법들은 클래스 firewall을 통해 프로그램 변화로 인해 영향을 받은 클래스들의 집합을 구별해내어 회귀 시험을 수행했다.

## 참고문헌

[1] L. J. Osterweil, et. al., "Strategic directions in software Quality", ACM Computing Surveys, 738-750, Dec. 1996.

[2] M. J. Harrold, "Testing: A Roadmap, in The Future of Software Engineering", A. Finkelstein, ed., ACM Press, New York, 2000.

[3] J. B. Goodenough and S. L. Gerhart, "Toward a Theory of Test Data Selection", IEEE Transactions on Software Engineering, June 1975.

[4] E. J. Weyuker, "Axiomatizing Software Test Data Adequacy", IEEE TSE, Dec. 1986.

[5] E. J. Weyuker, "The Evaluation of Program-based Software Test Data Adequacy Criteria", CACM, June 1988.

[6] S. H. Zweben and J. S. Gourlay, "Comments, with reply, on 'Axiomatizing software test data adequacy' by E.J. Weyuker", IEEE TSE, April 1989.

[7] E. J. Weyuker and T. J. Ostrand, "Theories of Program Testing and the Application of Revealing Subdomains", IEEE TSE pp. 236-246, May 1980.

[8] J. W. Duran and S. C. Ntafos, "An Evaluation of Random Testing", IEEE TSE, pp. 438-444, Jul 1984.

[9] E. J. Weyuker and B. Jeng, "Analyzing Partition Testing Strategies", IEEE TSE, vol. 17, no. 7, pp. 703-711, Jul 1991.

[10] T. Y. Chen and Y. T. Yu, "On the Expected Number of Failures Detected by Subdomain Testing and Random Testing", IEEE TSE, vol. 22, no. 2, pp. 109-119, Feb 1996.

[11] S. C. Ntafos, "On Comparisons of Random, Partition, and Proportional Partition Testing", IEEE TSE, vol. 27, no. 10, pp. 949-960, Oct 2001.

[12] Myers, G., The Art of Software Testing, Wiley-Interscience, 1979.

[13] P. G. Frankl and O. Iakounenko, "Further empirical studies of test effectiveness", *In ACM SIGSOFT Software Engineering Notes, Sixth International Symposium on Foundations of Software Engineering*, Volume 23, pages 153-162, November 1998.

[14] A. Paradkar, K. Tai, and M. A. Vouk, "Automatic test-generation for predicates", *IEEE Transactions on reliability*, 45(4):515-530, December 1996.

[15] M. J. Harrold and G. Rothermel, "Performing data flow testing on classes", *2nd ACM SIGSOFT Symposium on the Foundations of software engineering*, pages 154-163, December 1994.

[16] B. Rudner, "Seeding / tagging estimation of software errors: models and estimates", Rome Air Development Centre, Rome, NY, RADC-TR-77-15, also AD-A036 655, 1977.

[17] R. J. Lipton, et. al., "Hints on test data selection: Help for the practicing programmer", IEEE Computer, vol. 11, no. 4, pp. 3441, April 1978.

[18] R. A. DeMillo, D et. al., "An extended overview of the Mothra software testing environment",Proc. of the Second Workshop on Software Testing, Verification, and Analysis, pp. 142-151, July 1988.

[19] Y. S. Ma, Y. R. Kwon, and J. Offutt. "Inter-class mutation operators for Java", In Proc. of the 13th ISSRE, pp. 352-363, November 2002.

[20] A. T. Acree, On mutation. PhD Thesis, Georgia Institute of Technology, Atlanta, GA, 1980.

[21] A. J. Offutt, et. al., "An experimental determination of sufficient mutation operators" ACM TOSEM 1996; vol. 5, no. 2, pp. 99118.

[22] Choi B, Mathur AP. "High-performance mutation testing", Journal of Systems and Software 1993; vol. 20, no. 2, pp. 135152.

[23] DeMillo RA, Krauser EW, Mathur AP. "Compiler-integrated program mutation". In

Proc. of the 15th COMPSAC, September 1991. pp. 351356.

[24] Untch R, Offutt AJ, Harrold MJ. "Mutation analysis using program schemata", In Proc. of ISSTA, June 1993. pp. 139148.

[25] Y. S. Ma, Jeff Offutt and Y. R. Kwon, "MuJava: An Automated Class Mutation System", Journal of STVR will be appear at vol. 15, no. 2, 2005.

[26] D. Lee and M. Yannakakis, "Principles and Methods of Testing Finite State Machine - A Survey". IEEE TSE, vol. 84, no. 8, pp. 10891123, 1996.

[27] B. Y. Tsai, et. al., "An Automatic Test Case Generator Derived from State-based Testing", Proc. of APSEC, pp. 270-277. 1998.

[28] H. S. Hong, et. al., "A Test Sequence Selection Method for Statecharts", Journal of STVR, vol. 10, no. 4, pp. 203227, 2000.

[29] Huo Yan Chen, T. H. Tse, and T. Y. Chen, "TACCLE: a methodology for object-oriented software testing at the class and cluster levels" , ACM TOSEM, 10(1), 2001.

[30] R.V. Binder, Testing Object-Oriented SystemsModels, Patterns, and Tools, Object Technology. Addison-Wesley, 1999.

[31] H. S. Hong, Y. R. Kwon, and S. D. Cha, "Testing of Object-Oriented Programs Based on Finite State Machines," in Proc. of APSEC, pp. 234-241, 1995.

[32] M. J. Harrold, et. al., "Incremental Testing of Object-Oriented Class Inheritance Structures", In Proc. of the 14th ICSE '92, pp. 68-80, May 1992.

[33] J. Offutt, et. al., "A Fault Model for Subtype Inheritance and Polymorphism", In Proc. of ISSRE '01, pp. 84-95, November 2001.

[34] J. Voas, "Developing a Usage-based Software Certification Process", IEEE Computer, August 2000, pp. 32-37.

[35] Y. Wang, et. al., "A method for built-in tests in component-based software maintenance", In European Conference on software Maintenance and Reengineering(CSMR), pp186-189, 1999.

[36] V. S. Alagar, et. al., "Automated Test Generation from Object-Oriented Specifications of Real-Time Reactive Systems", In Proc. of the 10th APSEC, pp. 406-414, 2003.

[37] A. Bertolino and A. Polini, "WCT" a Wrapper for Component Testing", in Proc. of Fidji'2002, Nov. 28-29, 2002.

[38] A. Orso, M. J. Harrold, and D. Rosenblum "Component Metadata for Software Engineering Tasks", in W. Emmerich and S. Tai(Eds) EDO2000, LNCS 1999, pp. 129-144.

[39] JUnit.org, http://www.junit.org/index.htm

[40] J. Hartmann, and D. Robson, "Techniques for selective revalidation", IEEE Software 16, pp. 3138, Jan. 1990.

[41] M. J. Harrold and M. L. Soffa, "An incremental approach to unit testing during maintenance", In Proc. of the Conference on Software Maintenance (Oct.), pp. 362367, 1988..

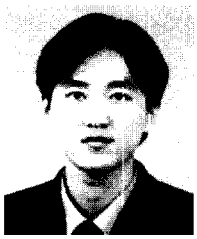
[42] G. Rothermel and M. J. Harrold. "A safe, efficient regression test selection technique", ACM TOSEM, 6(2):173210, Apr. 1997.

[43] G. Rothermel, et. al., "Prioritizing Test Cases for Regression Testing," IEEE TSE, vol. 27, no. 10, pp. 929-948, Oct. 2001.

[44] P. Hsia, et. al., "A technique for the selective revalidation of OO software", Software Maintenance: Research and Practice, 9:217233, 1997.

[45] Paul E. Rook, "Controlling software projects" IEE Software Engineering Journal 1, 1 (January 1986), 7-16.

## 저자약력

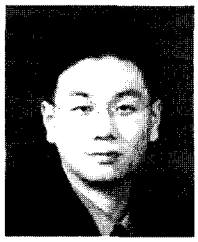


**김 상 운**

2000년 한국과학기술원 전산학 학사
2002년 한국과학기술원 전산학 석사
2002년~현재 한국과학기술원 박사과정
관심분야: 소프트웨어 공학, 컴포넌트 시험, 웹 어플리케이션 시험
E-mail: swkim@salmosa.kaist.ac.kr

**마 유 승**

1998년 한국과학기술원 전산학 학사
2000년 한국과학기술원 전산학 석사
2001년~현재 한국과학기술원 박사과정
관심분야: 소프트웨어 시험, 객체지향, 뮤테이션 시험
E-mail: ysma@salmosa.kaist.ac.kr

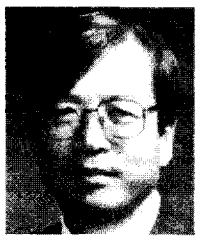


**신 종 민**

2003년 한동대학교 경영경제학 학사
2004년~현재 한국과학기술원 석사과정
관심분야: 객체지향 소프트웨어 시험, 소프트웨어 프로세스 개선, 웹서비스
E-mail: jmshin@salmosa.kaist.ac.kr

**이 숙 희**

2002년 한국과학기술원 전산학 학사
2004년 한국과학기술원 전산학 석사
2004년~현재 한국과학기술원 박사과정
관심분야: 소프트웨어 메트릭, 리팩토링, 소프트웨어 프로세스
E-mail: shlee@salmosa.kaist.ac.kr



**권 용 래**

1969년 서울대학교 물리학과 이학사
1971년 서울대학교 대학원 이학석사
1971년~1974년 육군사관학교 전임강사
1978년 미국 피츠버그대학 이학박사
1978년~1983년 미국 Computer Science Corporation 연구원
1983년~현재 한국과학기술원 전자전산학과 교수
관심분야: 소프트웨어 공학, 시험
E-mail: kwon@salmosa.kaist.ac.kr