CBSD에서의 컴포넌트 조립 테스트 기법
(A Component Composition Testing Technique in CBSD)

윤 희 진 † 최 병 주 ††
(Hoijin Yoon) (Byoungju Choi)

요 약 컴포넌트 기반 소프트웨어 개발(Component-Based Software Development : CBSD)로 만들어진 소프트웨어는 현재 개발자에 의해 새로 만들어진 컴포넌트들과 재사용되는 컴포넌트들의 '조립'으로 이루어진다. 본 논문에서는 이들을 각각 화이트박스 컴포넌트와 블랙박스 컴포넌트로 정의한다. 따라서 컴포넌트 조립에 의한 오류는 소프트웨어를 이루는 블랙박스 컴포넌트들과 화이트박스 컴포넌트들의 상호작용에 의해 발생한다. 본 논문은 이러한 조립 오류들을 테스트하는 방법을 제안하고, 엔터프라이즈 자바빈즈 아키텍쳐에서의 컴포넌트 조립 테스트에 적용한다.

본 기법은 화이트박스 컴포넌트의 특정 영역에만 오류를 삽입하여 테스트 케이스를 선정한다. 오류가 삽입되는 특정 영역은 컴포넌트 조립의 패턴들을 분석하여 선정되며, 이는 본 기법의 테스트 케이스가 높은 효율성을 갖도록 한다. 본 논문에서는 테스트 케이스의 효율성을 실험을 통해 평가하며, 나아가 컴포넌트 조립 테스트 기법의 자동화 방안을 제안한다.

키워드 : CBSD, 컴포넌트 테스트, 컴포넌트 조립 테스트

Abstract An application in Component-Based Software Development (CBSD) is built by 'composing' two kinds of components; One is a component that is made by current developer himself, and the other is a component that is from other developments. We define the former as a 'White-box component' and the latter as a 'Black-box component.' The error from the composition can be said to be caused by interactions of Black-box components and White-box components. This paper proposes a new testing technique for composition errors, and applies the technique to Enterprise Java Beans component architecture.

Our technique selects test cases by injecting a fault only into the specific parts of a White-box component. This specific parts for injecting a fault are selected by analyzing composition patterns, and lead to make our test cases have a good effectiveness. We show the effectiveness of our test cases through an experiment. Moreover, we also mention an automation tool for our technique.

Key words : CBSD, Component Test, Composition Test

1. 서 론

CBSD(Component-Based Software Development)는 이미 개발된 컴포넌트들을 재사용함으로써, 소프트웨어 개발 비용을 절감시킨다. 그러나 다른 개발자에 의해, 또는 다른 개발환경에서 만들어진 컴포넌트들을 재사용하는 데에는 위험부담이 있다. 어떤 시스템에서 오류 없이 잘 수행되던 컴포넌트들이 또 다른 시스템에서는 오류를 일으킬 수 있기 때문이다. 따라서 좀 더 정확하고 심도 있는 테스트 기법이 컴포넌트에 요구된다. 컴포넌트 테스팅은 CBSD의 활동들에 따라 몇 가지 수준을 갖는다. 이 가운데 하나가 컴포넌트 조립 테스트 수준이다. 본 논문에서는 컴포넌트 조립 테스트 수준에서 이용할 수 있는 테스트 기법을 엔터프라이즈 자바빈즈 아키텍쳐에 맞추어 개발한다.

CBSD 패러다임을 기반으로 소프트웨어를 개발할 경우, 일부분은 기존에 만들어진 컴포넌트들을 가져다 쓸 수 있으며, 또 일부분은 개발자가 직접 컴포넌트를 개발

· 본 연구는 한국과학재단 목적기초연구(과제번호:R04-2000-000-000792-0) 지원으로 수행되었음.

† 비 회 원 : 이화여자대학교 대학원 컴퓨터학과
hoijin@ewha.ac.kr
†† 종신회원 : 이화여자대학교 컴퓨터학과 교수
bjchoi@ewha.ac.kr

논문접수 : 2001년 8월 13일
심사완료 : 2002년 7월 15일

하여 사용할 수 있다. 따라서 CBSD를 통해 작성된 소프트웨어는 다른 사람에 의해 또는 다른 목적으로 개발된 컴포넌트들과 현재 개발자에 의해 개발된 컴포넌트들의 '조립(Composition)'을 통해 이루어진다. 따라서 컴포넌트들을 조립하는 일은 CBSD에서 필수적인 작업이다. 앞서 말한 대로 개발자는 두 가지 종류의 컴포넌트들을 조립한다. 하나는 다른 개발자에 의해 개발된 컴포넌트이고, 또 다른 하나는 현재 개발자가 기존의 개발에서 구할 수 없거나 현재 개발 환경의 특정한 기능을 구현한 컴포넌트이다. 전자는 일반적으로 소스코드를 공개하지 않는 블랙박스 특성을 갖으며, 후자는 개발자가 직접 개발한 것이므로 화이트박스 특성을 갖는다. 각 특성을 갖는 컴포넌트들을 조립할 때, 예기지 못한 오류가 발생할 가능성이 있고, 본 논문에서는 이 오류를 테스트하는 기법을 제안한다.

나아가 본 논문에서 제안하는 기법은 상대적으로 적은 수의 테스트 케이스를 선정하고, 이들 테스트 케이스는 조립 오류가 발생할 가능성이 높은 부분에서의 오류를 감지할 수 있는 특징이 있다. 이는 테스트 케이스의 높은 효율성을 제공함으로써, CBSD의 궁극적 목표인 개발 비용 절감에 부응하는 효과를 보인다. 여기서 말하는 효율성이란, 선정된 전체 테스트 케이스에 대한 오류를 감지하는 테스트 케이스 수의 비율로 평가한다[3]. 본 논문은 실험을 통해, 효율성 높은 적정수의 테스트 케이스를 선정하여 테스트 비용 절감 효과를 제공함을 보인다. 이는 CBSD의 주요 의의가 소프트웨어 개발의 비용절감 효과에 있는 것을 감안할 때, 본 기법의 효율성으로 인한 테스트 비용 절감효과는 CBSD에 더욱 부합한다. 이는 CBSD의 실제 개발 현장에 이용될 가치가 있음을 나타낸다.

본 논문은 2장의 관련 연구에 이어, 3장에서는 본 논문에서 개발하는 컴포넌트 조립 테스트 기법과 현재 진행중인 자동화 도구 구현 내용을 기술한다. 4장에서는 실험을 통해 본 기법이 오류 감지 효율이 높은 테스트 케이스를 선정하여, 테스트 비용 절감 효과를 갖음을 보인다. 마지막 5장에서는 본 논문의 결론과 향후 연구 과제에 대해 기술한다.

2. 관련 연구

2.1 컴포넌트 테스트 수준

EJB에서 CBSD를 수행할 때, 맞춤 테스트, 어셈블 테스트, 조립 테스트 등의 세 가지 테스트 수준이 요구된다. 맞춤 테스트 수준에서는, 이미 작성된 컴포넌트를 새로운 도메인에서 재사용하기 위하여 컴포넌트 맞춤을 수행할 때 발생할 수 있는 오류를 테스트하고, 어셈블 테스트 수준에서는, EJB의 하나의 JAR파일에 들어가는 컴포넌트들의 상호작용에서 발생하는 오류를 테스트한다. 조립 테스트 수준에서는, 두 개의 컴포넌트를 조립할 때 조립된 컴포넌트들 사이의 상호작용을 통해 발생할 수 있는 오류를 테스트한다.

본 논문에 앞서 [6,7]에서, 컴포넌트 맞춤 테스트 수준을 위한 테스트 기법을 제안하였다. [6,7]은 공개된 인터페이스의 특정 부분을 선정하고 그 부분에만 임의의 오류를 심는 방법으로 테스트 케이스를 추출하여, 효율성 높은 테스트 케이스를 추출하였다. 본 논문은 컴포넌트 조립과 관련이 있는 부분을 오류를 심는 특정 부분으로 정의하여, EJB의 조립 테스트를 효율적으로 수행할 수 있는 테스트 케이스를 선정한다.

2.2 인터페이스 뮤테이션 기법

본 기법은 소프트웨어 오류 삽입 기법과 뮤테이션 테스트 기법[2]을 기반으로 하고 있다. 즉, 조립된 컴포넌트에 임의의 오류를 삽입하여, 오류가 삽입된 것과 삽입되기 이전의 것을 차별화 시키는 테스트 케이스를 선정한다. 그러나 기존의 뮤테이션 테스트 기법이 테스트 대상의 모든 문장에 오류를 삽입하여 변형시키는 반면, 본 기법은 테스트 대상이 되는 조립된 컴포넌트에서 특정 부분을 추출하고, 해당 부분에만 오류를 삽입하여 변형시킨다. 이 특정 부분은 조립으로 인한 오류에 더욱 민감한 테스트 케이스를 만들어 내는데 기여한다.

인터페이스 뮤테이션[1]은 본 기법과 유사하게 전체 문장을 변형시키지 않고, 단지 다른 모듈을 호출하는 문장만을 변형시키는 방법을 통해, 두 모듈 사이의 통합 테스트를 수행하는 기법이다. 이는 기존의 C프로그램에 적용하기 위한 방법으로서, 호출하는 문장의 매개변수, 호출되는 모듈의 반환변수, 그리고 전역변수를 변형 대상으로 하고 있다. 이는 본 기법에서 선정한 EJB에서의 변형대상과 유사한 모습을 띈다. 이는 본 기법의 적용 대상을 EJB로 함으로써 우연히 같은 결과를 나타내는 것일 뿐이다. 즉, EJB의 특성상 컴포넌트들의 조립을 메소드 호출 방식으로 하기 때문에, 본 기법의 특정 부분에 대한 정의에 따라서 추출한 결과가 인터페이스 뮤테이션의 대상과 유사하게 나타난다. 본 논문의 특성은 인터페이스 뮤테이션과 같이 메소드 호출 부분을 중심으로 오류를 심는다는 것에 있지 않고, 컴포넌트들의 조립 패턴을 추출하고, 각 조립 패턴에 따라 오류를 삽입할 특정 부분을 찾는데 있다. 따라서 조립을 메소드 호출이 아닌 다른 형태로 하는 컴포넌트 아키텍쳐에서는 다른 모습의 특정 부분이 생성된다. 또한 본 기법에서는 인터페이스 뮤테이션과는 달리 컴포넌트들의 조립 패턴을 선정하였다. 조립 패턴에 따라 서로 다른 대상에 오류를 삽입함으로써, 인터페이스 뮤테이션을 적용했을 때

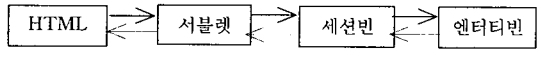
보다 효율적으로 테스트 케이스를 선정할 수 있다.

2.3 엔터프라이즈 자바빈즈의 컴포넌트 조립

엔터프라이즈 자바빈즈(Enterprise JavaBeans : EJB)는 자바를 기반으로 하는 분산 컴포넌트 기술 아키텍처이다. 이는 분산개념과 컴포넌트 개념을 모두 포함하고 있어서, 향후 소프트웨어 개발의 강력한 아키텍처가 될 것으로 전망되고 있다. 이에 따라, 본 논문도 EJB 아키텍처를 기반으로 하는 기법을 제안한다.

일반적으로 컴포넌트 조립은 CBSD를 이루는 여러 작업들 가운데 하나로서, 컴포넌트들을 서로 연동시켜서 하나의 소프트웨어를 만드는 작업을 의미한다. EJB에서 작성된 어플리케이션을 예로 들면 다음과 같다. 사회보장번호와 승수를 입력으로 넣어서, 해당 번호를 갖은 사람의 보너스를 수정해주는 '보너스 프로그램[5]'을 그림 1과 같이 EJB에서 CBSD 개념으로 작성한다고 하자. HTML과 서블렛은 각각 웹 컴포넌트이고, 세션빈과 엔터티빈은 각각 EJB 컴포넌트가 된다. 따라서 '보너스 프로그램'은 네 개의 컴포넌트들을 조립하여 사용하게 된다. 그림1에서 엔터티빈은 다른 개발자에 의해 개발되어 재사용하는 컴포넌트이고, 세션빈은 현재 개발자에 의해 개발된 컴포넌트이다.

EJB 아키텍처에서 위의 컴포넌트들은 두 가지 종류의 파일에 나누어 저장된다. HTML과 서블렛은 WAR 파일의 형태로 다루어지며, 세션빈과 엔터티빈은 JAR파일의 형태로 다루어진다. 이 두 가지의 파일들은 최종적으로 EAR파일 형태에 저장됨으로써, EJB 어플리케이션이 완성된다.



- HTML : 사회보장번호와 승수를 입력으로 하기 위한 폼문
- 서블렛 : HTML에서 입력으로 받는 사회보장번호와 승수를 세션빈으로 넘겨주고 그 결과를 받아서 브라우저에 다시 뿌려주는 역할. 일종의 CGI 프로그램.
- 세션빈 : 들어온 두개의 입력을 이용하여 새로운 보너스를 계산하고, 그 결과를 엔터티빈에 전달하여 DB에 저장하는 메소드와, 사회보장번호를 엔터티빈에 보내서 관련된 기록을 가져오는 메소드로 구성된다.
- 엔터티빈 : 다른 사람이 작성해 놓은 컴포넌트로서, 사회보장번호와 보너스를 저장하는 DB를 관리한다.

그림 1 보너스 프로그램

3. 조립 테스트 기법

본 장에서는 컴포넌트 조립으로 발생하는 오류를 테스트하는 기법을 제안한다. 우선 본 기법에서 사용되는 용어에 대한 정의를 내린 후, 본 기법의 기반이 되는 조립 패턴을 추출한다. 추출한 조립 패턴에 따라서 오류를 삽입할 특정 장소를 추출하고, 이들을 이용한 테스트 케이스 선정 방법을 기술한다.

3.1 용어 정의

본 장에서는 본 논문에서 사용되는 용어들에 대한 정의를 내린다.

정의 1 : 블랙박스 컴포넌트 ($\underline{B}$)

블랙박스 컴포넌트는 현재 개발자가 그 코드에 접근할 수 없는 컴포넌트를 의미한다. 이는 다른 개발자에 의해서 다른 시스템을 위해 개발된 컴포넌트로서, 현재 시스템에서는 재사용되는 컴포넌트가 이에 해당한다. 본 논문은 이를 간단하게 $\underline{B}$로 표현한다.

예 : EJB를 예로 들면, 그림 1의 엔터티빈은 이미 작성되어 재사용되는 컴포넌트로서, 현재 개발자는 이에 대한 접근 권한이 없다. 따라서 이는 $\underline{B}$에 해당한다. ■

정의 2 : 화이트박스 컴포넌트 ($\underline{W}$)

화이트박스 컴포넌트는 블랙박스 컴포넌트와 대치되는 개념으로서, 현재 개발자에게 그 소스코드까지 공개한 컴포넌트를 일컫는다. 이는 현재 개발자가 제어권을 가질 수 있는 컴포넌트로서, 현재 개발자에 의해 현재 시스템을 위해 개발된 컴포넌트가 이에 속한다. 본 논문은 이를 간단히 $\underline{W}$로 표현한다.

예 : EJB를 예로 들면, 그림 1에서 세션빈, 서블렛, HTML문서는 현재 개발자에 의해 직접 개발된 것으로서 $\underline{W}$이다. ■

정의 3 : 조립 단위 (CU)

조립 단위는 두 컴포넌트를 조립하여 생성된 단위를 의미한다. 이때 조립되는 컴포넌트들은 $\underline{B}$가 될 수도 있고 $\underline{W}$가 될 수도 있다. 본 논문에서는 간단히 이를 CU로 표시한다. CU에는 $\underline{WB}$, $\underline{WW}$, $\underline{BB}$, $\underline{BW}$등이 있다. $\underline{WB}$는 $\underline{W}$가 B의 메소드를 호출하여 형성된 CU를 의미한다. 또한 $\underline{WW}$는 $\underline{W}$가 또 다른 $\underline{W}$의 메소드를 호출하여 만들어진 CU를 의미하며, 이때 메소드를 호출하는 $\underline{W}$를 $\underline{W}_s$로 표현하고, 메소드 호출을 당하는 $\underline{W}$를 $\underline{W}_d$로 표현한다. $\underline{B}$가 또 다른 $\underline{B}$의 메소드를 호출하여 생성된 CU를 $\underline{BB}$라 하고, $\underline{BW}$는 $\underline{B}$가 $\underline{W}$의 메소드를 호출하는 CU를 나타낸다. 나아가, 임의의 오류가 삽입된 $\underline{WW}$를 $\underline{fWW}$라 하고, 오류가 삽입된 $\underline{WB}$를 $\underline{fWB}$라 한다. $\underline{fWW}$나 $\underline{fWB}$는 오류가 삽입된 CU라고 할 수 있으므로, 이들을 합하여 $\underline{fCU}$라고 부를 수 있다.

예 : EJB를 예로 들면, 그림 1의 세션빈과 엔터티빈은 $\underline{WB}$로서 하나의 CU로 볼 수 있다. 또한 오류가 삽입된 $\underline{WB}$, 즉 $\underline{fWB}$는 fCU로 볼 수 있다. $\underline{W}$인 세션빈이 $\underline{B}$인

엔터티빈의 메소드를 호출하여 사용하므로, 이들 조립을 *WB*라고 할 수 있다. 이때 세션빈은 W_s가 된다. 테스팅을 위해 *WB* 조립에 오류를 삽입하면, 이는 *fWB*가 된다. 또한 *W*인 서블렛에서 *W*인 세션빈의 메소드를 호출하여 사용하므로, 이들 조립을 *WW*라고 할 수 있다. 이때 서블렛이 W_s가 되며, 세션빈이 W_d가 된다. 테스팅을 위해, *WW*에 오류를 삽입하면 이는 *fWW*가 된다.

정의 4 : *P*

*P*는 *CU*에서 다른 컴포넌트의 메소드를 호출할 때 사용한 매개변수를 의미한다.

예 : EJB를 예로 들면, 그림 1의 '보너스 프로그램'에서의 컴포넌트들인 세션빈과 엔터티빈의 *WB* 조립에서 W_s인 세션빈의 Bean class 코드의 일부분은 다음과 같다. 여기에서 bold로 표시된 부분이 엔터티빈의 메소드를 호출하는 부분이며, 이곳에서 calc, socsec이 *P*가 된다

```
public class CalcBean implements SessionBean {
....
//Store data in entitybean
try {
theBonus = homebonus.create(calc, socsec);
} catch(java.rmi.RemoteException e) {
String message = e.getMessage();
e.printStackTrace();
}
return theBonus;
}
public Bonus getRecord(String socsec) {
            ...
//Use primary key to retrieve data from entity bean
try {
record = homebonus.findByPrimaryKey(socsec);
} catch(java.rmi.RemoteException e) {
...
}
```

정의 5: *R*

*R*은 호출되는 메소드에서 반환하는 변수를 지칭한다.

예 : EJB를 예로 들면, 그림 1의 서블렛과 세션빈사이의 *WW* 조립에서 서블렛은 세션빈의 calc bonus 메소드와 getRecord 메소드를 호출한다. 이때 다음에서 보듯이 W_d인 세션빈의 해당 메소드에서 반환변수인 theBonus와 record가 *R*이 된다.

```
public Bonus calcBonus(int multiplier, doublebonus,
String socsec)
{
.....
return theBonus;
}
public Bonus getRecord(String socsec) {
....
return record;
}
```

정의 6 : *ref(x)*

*ref(x)*는 x의 값에 변화를 주는 문장을 의미한다. *ref(P)*는 P가 존재하는 컴포넌트에 존재하고, *ref(R)*는 R이 존재하는 컴포넌트에 위치한다.

예 : EJB를 예로 들면, 그림 1의 서블렛과 세션빈의 *WW* 조립에서, 세션빈의 아래 메소드의 record는 *R*이다. 이때 *ref*(record)는 *Bonus record = null;*이 된다.

```
public Bonus getRecord(String socsec) {
Bonus record = null;
//Use primary key to retrieve data from entity bean
try {
record = homebonus.findByPrimaryKey(socsec);
} catch(java.rmi.RemoteException e) {
String message = e.getMessage();
} catch(javax.ejb.FinderException e) {
e.printStackTrace();
}
return record;
}
```

정의 7 : 오류 삽입 대상[6,7]

오류 삽입 대상은 오류를 삽입하는 특정 위치를 의미한다. 본 논문은 이를 간단히 *FIT*로 표현한다. 효율적인 테스트 케이스는 *FIT*에 오류를 삽입함으로써 얻어질 수 있다.

정의 8 : 오류 삽입 연산자 [6,7]

오류 삽입 연산자는 *FIT*에 오류를 삽입하는 연산자를 의미한다. 본 논문은 이를 간단히 *FIO*로 표현한다. 오류는 구문상 에러가 일어나지 않도록 삽입되어야 한다. 이는 Component architecture가 어떻게 *W*를 구현하고 있느냐에 따라서, 다른 모습으로 나타날 수 있다.

3.2 *FIT*와 *FIO*

EJB에서는 상대 컴포넌트의 메소드를 호출하는 방식으로 조립단위를 구성한다. 따라서 메소드를 호출하는 입장과 호출 받는 입장, 그리고 블랙박스컴포넌트와 화이트박스 컴포넌트로 나누어 *BB*, *BW*, *WB*, *WW*의 네 가지 조립패턴을 얻을 수 있다.

BB 조립패턴은 이미 컴포넌트 제공자에 의해 만들어진 것으로써, 컴포넌트 사용자 입장에서 수행하는 조립 테스트에서는 고려할 수도, 또 고려할 필요도 없다. *BW* 조립패턴은 *B*를 제공한 컴포넌트 제공자에 의해 이루어진 것이다. 왜냐하면 *B*를 받아서 사용하는 컴포넌트 사용자, 즉 현재 개발자는 *B* 내부에 *W*의 메소드를 호출하도록 하는 조립을 위한 코드를 삽입할 자격이 없기 때문이다. 정의1에 따라 *B*의 코드는 현재 개발자에게 공개되지 않는다.

*BB*와 *BW*를 제외한 나머지 두 개의 조립패턴, *WB*와 *WW*,이 조립 테스트에서 다루어야 하는 패턴이다. 따라서 *WB*와 *WW*에 대한 *FIT*와 *FIO*를 개발한다. 본 논문은 *CU*의 *FIT*에 임의로 오류를 삽입하여 높은 효율성을 갖는 테스트 케이스를 선정한다. *FIT*가 높은

효율성을 보장한다는 사실이 실험을 통해서 보여진다. 따라서 각 패턴에서 *FIT*를 선정하는 일이 본 논문의 핵심이라고 할 수 있다.

*FIT*는 *fCU*를 만들기 위해 오류를 삽입하는 위치이다. 본 기법은 $\underline{W}$가운데 조립관계를 직접 이루는 부분을 중심으로 *FIT*를 선정한다. 즉, 컴포넌트 아키텍처마다 정의하고 있는 조립관계 구축 방법에 따라, 그에 따른 *FIT*가 선정된다. 따라서 본 논문에서는 $\underline{W}$의 구성요소 가운데 조립관계를 직접 이루는 부분을 중심으로 *FIT*를 선정한다는 본 기법의 대전제아래, EJB 컴포넌트 아키텍처를 대상으로 구체적인 *FIT*를 다음과 같이 선정한다.

3.2.1 *FIT*

$\underline{WB}$ 조립패턴에서, $\underline{B}$는 조립으로 인해 수정될 수 없으므로, 오류는 $\underline{W}$에 존재할 것이며, 또한 임의의 오류도 $\underline{W}$에 삽입될 수밖에 없다. EJB는 메소드를 호출하는 방식으로 조립관계를 구축하므로, *FIT*로 우선 고려할 수 있는 부분이 *ref(P)*와 *ref(R)*이다. 그러나, *ref(R)*은 $\underline{B}$내에 존재하는 것이므로 그 대상에서 제외되며, *ref(P)*가 FIT로서 고려될 수 있다. 그러나 본 기법에서는 *ref(P)*의 전체를 FIT로 하지 않고, 테스트 케이스의 효율성을 고려하여, 단지 *P*만을 *FIT*로 선정한다.

$\underline{WW}$ 조립패턴에서는, 오류가 $\underline{W}_s$ 와 $\underline{W}_d$에 존재할 수 있다. 따라서 임의의 오류도 $\underline{W}_s$ 와 $\underline{W}_d$에 삽입될 수밖에 없다. $\underline{W}_s$의 메소드에서 $\underline{W}_d$의 메소드를 호출하는 방식으로 조립이 이루어지므로, $\underline{W}_s$의 *ref(P)*와 $\underline{W}_d$의 *ref(R)*가 *FIT*로서 고려될 수 있다. 그러나 본 기법에서는 *ref(P)*와 *ref(R)*을 *FIT*로 하지 않고, 테스트 케이스의 효율성을 고려하여, 단지 *P*와 *R*만을 *FIT*로 선정한다.

본 논문에서 *ref(P)*와 *ref(R)*대신 *P*와 *R*만을 *FIT*로 선정하기 위해서는, "오직 *P*또는 *R*에만 임의의 오류를 삽입하여 선정된 테스트 케이스가 *ref(P)*또는 *ref(R)*에 존재하는 오류까지도 감지할 수 있다."는 전제 하에 가능하다. 이는 다음의 정리를 통해 정당화하였다.

정리를 증명하기 위해 우선 2개의 보조정리를 이용하였다.

보조정리 1.

fCU_i : $\underline{W}$의 구성요소 i에 오류를 삽입하여 생성된 *fCU*.

tc_i : $fCU_i(x) \neq CU(x)$인 x가운데, $\underline{W}$의 i를 참조하는 테스트 케이스 x.

이때 **$\forall x \in tc_i$는 $\underline{W}$의 i에 존재하는 오류를 감지할 수 있다.**

=> $\forall x \in tc_i$의 정의에 따르면 tc_i에 속하는 x를 fCU_i에 적용한 결과와 x를 CU에 적용할 결과가 다르다. 또한 $\forall x \in tc_i$는 $\underline{W}$의 i를 참조하고 있다. 만일 $\underline{W}$의 i에 오류가 존재한다며, $\forall x \in tc_i$에 속하는 메소드들은 오류가 존재하는 I을 항상 참조할 것이고, 그 결과 $CU(x)$의 예상 결과를 $CU(x)$의 실제 결과는 항상 달라진다. 따라서 $\forall x \in tc_i$는 $\underline{W}$의 i에 존재하는 오류를 감지한다.

보조정리 2.

fCU_i : $\underline{W}$의 구성요소 i에 오류를 삽입하여 생성된 *fCU*.

tc_i : $fCU_i(x) \neq CU(x)$인 x가운데, $\underline{W}$의 i를 참조하는 테스트 케이스 x.

$TC_i = \{x | fCU_i(x) \neq CU(x)\}$

p, q :p와 q는 $\underline{W}$의 구성요소이고 p는 q를 참조하는 관계에 있다.

이때, $\boldsymbol{TC_p \supset tc_p}$ **and** $\boldsymbol{TC_q \supset tc_q}$ 이다.

=> TC_i의 정의에 따르면, x가 TC_p에 속하기 위해서는 $fCU_p(x) \neq CU(x)$을 만족해야 한다. 또한 fCU_i의 정의에 따르면, fCU_p와 CU는 p에 대한 서로 다른 값을 갖는다. 따라서, $\forall x \in tc_p \rightarrow fCU_p(x) \neq CU(x)$이고, $\forall x \in tc_p \nleftarrow fCU_p(x) \neq CU(x)$이므로, $TC_p \supset tc_p$이다. 그리고 p는 q를 참조하므로, q에 삽입된 오류를 p에 영향을 준다. 따라서 $\forall x \in tc_q \rightarrow fCU_q(x) \neq CU(x)$이고, $\forall x \in tc_q \nleftarrow fCU_q(x) \neq CU(x)$이므로, $TC_q \supset tc_q$이다.

위의 2개의 보조정리를 통해 다음 정리를 보인다.

정리 1.

fCU_i : $\underline{W}$의 구성요소 i에 오류를 삽입하여 생성된 *fCU*.

tc_i : $fCU_i(x) \neq CU(x)$인 x가운데, $\underline{W}$의 i를 참조하는 테스트 케이스 x.

$TC_i = \{x | fCU_i(x) \neq CU(x)\}$

이때, **TC_{FIT}는 *ref(FIT)*또는 *FIT*에 존재하는 오류를 감지할 수 있다.**

=> 보조정리 2에 따르면, $TC_{FIT} \supset tc_{FIT}$이다. 또한 보조정리1에 의하면, $tc_{ref(FIT)}$는 *CU*의 *ref(FIT)*에 있는 오류를 감지할 수 있다. 그리고 tc_{FIT}는 *CU*의 *FIT*에 있는 오류를 감지한다. 따라서 $TC_{FIT} \supset tc_{FIT}$이므로, TC_{FIT}는 *ref(FIT)*또는 *FIT*에 존재하는 오류를 감지할 수 있다.

3.2.2 *FIO*

본 논문은 $\underline{WB}$ 조립패턴의 *FIT*를 *P*로 정의하였다. 더불어 *FIT*에 오류를 어떻게 삽입할지를 결정하는 *FIO*도 선정하여야 한다. EJB에서 *P*는 반드시 주어진 메소드 시그너쳐를 지켜야하므로 삭제와 추가는 *FIO*가 될 수 없다. 또한 *FIO*의 정의8에 따르면, $\underline{fBW}$도 반드

시 실행 가능해야 하므로, P를 다른 값이나 변수로 대체하는 연산자가 $\underline{WB}$ 조립 패턴의 연산자가 된다. 본 논문은 이 FIO를 특히 $Replace(P)$라고 표현한다.

또한 $\underline{WW}$ 조립 패턴에서, FIT를 P와 R로 선정하였다. 더불어 FIT에 오류를 어떻게 삽입할지를 결정하는 FIO도 선정하여야 한다. P와 R은 반드시 주어진 메소드 시그너쳐를 지켜야하므로 삭제와 추가는 FIO가 될 수 없다. 또한 FIO의 정의8에 따르면, $\underline{fBW}$도 반드시 실행 가능해야 하므로, P와 R을 다른 값이나 변수로 대체하는 연산자가 $\underline{WW}$ 조립 패턴의 연산자가 된다. 본 논문은 이 각각의 FIO를 $Replace(P)$와 $Replace(R)$이라고 표현한다.

결과적으로 다음 표1은 EJB의 각 조립 패턴의 FIT와 FIO를 나타낸다.

표 1 EJB의 조립을 위한 FIT와 FIO

조립 패턴	FIT	FIO	설명
$\underline{WB}$	P	$Replace(P)$	W의 P를 다른 변수나 상수로 대체한다.
$\underline{WW}$	P	$Replace(P)$	W_s의 P를 다른 변수나 상수로 대체한다.
	R	$Replace(R)$	W_d의 R을 다른 변수나 상수로 대체한다.

위의 두 가지 패턴에서의 FIT들은 [1]의 오류 삽입 대상과 유사하다. 이는 단지 EJB에서 컴포넌트들을 메소드 호출 방법으로 조립하기 때문이다. 조립 오류를 테스트하기 위해서, 본 논문의 초점은 P 또는 R에 임의의 오류를 삽입한다는 것이 아니라, EJB 각 패턴에 따라 FIT 선정을 선정한다는 것에 있다. 따라서, 만일 EJB가 아닌 다른 컴포넌트 아키텍처에서 메소드 호출이 아닌 다른 방법으로 컴포넌트를 조립한다면, 본 기법은 P나 R이 아닌 다른 형태의 FIT를 선정할 것이다.

3.3 테스트 케이스 선정

본 기법은 $\underline{fCU}$를 사용하여 테스트 케이스를 선정한다. 본 논문은 CU와 fCU를 가지고 테스트 케이스를 다음과 같이 선정한다.

> 컴포넌트 조립 테스트를 위한 테스트 케이스의 집합, TC는 CU와 fCU를 차별화 하는 일련의 메소드들이다. TC는 이들 테스트 케이스들의 집합이다.
>
> $$TC = \{x | CU(x) \neq fCU(x)\}$$

본 논문의 테스트 케이스 선정 방법은 뮤테이션 테스트 케이스 선정[2]과 유사하다. 그러나 정의9는 단순히 CU에 뮤테이션 테스트 기법을 적용한 것과는 다르다. 조립 패턴을 가지고, FIT를 정의하고, FIT에 임의의 오류를 삽입했다. 따라서, 본 기법은 뮤테이션 테스팅이 갖는 'computational bottleneck' 문제를 감소시킬 수 있고, 높은 효율성을 갖는 테스트 케이스를 선정할 수 있다.

3.3 조립 테스트 기법의 자동화

본 논문에서는 컴포넌트 조립을 위한 새로운 테스트 기법을 개발하였으며, 현재 이를 자동으로 수행할 수 있는 도구를 구현하고 있다. 그림2에 나타내었듯이, 본 도구는 기법의 조립패턴 찾기, FIT와 FIO 찾기, 오류 삽입, 테스트 케이스 선정, 그리고 분석을 자동으로 수행한다.

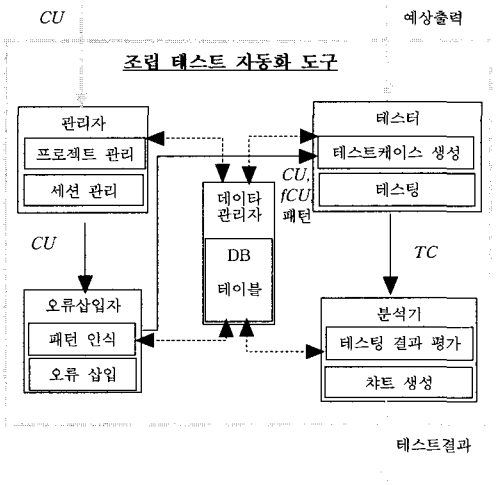


그림 2 조립 테스트 자동화 도구의 구조

본 도구는 사용자로부터 테스트 대상이 되는 CU를 입력으로 받아, 오류삽입자와 테스터를 거쳐 테스트 케이스를 생성한다. 생성한 테스트 케이스를 이용하여 실제 테스팅하는 과정에서 사용자로부터 받은 해당 테스트 케이스에 대한 예상 출력을 이용한다. 테스팅의 결과를 분석하는 분석기를 통해 다양한 면에서 보는 테스팅 결과가 만들어지며, 이중 몇몇은 챠트로 표현되어 본 도구의 결과로서 제공된다. 이때 각 과정들은 데이타 관리자와 긴밀한 관계를 통해 자동화를 위한 거대한 데이타들을 관리한다.

이는 자바기술기반으로 개발되고 있다. 지금까지 다음의 API 기술들이 본 도구 구현에 사용되었다.

- JDK 1.3.1
- Java2 Enterprise Edition 1.3
- MySQL 3.23.42

- mm.mysql.jdbc-1.2c
- Swing API
- Jakarta-Ant 1.3
- Java AIP for XML processing : jaxp-1.1

본 도구의 개발은 컴포넌트 아키텍처를 명확하게 표현하고 있는 EJB를 기반으로 이루어진다. 따라서 EJB로 작성되는 어플리케이션에 대한 테스팅을 본 도구를 이용하여 수행할 수 있도록 한다.

4. 실 험

본 장은 본 기법의 *FIT*가 테스트 케이스의 효율성을 이끌어냄을 실험을 통해 보인다.

'테스트 케이스의 효율성'은 선정된 테스트 케이스 가운데 얼마나 많은 테스트 케이스가 오류를 감지하느냐를 의미한다[3]. 즉 테스트 케이스의 효율성 = (오류를 감지한 테스트 케이스의 수 / 총 테스트 케이스 수) * 100 이다. 선정한 테스트 케이스 수에 대해 오류를 감지하는 테스트 케이스 수의 비율이 높을수록 효율성이 좋다.

■ 비교대상 선정

*FIT*이 비교대상으로 가장 쉽게 생각할 수 있는 부분이 $\underline{W}$전체 부분이다. 오류를 심는 부분이 넓을수록 테스트 케이스의 수는 많아질 것이므로 W전체부분은 효율성 면에서 $\underline{FIT}$보다는 떨어질 것이라고 예측할 수 있다. 따라서 좀더 경쟁력 있는 비교대상을 선정해야 한다. 그래서 선정한 부분이 *ref(FIT)*부분이다. 실제 컴포넌트 사용자가 새로운 조립단위를 만들거나, 기존의 조립단위의 조립관계를 수정하기 위해 변화를 주는 부분이 *ref(FIT)*부분일 것이다. 이 부분을 수정한 결과가 바로 조립관계에 영향을 줄 수 있기 때문이다. 따라서 이 부분이 가장 조립 오류가 발생할 확률이 높은 부분이며, 테스트를 위한 오류를 삽입할 대상으로 선정하기에 가장 합리적인 부분이기도 하다.

결과적으로 *W*전체와 *ref(FIT)*가 *FIT*와 비교대상이 될 수 있으므로, 실험을 이들 세 부분, 즉 *FIT*, *ref(FIT)*와 $\underline{W}$전체에 대하여 각각 수행하여, 비교 분석한다. 지금부터, *FIT*에 임의의 오류를 삽입하여 테스트 케이스를 선정하는 것을 α, *ref(FIT)*에 임의의 오류를 삽입하여 테스트 케이스를 선정하는 것을 β, 그리고 $\underline{W}$전체에 임의의 오류를 삽입하여 테스트 케이스를 선정하는 것을 γ로 지칭한다. 또한 인터페이스 뮤테이션[1]을 적용했을 경우를 각 실험 분석 내용에서 고려하여 기술한다.

■ 실험 대상

보다 객관적인 실험을 위해, 표4와 같이 다양한 특성을 갖는 *CU*들을 실험대상으로 구현하여 10번의 실험을 수행하였다. 생성한 테스트 케이스의 효율성을 보이기 위해, 각 *CU*들이 에러를 갖도록 구현하였다. 물론 *CU*에 임의로 생성한 에러는 단순한 에러이다. "coupling effect"[2,4]에 따르면 이렇게 단순한 에러를 감지하는 태스트 케이스가 더 복잡한 에러도 감지할 수 있다.

■ 실험 수행 방법

본 실험은 앞서 미리 구현한 10개의 *CU*의 *FIT*, *ref(FIT)*, $\underline{W}$전체에 각각 임의의 오류를 심어서 각각의 *fCU*를 생성하고, 그로부터 선정된 각각의 테스트 케이스들의 효율성을 계산한다. 효율성 계산을 객관적으로 하기 위하여, α, β, γ는 동일하게 n개의 테스트 케이스만을 선정한다. 본 실험에서는 n을 임의로 20으로 하였다.

■ 실험 결과 분석

위의 실험 수행 방법에 따라, 실험을 수행하여, 생성된 테스트 케이스의 수와 그 가운데 오류를 감지하는 테스트 케이스를 얻고, 이들을 기반으로 (1) α,β,γ의 테스트 케이스 효율성과 (2) α,β,γ에서 생성되는 *fCU* 수를 비교하여 다음의 결과를 추출해 내었다.

(1) α,β,γ의 테스트 케이스 효율성은 각 α,β,γ에서 생성된 테스트 케이스의 수에 대한 오류를 감지하는 테스트 케이스의 수의 비율로 계산하였다. 그 결과 10개의 *CU*에서 α가 다른 두 경우에 비해 높은 효율성을 나타내었다. 인터페이스 뮤테이션의 경우에는 조립 패턴 구분 없이 *P*와 *R* 모두에 오류를 삽입할 것이므로, α보다는 낮은 효율성을 나타낼 것으로 추측된다.

(2) α,β,γ에서 생성되는 *fCU* 수 비교를 위해 각 α, β,γ에서 만들어지는 *fCU*의 수를 계산하였다. 그 결과 α에서 생성되는 *fCU*의 수가 적게 나타났다. *fCU*를 생성하는데 요구되는 비용을 고려할 때, 본 기법이 비교적 적은 비용을 요구할 것임을 추측할 수 있다. 또한 적은 수의 *fCU*를 생성함으로써, 뮤테이션 테스트 기법이 갖는 computational bottleneck문제를 감소시킬 수도 있다. α에서 평균 하나의 *fCU*를 생성하는 것에 비해, 인터페이스 뮤테이션의 경우에는 조립 패턴 구분 없이 *P*와 *R* 모두에 오류를 삽입할 것이므로, 평균 두 개의 *fCU*를 생성할 것이다.

위의 (1),(2) 두 가지 실험 결과를 통합하여 본 기법의 효율성을 보여주기 위해, "α,β,γ에서 하나의 *fCU*마다 갖는 테스트 케이스 효율성"을 계산하였다. 위의 (1),(2) 결과를 기반으로 계산하여 그림3의 챠트를 얻었다.

(2)의 결과에서 언급한대로, α는 β나 γ보다 적은 수의 *fCU*를 생성한다. 많은 *fCU*중에는 오류를 감지하

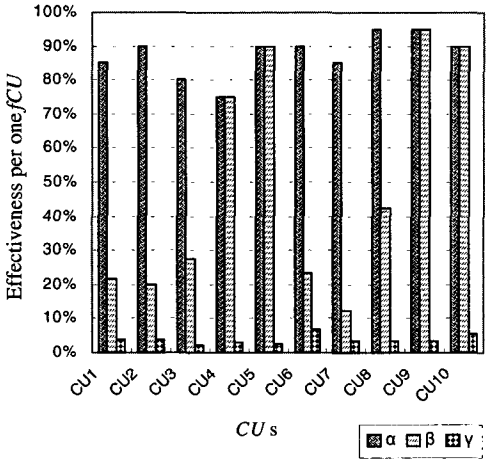


그림 3 α,β,γ에서 하나의 fCU마다 갖는 테스트 케이스 효율성

지 못하는 테스트 케이스를 생성하는 *fCU*가 포함될 수 있기 때문에, 생성되는 *fCU*의 수는 (1)에서 계산한 테스트 케이스의 효율성에 반비례함을 예측할 수 있다. 인터페이스 뮤테이션의 경우, α보다 작은 효율성을 갖으며, α보다 많은 수의 *fCU*를 생성하므로, 이 둘을 이용하여 산정한 하나의 *fCU*마다 갖는 테스트 케이스의 효율성도 작게 나타날 것이다. 그림 3은 하나의 *fCU*가 생성하는 테스트 케이스의 효율성을 보여준다. 즉 이 값이 높은 것은, 높은 효율성 보장하는 *fCU*를 생성한다는 것이고, 이는 높은 효율성을 보장하는 곳에 임의의 오류를 심었음을 나타낸다. 따라서 그림 3에서 보듯이, *FIT*에 오류를 삽입한 경우가 다른 두 경우보다 높은 값을 타나내므로 *FIT*가 테스트 케이스의 효율성을 보장함을 알 수 있다. 이는 *FIT*가 적정수의 *fCU*를 생성하며, 이들이 오류를 감지하는 능력이 있는 테스트 케이스를 선정하기 때문이다.

5. 결론 및 향후 연구 과제

본 논문은 CBSD에서 컴포넌트 조립에 의해 발생하는 오류를 테스트하는 기법과 그의 자동화 방안을 제안하였다. 본 논문은 다음과 같은 세 가지 특징을 갖는다.

첫째, 본 기법은 실제 컴포넌트 아키텍처에 적용될 수 있다. 본 논문에서는 EJB를 기반으로 적용하였으나, 다른 컴포넌트 아키텍처의 특성에 따라 조립 패턴을 추출하고, 그에 대한 *FIT*와 *FIO*를 본 논문에서의 정의에 따라 선정하는 방법으로, CBSD를 지원하는 다른 컴포넌트 아키텍처에도 본 기법을 적용시킬 수 있다.

둘째, 본 기법은 비용 절감 효과를 갖는다. 본 기법의 비용 절감 효과는 CBSD가 갖는 비용 절감 효과와 일치한다. 본 비용 절감 효과는 테스트 케이스의 높은 효율성에서 나오며, 본 논문은 이를 4장의 실험에서 보였다. 본 실험에서 얻은 (1),(2),(3)을 통해, *FIT*가 적정수의 *fBW*를 생성하며, 생성된 *fCU*로부터 오류 감지 능력이 있는 테스트 케이스를 선정함으로써, 높은 테스트 케이스 효율성을 갖음을 확인했다. 즉, *FIT*가 이 효율성을 보장한다.

셋째, 본 논문에서는 제안한 기법을 자동화하는 도구를 구현함으로써, 본 기법을 적용하는 테스트 비용을 절감시킬 수 있도록 한다. 본 도구는 현재 구현중이며, 향후 CBSD의 맞춤 테스트와 조립 테스트 수준의 테스팅 과정을 자동으로 수행하게 된다. 따라서 이는 테스팅 수행 시간을 줄여줌으로써, 비용 절감 효과를 증대시킬 것이며, 이를 통해 본 기법이 CBSD의 비용절감 효과에 더욱 부합하여, CBSD의 개발현장에 더욱 쉽게 적용될 수 있게 할 것으로 기대된다.

참 고 문 헌

[1] Marcio E. Delamaro, Jose C. Maldonado, and Aditya P. Mathur, Interface Mutation: An Approach for Integration Testing, *IEEE Trans. on Software Engineering*, Vol. 27, No. 3, pp.228-247, Mar. 2001.

[2] R.A.DeMillo, R.J.Lipton, and F.G.Sayward, "Hints on Test Data Selection : Help for the Practicing Programmer," *IEEE Computer*, Vol.11, No.4, pp. 34-41, Apr 1978.

[3] Aditia P.Mathur and W.Eric Wong. 1993. Comparing the Fault Detection Effectiveness of Mutation and Data Flow Testing: An Empirical Study. *SERC-TR-146-P*, Dec.29

[4] A.Jefferson Offutt. 1992. Investigations of the Software Testing Coupling Effect. *ACM Trans. on Software Engineering and Methodoloy*, 1(1): 5-20, Jan.

[5] Monica Pawlan, "Writing Enterprise Application with JavaTM 2 Platform, Enterprise Edition," at URL : *http://developer.java.sun.com/developer/onlineTraining/J2EE/Intro/*

[6] Hoijin Yoon and Byoungju Choi, "Inter-class Test Technique between Black-box-class and White-box-class for Component Customization Failures," *Asia-Pacific Software Engineering Conference*, page162-165, Japan, Dec 8-10, 1999.

[7] Hoijin Yoon and Byoungju Choi, "Component Customization Testing Technique Using Fault

Injection Technique and Mutation Test Criteria, i±Proceeding on Mutation2000, Oct. 2000, USA.

윤 희 진

1993년 2월 이화여자대학교 전자계산학과 학사. 1998년 2월 이화여자대학교 대학원 컴퓨터학과 석사. 1998년 3월 ~ 현재 이화여자대학교 대학원 컴퓨터학과 박사과정. 관심분야는 소프트웨어공학, 분산 컴포넌트 테스팅, 테스트 프로세스, 객체지향 소프트웨어 테스팅

최 병 주

1979년 ~ 1983년 이화여대 수학과 학사. 1984년 ~ 1985년 Purdue Univ. Computer Science 학사수료. 1986년 ~ 1987년 Purdue Univ. Computer Science 석사. 1987년 ~ 1990년 Purdue Univ. Computer Science 박사. 1991년 ~ 1992년 삼성종합기술원 1992년 ~ 1995년 용인대 전산통계학과 조교수. 1995년 ~ 현재 이화여대 컴퓨터학과 부교수. 관심분야는 소프트웨어공학, 소프트웨어 테스트, 소프트웨어 및 데이타 품질 측정