pISSN: 2287-5905

http://dx.doi.org/10.3745/KTSDE.2013.2.3.151

Effect Relation-based Coverage and Test Case Generation for GUI Testing of iOS Applications

Yongjin Seo† · Daegeon Mun† · Hyeon Soo Kim††

ABSTRACT

iOS applications refer to the apps operating on iOS, a mobile OS developed by Apple. As iOS provides graphic user interfaces based on touch screens, most of iOS apps support GUIs. GUIs become increasingly important for iOS apps. So are GUI tests. As GUI functions are performed by event handlers, faulty event handlers could cause defects in GUIs. Hence, this study detects faults in event handlers as a way to test GUIs for iOS apps, and suggests how to generate test cases by re-defining input domains of event handlers.

Keywords : iOS Application, GUI Testing, Test Case Generation, Effect Relation Coverage

iOS 애플리케이션 GUI 테스팅을 위한 영향 관계 기반 커버리지 및 테스트 케이스 생성

서 용 진† · 문 대 건† · 김 현 수††

요 약

iOS 애플리케이션은 애플에서 개발한 모바일용 운영체제인 iOS 위에서 동작하는 애플리케이션을 말한다. iOS가 터치스크린을 기반으로 하는 그래픽 사용자 인터페이스를 제공하기 때문에, iOS 애플리케이션은 대부분 GUI를 사용자 인터페이스로 제공한다. iOS 애플리케이션에서 GUI의 비중이 높아질수록 GUI 테스트에 대한 중요성도 높아지게 된다. GUI의 기능은 이벤트 핸들러에 의해 수행되기 때문에 이벤트 핸들러 자체에 결함이 존재한다면, 그것은 곧 GUI의 결함을 유발할 수 있다. 이에 본 논문에서는 이벤트 핸들러 자체의 결함을 검출하는 방식으로 iOS 애플리케이션의 GUI 테스트를 수행하고자 한다. 이를 위해서 이벤트 핸들러의 입력 도메인을 재정의하고 이로부터 테스트 케이스를 생성하는 방법을 제안한다.

키워드 : iOS 애플리케이션, GUI 테스팅, 테스트 케이스 생성, 영향 관계 커버리지

1. 서 론

iOS는 애플에서 개발한 모바일용 운영체제를 말하며, 이를 사용할 수 있는 장치로는 iPhone, iPod Touch, iPad 등이 있다. iOS의 사용자 인터페이스는 멀티 터치 제스처에 의한 직접 조작(Direct Manipulation)을 기반으로 하고 있다[1]. 여기서 직접 조작이란, 사용자에게 시각적인 요소를 제공함으로써, 시스템을 제어할 수 있도록 하는 것을 의미하며[2], 그래픽 사용자 인터페이스(GUI)와 유사한 개념이다. 애플에서는 코코아 터치 프레임워크(Cocoa Touch Framework)를 통하여 사용자에게 직접 조작을 기반으로 한 인터페이스를 제공하고 있다. 코코아 터치 프레임워크는 터치스크린을 내장하고 있는 모바일 장치를 위한 사용자 인터페이스 프레임워크로, 다양한 하위 프레임워크로 구성된다. 그 중에서 UIKit 프레임워크가 iOS 애플리케이션의 사용자 인터페이스를 위해 제공되는 주요 프레임워크이다. UIKit 프레임워크에서는 기본적인 GUI 컴포넌트와 멀티 터치 제스처를 제공한다. 따라서 iOS는 터치스크린 기반의 GUI를 제공하고 있으며, iOS 위에서 동작하는 iOS 애플리케이션 역시 GUI를 사용자 인터페이스로 제공한다.

이와 같은 이유로 iOS 애플리케이션에서 GUI가 갖는 비중은 매우 크다. 따라서 GUI가 문제없이 동작하는지에 따라 iOS 애플리케이션의 품질에 커다란 영향을 미칠 수 있다. iOS 애플리케이션의 품질을 향상시키기 위해서는 GUI가 문제없이 동작한다는 것을 보장할 수 있어야 하며, 이는 곧 GUI에 대한 충분한 테스트가 필요하다는 것을 의미한다.

※ 이 논문은 2010년도 정부(교육과학기술부)의 재원으로 한국연구재단의 기초연구사업 지원을 받아 수행된 것임(No.2010-0025329).
† 준 회 원 : 충남대학교 컴퓨터공학과 석사과정
†† 종신회원 : 충남대학교 컴퓨터공학과 교수
논문접수 : 2012년 10월 5일
심사완료 : 2012년 12월 2일
* Corresponding Author : Hyeon Soo Kim(hskim401@cnu.ac.kr)

iOS 애플리케이션에서 사용자의 입력은 시스템에 의해 이벤트로 변환되어 애플리케이션에 전달된다. 이벤트를 전달 받은 애플리케이션은 이를 처리할 수 있는 GUI 컴포넌트를 찾는다. 만일 적합한 GUI 컴포넌트를 찾으면 해당 GUI 컴포넌트와 연관된 이벤트 핸들러를 통해 이벤트가 처리되며, 그렇지 않으면 해당 이벤트는 무시된다. 이와 같은 처리 과정을 통해서 사용자는 애플리케이션의 기능을 사용하게 된다. 이를 통해서 우리는 두 가지 사실을 알 수 있다. 첫 번째는 이벤트의 전달은 시스템에 의해서 수행되기 때문에 테스트의 범위에서 벗어난다는 점이다. 두 번째는 이벤트는 단순히 애플리케이션의 기능을 사용하기 위한 입력일 뿐이라는 것이다. 반면에, 이벤트 핸들러는 개발자에 의해서 정의되고 실질적으로 애플리케이션의 기능을 수행한다. 이와 같은 사실을 통해서 이벤트보다는 이벤트 핸들러가 GUI에 더 많은 영향을 끼친다는 것을 알 수 있다. 이벤트의 다양한 조합은 GUI가 가지고 있는 결함을 발견할 수 있게 도와주지만, 이벤트 핸들러의 결함은 그 자체가 GUI의 결함이 되기 때문이다. 따라서 효과적인 GUI 테스트를 수행하기 위해서 이벤트 핸들러를 중심으로 GUI 테스트가 이루어져야 한다.

본 논문에서는 이벤트 핸들러의 결함을 검출함으로써 iOS 애플리케이션의 GUI 테스트를 수행하고자 한다. 이를 위해 이벤트 핸들러의 입출력 도메인을 재정의하고, 이를 기반으로 테스트 케이스를 생성하는 방법을 제안한다.

2. 관련 연구

GUI는 기본적으로 이벤트의 순차적인 입력을 통해 동작하기 때문에, GUI 테스트에 대한 연구는 일련의 이벤트를 생성하는 것을 중심으로 진행되어 왔다. 따라서 이러한 연구를 통해 생성된 테스트 케이스는 주로 일련의 이벤트가 되었다. 각 연구들은 사용되는 기술에 따라 몇 가지로 분류할 수 있는데, 첫 번째로 Record-Playback을 활용한 연구[3-5]를 들 수 있다. Record-Playback은 사용자가 입력한 일련의 이벤트를 저장한 뒤, 저장된 일련의 이벤트를 순서대로 발생시킴으로써 GUI 테스트를 수행하는 기술이다. 다음으로 모델을 기반으로 테스트 케이스를 생성하는 연구[6-9]가 존재한다. 해당 연구들은 이벤트에 따라 변화하는 GUI의 상태에 초점을 맞추어 진행된다. 이벤트에 따라 변하는 GUI의 상태를 모델로 표현하고, 생성된 모델을 기반으로 테스트 케이스를 생성한다. 이 때, 사용되는 모델은 Event Flow Graph(EFG), State Diagram, Finite State Machine Model 등이 존재한다. 마지막으로 무작위로 이벤트를 발생시켜 GUI의 결함을 검출하려는 연구[10]도 있다. 이 연구에서는 구글에서 제공하는 Monkey 도구를 이용하여 무작위로 이벤트를 발생시켜 애플리케이션의 GUI를 테스트한다. 앞서 나열한 연구들은 그 방법은 서로 다르지만, 결론적으로 테스트 케이스로써 일련의 이벤트를 생성한다는 것이 동일하다. 그러나 이벤트를 통해서 GUI가 동작하는 것은 맞지만, 이를 통해서 GUI 테스트가 충분히 수행되었다고 볼 수 없다. 그 이유는 입력 도메인이 너무 방대하기 때문이다. 한 애플리케이션에서 발생할 수 있는 이벤트의 개수가 적더라도 그 조합의 수는 기하급수적으로 늘어나게 된다. 따라서 생성된 테스트 케이스를 통해서 애플리케이션의 GUI를 테스트 하였을 때, 충분히 수행되었음을 보장하기 어렵다. 더 나아가 연구 [10]의 경우에는 생성된 테스트 케이스가 의미있다고 보기 힘들다.

논문 [11]에서는 위와 같은 기존 연구의 단점을 보완하기 위해서 이벤트 핸들러 기반의 테스트 커버리지(test coverage)를 제안하고 있다. [11]에서는 두 가지 사실을 기반으로 연구를 진행하였다. 첫 번째는 특정 GUI 컴포넌트에 대해서 발생한 특정 이벤트를 통해서 실행되는 이벤트 핸들러는 항상 동일하다는 것이며, 두 번째는 경우에 따라 이벤트 핸들러 간에 의존성이 존재한다는 것이다. 이와 같은 사실은 바로 이벤트 간의 의존성이 존재한다는 것으로 귀결된다. [11]에서는 이와 같은 의존성을 "Definition-Use" 관계로 표현하고 있다. 즉, 한 이벤트 핸들러에서 정의된 변수의 값을 다른 이벤트 핸들러에서 사용하는 경우에 의존성이 존재한다고 정의한다는 것이다. 이러한 사실은 일련의 이벤트를 생성할 때, 그 조합의 수를 줄여줄 뿐만 아니라 의미 있는 일련의 이벤트를 생성할 수 있음을 알려준다.

그러나 논문 [11]에서 언급했던 것처럼 발생된 이벤트를 통해 실행되는 이벤트 핸들러에서는 어떠한 값을 생성하거나 사용하는 과정이 발생하게 된다. 따라서 이벤트 자체보다는 이벤트 핸들러와 관련된 값이 이벤트 핸들러에 더 많이 영향을 끼칠 수 있다고 볼 수 있다. 그렇기 때문에 단순히 이벤트의 조합을 발생시켜 이벤트 핸들러를 동작시키는 것보다는 이벤트 핸들러와 관련된 값의 변화에 따라 이벤트 핸들러가 문제를 일으키지 않는지 확인하는 것이 더 효과적이라고 할 수 있다. 따라서 본 논문에서는 위와 같은 사실을 토대로 이벤트에 한정되었던 입력 도메인을 재정의하고자 한다. 또한 이벤트 핸들러 자체의 결함을 검출하는 방법으로 GUI 테스트를 수행하고자 한다.

3. iOS 애플리케이션의 구조

이번 절에서는 iOS 애플리케이션의 구조에 대해서 알아본다. 먼저 iOS의 주요 디자인 패턴인 MVC 패턴에 대해 알아본 뒤, iOS 애플리케이션의 구조에 대해서 알아본다.

3.1 MVC 패턴

MVC 패턴[13][14]에서는 애플리케이션을 세 가지 컴포넌트로 나누고 있으며, 각 컴포넌트는 모델(Model), 뷰(View), 컨트롤러(Controller)가 된다. 일반적으로 모델은 애플리케이션의 주요 데이터와 기능을 포함하고, 뷰는 모델로부터 데이터를 얻어 사용자에게 출력하는 역할을 담당한다. 마지막으로 컨트롤러는 사용자로부터 입력을 받아 이를 처리하는 역할을 담당한다.

그러나 코코아 터치 프레임워크에 적용된 MVC 패턴은 기존 MVC 패턴에서 변형된 구조를 갖는다. Fig. 1과 같은 구조로, 뷰가 모델로부터 직접 데이터를 얻어오는 것이 아니라 컨트롤러를 통해 간접적으로 데이터를 얻어오는 방식으로 동작한다. 이와 같은 MVC 패턴을 "Passive View"라고 부르며[13], 이를 통해 모델과 뷰의 재사용성을 향상시킬 수 있다. 대신 컨트롤러가 중간에서 모델과 뷰의 상태를 동기화시키는 역할을 담당한다.

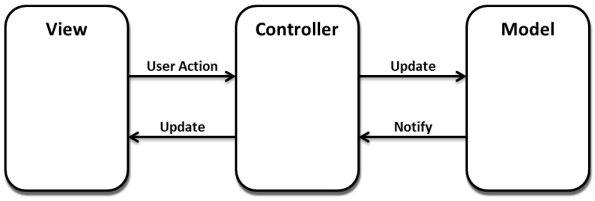


Fig. 1. MVC Pattern in Cocoa Touch Framework

3.2 iOS 애플리케이션 구성 요소

앞서 살펴본 MVC 패턴을 통해서 iOS 애플리케이션이 크게 모델 컴포넌트, 뷰 컴포넌트, 컨트롤러 컴포넌트로 구성된다는 것을 알 수 있다. 또한 애플에서 제공하는 개발자 가이드 문서[16]를 보면 iOS 애플리케이션의 구성 요소를 Fig. 2와 같이 표현하고 있다.

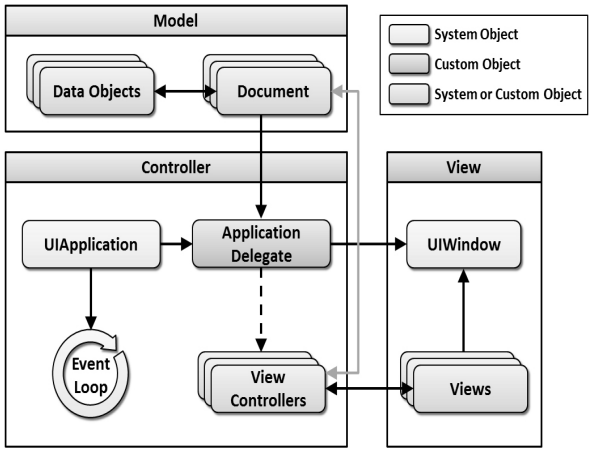


Fig. 2. Component of iOS Application

iOS 애플리케이션은 휴대용 장치에서 동작하기 때문에, 휴대용 장치의 제약적인 화면 크기에 맞는 GUI를 제공해야 한다. 따라서 GUI가 여러 개의 화면을 나뉘어 표현된다. 이와 같은 내용은 Fig. 2에서도 확인할 수 있다. Fig. 2의 뷰 컴포넌트를 보면 UIWindow 객체와 여러 개의 뷰로 구성되어 있음을 알 수 있다. 여기서 뷰는 단일 화면을 의미하며, UIWindow 객체는 애플리케이션의 모든 뷰를 관리한다. 그림에는 나타나지 않았지만, 뷰는 버튼, 텍스트필드, 토글스위치와 같은 하위 요소들로 구성되며, 이러한 요소를 UI Controls이라 부른다.

3.3 iOS 애플리케이션의 이벤트 핸들러

iOS 애플리케이션에서는 이벤트를 처리하기 위해서 세 가지의 이벤트 핸들러를 사용한다. 각각은 Action, Delegate, DataSource이며, 정의는 다음과 같다.

- **Action**은 UI Controls에서 특정 이벤트가 발생했을 때, 해당 이벤트를 처리하는 사용자 정의 함수를 말한다. Action은 이벤트가 발생한 UI Controls에 대한 정보만을 파라미터로 받는다.
- **Delegate**는 UI Controls에서 처리해야 하는 이벤트의 집합을 정의하고 있는 클래스이다. 이벤트 핸들러는 Delegate 내부에 정의되어 있다. 개발자는 이를 상속받아 이벤트 핸들러를 재정의함으로써 이벤트를 처리할 수 있다. Action과 달리 파라미터로 UI Controls 외의 다른 정보를 요구하는 경우가 존재한다.
- **DataSource**는 데이터의 출력과 관련된 이벤트 핸들러의 집합을 정의하고 있는 클래스이다. 데이터의 출력을 담당한다는 점 외에는 Delegate와 동일한 속성을 갖는다.

iOS 애플리케이션의 UI Controls는 각자의 특성에 따라 고유한 이벤트 핸들러를 사용한다. Table 1은 UI Controls마다 사용할 수 있는 이벤트 핸들러를 정리한 것이다.

Table 1. Event Handler Types of UI Controls

UI Controls	Event Handler
Button	Action
Segmented Control	Action
Text Field	Delegate
Slider	Action
Switch	Action
Page Control	Action
Table View	Delegate, DataSource
Text View	Delegate
Web View	Delegate
Map View	Delegate
Scroll View	Delegate
Date Picker	Action
Picker View	Delegate, DataSource
Ad Banner View	Delegate
Navigation Bar	Delegate
Search Bar	Delegate

4. 이벤트 핸들러 기반의 테스트 케이스 생성 기법

본 연구의 목적은 이벤트 핸들러에 대한 테스트를 수행함으로써 iOS 애플리케이션의 GUI에 대한 품질을 보장하는 것에 있다. 이를 위해서 본 논문에서는 이벤트 핸들러에 대해서 단위 테스트를 수행하고자 한다. 특정 함수에 대한 단위 테스트를 수행할 때는 보통 함수가 갖는 파라미터를 입력 도메인으로 활용한다. 그러나 iOS 애플리케이션의 이벤트 핸들러가 갖는 파라미터는 대부분 이벤트가 발생한 UI Controls에 대한 정보뿐이다. 이와 같은 정보를 기반으로 단위 테스

트를 수행하는 것은 이벤트를 입력 도메인으로 사용하는 것과 다르지 않으므로, 기존 연구와 차별성이 없다. 따라서 본 논문에서는 이벤트 핸들러의 입력 도메인을 재정의하고, 이로부터 테스트 케이스 생성 전략을 제안하고자 한다.

4.1 이벤트 핸들러의 입력 도메인 재정의

우리는 [11]의 연구 내용으로부터 이벤트 핸들러에 대한 몇 가지 흥미로운 사실을 도출할 수 있었다. 첫 번째는 이벤트 핸들러가 동작하는데 있어 이벤트가 미치는 영향이 크지 않다는 것이다. 두 번째는 이벤트 핸들러의 동작에 많은 영향을 주는 것은 애플리케이션이 가지고 있는 특정 데이터라는 것이다. [11]에서는 이벤트와 이벤트 핸들러의 관계를 $f_{EH}: E \rightarrow H$로 정의하고 있다. 여기서 E는 이벤트의 집합이며, H는 애플리케이션이 갖고 있는 모든 이벤트 핸들러의 집합을 의미한다. 이 정의를 통해 이벤트를 통해서 이벤트 핸들러가 동작한다는 사실을 알 수 있다. 그러나 이벤트의 역할은 여기까지이다. [11]에서 이벤트 핸들러 간의 의존성을 나타내는 Definition Use Handler Pair에서는 이벤트에 대해 고려하지 않는다. Definition Use Handler Pair를 도출할 때, 중요한 것은 “이벤트 핸들러 간에 공통적으로 사용하는 데이터가 존재하는가”이다. 따라서 [11]에서 Definition Use Handler Pair에 대한 테스트 커버리지를 고려하는 것은 이벤트 핸들러 내에서 다루는 데이터가 변경되는 것이 이벤트 핸들러가 동작하는데 있어 영향을 줄 수 있다고 판단했기 때문이다. 이는 곧 특정 데이터가 이벤트 핸들러의 동작에 영향을 미친다는 의미이다. 논문 [12]에서도 유사한 이야기를 하고 있다. [12]에서는 이벤트가 처리되는 과정에서 애플리케이션 상태를 변경할 수 있으며, 이는 곧 다른 이벤트의 처리에 영향을 미칠 수 있다고 말한다. 이벤트의 처리는 이벤트 핸들러에 의해서 이루어지므로, 결국 애플리케이션 상태가 이벤트 핸들러의 동작에 영향을 미칠 수 있음을 의미한다.

따라서 본 논문에서는 이벤트 핸들러에 대한 단위 테스트를 수행할 때, 이벤트 핸들러 내에서 다루는 데이터를 입력 도메인으로 활용하는 것이 더 효과적이라고 판단하였으며, 이를 다음과 같이 정의한다. 파라미터가 포함되는 이유는 Delegate나 DataSource의 경우에는 이벤트 핸들러의 기능에 영향을 줄 수 있는 파라미터가 존재하기 때문이다.

Definition 파라미터 v를 갖는 이벤트 핸들러 h에 대해서 내부에서 사용되는 애플리케이션 상태의 집합을 S_u라 하고 내부에서 변경되는 애플리케이션 상태의 집합을 S_m라고 할 때, 우리는 이벤트 핸들러 h에 대해서 $h: (S_u, v) \rightarrow S_m$라고 정의할 수 있다.

애플리케이션 상태는 [12]에서 가져온 개념으로 이벤트 핸들러 내부에서 사용 혹은 변경되는 데이터를 의미한다. 그렇다면 어떤 데이터가 애플리케이션 상태에 해당될 수 있을까? 이는 Fig. 2를 통해 알 수 있다. iOS 애플리케이션은 기본적으로 MVC 패턴을 따르고 있다. 뷰 컴포넌트를 통해 받은 사용자의 입력은 컨트롤러 컴포넌트에서 처리되며, 이 과정에서 모델 컴포넌트나 뷰 컴포넌트가 변경될 수 있다. 예를 들어, 계수기 애플리케이션을 생각해 보자. 계수기 애플리케이션은 텍스트필드 T와 버튼 A, B로 구성된 뷰와 계수 이력이 저장되는 모델 클래스 M을 가지고 있다. 버튼 A를 누르면 텍스트필드 T에 적힌 숫자가 하나씩 증가한다. 버튼 B를 누르면 텍스트필드 T의 값이 모델 클래스 M에 저장된 후, 0으로 초기화된다. 이 때, 버튼 A와 연결된 이벤트 핸들러는 기능을 수행하는 과정에서 텍스트필드 T의 값을 변경시킨다. 또한 버튼 B와 연결된 이벤트 핸들러는 기능을 수행하는 과정에서 텍스트박스 T의 값과 모델 클래스 M의 값을 변경시킨다. 이를 통해서 이벤트 핸들러에 의해 애플리케이션이 갖는 모델 컴포넌트와 뷰 컴포넌트가 변경될 수 있음을 알 수 있다. 여기서 변경의 범위에 주목할 필요가 있다. 계수기 애플리케이션에서 버튼 A를 눌렀을 때, 발생되는 변경의 범위는 뷰를 구성하는 UI Controls의 속성에 한정된다. 모델 클래스 M도 마찬가지로, 변경의 범위가 전체가 아닌 일부 속성이 된다. 따라서 이벤트 핸들러 내부에서 사용되는 데이터는 뷰 컴포넌트와 모델 컴포넌트가 가지고 있는 속성 값이라고 볼 수 있다.

4.2 이벤트 핸들러 기반의 테스트 케이스

재정의된 이벤트 핸들러의 입력 도메인을 통해 애플리케이션 상태를 이벤트 핸들러의 파라미터로 가정하여 테스트를 수행할 수 있다.

예를 들어, Fig. 3의 이벤트 핸들러 h_1은 iOS 애플리케이션이 갖는 Action 타입의 이벤트 핸들러다. 이벤트 핸들러 h_1이 갖는 애플리케이션 상태 S_u는 TextField_Name의 text 값과 TextField_PhoneNumber의 text 값이며, 각각은 UI Controls의 속성 값이다. 이를 통해서 이벤트 핸들러 h_1을 이벤트 핸들러 h_2와 같이 변경하여 생각해 볼 수 있다. 이벤트 핸들러 h_1을 이벤트 핸들러 h_2와 같이 변경함으로써 이벤트 핸들러에 대한 단위 테스트를 일반적인 함수에 대한 단위 테스트와 동일하게 바라볼 수 있다. 즉, 각 애플리케이션 상태가 갖는 값의 조합으로 테스트 케이스를 구성할 수 있으며, 이 때, 테스트 케이스 $t = \langle d_1, d_2, ..., d_n \rangle$로 표현할 수 있다.

그러나 애플리케이션 상태가 갖는 값은 방대하기 때문에, 이를 이용하여 의미 있는 조합을 구성하기란 쉽지 않다. 따라서 일반적으로 함수에 대한 단위 테스트를 수행할 때, 사용하는 다양한 테스트 방법[17] 중에서 본 논문에서는 블랙박스 테스트 기법 중 하나인 동등 분할 기법을 사용하고자 한다. 이를 통해서 애플리케이션 상태를 매개변수로 사용하는 이벤트 핸들러에 대해서 다음과 같은 테스트 케이스 t를 갖는다고 할 수 있다.

$$t = \langle c_1, c_2, ..., c_n \rangle \quad (1)$$

Equation (1)에서 $c_1, c_2, ..., c_n$은 애플리케이션 상태가 갖는 동등 분할 클래스를 의미한다. 즉, 각 애플리케이션 상태가

Event Handler h_1

```
- (IBAction)requestRegister
{
  if([TextField_Name.text isEqual:@""] || [TextField_PhoneNumber.text isEqual:@""])
  {
    NSLog(@"Check value of textfield");
  }
  else
  {
    [[[MyInfoManager myInfoManager] myInfo] setName:TextField_Name.text];
    [[[MyInfoManager myInfoManager] myInfo] setPhoneNumber:TextField_PhoneNumber.text];
  }
}
```

Event Handler h_2

```
- (IBAction)requestRegister:(NSString *)name  phone:(NSString *)phoneNumber
{
  if([name  isEqual:@""] || [phoneNumber  isEqual:@""])
  {
    NSLog(@"Check value of textfield");
  }
  else
  {
    [[[MyInfoManager myInfoManager] myInfo] setName: name];
    [[[MyInfoManager myInfoManager] myInfo] setPhoneNumber:phoneNumber];
  }
}
```

Fig. 3. Redefinition of Event Handler

갖는 동등 분할 클래스의 조합을 통해서 테스트 케이스를 표현할 수 있다.

4.3 애플리케이션 상태 간의 영향 관계를 이용한 커버리지

동등 분할 기법을 이용하여 테스트 케이스를 생성하는 가장 간단한 방법은 모든 동등 분할 클래스의 조합을 고려하는 것이다. 그러나 모든 조합을 고려하는 것은 경우에 따라 불필요하거나 중복된 결과를 도출하는 조합의 테스트 케이스를 생성한다. 따라서 적지 않은 수의 테스트 케이스를 생성하기 때문에, 생성되는 테스트 케이스의 수를 줄일 필요가 있다. 앞서 테스트 케이스를 생성할 때, 모든 조합을 고려한 이유는 애플리케이션 상태 S_u와 파라미터 v가 애플리케이션 상태 S_m에 어떤 영향을 미칠 수 있는지 예측할 수 없기 때문이었다. 이를 달리 생각하면, 만일 애플리케이션 상태 S_u와 파라미터 v, 그리고 애플리케이션 상태 S_m 사이의 영향 관계를 파악할 수 있다면, 모든 조합을 고려하는 것에 비해 적지만 더 의미 있는 테스트 케이스를 생성할 수 있다는 의미가 된다.

따라서 본 논문에서는 테스트 케이스의 수를 줄이기 위해서 애플리케이션 상태 S_u와 파라미터 v, 그리고 애플리케이션 상태 S_m 사이의 영향 관계를 정의하고자 한다. 그리고 정의된 영향 관계를 기반으로 테스트 커버리지의 기준을 정의한다.

1) 애플리케이션 상태 간의 영향 관계

이벤트 핸들러는 애플리케이션 상태 S_u와 파라미터 v의 값을 이용하여 기능을 수행하며, 그 결과로 애플리케이션 상태 S_m의 값이 변경된다. 다시 말해, 애플리케이션 상태 S_u와 파라미터 v는 직간접적으로 애플리케이션 상태 S_m에 영향을 주게 된다. 예를 들어, 두 가지 경우가 존재할 수 있다. 첫 번째 경우는 애플리케이션 상태 S_u와 파라미터 v을 이용한 연산의 결과를 애플리케이션 상태 S_m에 대입하는 경우이다. 여기서 연산은 기본적으로 제공되는 연산자를 이용한 사칙연산이 될 수도 있고 특정 함수를 이용하는 연산이 될 수도 있다. 이 경우에는 직접적으로 영향을 준다고 판단할 수 있다. 두 번째 경우는 간접적으로 영향을 주는 경우로써, 조건문이나 반복문의 조건식으로 애플리케이션 상태 S_u와 파라미터 v를 이용한 경우이다. 첫 번째 경우에 해당하는 문장이 조건문이나 반복문 내부에 존재할 경우, 조건식이 간접적으로 애플리케이션 상태 S_m이 변경되는데 영향을 줄 수 있다. 이는 조건식이 성립하는 경우에만 내부 문장이 수행될 수 있기 때문이다.

따라서 애플리케이션 S_m는 애플리케이션 상태 S_u와 파라미터 v으로부터 직간접적으로 영향을 받는다는 것을 알 수 있다. 또한 두 번째 경우를 통해서 애플리케이션 상태 S_m가 애플리케이션 상태 S_u와 파라미터 v가 가질 수 있는 모든 값에 대해서 영향을 받는 것이 아니라 특정 조건에 부합하는 값을 갖는 경우에 대해서만 영향을 받는 경우가 존재한다는 사실도 알 수 있다. 이와 같은 사실로부터 우리는 애플리케이션 상태 S_u와 파라미터 v, 그리고 애플리케이션 상태 S_m 사이의 영향 관계에 대해서 다음과 같이 정의한다.

Definition 파라미터 v를 갖는 이벤트 핸들러 h가 기능을 수행하는 과정에서 애플리케이션 상태 S_u나 파라미터 v의 동등 분할 클래스 $c_1, c_2, \ldots, c_n$에 속하는 값으로 인해 애플리케이션 상태 S_m의 값이 변경되었을 때, 우리는 애플리케이션 상태 S_u와 파라미터 v가 애플리케이션 상태 S_m와 영향 관계 $(c_1, c_2, \ldots, c_n) \in R$가 존재한다고 말한다.

동등 분할 클래스는 입력 값에 대해서 내부적으로 동일한 방식으로 처리될 것이라 판단되는 값의 집합을 의미하며, 일반적으로 입력 값의 특정 범위로 표현된다. 따라서 동등 분할 클래스는 애플리케이션 상태 S_m에 영향을 주는 애플리케이션 상태 S_u나 파라미터 v의 값의 범위를 표현하기에 적합하며, 이러한 이유로 영향 관계를 정의하는데 사용되었다. 또한 앞서 설명한 경우를 통해서 영향 관계 R을 두 가지로 분류할 수 있으며, 각각은 직접적인 영향 관계 R_D와 간접적인 영향 관계 R_I가 된다.

2) 영향 관계 기반의 테스트 커버리지

이번 절에서는 앞서 정의한 영향 관계를 기반으로 이벤트 핸들러의 단위 테스트에 대한 커버리지를 정의한다. 영향 관계는 이벤트 핸들러의 동작에 직간접적으로 영향을 주는 입력 값의 집합이기 때문에, 영향 관계를 포함하는 테스트 케이스를 통해 테스트가 수행되었다고 한다면 이벤트 핸들러의 동작을 모두 확인하였다고 볼 수 있다. 따라서 영향 관계를 테스트 커버리지의 기준으로 삼고자 한다. 테스트 케이스 T에 대한 영향 관계 커버리지의 충족도는 다음과 같다.

$$Coverage(T) = |R_c| / |R_t| \quad (2)$$

Equation (2)에서 R_c는 테스트 케이스에 의해 충족된 영향 관계의 집합을 의미하며, R_t는 모든 영향 관계의 집합을 의미한다. 따라서 우리는 영향 관계 커버리지의 충족도를 계산하기 위해서 두 가지 물음에 대해서 대답할 수 있어야 한다. 첫 번째는 "이벤트 핸들러 h가 갖는 영향 관계를 어떻게 도출할 수 있는가?"이며, 두 번째는 "테스트 케이스가 영향 관계를 충족한다는 것을 어떻게 판단할 것인가?"이다. 이에 본 논문에서는 이벤트 핸들러로부터 영향 관계를 도출하는 방법과 테스트 케이스가 영향 관계를 충족함에 대해서 정의하고자 한다.

a) 직간접적인 영향 관계 도출 방법

영향 관계는 직접적으로 영향을 미치는지 간접적으로 영향을 미치는지에 따라서 도출할 수 있는 위치가 다르다. 직접적인 영향 관계 R_D는 애플리케이션 상태의 변경이 발생할 수 있는 대입 연산, Set 함수의 호출과 같은 부분에서 도출할 수 있는 반면 간접적인 영향 관계 R_I는 분기문이나 반복문의 조건식에서 도출할 수 있다. 따라서 본 논문에서는 영향 관계를 도출하기 위해서 직접적인 영향 관계와 간접적인 영향 관계로 나누어 도출하고자 한다. 이후에 추가적인 병합 연산을 통해서 영향 관계 R을 도출한다.

먼저, 간접적인 영향 관계를 도출하는 방법을 알아본다. Fig. 4는 이벤트 핸들러가 갖고 있는 간접적인 영향 관계를 도출하기 위한 알고리즘이다.

Algorithm – get a Indirect Effect Relation R_I:
In : event handler h
Out : void

```
1  function getIndirectEffectRelation(h) {
2      BB = constuctBasicBlock(h);
3      for each b∈BB do {
4          R_I^b = ⋂_{a∈predecessor(b)} COND(a, b);
5          for each c∈successor(b) do {
6              COND(b, c) =
                   M(R_I^b, getEquiClass(c, cond));
7          }
8      }
9  }
```

Fig. 4. Algorithm to Derive the Indirect Effect Relation

2번째 줄의 constructBasicBlock()은 주어진 이벤트 핸들러 h에 대한 기본 블록(Basic Block)을 구성한다. 단일 기본 블록은 내부에 순차적인 프로그램의 흐름만을 담고 있으며, 분기나 반복과 같은 예외적인 흐름은 기본 블록 간의 관계를 통해 표현하게 된다. 결론적으로 간접적인 영향 관계를 도출하는 것은 조건식에서 사용되는 애플리케이션 상태 S_u와 파라미터 v를 파악하는 것과 같기 때문에 이를 수이 판단할 수 있도록 기본 블록을 사용하였다. COND(b_i, b_j)는 기본 블록 b_i와 b_j 사이에서 이동이 발생하기 위한 조건을 의미하며, b_i는 b_j의 Predecessor이다. 6번째 줄의 getEquiClass()는 주어진 조건식 $cond$를 바탕으로 주어진 기본 블록 c에 맞는 동등 분할 클래스를 도출하는 작업을 수행한다. 또 $M()$이 사용된 것을 볼 수 있는데, 이는 "b) 영향 관계 도출 방법"에서 자세히 다루도록 한다. Fig. 4의 알고리즘을 통해서 기본 블록의 간접적인 영향 관계는 자신의 Predecessor로부터 전달된 COND의 조합으로 결정된다는 것을 알 수 있다.

다음으로 직접적인 영향 관계를 도출하는 방법을 알아본다. Fig. 5는 이벤트 핸들러가 갖고 있는 직접적인 영향 관계를 도출하기 위한 알고리즘이다.

Algorithm – get a Direct Effect Relation R_D:
In : event handler h
Out : void

```
1  function getDirectEffectRelation(h) {
2      BB = constuctBasicBlock(h);
3      for each b∈BB do {
4          for each stmt∈b do {
5          /* S_i = {c_1, c_2, ..., c_n}, S_i∈S_u ∨ S_i∈v */
6              for each S_i∈RHS in stmt do {
7                  R_D^b = ∏_i S_i;
8              }
9          }
10     }
11 }
```

Fig. 5. Algorithm to Derive the Direct Effect Relation

직접적인 영향 관계는 대부분 대입 연산과 관련된 문장으로부터 도출할 수 있기 때문에, Fig. 5의 알고리즘은 기본 블록을 구성하는 문장 중에서 RHS(Right Hand Side)에 속하는 애플리케이션 상태 S_u와 파라미터 v를 찾아 각각의 동등 분할 클래스를 조합하는 방식으로 구성되었다.

b) 영향 관계 도출 방법

우리는 Fig. 4과 Fig. 5의 알고리즘을 통해서 간접적인 영향 관계 R_I와 직접적인 영향 관계 R_D를 도출할 수 있다. 이제 도출된 직간접적인 영향 관계에 대해 병합연산을 수행하여 영향 관계 R를 도출하여야 한다. 병합 연산을 수행하기에 앞서 간접적인 영향 관계 R_I와 직접적인 영향 관계 R_D의 관계를 알아볼 필요가 있다.

간접적인 영향 관계 R_I는 단일 기본 블록 내의 정보만이 아니라 자신의 Predecessor에 해당하는 기본 블록의 정보를 바탕으로 도출된다. 반면에 직접적인 영향 관계 R_D는 단일 기본 블록 내의 정보만으로 도출된다. 즉, 간접적인 영향 관계 R_I는 기본 블록 간의 정보를 의미하며, 해당 정보로부터 외부로부터 기본 블록 내에 들어올 수 있는 값의 범위를 알 수 있다. 따라서 단일 기본 블록은 자신과 관련된 간접적인 영향 관계 R_I의 범위에 해당하는 값만이 입력될 수 있으며, 단일 기본 블록 내의 정보만으로 도출되는 직접적인 영향 관계 R_D 역시 간접적인 영향 관계 R_I의 범위에 해당하는 값으로 재구성되어야 한다.

또한 병합 연산을 수행하는 과정에서 두 가지의 예외 상황이 발생할 수 있다. 첫 번째 경우는 기본 블록 내에 직접적인 영향 관계 R_D가 존재하지 않는 경우이다. 이와 같은 경우는 해당 기본 블록 내에서 애플리케이션 상태 S_m의 변경이 발생하지 않을 때 발생한다. 애플리케이션 상태 S_m의 변경이 발생하지는 않지만, 이 경우에도 최소한 한 번의 테스트가 수행되어야 한다. 따라서 해당 기본 블록과 관련된 간접적인 영향 관계 R_I가 갖는 요소 중에서 하나를 영향 관계 R로 도출한다. 두 번째 경우는 기본 블록 내에 간접적인 영향 관계 R_I가 존재하지 않는 경우이다. 이 경우는 기본 블록이 분기문이나 반복문에 해당하지 않는 경우로, 직접적인 영향 관계 R_D 자체가 영향 관계 R로 도출된다.

본 논문에서는 위와 같은 내용을 기반으로 병합 연산 $merge(R_I, R_D)$을 정의하였으며, 이를 통해 영향 관계는 $R = \bigcup_{b \in BB} merge(R_I^b, R_D^b)$와 같이 도출된다. 합병 연산 $merge(R_I, R_D)$에 대한 상세한 알고리즘은 Fig. 6에 나타난다.

Algorithm – merge a Indirect Effect Relation and Direct Effect Relation:
In : indirect effect relation R_I, direct effect relation R_D
Out : effect relation R

```
1  function merge(R_I, R_D) {
2      R = ∅;
3      if (R_D = ∅) {
4          R = selectEffectRelation(R_I);
5      }
6      else if (R_I = ∅) {
7          R = R_D;
8      }
9      else {
10         eliminateEffectRelation(R_I, R_D);
11         R = M(R_I, R_D);
12     }
13     return R;
14 }
```

Fig. 6. Algorithm to Merge the Indirect Effect Relation and Direct Effect Relation

4번째 줄의 selectEffectRelation()은 간접적인 영향 관계 R_I가 갖는 요소 중에서 하나를 반환하는 작업을 수행한다. 이는 병합 연산 중에 발생할 수 있는 예외 상황 중 첫 번째 경우를 해결하기 위해 사용된다. 10번째 줄의 eliminate EffectRelation()는 직접적인 영향 관계 R_D가 간접적인 영향 관계 R_I의 범위를 벗어나는 값을 갖는 경우, 이에 해당하는 값을 제거하는 작업을 수행한다. 그 후, $M(R_x, R_y)$를 통해 영향 범위 R을 도출한다. 여기서 $M(R_x, R_y)$는 다음과 같이 동작한다.

Equation (3)은 동일한 애플리케이션 상태로부터 추출된 동등 분할 클래스 간의 조합이 발생되지 않도록 하기 위해

$$
\begin{aligned}
&where: \\
&\quad n = |S_u \cup v| \\
&\quad R_x = \{(c_1,...,c_k) | \forall k(c_k \in S_k, S_{k-1} \neq S_k), 1 \le k \le n\} \\
&\quad R_y = \{(c_1,...,c_l) | \forall l(c_l \in S_l, S_{l-1} \neq S_l), 1 \le l \le n\} \\
&M(R_x, R_y) = \begin{cases} \varnothing & , \exists (c_i, c_j), S_i = S_j \wedge c_i \neq c_j, 1 \le i \le k, 1 \le j \le l \\ R_x \cup R_y \end{cases}
\end{aligned}
\tag{3}
$$

정의되었다. 예시를 위해 애플리케이션 상태 $S_u = \{S_1, S_2, S_3\}$를 갖는 이벤트 핸들러가 존재한다고 하자. 각 애플리케이션 상태가 갖는 동등 분할 클래스는 $S_1 = \{c_1, c_2\}$, $S_2 = \{c_3, c_4, c_5\}$, $S_3 = \{c_6, c_7, c_8\}$와 같다. 예를 들어 (c_1, c_4)와 (c_1, c_8)에 대해서 $M(R_x, R_y)$ 연산을 수행하면 (c_1, c_4, c_8)의 결과를 얻어낼 수 있다. 그러나 (c_1, c_4)와 (c_2, c_4, c_8)에 대해서 $M(R_x, R_y)$ 연산을 수행하면 $\varnothing$를 반환한다. c_1와 c_2는 동일한 애플리케이션 상태 S_1의 동등 분할 클래스이기 때문에, c_1와 c_2의 조합이 발생된다면 S_1는 동시에 두 개의 값을 갖는 모순이 발생한다. 따라서 이와 같은 상황을 피하기 위해 $M(R_x, R_y)$ 연산을 정의한 것이다.

c) 영향 관계 커버리지 충족 요건

본 논문에서 제안한 이벤트 핸들러 기반의 테스트 케이스는 애플리케이션 상태 S_u와 파라미터 v가 갖는 동등 분할 클래스의 조합으로 구성된다. 영향 관계도 마찬가지로 동등 분할 클래스로 구성된다. 이처럼 테스트 케이스와 영향 관계는 서로 구성하는 요소가 동일하기 때문에 테스트 케이스가 영향 관계를 충족하는지 확인하는 작업은 서로 일치하는지 확인하는 것과 같다. 그러나 특정 이벤트 핸들러에 대한 테스트 케이스의 길이는 항상 일정한 것과 달리 영향 관계는 기본 블록마다 영향 받는 애플리케이션 상태 S_u와 파라미터 v의 종류가 다르기 때문에 그 길이가 일정하지 않다. 따라서 테스트 케이스의 일부와 영향 관계의 튜플을 구성하는 요소가 완전히 일치할 때 우리는 테스트 케이스가 영향 관계를 충족한다고 말할 수 있다. 이와 같은 사실을 통해 우리는 테스트 케이스 t가 영향 관계 $R_n \in R$를 충족함에 대해서 다음과 같이 정의할 수 있다.

Definition 테스트 케이스 $t = \langle c_1, c_2, ..., c_n \rangle$와 영향 관계 $(c_1, c_2, ..., c_m) \in R$에 대해서 $M(t, (c_1, c_2, ..., c_m)) = \varnothing$을 만족한다면, 우리는 테스트 케이스 t가 영향 관계 R을 충족한다고 말한다.

테스트 케이스와 영향 관계 간에 서로 동일하지 않은 동등 분할 클래스가 존재한다면, $M(t, (c_1, c_2, ..., c_m))$는 항상 $\varnothing$을 반환한다. 동일하지 않은 동등 분할 클래스가 존재함에도 불구하고 $t \cup (c_1, c_2, ..., c_m)$이 수행되기 위해서는 서로 속해 있는 애플리케이션 상태가 겹치지 않아야 한다. 그러나 $size(t) = |S_u \cup v|$이기 때문에, 애플리케이션 상태가 겹치지 않는 상황은 발생할 수 없다. 따라서 위 정의를 통해 테스트 케이스가 영향 관계를 충족하는지 확인할 수 있다.

4.4 영향 관계를 이용한 테스트 케이스 생성 방법

이번 절에서는 앞서 정의된 영향 관계를 기반으로 테스트 케이스를 생성하는 방법을 제안한다. 나아가 예시를 통하여 이벤트 핸들러로부터 영향 관계를 도출하고 영향 관계 커버리지를 만족하는 테스트 케이스를 만드는 방법을 설명한다.

"c) 영향 관계 커버리지 충족 요건"에서 설명한 것처럼 대부분의 영향 관계는 테스트 케이스보다 길이가 짧기 때문에, 하나의 테스트 케이스로 여러 개의 영향 관계를 충족시킬 수 있다. 이는 곧 여러 개의 영향 관계를 병합함으로써 하나의 테스트 케이스를 구성할 수 있다는 의미가 된다. 따라서 앞서 소개한 $M(R_x, R_y)$ 연산을 통하여 영향 관계로부터 테스트 케이스를 생성하는 방법에 대해서 제안한다. 생성 방법은 Fig. 7의 알고리즘을 통해 설명한다.

Algorithm – generate a test case:
In : effect relation R
Out : test case T

```
1  function genTestCase_EffectRelation(R) {
2      T=∅; R_new=R; R_m=∅;
3      while R_new ≠ ∅ do {
4          R_m = R_new; R_new = ∅;
5          for each r_i∈R_m do {
6              for each r_j∈R_m do {
7                  if (r_i ≠ r_j) {
8                      r_k = M(r_i, r_j);
9                      if (r_k ≠ ∅) {
10                         R_new = r_k;
11                     }
12                 }
13             }
14         }
15     }
16     n = |S_u ∪ v|;
17     for each r∈R_m do {
18         if (getSize(r) ≠ n) {
19             fill_DontCare(r);
20         }
21         T=⋃r;
22     }
23     return T;
24 }
```

Fig. 7. Algorithm for Test Case Generation using Effect Relation

3~15번째 줄은 영향 관계 R의 모든 원소에 대해서 병합 연산을 수행하는 과정이다. 더 이상 합쳐질 수 없을 때까지 반복하여 수행한다. 19번째 줄의 fill_DontCare()는 채워지지 않은 애플리케이션 상태 S_u나 파라미터 v의 동등 분할 클래스가 존재할 경우, 이에 대한 Don't Care를 채우는 작업을 수행한다. Fig. 7의 알고리즘을 통해 이벤트 핸들러의 모든 애플리케이션 상태 S_u와 파라미터 v에 대해서 구성된 영향 관계는 그 자체가 테스트 케이스로 활용될 수 있으며, 이를 통해 영향 관계 커버리지를 만족하는 테스트 케이스를 생성할 수 있다.

명확한 이해를 위해 제시된 Fig. 7의 알고리즘으로부터 테스트 케이스를 생성하는 과정을 예를 들어 설명한다. Fig. 3의 이벤트 핸들러 h_1는 5절의 Case Study에서 사용하는 위치 기반 친구찾기 애플리케이션에 존재하는 이벤트 핸들러로써, 이 이벤트 핸들러를 바탕으로 진행한다. 먼저 테스트 케이스를 생성하기에 앞서 이벤트 핸들러 h_1이 갖는 애플리케이션 상태를 분석한다. Table 2는 이벤트 핸들러 h_1이 갖는 애플리케이션 상태를 정리한 것이다.

Table 2. Application State of Event Handler h_1

ID	Name	$EquiClass$
S_1	TextField_Name.text	c_1, c_2, c_3
S_2	TextField_PhoneNumber.text	c_4, c_5, c_6
S_3	MyInfo.Name	-
S_4	MyInfo.PhoneNumber	-

Table 2의 애플리케이션 상태는 각각 $S_u = \{S_1, S_2\}$와 $S_m = \{S_3, S_4\}$가 된다. 또한 각각의 동등 분할 클래스는 정규표현식에 맞는 값, 빈 문자열, 정규표현식과 맞지 않는 값으로 나뉜다. 애플리케이션 상태에 대해서 정리를 한 후에는 영향 관계를 도출한다. 이벤트 핸들러 h_1의 기본 블록은 Fig. 8과 같으며, 이로부터 영향 관계를 도출한다.

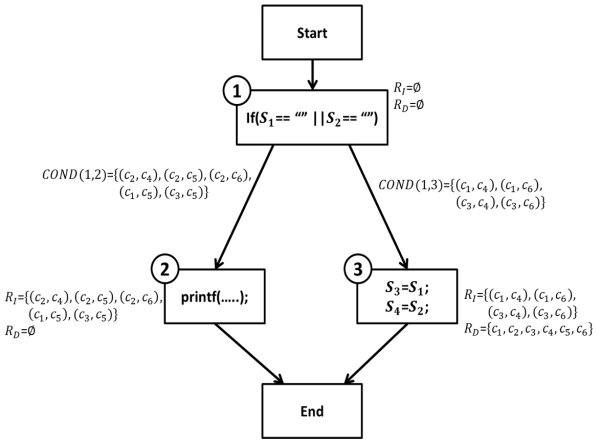


Fig. 8. Basic Block of Event Handler h_1

1번 기본 블록은 Predecessor로부터 전달된 $COND$와 애플리케이션 상태가 변경하는 부분이 존재하지 않으므로, 관련된 영향 관계가 존재하지 않는다. 2번과 3번 기본 블록은 1번 블록부터 전달된 $COND$를 통해 자신의 R_I를 도출한다. 대신 2번 기본 블록은 내부에서 애플리케이션 상태를 변경하는 부분이 존재하지 않으므로 R_D는 공집합이 된다. 3번 기본 블록은 애플리케이션 상태 S_3와 S_4가 각각 애플리케이션 상태 S_1과 S_2에 의해 변경되므로 이를 R_D에 반영한다. 기본 블록으로부터 도출된 R_I와 R_D를 기반으로 영향 관계 R을 도출한다. 병합 연산인 $merge(R_x, R_y)$를 통해 도출된 영향 관계 R은 다음과 같다.

$$R = \bigcup_{b \in BB} merge(R_I^b, R_D^b)$$
$$= merge(R_I^1, R_D^1) \cup merge(R_I^2, R_D^2) \cup merge(R_I^3, R_D^3)$$
$$= \varnothing \cup \{(c_2, c_4)\} \cup \{(c_1, c_4), (c_1, c_6), (c_3, c_4), (c_3, c_6)\}$$
$$= \{(c_1, c_4), (c_1, c_6), (c_2, c_4), (c_3, c_4), (c_3, c_6)\}$$

도출된 영향 관계 R은 총 다섯 개이며, 각각은 애플리케이션 상태 S_u에 포함되는 모든 변수를 다루고 있다. 따라서 테스트 케이스 생성 알고리즘 내에서 영향 관계 R을 병합하는 과정을 수행하여도 동일한 집합을 얻게 된다. 또한 Don't Care에 해당하는 부분이 존재하지 않기 때문에, 위에서 도출된 영향 관계 R은 그 자체로 테스트 케이스로 활용할 수 있다. 따라서 이벤트 핸들러 h_1에 대한 테스트 케이스 T는 다음과 같다.

$$T = \{\langle c_1, c_4 \rangle, \langle c_1, c_6 \rangle, \langle c_2, c_5 \rangle, \langle c_3, c_4 \rangle, \langle c_3, c_6 \rangle\}$$

5. Case Study

본 논문에서는 제안한 테스트 케이스 생성 알고리즘을 이용하여 위치기반 친구 찾기(Location based Friend Finder) 애플리케이션에 대한 테스트 케이스를 생성하고 테스트를 수행하였다. 위치기반 친구 찾기 애플리케이션은 친구와 위치정보의 공유 기능을 제공한다. 자신의 위치정보를 서버에 등록하고, 서버로부터 자신이 원하는 친구의 정보를 주기적으로 수신할 수 있다.

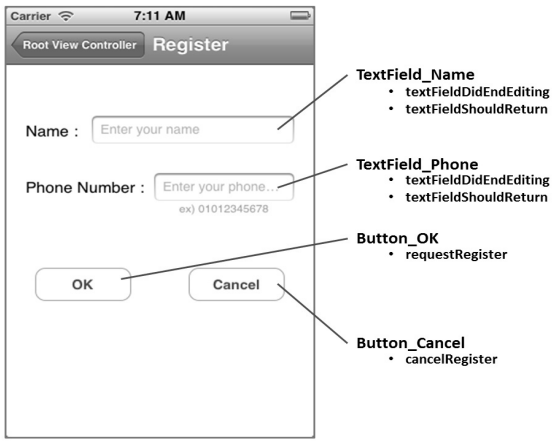


Fig. 9. RegisterView on "Location based Friend Finder"

애플리케이션은 총 여덟 개의 뷰로 구성된다. 분량 관계상, 그 중에서 회원 등록과 관련된 뷰에 대해서 생성된 테스트 케이스에 대해서만 포함한다. Fig. 9는 회원 등록 뷰의 구조를 보여준다.

다음으로 Table 3은 테스트 케이스 생성 알고리즘으로부터 생성된 회원 등록 뷰의 테스트 케이스 t_1, t_2를 정리한 것이다. Table 3에 포함된 이벤트 핸들러 중에서 requestRegister가 4.4절에서 예시로 든 이벤트 핸들러이다.

Table 3. Test Cases for RegisterView

Event Handler	S_u		$n(t_1)$	$n(t_2)$
	Name	$n(EquiClass)$		
request Register	name	3	9	5
	phone	3		
cancel Register	-	-	1	1
textfieldDid EndEditing	name	3	9	5
	phone	3		
textfield ShouldReturn	-	-	1	1

Table 3에서 $n(EquiClass)$는 해당 애플리케이션 상태에 대한 동등 분할 클래스의 개수를 의미한다. name과 phone의 자료형은 문자열인데, 각각은 빈 문자열, 주어진 정규 표현식에 맞는 문자열, 주어진 정규 표현식에 맞지 않는 문자열로, 총 세 가지의 동등 분할 클래스를 갖는다. $n(t_1)$은 동등 분할 클래스의 모든 조합을 고려하여 생성된 테스트 케이스의 개수를 의미하며, $n(t_2)$는 Fig. 7의 알고리즘, 즉 영향관계를 통해 생성된 테스트 케이스의 개수를 의미한다. 회원 등록 뷰를 대상으로 생성된 이벤트 핸들러의 테스트 케이스는 알고리즘에 따라 각각 20개와 12개이다. 이를 통해서 모든 조합을 고려하는 것에 비해 영향 관계를 이용한 경우가 더 적은 수의 테스트 케이스를 생성함을 알 수 있다.

Table 4. Test Cases for "Location-based Friend Finder"

View	t_1		t_2	
	$n(t_1)$	$n(t_1^{FD})$	$n(t_2)$	$n(t_2^{FD})$
Main View	2	2	2	2
Register View	20	5	12	3
Map View	27	3	19	3
Configuration View	69	10	39	8
Friend List View	39	7	21	5
Checked Friend List View	33	3	14	2
Location View	1	0	1	0
Developer View	2	0	2	0

Table 4는 테스트 케이스 생성 알고리즘을 통해 생성된 모든 테스트 케이스와 테스트 케이스의 수행 결과를 정리한 것이다. 여기서 $n(t_x)$와 $n(t_x^{FD})$는 각각 생성된 테스트 케이스의 개수와 결함을 발견한 테스트 케이스의 개수를 의미한다. 위 결과를 보면 영향 관계를 통해 생성된 테스트 케이스의 수가 확연히 적음을 알 수 있다. 각 방법의 $n(t_x^{FD})$를 비교하면 그 수가 차이가 나는 것을 볼 수 있다. 이는 발견하는 결함의 수가 줄어드는 것이 아니라 t_1에서 모든 조합을 고려하다 보니 동일한 결함을 발견한 테스트 케이스들이 존재하여 발생한 현상이다. 즉, 모든 조합을 고려하는 것에 비해 적은 수의 테스트 케이스를 생성하는 반면에 대부분의 결함을 찾아내는 것을 볼 수 있다.

6. 결 론

본 논문에서는 iOS 애플리케이션의 GUI를 테스트하기 위한 테스트 케이스를 생성하는 방법에 대해 제안하였다. 기존 연구들은 일련의 이벤트를 테스트 케이스로써 사용하였다면, 본 논문에서는 실제 기능을 수행하는 이벤트 핸들러에 대한 단위 테스트를 위한 테스트 케이스를 생성하는 방법으로써, 영향 관계를 통하여 테스트 케이스 생성 방법을 제안하였다. 제안된 방법은 생성되는 테스트 케이스의 수를 줄이면서도, 의미 있는 테스트 케이스를 생성할 수 있는 방법이다. 본 논문에서 제안하는 방법을 통해 생성된 테스트 케이스는 애플리케이션이 가지고 있는 이벤트 핸들러의 결함을 검출하는데 사용될 수 있다. 이를 통해 이벤트 핸들러의 품질 향상을 기대할 수 있을 뿐만 아니라 결국 GUI의 품질이 향상될 것이다.

그러나 본 논문에서 진행된 테스트는 이벤트 핸들러에 대한 단위 테스트만을 수행하였기 때문에, 기존 연구에 비해서 GUI 간의 연동에 대한 테스트가 미비하다 할 수 있다. 따라서 향후 연구로는 이벤트 핸들러에 초점을 맞추어 GUI 간의 연동 테스트를 위한 테스트 시나리오를 생성하고자 한다.

참 고 문 헌

[1] iOS [Internet], http://en.wikipedia.org/wiki/IOS

[2] Ben Shneiderman, “Direct manipulation: A step beyond programming languages,” *IEEE Computer*, Vol.16, No.8, pp.57-69, 1983.

[3] Mercury Interactive WinRunner, http://www.mercury interactive.com/products/winrunner, 2003.

[4] Bitbar, “Testdroid Recorder,” http://testdroid.com/product/ testdroid-recorder, 2012.

[5] Jung Gyw Lee, Seung Hak Kuk, and Hyeon Soo Kim, “Test Cases Generation Method for GUI Testing with Automatic Scenario Generation,” *Journal of KIISE: Software and Applications*, Vol.36, No.1, pp.45-53. 2009. (in Korean)

[6] Q. Xie and A.M. Memon, “Using a Pilot Study to Derive a GUI Model for Automated Testing,” *ACM Trans. Software Eng. And Methodology*, Vol.18, pp.1-35, 2008.

[7] A.M. Memon and Q. Xie, “Studying the Fault-Detection Effectiveness of GUI Test Cases for Rapidly Evolving Software,” *IEEE Trans. Software Eng.*, Vol.31, No.10, pp.884-896, 2005.

[8] L. White and H. Almezen, “Generating Test Cases for GUI Responsibilities Using Complete Interaction Sequences,” *Proc. 11th Int'l Symp. Software Reliability Eng.*, pp.110, 2000.

[9] T. Takala, M. Katara and J. Harty, “Experiences of System-Level Model-Based GUI Testing of an Android Application,” *4th IEEE Int'l Conference on Software Testing*, Verification and Validation, pp.377-386, 2011.

[10] C. Hu and I. Neamtiu, “Automating GUI Testing for Android Application,” *6th Int'l Workshop on Automation of Software Test*, pp.77-83, 2011.

[11] Lei Zhao and Kai-Yuan Cai, “Event Handler-Based Coverage for GUI Testing,” in *Proc. of 10th Int'l Conference on Quality Software*, pp.326-331, 2010.

[12] X.Yuan, M.B. Cohen and A.M. Memon, “GUI Interaction Testing: Incorporating Event Context,” *IEEE Trans. On Software Eng.*, Vol.37, No.4, pp.559-574, 2011.

[13] S. Alpaev, “Applied MVC Patterns. A Pattern Language,” presented at CoRR, 2006.

[14] D. Plakalovic and Simic D., “Applying MVC and PAC patterns in mobile applications,” *Journal of Computing*, Vol.2, No.1, pp.65-72, 2010.

[15] Apple, “Concepts in Objective-C Programming,” 2012.

[16] Apple, “iOS App Programming Guide,” 2012.

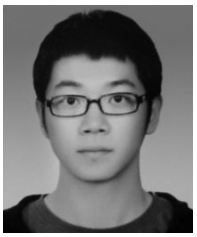
서 용 진

e-mail : yjseo082@cnu.ac.kr

2011년 충남대학교 컴퓨터공학과(학사)

2010년~현 재 충남대학교 컴퓨터공학과 석사과정, 석박사통합

관심분야 : 소프트웨어 테스팅, 스마트폰, UX/UI



문 대 건

e-mail : ppy333@cnu.ac.kr

2011년 충남대학교 컴퓨터공학과(학사)

2011년~현 재 충남대학교 컴퓨터공학과 석사과정

관심분야 : 소프트웨어 테스팅, 소프트웨어 품질관리, 스마트폰

김 현 수

e-mail : hskim401@cnu.ac.kr

1988년 서울대학교 계산통계학과(학사)

1991년 한국과학기술원 전산학과 (공학석사)

1995년 한국과학기술원 전산학과 (공학박사)

1995년~1995년 한국전자통신연구원 Post Doc.

1996년~2001년 금오공과대학교 조교수

1999년~2000년 Colorado State University 방문교수

2007년~2008년 Purdue University 방문연구교수

2001년~현 재 충남대학교 컴퓨터공학과 교수

관심분야 : 소프트웨어 공학, 소프트웨어 테스팅, SOA