http://dx.doi.org/10.3745/KTSDE.2013.2.10.679

A Test Case Generation Method Based on Activity for Android Application Testing

Minhyuk Ko[†] · Yongjin Seo[††] · Sangpil Yun[†††] · Hyeon Soo Kim[††††]

ABSTRACT

Smartphones have features that users feel free to install/delete the program they want. Their emergence makes many developers rush into the Smartphone application development market. Thus, developing good applications quickly is becoming even more intense competition in the market. Because, however, the application development and deployment procedures are simple in the Android environments and anyone can participate in the development easily, applications not validated thoroughly are likely to be deployed. Therefore, a systematic approach that can verify Android-based applications with fewer burdens is required. In this paper, we propose a method that generates automatically GUI-based testing scenarios for the Android applications. The automated test scenario generation can reduce the time which the developer spends on testing, thus it can improve the productivity of the development in the testing phase.

Keywords : Android Application, GUI Testing, Test Case Generation

안드로이드 애플리케이션을 테스트하기 위한 액티비티 기반의 테스트 케이스 생성 방법

고 민 혁[†] · 서 용 진[††] · 윤 상 필[†††] · 김 현 수[††††]

요 약

사용자가 원하는 프로그램을 자유롭게 설치/삭제 할 수 있는 특징을 가진 스마트폰의 등장으로 인해, 수많은 개발자들이 스마트폰 애플리케이션 개발 시장에 뛰어들면서 좋은 애플리케이션을 빨리 개발하려는 경쟁이 더욱 치열해지고 있다. 그러나 안드로이드 환경은 애플리케이션 개발 및 배포 절차가 간단하여 누구나 쉽게 개발에 참여할 수 있어서 충분히 검증되지 않은 애플리케이션들이 배포될 가능성이 높다. 따라서 적은 부담으로 안드로이드 기반의 애플리케이션을 검증할 수 있는 체계적인 방법이 필요하다. 이에 본 논문에서는 안드로이드 애플리케이션을 위한 GUI 기반의 테스트 시나리오 자동 생성 방법을 제시한다. 자동화된 테스트 시나리오 생성을 통해 테스트에 소요되는 시간을 줄임으로써 테스트 단계에서의 생산성을 향상시킬 수 있다.

키워드 : 안드로이드 애플리케이션, GUI 테스팅, 테스트 케이스 생성

1. 서 론

스마트폰(Smartphone)은 PC와 유사한 성능을 가지며 범용 운영체제를 내장한 휴대폰이다. 기존의 모바일 플랫폼과 달리 스마트폰에서는 사용자가 원하는 프로그램을 자유롭게 설치하고 삭제할 수 있다. 이러한 특성은 스마트폰 애플리케이션에 대한 관심을 증가시키는 요인이 된다. 이를 지원하는 스마트폰 운영체제로는 iOS, 안드로이드, 윈도우즈 폰, 블랙베리 OS 등이 존재한다. 특히, 그 중에서 안드로이드는 가장 많은 장치와 사용자를 확보하고 있기 때문에[1] 가장 많은 관심을 받고 있다.

안드로이드 기반의 애플리케이션을 개발하기 위해서는 개발자 등록을 수행하여야 하며 구글에서 제시하는 배포 절차를 따라야 한다. 그러나 개발자 등록에 대한 제약 조건이 없어서 누구나 쉽게 개발자로 등록할 수 있으며, 배포 절차 또한 간단하다. 이는 충분히 검증이 되지 않은 애플리케이션들이 사용자에게 배포될 수 있다는 것을 의미한다[2][3]. 따라서 애플리케이션에 대한 신뢰성을 보장하기 위해서 개발자가 따로 테스트를 수행하여야 한다. 그러나 안드로이드 기반의 애플리케이션은 짧은 개발 주기를 갖기 때문에[4] 애플리케이션 테스트에 할애할 수 있는 시간이 충분하지 않다. 이러한 이유로 안드로이드 기반 애플리케이션들은 사용자의 버그 리포트에 의존하고 있다. 제품을 출시한 후, 사용자가 애플리케이션을 사용하는 과정에서 발생된 오류에 대

※ 이 논문은 2010년도 정부(교육과학기술부)의 재원으로 한국연구재단의 기초연구사업 지원을 받아 수행된 것임(No.2010-0025329).
† 정 회 원 : 충남대학교 컴퓨터공학과 석사
†† 준 회 원 : 충남대학교 컴퓨터공학과 석·박사통합
††† 준 회 원 : 충남대학교 컴퓨터공학과 석사과정
†††† 종신회원 : 충남대학교 컴퓨터공학과 교수
논문접수 : 2013년 5월 14일
수 정 일 : 1차 2013년 7월 11일
심사완료 : 2013년 7월 18일
* Corresponding Author : Hyeon Soo Kim(hskim401@cnu.ac.kr)

해서 보고 받는 것이다. 하지만 이와 같은 방법은 애플리케이션에 대한 신뢰를 떨어뜨릴 뿐만 아니라 유지보수에 많은 시간을 소모하도록 만든다. 이러한 문제를 해결하기 위해서는 자동화된 테스트 방법이 요구된다. 테스트 자동화를 통해 개발자에게 주어지는 테스트 케이스 생성과 같은 부담과 테스트 단계에서 반복 작업의 시간을 줄여줌으로써 테스트 단계의 생산성을 향상시킬 수 있다.

안드로이드 기반의 애플리케이션은 터치스크린을 통해 동작하는 GUI(Graphical User Interface) 기반 응용 프로그램이다. 따라서 사용자는 GUI를 통해 안드로이드 기반의 애플리케이션을 조작하여야 한다. 이는 안드로이드 기반의 애플리케이션에게 있어 GUI가 차지하는 중요성을 보여준다[5]. GUI 중심의 애플리케이션이 갖는 기능은 결국 GUI와 밀접한 관련이 있기 때문이다. 이는 곧 GUI에 대한 검증을 통해 안드로이드 기반의 애플리케이션의 신뢰성을 향상시킬 수 있음을 말한다. 안드로이드 기반의 애플리케이션에서 GUI의 구현은 액티비티(Activity)를 통해 이루어진다. 액티비티는 안드로이드 애플리케이션의 하나의 화면(View)을 담당하는 핵심 컴포넌트로, GUI를 형성하고 사용자로부터의 이벤트를 수신하고 처리하는 기능을 수행한다. 정리하면, 안드로이드 기반의 애플리케이션을 검증하기 위해서는 GUI의 결함 여부를 확인하여야 하며, GUI의 결함 여부는 액티비티의 결함 파악을 통해 확인할 수 있다.

이를 위해 본 논문에서는 액티비티의 특성과 이로부터 발생할 수 있는 오류들을 파악한다. 파악된 내용을 바탕으로 이를 검출하기 위한 테스트 케이스 및 테스트 시나리오 생성 방법과 이에 대한 자동화 방안을 제시한다. 제시된 방법을 이용한다면 테스트에 대한 전문지식이 없는 개발자도 쉽게 테스트를 수행할 수 있으며, 테스트 수행에 소요되는 시간 역시 줄임으로써 안드로이드 기반의 애플리케이션이 갖는 짧은 개발 주기를 충족시킬 수 있다.

2. 관련 연구

안드로이드 애플리케이션은 다수의 액티비티와 그 사이의 관계들로 구성되는 복잡한 구조를 갖고 있다. 따라서 애플리케이션을 구동시키기 위한 다양한 경로가 존재할 수 있으며, 모든 경로에 대해 테스트를 수행하는데 많은 시간과 노력이 요구된다. 이 문제들은 기존의 GUI 테스트에서 갖고 있던 문제들로, 이를 해결하기 위해 다양한 방법들이 연구되었다.

먼저, Record-Playback 기법이 존재한다. 이 방법은 테스터가 발생시킨 일련의 GUI 이벤트를 발생 시간 순서대로 기록한 뒤, 기록된 이벤트들을 재생하여 테스트를 수행하는 방법이다. 이 방법은 테스트 자체는 자동으로 수행되지만, 테스터가 직접 이벤트를 발생시켜 테스트 시나리오를 생성해야 하기 때문에 테스터에게 테스트 시나리오 생성에 대한 부담을 줄여주지 못한다. 이를 해결하기 위해 [6]은 GUI 이벤트를 그룹화하는 방법을 제시한다. [6]에서는 유사한 기능을 수행하는 이벤트를 하나의 그룹으로 지정하고 테스터가 작성한 최소한의 테스트 시나리오를 통해서 더 많은 개수의 테스트 시나리오를 도출해낼 수 있다. 그러나 이 방법 역시 테스트 시나리오 생성에 대한 테스터의 부담을 완전히 해소하지 못한다.

다음으로 모델 기반 테스트 기법이 존재한다. 이 방법은 애플리케이션의 동작을 모델로 작성하고, 작성된 모델을 테스트에 활용하는 방법이다. [7]에서는 모델 기반 테스트 기법을 적용하여 안드로이드 애플리케이션을 테스트한다. [7]에서는 작성한 모델로부터 테스트 케이스를 도출하기 때문에 모델을 관리함으로써 테스트 케이스의 관리와 재사용이 가능하다는 장점이 있다. 여기서 사용되는 모델은 애플리케이션의 명세로부터 작성된다. 그러나 모바일 애플리케이션 개발자들은 명세에 대한 고려가 거의 없기 때문에[8] 안드로이드 애플리케이션에 적용하기 어렵다.

반면, [9]에서는 GUI 크롤링 기법을 이용하여 애플리케이션의 GUI 구조를 분석하고 이를 통해 이벤트 트리를 구성한다. 이벤트 트리를 탐색함으로써 도달 가능한 일련의 이벤트, 즉 테스트 시나리오를 생성할 수 있다. 이 방법은 명세가 아닌 실제 구현물을 통해서 모델을 작성한다. 따라서 테스터가 모델을 생성하기 위한 노력과 시간이 필요하지 않다.

마지막으로 [10]는 액티비티에 무작위 이벤트를 발생시켜 테스트를 수행하는 방법을 사용한다. 이 연구에서는 구글에서 제공하는 테스트 도구인 Monkey Runner를 사용하여 애플리케이션을 구성하는 모든 액티비티에 무작위로 이벤트를 발생시킨다. 무작위로 수행되는 만큼 테스터가 수행하여야 하는 작업은 수행 결과를 확인하는 것만 존재한다. 그러나 테스터의 부담을 줄여줄지라도 특정 목적을 가지고 수행되는 테스트가 아니기 때문에 의미 있는 테스트 케이스를 만들어 내지 못한다는 단점이 있다.

본 논문에서는 기존의 연구들이 가진 문제점을 해결하기 위해서 두 가지의 원칙을 바탕으로 테스트를 수행한다. 첫째, 테스트는 명확한 목적을 기반으로 수행되어야 한다. 따라서 테스트 대상과 그로부터 검출한 오류에 대해서 명확히 정의하여야 한다. 본 논문에서 제안하는 테스트 방법은 안드로이드 애플리케이션을 구성하는 액티비티를 대상으로 하며, 검출하고자 하는 오류는 생명주기 오류, GUI 이벤트 처리 오류, 인텐트(Intent) 오류이다. 오류에 대한 자세한 설명은 4.1절에 기술한다. 둘째, 테스터의 개입을 최소화한다. 앞서 살펴본 대부분의 연구들은 테스트의 개입 없이는 테스트가 불가능하다. [9]의 경우에는 애플리케이션의 구현물로부터 모델을 자동으로 생성하는 방법을 사용함으로써 테스터의 개입을 최소화할 뿐만 아니라 테스트에 치중할 수 있는 환경을 제공해준다. 따라서 본 논문에서도 [9]의 접근법을 참고하여 구현물로부터 테스트 케이스를 자동 생성하는 방법을 제시한다. 구현된 애플리케이션의 소스코드와 GUI 레이아웃을 바탕으로 테스트 케이스를 생성한다.

3. 안드로이드 애플리케이션

안드로이드 애플리케이션은 모바일 환경에서 동작하도록 구현되었기 때문에 PC에서 동작하는 일반적인 소프트웨어

와는 동작 방식이 다르다. 이 절에서는 테스트 대상인 액티비티를 중심으로 안드로이드 애플리케이션의 동작 방식 및 구성요소에 대하여 분석한다.

3.1 안드로이드 애플리케이션 동작 메커니즘

모바일 장치는 PC와 달리 제한된 디스플레이 크기를 갖는다. 이러한 이유로 인해 안드로이드 애플리케이션은 하나의 화면에 모든 기능을 구현하기 어렵다. 따라서 대부분의 안드로이드 애플리케이션은 여러 개의 화면으로 분리되어 구현된다. Fig. 1은 일곱 개의 액티비티로 구성된 안드로이드 애플리케이션의 구조를 나타낸다. Fig. 1을 통해 알 수 있듯이 안드로이드 애플리케이션은 하나 이상의 액티비티와 그 사이의 관계를 정의함으로써 구성된다.

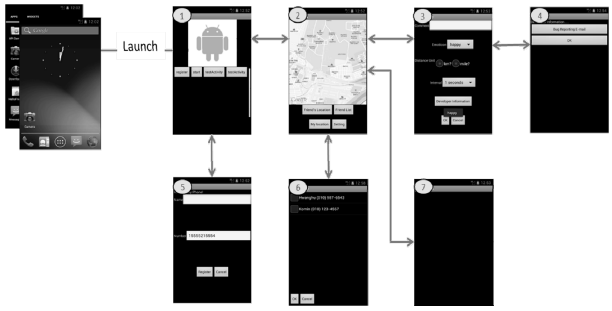

Fig. 1. Structure of Android Application

3.2 액티비티

액티비티는 GUI를 담당하는 애플리케이션 구성요소로써, 화면을 구성하고 사용자 이벤트를 처리한다. 액티비티는 두 가지의 GUI 구현 방법을 제공한다. 하나는 소스코드 내에 하드코딩(Hard Coding)을 통해 GUI를 구현하는 것이다. 그러나 GUI는 변경이 잦은 요소이기 때문에, 이와 같은 방법은 권장하지 않는다. 다른 하나는 GUI와 애플리케이션 로직을 분리하여 개발하는 방법으로, XML 레이아웃이라고 하는 XML 형식의 구성 방법을 사용하는 것이다.

액티비티 사이의 관계는 인텐트를 통해 구현된다. 액티비티는 인텐트를 사용하여 다른 액티비티를 활성화시킬 수 있다. 다른 액티비티를 활성화시킨 액티비티는 화면에 보이지 않게 된다. 이때, 화면에 보이지 않을 뿐 액티비티 자체가 삭제되지는 않는다. 이는 각 액티비티는 고유의 생명주기를 기반으로 동작하기 때문이며, 화면에서 보이지 않는 액티비티는 생명주기에 의해 정지 상태에 머물게 된다. 액티비티의 생명주기 상태는 사용자 이벤트나 시스템 내부의 요인에 의해 전환된다.

3.3 인텐트

인텐트는 액티비티 사이에 주고받는 하나의 메시지로, 주로 다른 액티비티를 활성화시키기 위하여 사용된다. 인텐트는 Action, Category, Data라는 세 가지 필드로 구성되어 있으며, 이를 통해서 다양한 작업을 수행할 수 있다. Action 필드와 Category 필드는 인텐트를 수신하는 액티비티가 수행하여야 하는 동작을 명시하기 위해 사용된다. 이를 통해서 액티비티에게 특정 작업을 위임할 수 있다. 예를 들어 사진을 출력해야 하는 경우, 사진 출력 기능을 직접 구현하는 대신 다른 액티비티에게 사진을 출력하도록 요청할 수 있다. 간혹 작업을 위임하기 위해서 추가 정보가 필요할 수 있는데 이를 위한 필드가 Data 필드이다. Data 필드에는 인텐트를 수신하는 액티비티에게 전달하고자 하는 값을 저장할 수 있다. Data 필드는 원시 자료형(Primitive data type) 뿐만 아니라 클래스 형태의 사용자 지정 데이터도 담을 수 있으며, 저장되는 값은 키-값 쌍이다.

4. 안드로이드 애플리케이션을 위한 테스트 케이스 정의

4.1 액티비티 오류

구글에서는 액티비티를 검증할 때 확인하여야 하는 요소로 다음 항목을 제시하고 있다[11].

- GUI 이벤트
- 생명주기 이벤트
- 인텐트
- 런타임 환경 변화

본 논문에서는 위 요소로부터 액티비티 오류를 세 가지로 정의한다. 각각은 생명주기 오류, GUI 이벤트 처리 오류, 인텐트 오류이다. 이 때, 런타임 환경 변화는 생명주기를 통해 확인할 수 있기 때문에 생명주기 오류에 포함된다.

생명주기 오류는 액티비티의 생명주기가 전환되는 과정에서 호출되는 생명주기 메서드에서 발생할 수 있는 오류로써, 생명주기 메서드 내에서 애플리케이션의 주요 데이터가 손실되는 것을 의미한다. 액티비티는 생명주기 상태를 가지며, 각 상태는 시작, 동작, 일시정지, 정지, 종료로 나눌 수 있다. 액티비티는 변경될 생명주기 상태에 맞는 작업을 수행하기 위해 생명주기 메서드를 호출한다. 생명주기 메서드에서는 주로 애플리케이션에서 다루고 있는 데이터를 저장하거나 복원하는 작업을 수행한다. 실질적인 구현은 애플리케이션마다 다르기 때문에, 생명주기 메서드는 개발자가 구현해야 한다. 생명주기 메서드는 선택적으로 구현할 수 있기 때문에 만일 구현할 필요가 없다면 구현하지 않아도 된다. 따라서 개발자가 생명주기 메서드를 잘못 구현하거나 구현하지 않음으로 인해 저장 및 복원하여야 할 애플리케이션의 데이터가 손실될 수 있다. 따라서 발생할 수 있는 생명주기 변화를 모두 확인하여 오류를 내포하고 있는지 확인할 필요가 있다.

GUI 이벤트 처리 오류는 발생된 이벤트의 처리 결과가 예상된 결과와 다른 것을 말한다. 애플리케이션에서 발생한 이벤트를 처리하기 위해서 이벤트 핸들러를 구현하여야 한다. 안드로이드 애플리케이션은 자바 기반으로 구현되므로 이벤트 핸들러를 구현하기 위해서 이벤트가 발생할 수 있는 GUI 컴포넌트에 이벤트 리스너를 등록하고 구현하여야 한

다. 만일 이벤트 리스너를 구현하는 과정에서 개발자가 실수를 한다면, 애플리케이션은 의도하지 않은 동작을 할 수 있다. 따라서 다양한 이벤트를 발생시켜 이벤트 리스너에 잠재된 오류가 존재하는지 확인하여야 한다.

인텐트 오류는 인텐트의 Data 필드에 담긴 예상치 못한 값으로 인해 애플리케이션이 오작동하거나 비정상 종료를 하는 경우를 의미한다. 인텐트를 구성하는 세 가지 필드 중에서 Action 필드와 Category 필드는 시스템에서 의해서 처리되기 때문에 오류가 발생할 수 있는 부분은 Data 필드에 한정된다. Data 필드에 담길 수 있는 값은 액티비티마다 다를 뿐만 아니라 런타임 시점에 값이 결정되기 때문에 예상치 못한 값이 담겨있더라도 오작동 하지 않도록 예외처리 되어야 한다. 따라서 인텐트 오류가 발생한다는 것은 예외처리를 하지 않았다는 의미가 된다. 즉, 인텐트 오류를 확인하기 위해서는 Data 필드의 자료형에서 맞는 다양한 값에 대해서 애플리케이션이 문제없이 동작할 수 있음을 검증하여야 한다.

4.2 테스트 케이스 정의

이번 절에서는 4.1절에서 언급한 오류를 검출할 수 있는 테스트 케이스를 정의한다. 각 오류는 검출하는 방법이 다르기 때문에 각각 다른 테스트 케이스를 정의한다.

1) 생명주기 테스트 케이스

액티비티의 생명주기 상태는 다섯 가지로 나눌 수 있다. 생명주기 전환은 한 생명주기 상태에서 다른 생명주기 상태로 바뀌는 것을 의미하며, 이를 상태 다이어그램으로 표현하면 Fig. 2와 같다.

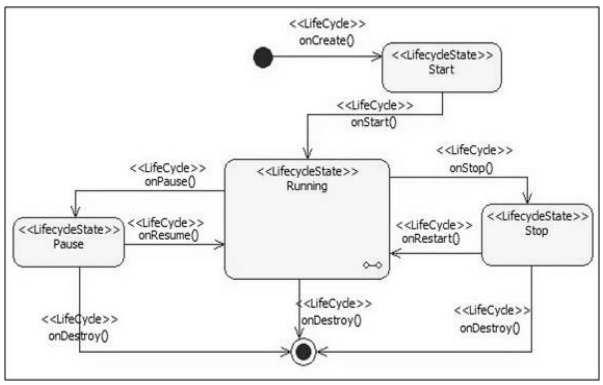

Fig. 2. State Diagram for Activity

- **시작(Start)** : 액티비티가 생성된 상태로, 사용자에게 보이기 이전의 상태이다.
- **동작(Running)** : 사용자가 볼 수 있는 상태로, 이 상태에 도달하여야 사용자 이벤트를 수신할 수 있다.
- **일시정지(Pause)** : 모달 액티비티나 알림 메시지가 실행되어 화면을 가리고 있는 상태
- **정지(Stop)** : A 액티비티에서 B 액티비티로의 전환이 발생하였을 때, A 액티비티가 도달하는 상태이다. 정지 상태가 되면 사용자에게 보이지 않게 되며, 메모리의 부족으로 종료될 수 있다.
- **종료(Termination)** : 액티비티가 제거된 상태이다.

액티비티의 생명주기는 Fig. 2의 상태 다이어그램을 기반으로 전환된다. 따라서 Fig. 2의 상태 다이어그램에서 발생 가능한 흐름을 통해서 액티비티에서 발생될 수 있는 모든 생명주기 전환을 확인할 수 있다. 또한 그 흐름은 생명주기 테스트 케이스로 사용될 수 있다. Table 1은 Fig. 2에서 발생 가능한 흐름, 즉 생명주기 테스트 케이스를 정리한 것이다[12].

Table 1. Lifecycle Test Case

ID	Lifecycle Scenario
LT1	Create → Start → Running → Termination
LT2	Create → Start → Running → Pause → Running → Termination
LT3	Create → Start → Running → Pause → Termination
LT4	Create → Start → Running → Stop → Running → Termination
LT5	Create → Start → Running → Stop → Termination

- **LT1** : 통상적으로 실행되어 종료되는 상태 전환 시나리오이다.
- **LT2** : 모달 액티비티가 실행되거나 알림 메시지가 출력되는 경우이다. 이 경우에 액티비티는 일시정지 상태에 도달하며, 모달 액티비티의 작업이 종료되거나 알림 메시지를 확인하면 동작 상태로 돌아온다.
- **LT3** : 이 시나리오는 모달 액티비티 호출 후에 사용자의 홈 버튼 클릭 이벤트 발생으로 강제로 종료되는 경우이다.
- **LT4** : 다른 액티비티로 전환이 발생한 경우이다. 이 경우에 액티비티는 정지 상태에 도달한다. 만일 사용자가 Back 버튼을 누르면 다시 동작 상태로 돌아온다.
- **LT5** : 다른 액티비티로 전환이 발생한 경우이다. LT4와의 차이점은 정지 상태에서 어떤 이유에 의해(예, 메모리 부족) 강제 종료되는 상황이다.

모든 액티비티는 내부 구성이 다르더라도 각각의 액티비티가 갖는 생명주기는 동일하다. 따라서 Table 1의 생명주기 테스트 케이스는 모든 액티비티에 적용될 수 있다.

2) GUI 이벤트 테스트 케이스

이벤트 처리 오류는 개발자가 구현한 이벤트 핸들러에서 발생한다. 따라서 이를 검출하기 위해서는 애플리케이션에서 발생할 수 있는 모든 이벤트를 실행시켜 보아야 한다. 연구 [13]에 따르면 이벤트는 서로 영향을 주고받는 관계이므로, 가능한 다양한 조합의 이벤트를 생성하여 실행시켜야 한다. 단, 이벤트의 조합은 무한하게 생성될 수 있기 때문에 조합의 수를 최소화 할 필요가 있다. 이를 위해서 애플리케이션 관점과 액티비티 관점의 테스트 케이스의 최소 생성 방법을 제시한다.

먼저, 애플리케이션 관점에서는 단일 액티비티 내의 이벤트만을 고려하는 것이다. 하나의 액티비티 내에서 발생되는 이벤트는 다른 액티비티에 영향을 주지 않는다. 따라서 이 점을 활용하면 조합해야 하는 이벤트의 개수가 줄어들기 때문에 생성되는 테스트 케이스의 수를 최소화 할 수 있다. 액티비티 관점에서의 방법은 이벤트 조합의 길이를 한정하는 것이다. 이벤트는 애플리케이션이 종료되기 전까지 무한정 발생할 수 있으며 그 조합의 길이도 무한대가 된다. 조합의 길이가 길어지면 그 경우의 수도 증가하기 때문에 길이를 한정하는 것은 테스트 케이스를 최소화 하는데 도움이 된다. 본 논문에서는 이벤트 조합의 길이, 즉 테스트 케이스의 길이를 발생 가능한 이벤트의 개수($=n_e$)로 한정한다. $T(n_e)$는 n_e의 길이를 갖는 테스트 케이스의 집합이라 정의한다. 이에 대한 타당성을 제시하기 위해서는 이벤트 조합의 길이가 (1)발생 가능한 이벤트의 개수보다 짧은 경우 ($T(m), m < n_e$)와 (2)발생 가능한 이벤트의 개수보다 긴 경우($T(l), l > n_e$)에 대해서 $T(n_e)$가 갖는 이점을 확인할 필요가 있다.

a) 발생 가능한 이벤트의 개수보다 짧은 경우

이를 확인하기 위해서는 부분 조합 $Subseq(t)$에 대해 설명할 필요가 있다. 기본적으로 테스트 케이스 t는 이벤트의 순차적인 흐름으로 구성되며, $t=\langle e_1,e_2,...,e_i\rangle, i \geq 1$와 같이 표기할 수 있다. $t=\langle e_1,e_2,...,e_i\rangle$는 이벤트 e_1부터 이벤트 e_i까지 순차적으로 발생시키는 것을 말하며, i는 이벤트의 순서를 표기하기 위해 사용되었다. 테스트 케이스 $t=\langle e_1,e_2,...,e_i\rangle$에 대해서 부분 조합 $Subseq(t)$는 수식 (1)과 같이 정의된다.

$$Subseq(t) = \{\langle e_1,e_2,...,e_j\rangle | j \leq i\} \tag{1}$$

부분 조합 $Subseq(t)$는 테스트 케이스 t의 부분 흐름의 집합이다. 발생된 이벤트가 다음에 발생될 이벤트에 미치는 영향은 예측할 수 없기 때문에, 부분 조합을 생성할 때는 순서를 준수하여야 한다. 따라서 첫 번째 이벤트부터 차례대로 길이를 늘려가며 부분 조합을 생성하여야 한다. 이렇게 생성된 부분 조합 $Subseq(t)$는 테스트 케이스 t에서 확인할 수 있는 이벤트의 흐름을 알려준다. 부분 조합에 대한 예를 들기 위해 특정 액티비티에서 발생 가능한 이벤트 $E=\{e_1,e_2,e_3\}$가 있다고 가정하자. 이로부터 생성할 수 있는 $T(n_e=3)$는 다음과 같다.

$$T(n_e) = \{t_1,t_2,t_3,t_4,t_5,t_6\}$$

$$t_1=\langle e_1,e_2,e_3\rangle, t_2=\langle e_1,e_3,e_2\rangle, t_3=\langle e_2,e_1,e_3\rangle$$
$$t_4=\langle e_2,e_3,e_1\rangle, t_5=\langle e_3,e_1,e_2\rangle, t_6=\langle e_3,e_2,e_1\rangle$$

위 테스트 케이스 중에서 테스트 케이스 t_1에 대한 부분 조합 $Subseq(t_1)$는 $\langle e_1\rangle, \langle e_1,e_2\rangle, \langle e_1,e_2,e_3\rangle$이다. 이를 통해서 테스트 케이스 t_1은 $\langle e_1,e_2,e_3\rangle$ 외에도 $\langle e_1\rangle$, $\langle e_1,e_2\rangle$에 대해서도 확인할 수 있음을 알 수 있다. 즉, 테스트 케이스 t_1보다 더 짧은 길이를 갖는 테스트 케이스에 대해서 테스트를 수행하지 않더라도 테스트 케이스 t_1을 통해 이를 확인할 수 있다는 의미이다. 물론 테스트 케이스 t_1만으로 $T(n_s)$의 모든 테스트 케이스를 대신할 수는 없다. 하지만 테스트 케이스 t_1으로 확인할 수 없는 부분은 나머지 테스트 케이스 t_2, t_3, t_4, t_5, t_6을 통해서 확인할 수 있다.

$$\begin{aligned} Subseq(T(n_e=3)) &= \bigcup_1^6 Subseq(t_k) \\ &= \{\langle e_1\rangle, \langle e_1,e_2\rangle, \langle e_1,e_2,e_3\rangle\} \cup \\ &\quad \{\langle e_1\rangle, \langle e_1,e_3\rangle, \langle e_1,e_3,e_2\rangle\} \cup ... \cup \\ &\quad \{\langle e_3\rangle, \langle e_3,e_2\rangle, \langle e_3,e_2,e_1\rangle\} \\ &= \{\langle e_1\rangle, \langle e_2\rangle, \langle e_3\rangle\} \cup \\ &\quad \{\langle e_1,e_2\rangle, \langle e_1,e_3\rangle, ..., \langle e_3,e_2\rangle\} \cup \\ &\quad \{\langle e_1,e_2,e_3\rangle, \langle e_1,e_3,e_2\rangle, ..., \langle e_3,e_2,e_1\rangle\} \\ &= T(1) \cup T(2) \cup T(3) \end{aligned}$$

위 계산을 통해서 $T(1)$과 $T(2)$의 테스트 케이스는 $T(n_e=3)$의 테스트 케이스를 통해서 테스트 될 수 있음을 알 수 있다. 여기서 $T(1)$과 $T(2)$는 곧 $T(m)$이므로 수식 (2)와 같이 정리할 수 있다.

$$Subseq(T(n_e)) = T(m) \cup T(n_e) \tag{2}$$

우리는 수식 (2)을 통해서 $T(m)$ 대신 $T(n_e)$을 이용하여 테스트하는 것이 효율적임을 알 수 있다.

b) 발생 가능한 이벤트의 개수보다 긴 경우

이 경우에는 명확한 근거를 제시하기 힘들다. 기본적으로 테스트 케이스의 길이가 길면 길수록 더 많은 이벤트의 조합을 확인할 수 있어 오류를 검출할 수 있는 확률이 증가하기 때문이다. 다만, 테스트 케이스의 효율성 측면에서 보았을 때 문제가 될 수 있다. 테스트 케이스의 길이가 늘어나면 생성되는 테스트 케이스의 수는 기하급수적으로 늘어나기 때문이다. 만일 발생 가능한 이벤트가 x개 존재한다고 할 때, 테스트 케이스의 길이 y에 따라 생성되는 테스트 케이스의 개수 $n(t)$는 수식 (3)과 같다.

$$n(t) = \begin{cases} {}_xP_y & , y \leq x \\ x! \bullet x^{(y-x)} & , y > x \end{cases} \tag{3}$$

수식 (3)을 통해서 $T(n_e)$보다 테스트 케이스의 길이가 하나 증가할 때마다 테스트 케이스의 길이는 n_e배만큼 증가하는 것을 알 수 있다. 또한 $T(n_e)$의 경우에는 모든 이벤트가 한 번씩만 발생하는데 반해서 $T(l)$의 경우에는 이미 발생된 이벤트를 중복으로 발생시키게 된다. 따라서 이를 통해 얻는 효과보다는 추가적으로 부담하여야 하는 테스

트 비용이 더 크다. 그러므로 $T(l)$ 대신 $T(n_e)$을 이용하여 테스트하는 것을 효율적임을 알 수 있다.

위 내용을 바탕으로 생성되는 Fig. 3의 액티비티의 GUI 이벤트 테스트 케이스는 24가지가 된다. 액티비티 내의 모든 GUI 컴포넌트에서 동일한 이벤트가 발생하므로 편의상 이벤트 대신 GUI 컴포넌트 이름으로 대신 표기한다: $\langle A,B,C,D\rangle, \langle A,B,D,C\rangle, ..., \langle D,C,B,A\rangle$.

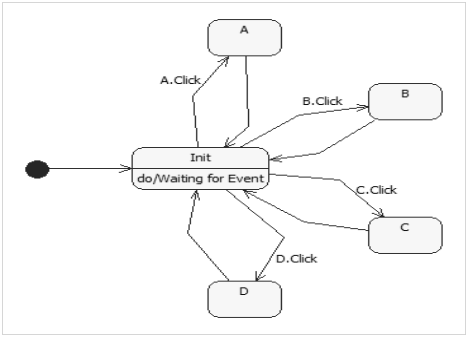

Fig. 3. Example of Activity Event-flow

만일 Fig. 3의 액티비티에서 발생할 수 있는 이벤트에 약간의 제약 조건이 존재한다고 가정하자. 첫째, GUI 컴포넌트 A의 클릭 이벤트가 처리되기 위해서는 GUI 컴포넌트 B의 클릭 이벤트가 선행되어야 한다. 둘째, GUI 컴포넌트 C의 클릭 이벤트가 발생되면 다른 액티비티로의 전환이 수행된다. 이와 같은 제약 조건은 소스코드로부터 도출할 수 있으며, 만약 이러한 조건이 존재한다면 생성할 수 있는 이벤트의 조합은 바뀌게 된다. 그리고 이러한 조건이 테스트 케이스를 최소화 시킬 수 있는 여지를 마련해준다.

먼저, GUI 컴포넌트 C의 클릭 이벤트가 발생하면 나머지 이벤트는 발생하지 못한다. 다른 액티비티로 전환되어 Fig. 3의 액티비티 내부 이벤트가 실행될 수 없기 때문이다. 다음으로 컴포넌트 A의 클릭 이벤트는 항상 컴포넌트 B의 클릭 이벤트를 수반하는 상황을 고려한다. 이런 상황을 고려하면 Fig. 3은 Fig. 4와 같이 수정된다.

Fig. 4를 보면 GUI 컴포넌트 C의 클릭 이벤트가 발생하면 종료 상태로 전이된다. 이와 같은 이벤트를 종단 이벤트라 한다. 또한 선행 이벤트가 존재하는 경우에는 이를 하나의 그룹 이벤트로 표현하고 GUI 컴포넌트 A에 대한 이벤트 대신 그룹 이벤트 G1을 발생시키는 것으로 한다. 이와 같은 제약 조건을 고려하여 GUI 이벤트 테스트 케이스를 생성하기 위해서는 다음 절차를 따른다.

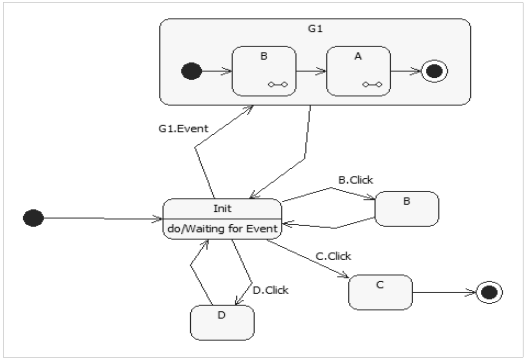

Fig. 4. Modified Activity Event-flow

1. 선행 이벤트가 존재하는 이벤트에 대해서 그룹 이벤트를 생성한다. 만일, 다수의 선행 이벤트가 존재한다면 선행 이벤트의 조합 중 하나에 대해서 그룹 이벤트를 생성한다. 이는 이벤트의 순서보다는 하나의 묶음으로써 이벤트가 발생되는 것이 중요하기 때문이다. 이 때, 생성되는 조합의 길이는 그룹 이벤트에 속하는 이벤트의 개수와 같다.
2. 그룹 이벤트에 속하는 이벤트는 이벤트 집합 E에서 제외하며, 대신 그룹 이벤트를 포함시킨다.
3. 종단 이벤트 E_t와 일반 이벤트(non-terminal event) E_n을 분류한다.
4. 일반 이벤트 E_n을 이용하여 이벤트 흐름 S를 구성한다. 이 때, 생성되는 이벤트 흐름의 길이는 이벤트 E_n의 개수와 같다.
5. 생성된 이벤트 흐름 S와 종단 이벤트 E_t를 조합하여 테스트 케이스 T를 생성한다. 이때, 이벤트 흐름 S와 종단 이벤트 E_t의 조합은 다음의 규칙을 따른다.

$$T = S \otimes E_t = \{\langle s,e\rangle \mid s \in S, e \in E_t, \\ \textit{each element of } S \textit{ appears at least once as } s, \\ \textit{each element of } E_t \textit{ appears at least once as } e\}$$

생성 절차에 대한 예시를 위해 Fig. 4과 같은 이벤트 흐름도로부터 GUI 이벤트 테스트 케이스를 생성한다. Fig. 3에서 발생 가능한 이벤트의 집합은 $E=\{A,B,C,D\}$와 같다. 먼저, 그룹 이벤트를 생성한다. GUI 컴포넌트 A의 클릭 이벤트는 GUI 컴포넌트 B의 클릭 이벤트를 선행 이벤트로 가지므로 그룹 이벤트 G1에 대해서 다음과 같이 표현할 수 있다. 그룹 이벤트를 생성한 뒤, 이벤트 집합 E를 갱신한다. G1에 속하는 이벤트 A와 B를 제외시키고 그룹 이벤트 G1를 추가한다.

$$G1 = \langle B,A\rangle,\ E = \{C,D,G1\}$$

갱신된 이벤트 집합 E로부터 종단 이벤트 E_t와 일반 이벤트 E_n을 분류한 뒤, 일반 이벤트 E_n으로부터 이벤트 흐름 S를 생성한다.

$$E_t = \{C\},\ E_n = \{D,G1\}$$
$$S = \{\langle D,G1\rangle, \langle G1,D\rangle\}$$

생성된 이벤트 흐름 S와 종단 이벤트 E_t을 통해 생성되는 GUI 이벤트 테스트 케이스는 다음과 같이 두 가지가 생성된다.

$$
\begin{aligned}
T &= S \otimes E_t \\
&= \{\langle G1, D\rangle, \langle D, G1\rangle\} \otimes \{C\} \\
&= \{\langle G1, D, C\rangle, \langle D, G1, C\rangle\} \\
&= \{\langle B, A, D, C\rangle, \langle D, B, A, C\rangle\}
\end{aligned}
$$

3) 인텐트 테스트 케이스

인텐트 오류를 확인하기 위해서는 인텐트 내부의 값이 어느 시점에 사용되는지 파악할 필요가 있다. 액티비티에서 인텐트 내부의 값을 사용하는 위치는 두 곳으로 나눠진다. Fig. 5는 액티비티에서 인텐트를 수신하여 이를 사용하는 부분을 도식화한 것이다.

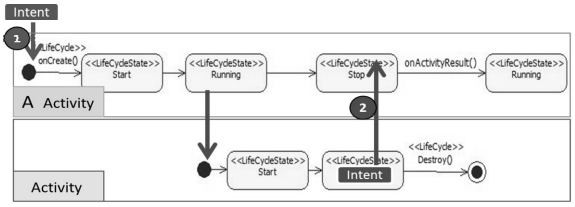

Fig. 5. Location to Use Intent Data

Fig. 5를 보면 액티비티가 생성될 때 수신한 인텐트는 onCreate()에서 처리하고, 액티비티가 다른 액티비티로부터 결과 값을 돌려받는 경우에는 onActivityResult()에서 처리하는 것을 알 수 있다. 인텐트의 송수신은 시스템을 통해서 이루어진다. 따라서 인텐트의 수신 여부를 검증하는 것이 아니라 수신한 인텐트의 처리 과정이 문제없이 동작하는지 검증하여야 한다. 이 때, 올바른 값뿐만 아니라 그렇지 않은 값을 수신하였을 경우도 고려해야만 정확한 테스트를 수행하는 것이다. 그러므로 인텐트 테스트 케이스는 해당 인텐트가 가질 수 있는 값의 집합으로 정의된다.

4) 조합 테스트 케이스

앞서 정의한 세 가지 테스트 케이스는 각각 생명주기 오류, GUI 이벤트 처리 오류, 인텐트 오류를 검출하기 위한 테스트 케이스이다. 각각 서로 다른 오류의 검출을 목적으로 정의되었기 때문에 동일한 애플리케이션에 대해서 세 가지의 테스트를 수행함으로써 모든 오류를 검출할 수 있다. 세 가지의 테스트는 따로 수행될 수도 있지만 서로 완전히 무관한 것이 아니기 때문에 동시에 수행할 수 있다. 예를 들어, GUI 이벤트 테스트를 수행하기 위해서는 액티비티는 동작중 상태에 있어야 한다. 이 때, 필연적으로 액티비티의 생명주기 상태가 시작 상태에서 동작중 상태로 전환되므로, 이에 대한 생명주기 테스트를 수행할 수 있다. 또한 동작중 상태에 도달한 액티비티에 대해 GUI 이벤트 테스트를 수행한 뒤, 액티비티를 종료시킴으로써 생명주기 테스트를 모두 완료할 수 있다. 뿐만 아니라 액티비티를 생성하는 과정에서 인텐트 테스트 또한 수행할 수 있다. 이처럼 독립적으로 테스트를 수행한다면 추가적으로 소요될 시간을 한 번에 테스트함으로써 시간을 절감할 수 있다. 게다가 그 수행 결과는 동일하게 얻을 수 있다. 따라서 본 논문에서는 Fig. 6과 같은 형태의 조합 테스트 케이스를 정의한다.

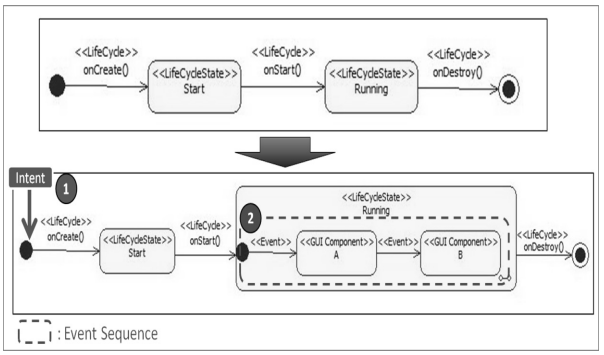

Fig. 6. Example of Combined Test Cases

앞서 언급한 것처럼 GUI 이벤트 테스트는 액티비티가 동작중 상태에 있을 때 수행할 수 있다. 따라서 조합 테스트 케이스는 기본적으로 생명주기 테스트 케이스를 기반으로 하며, GUI 이벤트 테스트 케이스는 각 생명주기 테스트 케이스의 동작중 상태에 내포시킨다. 또한 필요에 따라 액티비티 생성 단계에서 인텐트 테스트 케이스를 추가할 수 있다. 마지막으로 다른 액티비티의 결과 값을 돌려받는 상황을 표현한 생명주기 테스트 케이스 LT4에도 인텐트 테스트 케이스를 추가할 수 있다. 이렇게 생명주기 테스트 케이스를 기반으로 GUI 이벤트 테스트 케이스와 인텐트 테스트 케이스를 추가함으로써 조합 테스트 케이스를 생성할 수 있다.

5. 안드로이드 애플리케이션을 위한 테스트 케이스 자동 생성 방안

이 절에서는 4절에서 정의한 테스트 케이스를 자동으로 생성하는 방법을 제시한다. 본 논문에서 제시하는 방법은 구현된 애플리케이션의 소스코드와 XML 레이아웃으로부터 정보를 추출한 뒤, 추출된 정보를 바탕으로 테스트 케이스를 생성한다.

5.1 테스트 케이스 생성을 위한 정보 추출

이 논문에서는 안드로이드 애플리케이션의 GUI가 XML 레이아웃을 통해 구현되었다는 가정 하에 정보를 추출한다. 안드로이드 애플리케이션은 두 가지 요소를 통해 구현된다. 첫째는 자바 언어를 기반으로 작성된 소스코드이다. 소스코드에는 액티비티 클래스가 구현되어 있으며, 이 안에는 생명주기 메서드, 이벤트 핸들러 등과 같이 애플리케이션의 동작과 관련된 요소가 포함된다. 둘째는 XML 레이아웃이며, 한 액티비티의 화면 구성에 대한 내용이 포함된다. 소스코드와 XML 레이아웃으로부터 추출할 수 있는 정보가 다르기 때문에, 두 요소를 통해 테스트 케이스 생성을 위한 정보를 추출한다. 추출되는 정보는 Table 2와 같다.

소스코드로부터의 정보는 이클립스 자바 파서(Eclipse Java Parser)를 통해 AST를 구성하여 추출한다. XML 레이아웃은 XML 기반으로 작성되므로, XML 파서를 통해 정보를 추출한다.

Table 2. Extracted Information from Source Code and XML Layout

	Information
Source Code	• Activity class name • Intent • Event Listener of GUI component • Intent call method
XML Layout	• List of GUI components ✓ GUI component ID ✓ GUI component Type ✓ GUI component properties

5.2 단일 액티비티에 대한 GUI 맵 구성

소스코드와 XML 레이아웃으로부터 정보를 추출한 다음 테스트 케이스 생성을 위해서 GUI 맵을 구성한다. 테스트 케이스를 생성하기 위해서는 추출된 정보 사이의 관계를 파악하여야 한다. GUI 이벤트 테스트 케이스를 생성하기 위해서는 이벤트 사이의 의존 관계 뿐만 아니라 다른 액티비티를 호출하는 이벤트가 무엇인지 파악하여야 한다. 또한 해당 액티비티에서 인텐트 내부의 값을 사용하는지 역시 파악하여야 인텐트 테스트 케이스를 생성할 수 있다. 따라서 추출된 정보를 단순히 나열하는 것이 아니라 그들 사이의 관계를 표현할 수 있는 중간 표현 방법이 존재해야 한다. 이와 같은 역할을 수행하는 것이 GUI 맵이며, Fig. 7과 같은 구조를 갖는다.

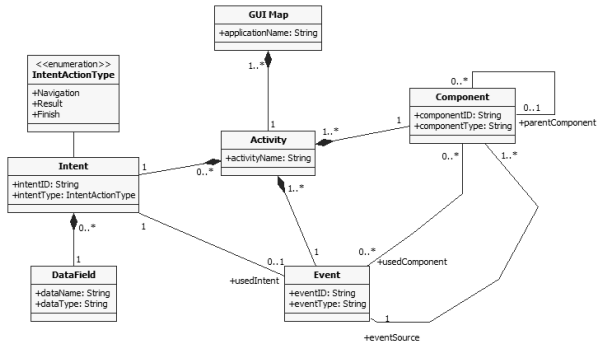

Fig. 7. Structure of GUI Map

GUI 맵은 각 액티비티마다 따로 구성되며, 액티비티에서 사용되는 GUI의 구조, 인텐트, 이벤트에 대한 정보뿐만 아니라 정보 간의 관계를 명시한다. GUI 맵을 이용하면 정보 간의 관계를 파악하는 것 외에도 정형화된 형식으로 테스트 대상을 명시할 수 있다. 본 논문에서는 GUI 맵을 XML 형식으로 기술하는데, 이를 통해 얻을 수 있는 이점은 컴퓨터가 테스트 대상의 정보를 쉽게 파악할 수 있다는 것이다. 따라서 GUI 맵을 사용하면 테스트 케이스를 자동으로 생성하기가 용이하다.

GUI 맵을 구성하는 과정에서 해당 액티비티에서 사용되는 GUI 컴포넌트와 같이 쉽게 파악할 수 있는 정보 외에도 간접적으로 유추하여야 하는 정보가 존재한다. 첫 번째가 이벤트 정보이며, 두 번째는 인텐트 정보이다. 소스코드에는 이벤트에 대한 직접적인 표기가 존재하지 않기 때문에 이벤트를 처리하기 위해 사용되는 이벤트 리스너의 종류를 통해 이벤트 정보를 파악한다. Table 3은 GUI 컴포넌트의 기본 클래스인 View 클래스에서 발생할 수 있는 이벤트와 그 이벤트를 처리하기 위한 이벤트 리스너를 정리한 것이다.

Table 3. Event/Event Listener for View Class

Event	Event Listener
Click	OnClickListener
LongClick	OnLongClickListener
FocusChange	OnFocusChangeListener
Key	OnKeyListener
Touch	OnTouchListener
CreateContextMenu	CreateContextMenuListener

Table 3을 보면 알 수 있듯이 이벤트 정보는 GUI 컴포넌트에 등록된 이벤트 리스너의 종류를 통해 유추할 수 있다. 다음으로 인텐트 정보 중에서 다른 액티비티로부터 결과 값을 넘겨받는 상황을 파악하기 위해서는 다른 액티비티를 실행시키는 메서드를 확인해야 한다. 액티비티에서 다른 액티비티를 실행시킬 때 사용하는 메서드는 두 가지가 있다. 하나는 startActivityForResult() 메서드이며, 다른 하나는 startActivity() 메서드이다. 이름에서 알 수 있듯이 startActivityForResult() 메서드를 통해 실행될 경우에만 결과 값을 기다리게 된다. 이 정보는 인텐트 테스트의 수행 여부를 결정하기 위해 파악하여야 한다. 이 두 가지 정보 외에는 소스코드나 XML 레이아웃을 통해 쉽게 파악할 수 있으며, 이와 같은 내용을 통해 GUI 맵을 구성한다.

5.3 조합 테스트 케이스 생성

조합 테스트 케이스는 생명주기 테스트 케이스를 기반으로 하여 GUI 이벤트 테스트 케이스와 인텐트 테스트 케이스를 추가하여 생성할 수 있다. 따라서 조합 테스트 케이스를 생성하기 위해서는 먼저 세 가지 테스트 케이스를 생성하여야 한다.

생명주기 테스트 케이스의 경우 모든 액티비티에서 동일하게 Table 1의 테스트 케이스를 사용하기 때문에 따로 테스트 케이스를 생성하는 절차는 필요하지 않다. GUI 이벤트 테스트 케이스는 GUI 맵의 내용을 바탕으로 생성한다. GUI 맵에는 해당 액티비티에서 발생할 수 있는 이벤트의 종류뿐만 아니라 이벤트 간의 의존 관계나 다른 액티비티를 호출하는 이벤트에 대한 정보도 포함되어 있다. 이벤트 간의 의존 관계를 통해서 이벤트 그룹을 생성할 수 있으며, 액티비티를 호출하는 이벤트를 통해서 종단 이벤트를 파악할 수 있다. GUI 이벤트 테스트 케이스를 생성한 후, 각 GUI 이벤트 테스트 케이스를 생명주기 테스트 케이스 내부에 내포한다. 이 때, 동작(running) 상태에 내포해야 하고 GUI 이벤트 테스트 케이스의 종단 이벤트에 따라 알맞은 생명주기 테스트 케이스를 선택해야 한다. 다음으로 인텐트 테스트 케이스를 생성하기 위해서 해당 액티비티가 인텐트의 Data 필드를 사용하는지 파악하여야 한다. 만일 인텐트의 Data 필드를 사용하지 않는다면 인텐트 테스트를 수행할 필요가 없기 때문이다. 만일 인텐트의 Data 필드를 사용한다면, 어떤 자

료형의 값을 사용하는지 파악하여야 한다. 인텐트 테스트 케이스를 생성하기 위해서는 인텐트가 가질 수 있는 다양한 값을 생성하여야 하며, 이는 곧 값의 자료형을 통해서 결정되기 때문이다. 인텐트 테스트 케이스를 생성하기 위해 필요한 정보들은 모두 GUI 맵에 명시되어 있으며, 이를 통해서 인텐트 테스트 케이스를 생성할 수 있다. 이와 같은 과정을 통해 조합 테스트 케이스를 생성할 수 있다.

6. 적용 예제

본 논문에서 제시한 테스트 케이스가 어떻게 생성되는지 보이기 위하여 승차권 예매 애플리케이션에 적용한다. 승차권 예매 애플리케이션은 승차권 조회 및 예매, 예매한 승차권 확인과 같은 기능을 제공하는 애플리케이션으로 아홉 개의 액티비티로 구성된다. Fig. 8은 승차권 예매 애플리케이션의 구조를 보여준다.

Fig. 8. Structure of Ticket Reservation Application

아홉 개의 액티비티 중에서 ⑥번 액티비티로부터 테스트 케이스를 생성하는 과정을 설명한다. 먼저, ⑥번 액티비티와 관련된 소스코드와 XML 레이아웃으로부터 GUI 맵을 구성한다. ⑥번 액티비티로부터 구성된 GUI 맵은 Fig. 9와 같다.

⑥번 액티비티의 이름은 "JoinActivity"로, 다섯 개의 에디트 텍스트(Edit Text)와 네 개의 버튼(Button)으로 구성된다. 에디트 텍스트에는 텍스트 입력을 위한 키 이벤트가 발생되며, 버튼에서는 클릭 이벤트가 발생된다. Fig. 9를 보면 버튼 idCheckButton과 joinButton의 이벤트 핸들러에서 사용되는 에디트 텍스트가 존재함을 알 수 있다. 이를 통해서 Table 4와 같은 그룹 이벤트를 생성할 수 있다.

Table 4. Group Event in JoinActivity

Group ID	Event Sequence
G1	KE1 → KE2 → KE3 → CE1
G2	KE4 → KE5 → CE2

```
<GUIMap applicationName="Ticket Reservation">

    .....

    <Activity activityName="JoinActivity">
        <Components>
            <Component componentID="idCheckButton" componentType="Button" />
            <Component componentID="joinButton" componentType="Button" />
            <Component componentID="cancelButton" componentType="Button" />
            <Component componentID="termsButton" componentType="Button" />
            <Component componentID="idEditText" componentType="EditText" />
            <Component componentID="passEditText" componentType="EditText" />
            <Component componentID="passCheckEditText" componentType="EditText" />
            <Component componentID="phoneEditText" componentType="EditText" />
            <Component componentID="addressEditText" componentType="EditText" />
        </Components>
        <Intents>
            <Intent intentID="inStart" intentType="Finish" />
            <Intent intentID="inConfirm" intentType="Navigation" />
            <Intent intentID="finish" intentType="Finish" />
        </Intents>
        <Events>
            <Event eventID="CE1" eventSource="idCheckButton" eventType="Click">
                <UsedComponent componentID="idEditText" />
                <UsedComponent componentID="passEditText" />
                <UsedComponent componentID="passCheckEditText" />
            </Event>
            <Event eventID="CE2" eventSource="joinCheckButton" eventType="Click">
                <UsedComponent componentID="phoneEditText" />
                <UsedComponent componentID="addressEditText" />
                <UsedIntent intentID="inStart" />
            </Event>
            <Event eventID="CE3" eventSource="cancelButton" eventType="Click">
                <UsedIntent intentID="finish" />
            </Event>
            <Event eventID="CE4" eventSource="termsButton" eventType="Click">
                <UsedIntent intentID="inConfirm" />
            </Event>
            <Event eventID="KE1" eventSource="idEditText" eventType="Key" />
            <Event eventID="KE2" eventSource="passEditText" eventType="Key" />
            <Event eventID="KE3" eventSource="passCheckEditText" eventType="Key" />
            <Event eventID="KE4" eventSource="phoneEditText" eventType="Key" />
            <Event eventID="KE5" eventSource="addressEditText" eventType="Key" />
        </Events>
    </Activity>

    .....

</GUIMap>
```

Fig. 9. GUI Map of JoinActivity

이렇게 생성된 그룹 이벤트를 기반으로 Table 5와 같은 GUI 테스트 케이스를 생성할 수 있다.

Table 5. GUI Event Test Case for JoinActivity

TestCase ID	Event Sequence
GT1	G1 → G2
GT2	G1 → CE3
GT3	G1 → CE4

⑥번 액티비티에서는 인텐트 테스트 케이스가 존재하지 않으므로 생명주기 테스트 케이스와 GUI 이벤트 테스트 케이스만을 이용하여 조합 테스트 케이스를 생성한다. 이 때, GUI 이벤트 테스트 케이스의 종단 이벤트에 따라 알맞은 생명주기 테스트 케이스를 선택한다. 예를 들어, GT2의 종단 이벤트인 CE3는 액티비티를 종료시키므로 종료 상태 직전의 동작 상태에 내포시키는 것이 적합하다. GT3의 종단 이벤트인 CE4는 다른 액티비티로의 전환을 유발하므로 생명주기 테스트 케이스 LT4의 첫 번째 동작 상태에 내포시키는 것이 적합하다. 이를 기반으로 생성된 JoinActivity의 조합 테스트 케이스는 Table 6과 같다. 조합 테스트 케이스는 "<생명주기 테스트 케이스, GUI 이벤트 테스트 케이스, 인텐트 테스트 케이스>"의 형태로 구성된다. 여기서 ∅는 해당 테스트 케이스가 없음을, {}는 LT2나 LT4처럼 동작 상태가 두 번 이상 나타날 때 {}에 포함된 테스트 케이스 순서대로 동작 상태에 차례대로 대응됨을 의미한다.

Table 6. Combined Test Case for JoinActivity

TestCase ID	TestCase
TC1	<LT1, GT1, ∅>
TC2	<LT1, GT2, ∅>
TC3	<LT2, {∅, GT1}, ∅>
TC4	<LT2, {∅, GT2}, ∅>
TC5	<LT3, ∅, ∅>
TC6	<LT4, {GT3, GT1}, ∅>
TC7	<LT4, {GT3, GT2}, ∅>
TC8	<LT5, GT3, ∅>

GUI 이벤트 테스트 케이스는 적합한 생명주기 테스트 케이스에 내포되며, 내포시키기 적합한 GUI 이벤트 테스트 케이스가 없는 경우에는 공란으로 둔다. 인텐트 테스트 케이스 역시 적합한 것이 없는 경우에는 공란으로 둔다. GUI 이벤트 테스트 케이스가 공란인 경우에는 테스트 도구에서 자체적으로 상태를 변경시켜야 한다. 인텐트 테스트 케이스가 공란인 경우에는 인텐트 테스트를 수행하지 않는다. 이와 같은 방식으로 생성된 JoinActivity의 조합 테스트 케이스는 총 여덟 가지이다.

생성된 조합 테스트 케이스는 테스트를 수행할 때는 하나의 시나리오처럼 재구성된다. 예를 들어 조합 테스트 케이스 TC7는 Fig. 10과 같은 흐름의 시나리오로 재구성된다. 먼저, 액티비티가 생성된 후, 시작 상태로 전환 된다. 다음으로 동작 상태로의 전환이 발생하는데, 이때는 GUI 이벤트 테스트 케이스 GT3이 실행된다. GT3의 이벤트 흐름은 “KE1 → KE2 → KE3 → CE1 → CE4”가 된다. GT3을 수행하게 되면 액티비티는 정지 상태로 전환된다. 이는 시스템에 의해 발생되는 것이 아니라 GT3의 마지막 이벤트인 CE4에 의해서 자연스레 발생된다. 이후에는 다시 동작 상태로 전환되며, 이때는 GUI 이벤트 테스트 케이스 GT2가 실행된다. GT2의 이벤트 흐름은 “KE1 → KE2 → KE3 → CE1 → CE3”이 된다. GT2를 수행하면 액티비티가 종료 상태로 전환되며 테스트가 종료된다. 물론 액티비티의 종료는 CE3을 통해 수행된다.

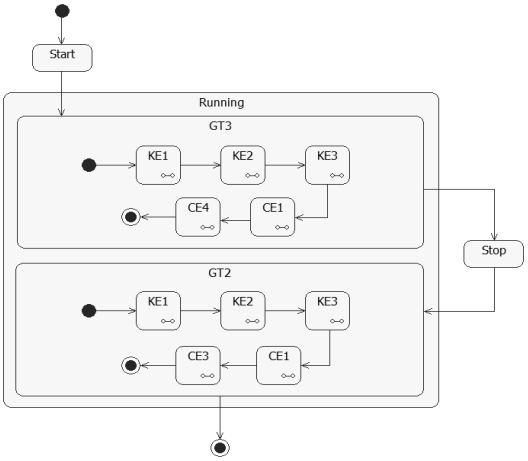

Fig. 10. Test Scenario for TC7

테스트 케이스 TC7은 Fig. 11과 같은 테스트 코드로 실현되어 테스트를 수행한다. Fig. 11의 테스트 코드는 구글에서 제공하는 ActivityInstrumentationTestCase2와 테스트 프레임워크인 Robotium을 이용하여 작성하였다. 정해진 이벤트만으로 테스트 케이스가 작성되기 때문에, 각 이벤트를 함수로 구성하여 테스트 메서드를 작성하였다. 아래 코드에 보이는 EventG1(), EventCE3(), EventCE4()가 해당된다. 테스트 수행 결과를 확인하기 위해서 Assert 함수를 사용하였다. 액티비티의 동작 상태가 알맞게 변경되었는지 혹은 알맞은 액티비티로 전환되었는지 등을 확인하도록 하였다.

```
public void testCase07() throws Exception
{
    Log.i("TestCase", "TestCase7");

    Assert.assertEquals(JoinActivity.ACTIVITY_STATE_RUNNING,
                        getActivity().getActivityState());

    // G1
    EventG1("T5W8y5JkK6", "ToiFEemO9a");
    Assert.assertEquals(getActivity().getCheckBit(), true);

    // CE4
    EventCE4();
    Thread.sleep(1000);
    solo.assertCurrentActivity("Navigate", "NoticeActivity");

    Thread.sleep(1000);
    Assert.assertEquals(JoinActivity.ACTIVITY_STATE_STOP,
                        getActivity().getActivityState());

    // Back to JoinActivity
    solo.goBack();
    Thread.sleep(1000);
    Assert.assertEquals(JoinActivity.ACTIVITY_STATE_RUNNING,
                        getActivity().getActivityState());

    // G1
    EventG1("qbEzkwG4nn", "PTigL3IKGs");
    Assert.assertEquals(getActivity().getCheckBit(), true);

    // CE3
    EventCE3();
    Thread.sleep(2000);
    Assert.assertEquals(JoinActivity.ACTIVITY_STATE_TERMINATE,
                        getActivity().getActivityState());
}
```

Fig. 11. Test Method for TC7

작성된 JoinActivity의 모든 테스트 케이스에 대한 수행 결과는 Fig. 12와 같다. 모두 정상 동작하는 것을 볼 수 있으며, 테스트를 수행하는데 약 170초 정도가 소요되었다.

그러나 Fig. 12와 같은 결과를 통해서는 테스트 케이스가 자신의 역할을 제대로 수행하는지 파악하기 어렵다. 아무런 오류를 찾을 수 없는 것처럼 비춰지기 때문이다. 따라서 본

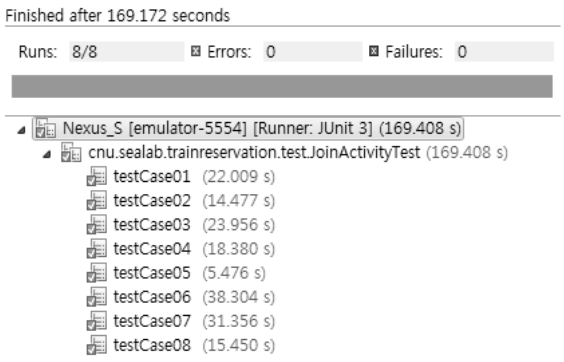

Fig. 12. Test Result for JoinActivity

논문에서 제시한 테스트 케이스를 통해 오류를 찾을 수 있음을 보이기 위해 뮤턴트(Mutant)[14]를 이용한다. 뮤턴트는 뮤테이션 테스트(Mutation Test)에서 사용되는 것으로, 본래 프로그램을 수정한 것을 말한다. 따라서 뮤턴트는 본래 프로그램과는 다른 동작을 하게 되고, 테스트 케이스가 이를 찾아낼 수 있는지를 통해서 테스트 케이스를 평가할 수 있다. 본 논문에서는 다음과 같은 기준으로 뮤턴트를 생성한다.

- **(MT1) 이벤트 핸들러 교환** : GUI 컴포넌트에 연결된 이벤트 핸들러가 서로 잘못 지정될 수 있다. 이에 대한 상황을 고려하여 두 이벤트 간의 이벤트 핸들러를 교환한다.
- **(MT2) 전환 액티비티 변경** : 실제 전환되어야 하는 액티비티가 아닌 다른 액티비티로 전환이 일어나도록 수정한다.
- **(MT3) 이벤트 종류 변경** : 본 논문에서는 이벤트를 종단 이벤트와 일반 이벤트로 나눈다. 종단 이벤트를 일반 이벤트로, 일반 이벤트를 종단 이벤트가 되도록 수정한다. 종단 이벤트의 경우, startActivity()와 finish() 같은 함수를 호출하므로 이를 호출하는 부분을 제거한다. 반대로 일반 이벤트는 각 함수를 호출하도록 수정한다.
- **(MT4) 로직 수정** : 개발자의 실수로 의도치 않은 연산자를 사용할 수 있다. 이를 고려하여 이벤트 핸들러 내부에서 사용된 연산자를 수정한다.

위와 같은 기준을 바탕으로 Table 7과 같이 아홉 개의 뮤턴트를 생성하였다. 각 기준은 뮤턴트 형식으로 사용되며, 각각은 MT1, MT2, MT3, MT4라는 식별자를 갖는다. 식별자는 위에서부터 차례대로 부여하였다.

Table 7. Mutant Lists for JoinActivity

Mutant ID	Modified Event	Mutant Type
M1	CE2, CE3	MT1
M2	CE1, CE4	MT1
M3	CE4	MT2
M4	CE3	MT3
M5	CE4	MT3
M6	CE1	MT3
M7	CE1	MT4
M8	CE1	MT4
M9	CE2	MT4

Table 7의 뮤턴트에 대해서 테스트를 수행한 결과는 Table 8과 같다. Table 8에서 O는 수정되지 않은 애플리케이션을 가리킨다. 통과된 테스트 케이스에 대해서는 P(Pass), 통과하지 못한 테스트 케이스에 대해서는 F(Fail)로 표기하였다. F는 주어진 테스트 케이스에 의해 수정된 애플리케이션의 오류(뮤턴트)가 발견되었음을 의미한다.

테스트 결과를 통해 알 수 있듯이 본 논문에서 제시한 테스트 케이스는 모든 뮤턴트를 판별해내는 것을 알 수 있다. 따라서 제시된 테스트 케이스를 통해 안드로이드 애플리케이션의 오류를 충분히 검출할 수 있음을 알 수 있다.

Table 8. Mutation Test Result

	TC1	TC2	TC3	TC4	TC5	TC6	TC7	TC8
O	P	P	P	P	P	P	P	P
M1	P	F	P	F	P	P	F	P
M2	F	F	F	F	P	F	F	F
M3	P	P	P	P	P	F	F	F
M4	P	F	P	F	P	P	F	P
M5	P	P	P	P	P	F	F	F
M6	F	F	F	F	P	F	F	F
M7	F	F	F	F	P	F	F	F
M8	F	F	F	F	P	F	F	F
M9	F	P	F	P	P	F	P	P

7. 결 론

이 논문에서는 안드로이드 애플리케이션을 테스트하기 위한 테스트 케이스를 분류하고 정의하였으며 이를 생성하기 위한 방법을 제시하였다. 이를 위해 구현으로부터 테스트에 필요한 정보를 추출하는 방법, 추출된 정보를 기반으로 테스트 케이스를 생성하는 방법을 제안하였다. 정보를 추출하고 테스트 케이스를 생성하는 과정이 자동화될 수 있기 때문에, 애플리케이션 테스팅을 위한 추가적인 노력이 요구되지 않는다. 또한 이 논문에서는 단순히 액티비티를 구성하는 GUI 뿐만 아니라 액티비티가 갖는 생명주기와 인텐트에 대한 테스트 방법도 고려하였다. 따라서 논문에서 제안하는 방법을 적용하면 애플리케이션을 구성하는 액티비티의 전반적인 부분을 테스트 할 수 있다.

물론, 안드로이드 애플리케이션은 액티비티 외에도 다른 요소(컨텐트 프로바이더, 브로드캐스트 리시버, 서비스)를 포함할 수 있다. 그러나 액티비티를 제외한 요소들은 그것들만의 부가적인 기능을 제공한다. 컨텐트 프로바이더는 다른 애플리케이션이 데이터베이스에 접근할 때 제공되는 요소이며, 브로드캐스트 리시버는 시스템 이벤트(메시지 수신, 전화 수신, 배터리 부족 등)를 수신하기 위한 요소이다. 서비스는 애플리케이션에서 수행될 수 있는 백그라운드 작업을 위해 제공되는 요소이다. 따라서 애플리케이션의 주요 로직은 액티비티를 통해서 구현되며, 액티비티를 테스트 한다는 것은 곧 안드로이드 애플리케이션에 대해 테스트 할 수 있음을 의미한다. 그러나 부가적인 기능만을 제공하는 요소일지라도 애플리케이션 자체에 문제를 일으킬 소지가 있다면 테스트 되어야 한다. 그런 이유로 향후에는 이 논문에서 다루지 않은 다른 요소를 고려한 테스트 케이스 생성 방법을 연구할 계획이다.

참 고 문 헌

[1] Gartner, “Gartner Says Worldwide Smartphone Sales Soared in Fourth Quarter of 2011 With 47 Percent Growth”, http://

www.gartner.com/it/page.jsp?id=1924314, February, 2012.
[2] S. Yoon, Y. Seo, M. Ko, and H. S. Kim, “Design of Test Cases and Automatic Generation of the Test Code for UI Unit Test of an Android Application”, Proc. of the KIISE Korea Computer Congress, Vol.39, No.1(B), pp.129-131, 2012.
[3] J. H. Kim, “Android and Android Market”, Journal of Contents Association, Vol.7, No.2, pp.29-36, 2009.
[4] S. M. Hwang, J. J. Kim, “A GUI Testing Method base on Scenario for Mobile Application Software”, Journal of the Korea Academia-Industrial cooperation Society, Vol.9, No.3, pp.681-689, 2008.
[5] S. Yoon, M. Ko, S. Kuk, H. S. Kim, “An automatic generation of test cases and test drivers for Android application's GUI testing”, Proc. of the KIISE Fall Conference 2011, Vol.38, No.2(B), pp.112-115, 2011.
[6] J. G. Lee, S. Kuk, and H. S. 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)
[7] T. Takala, M. Kataka, “Experiences of System-Level Model-Based GUI-Testing of an Android Application”, 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation(ICST), pp.377-386, 2011.
[8] R. Bareiss, T. Sedano, “Improving Mobile Application Development,” MSE workshop of MobiCASE 2011.
[9] D. Amalfitano, A. R. Fasolino, P. Tranmontana, “A GUI Crawling-based technique for Android Mobile Application Testing”, 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Work shops(ICSTW), pp.252-261, 2011.
[10] C. Hu, I. Neamtiu, “Automating GUI Testing for Android Applications”, in Proc. Of the sixth International Workshop on Automation of Software Test, pp.77-83, 2011.
[11] Google, “http://developer.android.com/tools/testing/acti vity_testing.html”
[12] M. Ko, “GUI based Automatic Test Scenario Generation for Testing Android Application”, Master's degree thesis. Chungnam National University, 2013.
[13] L. Zhao, K. Y. Cai, “Event Handler-Based Coverage for GUI Testing,” in Proc. of 10th Int'l Conference on Quality Software, pp.326-331, 2010.
[14] Aditya P. Mathur, “Foundations of Software Testing”, 1st ed., Addison-Wesley Professional, 2008.

고 민 혁
e-mail : minhko@cnu.ac.kr
2010년 우송대학교 IT경영정보학과(학사)
2013년 충남대학교 컴퓨터공학 석사
관심분야 : 소프트웨어 테스팅, 스마트폰

서 용 진
e-mail : yjseo082@cnu.ac.kr
2011년 충남대학교 컴퓨터공학과(학사)
2011년~현 재 충남대학교 컴퓨터공학과 박사재학, 석·박사통합
관심분야 : 소프트웨어 테스팅, 스마트폰, UX/UI

윤 상 필
e-mail : sambitious@cnu.ac.kr
2012년 충남대학교 컴퓨터공학과(학사)
2012년~현 재 충남대학교 컴퓨터공학과 석사과정
관심분야 : 소프트웨어 테스팅, 분산 컴퓨팅 개발 및 검증

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