Universidad de Málaga,

Dept. Lenguajes y Ciencias de la Computación,
E.T.S.I. Informática

Tesis Doctoral

Análisis de Aplicaciones Móviles utilizando Métodos Formales

(*Analyzing the Correctness of Smartphone Apps using Formal Methods*)

Doctorando:

Ana Rosario Espada Sandi

Directora:

María del Mar Gallardo Melgarejo

Málaga, Enero de 2017

AUTOR: Ana Rosario Espada Sandi

http://orcid.org/0000-0002-6900-4611

EDITA: Publicaciones y Divulgación Científica. Universidad de Málaga

Dra. Doña María del Mar Gallardo Melgarejo, Titular de Universidad del Área de Lenguajes y Sistemas Informáticos de la E.T.S. de Ingeniería Informática de la Universidad de Málaga,

Certifica que Doña Ana Rosario Espada Sandi, Ingeniera de Sistemas, ha realizado en el Departamento de Lenguajes y Ciencias de la Computación de la Universidad de Málaga, bajo mi dirección, el trabajo de investigación correspondiente a su Tesis Doctoral titulada:

Análisis de aplicaciones móviles utilizando métodos formales.

Revisado el presente trabajo, estimo que puede ser presentado al tribunal que ha de juzgarlo, y autorizamos la presentación de esta Tesis Doctoral en la Universidad de Málaga.

Málaga, Junio de 2017

Fdo.: María del Mar Gallardo Melgarejo
Titular de Universidad
Área de Lenguajes y Sistemas Informáticos

...Pesan poco más que el aire. Durante los cuatro mil quilómetros de travesía, unas cuantas caen volteadas por el cansancio, los vientos o las lluvias; pero las muchas que resisten aterrizan...
Allí descubren ese reino jamás visto, que desde lejos las llamaba.
Para volar han nacido: para volar este vuelo ...
Eduardo Galeano

Dedicado a mis papás
mis promotores de sueños

a Fabián

Agradecimientos

Debo agradecer de manera especial a mi directora de Tesis María del Mar por aceptar dirigir mi trabajo aun sin conocerme personalmente, por introducirme al grupo de investigación, por su paciencia para enseñarme y corregirme cuando era necesario, por exigirme un mejor trabajo y, a la vez, apoyarme cuando las situaciones eran adversas y complicadas. Pero sobre todo, por introducirme al mundo de la investigación que me ha abierto la mente a nuevas soluciones y me permite ver nuevos caminos por descubrir en el mundo de la Ingeniería del Software.

También debo agradecer a Pedro, que ha brindado con su conocimiento y su experiencia una gran riqueza a los casos de estudio de este trabajo. También debo agradecer a Alberto, por su paciencia, colaboración en el desarrollo de los experimentos y artículos presentados.

A todo el grupo de investigación, por su compañía, su amistad y su siempre bien recibimiento, que ha aligerado de gran manera el camino de desarrollo de la tesis.

A los amigos que he hecho en estos años, por creer en mi y apoyarme en mis proyectos aun sin conocerme del todo y su apoyo incondicional en los tiempos complicados tan lejos de mi hogar.

A mi familia, a quien he extrañado mucho, pero su cariño siempre me ha dado fortaleza que he necesitado.

A ti Fabián, por tu constante apoyo y tu inmenso cariño que me han ayudado a salir adelante.

Índice general

Índice de figuras

Índice de cuadros

Índice de códigos

Parte I

Preliminares

Capítulo 1

Introducción

La tecnología móvil ha surgido de la necesidad de las personas de llevar consigo un medio de comunicación con opciones de entretenimiento, una biblioteca y acceso a Internet. Actualmente, se ha masificado el uso de dispositivos móviles inteligentes (*smart*) que se utilizan como algo más que un mero medio de comunicación. Desde un dispositivo móvil *smart* se puede, por ejemplo, tomar fotografías, gestionar cuentas bancarias, localizar restaurantes y comercios cercanos a la ubicación geográfica del usuario, registrar las actividades físicas diarias, e incluso mostrar indicadores biométricos y de salud. En consecuencia, los dispositivos móviles modernos incrementan la capacidad del usuario para relacionarse con su entorno y con sus círculos sociales. La tecnología móvil está reemplazando el dinero en efectivo, los libros, las chequeras, los pasaportes, las llaves de acceso, los diarios, los teléfonos convencionales, los mapas, e incluso los maletines cargados de documentos.

Los dispositivos móviles *smart* poseen características de hardware muy diferentes de las de los PCs. La mayoría de ellos emplean procesadores basados en arquitectura ARM (*Advanced RISC Machine*), que está optimizada para utilizar un número reducido de instrucciones, un número menor de transistores, y un menor consumo energético. Por otra parte, los dispositivos móviles, en su mayoría, integran cámaras fotográficas y de vídeo, micrófonos, pantalla táctil, acelerómetros, GPS (*Global Positioning System*), y dispositivos de comunicación por radiofrecuencia (Bluetooth, WIFI, GSM, entre otros). Finalmente, es importante notar que los móviles contienen una gran cantidad de recursos hardware encapsulados en una estructura lo suficientemente pequeña para caber en la palma de la mano; por lo tanto, estos recursos deben trabajar de manera altamente coordinada, ya que la alta densidad componentes aumenta la predisposición a fallos en el dispositivo.

Las diferencias inherentes en el hardware, además de las crecientes expectativas de mejores y más avanzadas experiencias de interacción entre los dispositivos móviles *smart* y los usuarios, han dado lugar a sistemas operativos especialmente desarrollados para estos entornos, tales como Android, IOS, Windows Phone, Tizen, WebOS y Ubuntu Phone, entre otros. Estos sistemas operativos

se enfocan en optimizar el consumo energético, gestionar una limitada cantidad de memoria RAM, y en orquestar múltiples sensores y fuentes de información de manera simultánea y en tiempo real. Por otra parte, la interfaz de usuario de los sistemas operativos móviles presenta marcadas diferencias con respecto a sus contrapartidas en los PCs, principalmente, en cuanto a la utilización de una interfaz táctil como medio de entrada, al tamaño reducido de la pantalla, a la alta reactividad al usuario, y al uso constante de datos multimedia. En este contexto, cabe destacar que los fabricantes de dispositivos móviles normalmente personalizan los sistemas operativos para mejorar la compatibilidad con el hardware específico. Tal es el caso de ANDROID, cuyas versiones base se modifican significativamente para diferentes marcas y modelos de dispositivos.

Por todo lo anterior, no es difícil concluir que los dispositivos móviles *smart* poseen una serie de características que los diferencian de la tecnología de ordenadores convencionales, además de cumplir los principios de trasportabilidad, conectividad, adaptabilidad a las necesidades personales del usuario, y finalmente, integración con las redes sociales. Sin embargo, estas diferencias introducen nuevos retos para el desarrollo de aplicaciones móviles.

Las aplicaciones móviles (APPS) están en contacto directo con el usuario y están dirigidas por él. Se diseñan con propósitos puntuales como podrían ser, entre otros, comunicación, localización geográfica o diversión. Las aplicaciones coexisten unas con otras de manera que, si una necesita ejecutar una acción para la que no fue desarrollada, puede solicitar que la realice otra aplicación y, de esta manera, el usuario personaliza su dispositivo combinando las aplicaciones de su elección. Por ejemplo, una APP desarrollada para almacenar archivos en la nube puede requerir que el usuario seleccione la APP de reproducción instalada en su dispositivo, que debe abrir los archivos multimedia.

Por otro lado, las APP utilizan múltiples recursos de los dispositivos móviles *smart*, es decir, que entre otras cosas, consideran en su diseño factores de conectividad, de almacenamiento y de consumo energético. En este contexto, cuando hay cambios físicos en el dispositivo, el hardware genera notificaciones que son gestionadas por el sistema operativo. Si las APPS han sido programadas para reaccionar a las notificaciones ejecutan acciones cuando ocurren. Por ejemplo, si el hardware del dispositivo envía una señal de sobrecalentamiento de la batería, el sistema operativo libera recursos y propaga la alerta a las aplicaciones. Si las APPS no gestionan adecuadamente esta alerta y no liberan recursos, el dispositivo puede llegar a tener un daño físico serio. Por este motivo, las APPS están fuertemente integradas con sistema operativo y con el hardware del dispositivo.

Todo lo anterior describe un escenario donde el funcionamiento de una APP depende altamente de su coordinación con el sistema operativo y el hardware. En este sentido, los recursos de hardware que contiene un dispositivo dependen del criterio del fabricante. Además, las APPS se desarrollan para una versión de sistema operativo que, a su vez, es distinta para cada modelo de dispositivo. Esto puede provocar que una APP que no muestra comportamientos erróneos en un dispositivo los muestre en otro de un modelo de una marca distinta. Esto sugiere que para el análisis real de una APP es necesario hacer pruebas con la aplicación instalada en diversos modelos de dispositivos.

Adicionalmente, las características de gestión de recursos hardware y software, explicadas anteriormente, provocan que todo programa que se instala en un dispositivo tenga una influencia directa en su ejecución. En este sentido, el diseño de los programas que tienen el propósito de analizar las Apps debe tener en cuenta que cada pieza de código que se ejecuta en el dispositivo modifica su funcionamiento. Un programa de análisis puede alterar la coordinación del sistema operativo mermando el rendimiento, o provocar fallos en el hardware tales como el sobrecalentamiento y la reducción de vida de la batería. Por lo tanto, es recomendable utilizar dispositivos externos de alta potencia que, mediante una conexión con los móviles, recojan información necesaria para evaluar el funcionamiento de las Apps. Por ejemplo, se puede utilizar un voltímetro para medir de manera precisa la energía que utiliza una Apps cuando reproduce una película, o se puede utilizar un programa personalizado en un ordenador para evaluar si la traza de red que produce una App no tiene un comportamiento anómalo que indique un fallo de seguridad.

Dada la naturaleza altamente interactiva de las aplicaciones, descrita anteriormente, los usuarios habitualmente utilizan a su discreción todas las funcionalidades proporcionadas por los dispositivos móviles sin enfocarse en solo una App. A diferencia de la lógica de interacción en un PC, los usuarios hacen uso de una o más Apps para completar una tarea. Por ejemplo, una persona podría tomar una fotografía, editarla y, a continuación, enviarla a través del correo electrónico. Como puede observarse en este pequeño ejemplo, estas tareas consisten en secuencias de acciones que sigue un usuario. Cada secuencia es única y puede tener un efecto distinto en el dispositivo. Por ejemplo, algunas secuencias pueden provocar fallos inesperados como, por ejemplo, excepciones no controladas de las Apps, sobrecalentamiento del dispositivo, etc. Por este motivo, es importante emular el comportamiento del usuario para el análisis completo de las Apps.

En resumen, el escenario de verificación de las Apps es muy distinto y más exigente que el de las aplicaciones que se ejecutan en PC. En primer lugar, el funcionamiento de las Apps puede variar entre dispositivos, esto significa que las Apps se deben analizar físicamente en varios modelos de dispositivos. En segundo lugar, cualquier programa de análisis puede corromper el funcionamiento normal de un dispositivo, por lo que una buena alternativa para analizar las Apps sin comprometer los resultados es utilizar dispositivos externos. Finalmente, las secuencias de acciones que sigue el usuario utilizando las Apps pueden dar lugar a fallos que no son identificados con facilidad en tiempo de desarrollo, por este motivo el comportamiento del usuario debe formar parte del análisis de las Apps.

En este contexto, se observa que las múltiples interacciones hardware/software se comportan como un sistema concurrente en el sentido clásico, es decir, las interacciones provocadas por la utilización de una App forman parte de una serie de hebras independientes que se ejecutan de manera paralela y no determinista en el dispositivo. Por este motivo, las propiedades deseables de las Apps, como sistemas concurrentes, pueden analizarse utilizando *patrones del comportamiento* esperado sobre las trazas de ejecución. Estos patrones o propiedades se

pueden escribir utilizando diferentes lenguajes de especificación utilizando lógica temporal (LTL o CTL[1], por ejemplo), máquinas de estados, etc. Las propiedades se suelen agrupar en dos tipos principales: funcionales y extra-funcionales. Las propiedades funcionales describen requisitos que se refieren habitualmente al comportamiento correcto de las Apps con respecto al uso para el que han sido creadas, como son las propiedades de seguridad y viveza de los sistemas concurrentes. Por ejemplo, si un aplicación fue diseñada para calcular el promedio de dinero gastado en el mes pero, sin embargo, muestra el mínimo gastado, la aplicación violaría una típica propiedad funcional de seguridad. Por otro lado, debido a las evidentes limitaciones físicas de las plataformas móviles, hay otras propiedades, denominadas extra-funcionales, que son también esenciales para garantizar su funcionamiento correcto. Las propiedades extra-funcionales se refieren aspectos más físicos de comportamiento del dispositivo, como pueden ser, entre otros, el aceptable consumo de energía o que los perfiles de tráfico de red estén dentro de unos parámetros razonables.

La importancia del problema radica en que la alta coordinación que requieren estas plataformas las predispone a fallos, por lo tanto, las Apps no son unidades aisladas de su entorno físico y lógico. De esta manera, para analizar éste tipo de aplicaciones, es importante emular los posibles comportamientos de los usuarios utilizando una o varias Apps de manera simultanea sobre uno o varios dispositivos. Ademas, es igual de importante que el analizador modifique en lo mas mínimo el funcionamiento normal del dispositivo para que los resultados no se vean afectados.

Se han utilizado varias técnicas para analizar el comportamiento de las Apps que se describen a continuación. Por ejemplo, el *Model checking* [1, 2, 3] ha sido satisfactoriamente aplicado tanto al análisis de software como de hardware. Las herramientas de *Model checking* pueden analizar todo el espacio de estados de un sistema, es decir, todos sus posibles comportamientos para comprobar si una propiedad se satisface. Si se encuentra un error, dichas herramientas proporcionan una traza que muestra el camino desde el principio del sistema (el estado inicial) hasta el punto donde el error fue encontrado. De esta manera, los programadores pueden depurar el error y corregir el sistema. El problema principal de analizar una App con esta técnica es que es posible que el *Model checker* no tenga recursos suficientes para gestionar la explosión de estados producida por el dispositivo móvil. El tamaño de los estados a ser explorados crece en función de la complejidad de los sistemas. Por otro lado, verificar las Apps en un emulador significa replicar toda la arquitectura móvil descrita anteriormente para obtener un escenario real, convirtiendo esta tarea en compleja y poco factible.

Otra de las técnicas más utilizadas para analizar las aplicaciones durante el desarrollo es el *Testing* [4, 5]. Las técnicas de *Testing* pueden ser clasificadas como *white-box* o *black-box*. Para el *Testing white-box* se asume que se conoce el código del sistema, y que se tiene acceso a sus componentes. Esta técnica es útil para encontrar fallos, pero como no hace una exploración exhaustiva, no puede garantizar que el software es correcto respecto a una propiedad dada. Sin embargo, la técnica es una de las más frecuentemente utilizadas en el mercado. El *Testing* se puede hacer a varios niveles. Por ejemplo, los *Unit Tests* analizan

de forma aislada pequeñas partes de un sistema como clases o funciones. A un nivel más alto, se puede analizar la integración entre componentes, o el sistema como un todo.

Por otro lado, *Model Based Testing* [6, 7, 8] es una técnica de *Testing* [5, 4], que consiste en la generación automática de casos de prueba utilizando modelos que representan los requisitos y el comportamiento de los sistemas. El modelo es, en general, construido de forma manual lo que requiere conocer detalles de los sistemas. Para aplicar esta técnica se necesita mucho esfuerzo en el diseño del modelo pero, en contraposición, los tests generados a partir de un modelo cubren más funcionalidades que los obtenidos de forma aleatoria. Una vez generados los casos de *tests* pueden ser ejecutados en el dispositivo. Posteriormente, se analizarán las trazas de ejecución del dispositivo en busca de fallos.

Finalmente, la técnica denominada *Runtime verification* [9, 10, 11] permite el análisis de sistemas extrayendo información durante su ejecución para observar su comportamiento y determinar si las ejecuciones satisfacen o violan propiedades que describen los requisitos del programa. Las propiedades se especifican mediante formalismos como máquinas de estados finitas, expresiones regulares o lógica temporal, entre otros. Algunos ejemplos de propiedades son aquéllas que representan si existe bloqueo (*deadlock*) o condiciones de carrera (*dataraces*) en la ejecución. La técnica de verificación *Runtime* se utiliza mucho para el análisis de Apps. La mayoría de las propuestas son sólo capaces de analizar propiedades de forma independiente. Por ejemplo, pueden verificar si existen *back-doors* de seguridad, pueden medir el consumo de energía, o analizar los perfiles de tráfico de red.

En esta tesis, se presenta una propuesta de análisis de propiedades extra-funcionales sobre aplicaciones móviles ejecutadas en uno o varios dispositivos que utiliza de forma conjunta los métodos formales mencionados: *Model Checking* [1], *Model Based Testing* [6] y *Runtime Verification* [9]. Como resultado, la propuesta combina la generación exhaustiva de casos de prueba dirigida por modelos y monitores que leen información de las Apps, en tiempo de ejecución, de los sensores del dispositivo, del sistema operativo y de monitores externos de red, así como de energía. Esta información heterogénea se combina para obtener una sola traza enriquecida que permitirá el análisis de propiedades durante la ejecución. La propuesta implica simular la interacción real de un usuario con un dispositivo, y evaluar su impacto sobre la ejecución a través del análisis de propiedades sobre las trazas de ejecución. Los modelos se pueden ajustar para optimizar la búsqueda de errores. De igual manera, se propone la utilización de una notación para expresar propiedades sobre la traza enriquecida utilizando lógica de intervalos.

Capítulo 2

Sinopsis

Como se ha comentado en la Introdución, en esta tesis, se presenta un marco de trabajo integrado para el análisis dinámico de propiedades extra-funcionales de Apps que se ejecutan sobre dispositivos móviles *smart*. La propuesta se divide en dos fases que se describen en la Figura 2.1: (a) generación y ejecución de casos de prueba, y (b) verificación en tiempo de ejecución. La primera etapa es responsable de dirigir la creación y ejecución de casos de prueba obtenidos de los modelos de comportamiento de los usuarios, es decir, construidos a partir de sus posibles interacciones con una App (Apps) en uno o varios dispositivos.

Con el fin de completar la primera etapa, se deben llevar a cabo tres tareas: (a1) modelado del comportamiento de usuario, (a2) generación de secuencias de casos de test utilizando el model checker Spin [12] y (a3) ejecución de los casos de prueba. Durante la tarea de modelado del comportamiento, el desarrollador/-tester proporciona un modelo de la App o Apps a ser analizadas, incluyendo las propiedades que deben ser verificadas posteriormente. El modelado del comportamiento es crucial en nuestra propuesta debido a que está orientado a emular el comportamiento real del usuario en lugar de generar acciones de manera aleatoria. Después, en la tarea a2, el modelo del comportamiento de usuario se traduce al lenguaje *PROMELA* para ser interpretado por la herramienta Spin. Spin es una de las herramientas que emplean la técnica de *model checking* y, en consecuencia, puede generar todas las posibles trazas de ejecución de un modelo de software. Cada traza de ejecución generada por Spin constituye un caso de test diferente que luego será ejecutado en el dispositivo durante la tarea a3.

Por otro lado, la segunda etapa es responsable de la verificación de las propiedades extra-funcionales para el análisis del comportamiento del dispositivo, esto ocurre al mismo tiempo que cada caso de prueba se ejecuta. De esta manera, se ha aplicado un tipo de monitor del tipo *External Online*, como se describe en la Sección 6.5. La segunda etapa esta compuesta de dos tareas: (b1) composición de las trazas y (b2) la verificación de propiedades extra-funcionales. Las trazas pueden observarse en los dispositivos y monitores utilizando la apropiada instrumentación, con lo que es posible saber qué acciones son ejecutadas por el usuario, cuánta energía es consumida por el dispositivo, o cuántos paquetes

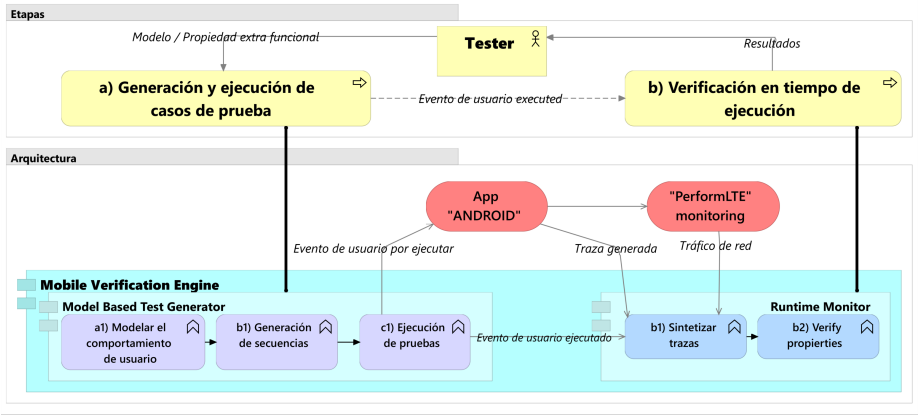


Figura 2.1: Resumen de la solución

son enviados/recibidos a través la red. Toda esta información se registra en una única traza enriquecida que puede ser analizada para comprobar si se satisfacen las propiedades extra-funcionales proporcionadas en la primera fase. Por ejemplo, se puede verificar si el tráfico de red de una aplicación sigue un determinado patrón cuando se ejecutan ciertas acciones. Finalmente, las propiedades extra-funcionales son verificadas al mismo tiempo que la aplicación se ejecuta. Esto significa que los resultados obtenidos tras analizar estas propiedades pueden ser conocidos tan pronto como se evalúan durante el proceso de ejecución de pruebas.

2.1. Contribuciones

La principales contribuciones de esta tesis son las siguientes:

1. *Un método dirigido por modelos para generar casos de pruebas* para aplicaciones que respondan a interacciones del usuario, eventos internos del dispositivo y de la red. Las interacciones internas y externas pueden ser expresadas en un modelo formal a partir de información proporcionada por el diseñador de una APPS y/o por el *Tester*. El modelo formal es explorado utilizando una herramienta de *Model checking* para generar secuencias reales de interacciones con las APPS ejecutándose en un dispositivo.

2. *Un lenguaje formal para describir propiedades extra-funcionales* que especifican el comportamiento esperado de las APPS en un intervalo de tiempo. El lenguaje definido se basa en la lógica de intervalos [13] que se ha extendido para representar los patrones deseados de energía, tráfico de red, memoria, etc. Estos patrones básicos pueden ser combinados para describir comportamientos complejos.

3. *Una arquitectura de verificación* que integra de la generación de casos de pruebas, la ejecución de las APPS en el dispositivo, el control de equipos adicionales, que permiten medir valores adicionales como la energía consumida y la verificación en tiempo de ejecución. Para la generación de los casos de prueba se combinan las técnicas de *Model Based Testing* y *Model Checking*. Esta integración permite que el proceso de verificación analizar todos los casos de prueba posibles en un modelo de una APP de manera automática.

4. *La formalización de toda la propuesta* que proporciona una semántica operacional formal tanto para la generación de los casos de prueba como para la verificación de las propiedades extra-funcionales. Dichas semánticas ofrecen contribuciones como la interacción entre varias *Apps*, y la interacción con eventos internos.

5. *Una implementación para el análisis de Apps sobre dispositivos* ANDROID que simula los comportamientos de usuario sobre un dispositivo y evalúa propiedades sobre las trazas de ejecución.

6. *La evaluación de la propuesta sobre aplicaciones reales* como Spotify®, Facebook®, YouTube®, o Yahoo Mail® para analizar propiedades de consumo de energía y comportamiento del tráfico de red.

Las distintas contribuciones se han publicado en los artículos [14] y [15], donde se presentaron las propuestas para la automatización de la generación de las pruebas y el método de análisis de propiedades de energía, respectivamente. El método para generar los casos de prueba fue inicialmente presentado en [14] considerando máquinas de estados de UML como lenguaje de modelado para interacciones de usuarios, mientras que en éste trabajo final se define una variante de la máquina de estados para ser implementada en la herramienta. El lenguaje para definir propiedades relacionadas a la energía fue introducido en [15], mientras que en este trabajo final se extiende para poder expresar otro tipo de propiedades. En la misma línea, toda la arquitectura de las herramientas incluyen la captura de información que al ser combinada ofrece posibilidades de verificación de las propiedades extra-funcionales. La implementación descrita de este trabajo muestra todas las mejoras conseguidas a través de los diferentes artículos. Las publicaciones muestran cómo se han utilizado APPS conocidas en el mercado para las pruebas para demostrar sus capacidades, como Facebook®, Spotify®, YouTube® y Yahoo Mail®.

2.2. Organización

Esta tesis se organiza de la siguiente manera. El capítulo II describe los fundamentos de la propuesta explicando las técnicas utilizadas y, la arquitectura de las aplicaciones móviles ANDROID. El capítulo III presenta el diseño de la propuesta y su implementación. El capítulo IV describe los diferentes experimentos

realizados, junto con sus resultados, que prueban la utilidad de nuestra propuesta. Finalmente, el capítulo V describe los trabajos relacionados, las conclusiones y posibles trabajos futuros.

2.3. Colaboración de proyectos

Este trabajo ha sido llevado a cabo gracias a la colaboración parcial de los proyectos P11-TIC-7659 (Gobierno regional de Andalucía), TIN2012-35669 (Ministerio Español de Economía), UMA-Keysight Technologies 8.06/5.47.3868-4Green y AUIP (Asociación Universitaria Iberoamericana de Postgrado) como patrocinador de Beca Académica.

Parte II

Antecedentes

Capítulo 3

ANDROID

ANDROID [16, 17] está en la vanguardia de los sistemas operativos para dispositivos móviles. Es utilizado por cientos de millones de dispositivos, y en más de 190 ciudades, alrededor del mundo. Cada día más de un millón de dispositivos se activan en el mundo. Esta plataforma es una de las que ha impulsado innovaciones de hardware y software para brindar nuevas prestaciones al público como, por ejemplo, el procesamiento *multicore*, *multimedia* de alto rendimiento, los de sensores más recientes del mercado, o las pantallas vibrantes brindando a los programadores muchas opciones para la creación de aplicaciones, y a los usuarios nuevas experiencias para relacionarse su móvil.

El sistema operativo ANDROID puede ser implementado en dispositivos móviles *Smart* de distinta índole como, por ejemplo, tabletas, teléfonos, relojes y otros similares. Además, la plataforma está diseñada para ser utilizada por múltiples modelos de distintas marcas. En consecuencia, las *Apps* construidas en ANDROID siguen un solo modelo de diseño, desarrollo e implementación. Un ejemplo claro es el funcionamiento de la interfaz de usuario que adapta el diseño visual de las *Apps* para cada dispositivo. Por ejemplo, una *App* que funciona tanto en un teléfono como en tabletas tiene, en la definición recursos de interfaz de usuario, unos archivos de configuración, algunos de los cuales son comunes para ambos tipos de dispositivos, mientras que otros son específicos para el teléfono o para la tableta. En tiempo de ejecución, ANDROID aplicará el recurso correcto, utilizando las variables de tamaño de pantalla, densidad, etc.

3.1. Arquitectura de ANDROID

En esta sección, se describe, la arquitectura que soporta la plataforma de ANDROID, conocida como ANDROID SDK (*Service Development Kit*). Esta introducción a la arquitectura es necesaria, para entender la complejidad del entorno de ejecución de las APPS de ANDROID que ha servido como un punto de partida para la propuesta. En la figura 3.1, se puede ver la estructura en capas de la plataforma [16, 17]. Cada capa utiliza los servicios proporcionados

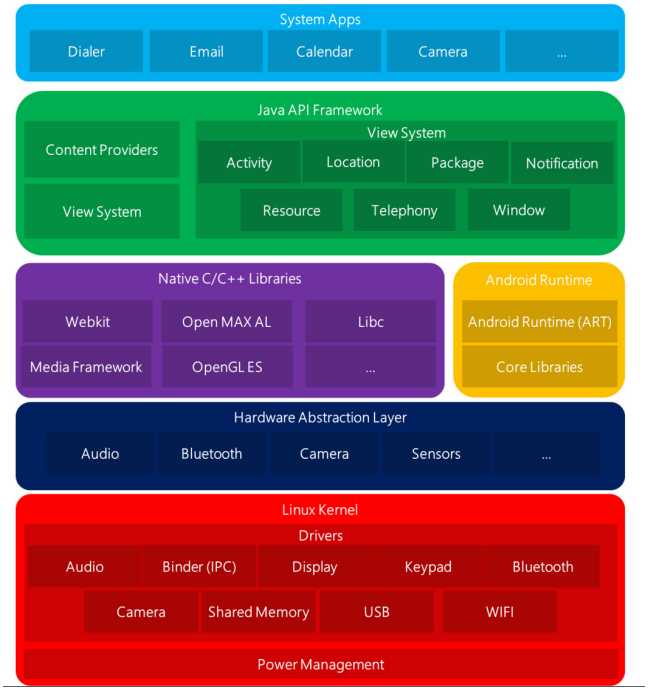


Figura 3.1: Arquitectura de ANDROID

por la capa inferior. Todas ellas se han diseñado para ser ejecutadas de manera coordinada con el objetivo de optimizar el uso de la memoria y del consumo de energía. En los siguientes párrafos, se describen cada una de las capas.

3.1.1. LINUX *Kernel*

ANDROID está construida sobre una versión comprimida del *kernel* de LINUX. En este sentido, el sistema operativo de LINUX para ANDROID encapsula los recursos de hardware para que puedan ser accesibles por las capas superiores y, de esta manera, gestionar su consumo para optimizar su ejecución.

Internamente esta capa se utiliza para la gestión de la memoria, la gestión del procesamiento, networking, y otros servicios del sistema operativo. El usuario del teléfono *smart* de ANDROID nunca ve la interfaz LINUX y las *Apps* no están diseñadas para hacer llamadas a LINUX directamente, es más, es muy posible que los programadores de las *Apps* no necesiten interactuar directamente con este sistema operativo.

3.1.2. Abstracción del *Hardware HAL*

En esta capa se encuentra el código que permite a ANDROID manipular directamente el hardware de los dispositivos. ANDROID ofrece una serie de pro-

gramas base para la gestión del hardware común a todos los dispositivos, como ser, batería, giroscopio, GPS, entre otros. Luego, los fabricantes manipulan estos programas y/o crean otros distintos para adaptarlos un dispositivo en concreto. El compendio de los programas agregados/modificados se agrega en una versión específica de ANDROID. Esta abstracción permite que el software del dispositivo se ejecute en varios tipos de dispositivos sin percatarse de las diferencias de hardware.

3.1.3. Librerías nativas C/C++(*Native C/C++ Libraries*)

La siguiente capa sobre el *kernel* de LINUX contiene las librerías nativas de ANDROID. Estas librerías están escritas en C o C++ y se compilan para una arquitectura hardware particular. Están preinstaladas de fábrica en cada dispositivo móvil *smart*. Estas librerías no están diseñadas para funcionar como aplicaciones independientes sino que, al contrario, están pensadas para trabajar de manera colaborativa con las aplicaciones de las capas superiores. De esta forma, encapsulan los recursos de hardware específicos de cada modelo de dispositivo, y permiten que las funcionalidades de las capas superiores de la plataforma no se vean alteradas por las diferencias hardware entre dispositivos distintos. Algunas librerías [18] son *Surface Manager*, *2D and 3D graphics*, *Media codecs*, *SQL database*, o *Browser engine*.

3.1.4. ANDROID *Runtime* (*ART*)

ANDROID Runtime (*ART*) es un entorno de ejecución de aplicaciones, encargado de realizar la traducción del código compilado de una aplicación a instrucciones nativas, que luego son ejecutadas en el dispositivo. Hasta la versión de ANDROID 4.4 *KitKat*, se utilizaba una maquina virtual llamada *Dalvik* que, a diferencia de *ART*, introduce el uso de la compilación anticipada (*AOT*), compilando aplicaciones enteras en código nativo de la máquina después de su instalación. Al eliminar la interpretación de *Dalvik*, *ART* mejora la eficiencia de ejecución general y el consumo de energía, lo que da como resultado, una autonomía de batería mejorada en dispositivos móviles. Al mismo tiempo, *ART* ofrece una ejecución más rápida de las aplicaciones, una mejor asignación de memoria, mecanismos de *garbage collector (GC)*, así como nuevas funciones de depuración de aplicaciones y un perfil de aplicaciones de alto nivel más preciso. Por otro lado, *ART* requiere tiempo adicional para completar la compilación cuando se instala una aplicación en el dispositivo; adicionalmente, las aplicaciones ocupan más espacio de almacenamiento requerido por el código compilado.

Para mantener la compatibilidad con versiones anteriores, *ART* utiliza internamente el núcleo de *Dalvik* y elimina el consumo de recursos por la interpretación de *Dalvik* para las *Apps* desarrolladas en versiones antiguas de ANDROID.

3.1.5. Plataforma JAVA (JAVA *API Framework*)

Esta capa contiene el JAVA *API Framework* que contiene librerías de desarrollo escritas en el lenguaje de programación JAVA. Estas librerías encapsulan las funcionalidades que proporcionan tanto *ART* como las *Native C/C++ Libraries* para que puedan ser accesibles por las *Apps*.

Las aplicaciones que se que se construyen con el *Framework* consisten en múltiples componentes: *Activities*, *Services*, y *Content Providers*. Los componentes pueden interactuar con otros componentes de la misma *App* o de una *App* diferente a través de llamadas denominadas *Intents*. El *Framework* utiliza *Managers* para gestionar la ejecución de cada uno de los componentes y su ciclo de vida cuando se ejecuta una aplicación.

3.1.6. Aplicaciones de sistema (*System Apps*)

Las *System Apps* de sistema son aplicaciones propias de ANDROID que se aplican a cualquier dispositivo y forman parte de la instalación del sistema operativo por defecto. Estas aplicaciones pueden ser llamadas desde otras *Apps* personalizadas.

3.2. Componentes de una *App* ANDROID

ANDROID dispone de cuatro tipos de componentes que conforman sus aplicaciones: *Activity*, *Service*, *BroadcastReceiver* y *ContentProvider*. A continuación, se describen cada uno de ellos.

Un *Activity* representa una pantalla de una aplicación como, por ejemplo, una ventana de búsqueda o de entrada de correos electrónicos recibidos. Las aplicaciones están formadas por múltiples *Activities* que pueden ser llamados indistintamente entre sí aunque no pertenezcan a la misma *App*. A cada *Activity* le corresponde un ciclo de vida que permite a la plataforma decidir qué recursos son prescindibles cuando se requiera liberar memoria. El ciclo de vida de un *Activity* puede pasar por cinco estados distintos:

- *Created*: Estado inicial de un *Activity* cuando es llamado por primera vez.
- *Started*: El *Activity* pasa a este estado cuando se muestra al usuario en pantalla.
- *Resumed*: Cambia a este estado y permanece en él mientras el usuario interactúe con el *Activity*.
- *Paused*: El *Activity* pasa a este estado cuando pasa a ser ejecutado en segundo plano después de estar visible. Por ejemplo, cuando se abre una segunda ventana modal [1] que cubre parcialmente la primera, o cuando

[1] La modalidad de una ventana se refiere a cómo mantiene ésta el foco respecto a las demás ventanas del sistema. Una ventana modal permite alternar el foco a otras ventanas del sistema, pero no a la ventana que le da origen (*ventana madre*) hasta que se realiza una acción sobre ella.

el usuario ha dejado de interactuar con el dispositivo mientras el *Activity* estaba visible.

- *Stopped*: Este estado indica que el *Activity* se ha detenido por estar invisible u oculto para el usuario, pero el estado del componente se mantiene, por ejemplo, cuando se atiende una llamada telefónica mientras se revisa el correo.

- *Destroyed*: Este estado indica que el *Activity* ha destruido la actividad para liberar recursos. Esto ocurre cuando el *Activity* ha estado detenido por mucho tiempo, o cuando la aplicación se ha cerrado.

Los *Services* y *BroadcastReceivers* son componentes que permiten a las aplicaciones ejecutar tareas (*jobs*) en segundo plano. Un *Service* se mantiene vivo siempre en el dispositivo y se puede utilizar, entre otras cosas, para actualizar la posición del GPS o la bandeja de mensajes recibidos. Por otro lado, un *BroadcastReceiver* se lanza a través de eventos disparados y solo se ejecuta durante un corto periodo de tiempo influyendo en el comportamiento de otros componentes, por ejemplo, si una llamada telefónica llega al dispositivo, el evento puede ser detectado por un *BroadcastReceiver* que puede hacer que una aplicación cambie el tono de llamada, dependiendo del número de teléfono que llama.

Finalmente, un *ContentProvider* representa al gestor de almacenamiento de datos en las *Apps* de ANDROID. Estos proveedores pueden ser utilizados para compartir datos entre múltiples aplicaciones (siempre y cuando tengan permisos suficientes). ANDROID tiene *ContentProviders* por defecto que le permiten, almacenar, por ejemplo, vídeos, imágenes, contactos, configuraciones, etc.

Cada *App* no tiene que estar compuesta, necesariamente, por los cuatro componentes pero, al menos, contener un *Activity* o un *Service*, dependiendo de su propósito.

3.3. Anatomía de una *App* ANDROID

ANDROID proporciona, para cada una de las capas de su arquitectura, una serie de escenarios de ejecución, composición y gestión de aplicaciones. Con este propósito, ANDROID diferencia claramente los conceptos de aplicación, proceso, tarea y hebra. En esta sección, se describe cada concepto así como su relación con los demás.

3.3.1. Procesos y hebras

Se distinguen cinco tipos de procesos en ANDROID para poder controlar el comportamiento del sistema, además de la ejecución de las aplicaciones.

Cada proceso tiene un nivel de importancia y un orden estricto que da lugar a una jerarquía. A continuación, se describe cada nivel:

- ***Foreground***: Este nivel contiene los procesos que se están ejecutando. Pueden ser un *Activity*, un servicio proporcionado por el *Activity*, el inicio o fin de un *Service* o, finalmente, la recepción de un evento de un *BroadcastReciever*.
- ***Visible***: En este nivel se encuentran los procesos que responden a componentes visibles pero sin ejecución.
- ***Service***: Estos procesos ejecutan los *Services* iniciados.
- ***Background***: Representa a los procesos que ejecutan *Services* y/o *Activities* en segundo plano.
- ***Empty***: Estos procesos no contienen componentes de aplicaciones activas, su propósito final es tener disponible un cache.

Cuando se necesita liberar memoria en ANDROID, los procesos *Empty* son los más susceptibles a ser eliminados, seguidos por los procesos *Background*, de manera que la experiencia del usuario no se vea afectada.

3.3.2. Tareas y pila de *Activities*

Una tarea es una colección de *Activities* con las que el usuario final ha interactuado para realizar una acción. Los *Activities* se organizan en una pila en el orden en el que se abren.

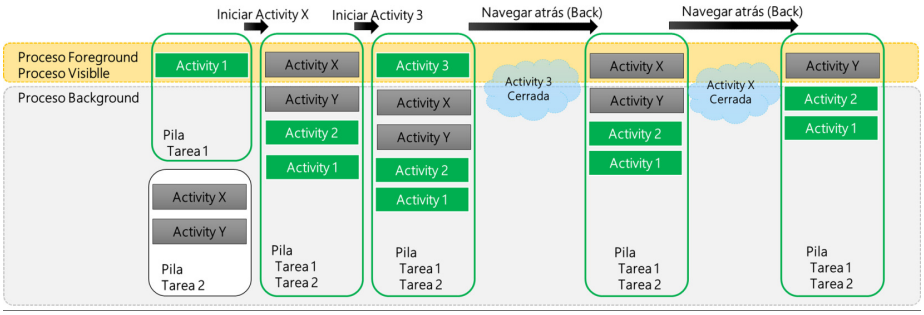


Figura 3.2: Pila de *Activities*

Cuando se inicia una *App*, la tarea de esa aplicación pasa a primer plano. Si no existe una tarea para la aplicación (la aplicación se abrió anteriormente y no se cerró), se crea una nueva tarea y el *Activity* inicial es insertando en la pila. Por ejemplo, cuando un *Activity* 1 llama a otro *Activity 2*, el *Activity* 2 es agregado a la parte superior de la pila y el *Activity* 1 se detiene (estado *stoped*) pero continúa en la pila. Cuando el usuario presiona el botón atrás, el *Activity* actual se cierra (pasa al estado *Destroyed*) y reanuda el siguiente *Activity* en la pila. Los *Activities* de la pila nunca se reorganizan, solo se insertan o se cierran.

La Figura 3.2 muestra la construcción de una pila de *Activities* es insertada. Por ejemplo, si se llama a un *Activity X* que pertenece a la tarea 2, toda la pila de *Activities* de la tarea 2 se inserta en la parte superior de la pila de la tarea 1. Las pilas de *Activities* de ambas tareas se combinan pero no dejan de ser 2 tareas distintas y los *Activities* nunca pierden la referencia a su tarea.

Capítulo 4

Model based testing

4.1. Testing

Según la asociación *IEEE Software Enginering Body of Knowledge* [19, 8] *Testing* es una actividad utilizada para evaluar la calidad de un producto y mejorarlo identificando sus defectos y problemas. El *Testing* consiste en la verificación *dinámica* del comportamiento de un programa utilizando un conjunto *finito* de casos de prueba, *seleccionados* a partir del conjunto de todas las posibles ejecuciones que podría ser infinito, contra un comportamiento *esperado*. En esta definición, las palabras *dinámica*, *finito*, *seleccionados* y *esperado* son conceptos claves en la técnica de *testing* [5][19].

- *Dinámico*: Significa que el análisis se realiza ejecutando un programa con unos valores de entrada específicos, para encontrar errores en su comportamiento. El programa puede ser ejecutado tanto en un entorno simulado como real. De esta última manera no solo se prueba el código de la aplicación sino también cómo reacciona el entorno en el que es ejecutado.

- *Finito*: Se establece un conjunto finito de pruebas. La cantidad de pruebas debe ser suficiente para analizar los diferentes aspectos de una aplicación y asegurar su calidad.

- *Seleccionado*: Cada aplicación tiene, en principio, una enorme o infinita cantidad de casos posibles, por lo que es necesario hacer una selección en función de ciertos requisitos. Esta tarea depende mucho del conocimiento de la aplicación. Un experto en la aplicación (desarrollador/Tester) debe determinar qué tipos de pruebas son relevantes para verificar la calidad de la aplicación.

- *Esperado*: Después de la ejecución de cada prueba se debe decidir si el comportamiento observado de la aplicación presenta errores, utilizando lo que se denomina *oráculo* del problema.

Un fallo (*Failure*) es un comportamiento no deseado de la aplicaciones que ocurre durante su ejecución. Un defecto (*Fault*) es la causa del error y puede deberse a una incorrecta especificación de los requisitos, un error en el diseño, o en la programación. Una vez que se han encontrado los fallos se pueden determinar los defectos para ser corregidos.

El *Testing* se distingue de las técnicas de verificación estática que no necesitan ejecutar las aplicaciones para analizarlas. También es distinto a la depuración de código (*Debug*) que se utiliza cuando el *Testing* ya ha detectado el fallo.

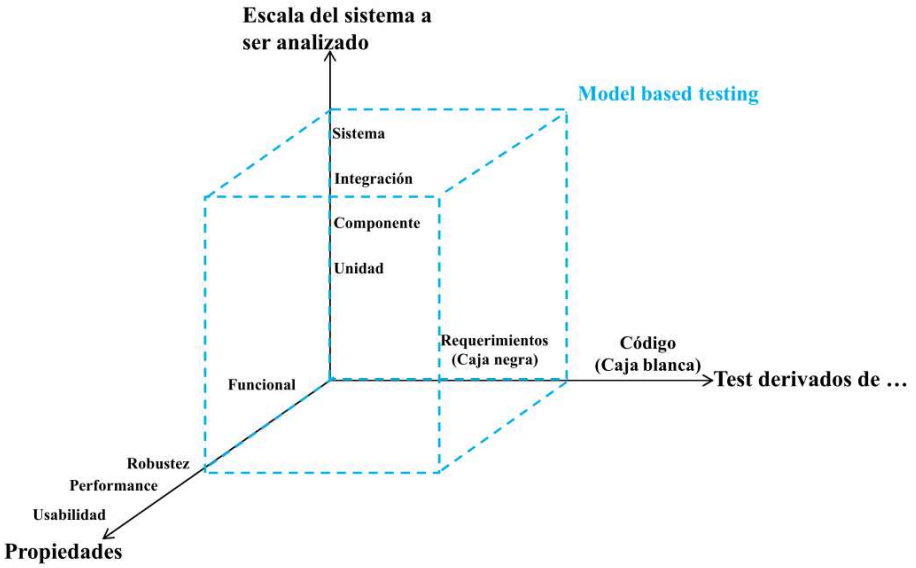


Figura 4.1: Diferentes tipos de *Testing*

Existen varios tipos de *Testing* como se muestra en la Figura 4.1 (extraído de [6]). Un eje muestra la escala del sistema (aplicación) a ser analizado desde las piezas más pequeñas hasta el sistema completo. Las pruebas de *unidad* (*Unit Test*) analizan de forma aislada pequeñas partes de un sistema como clases o funciones. Las pruebas de *componente* (*Component testing*) analizan cada componente de manera separada. Las pruebas de *integración* (*Integration Testing*) se aseguran de que la coordinación entre componentes no tenga fallos. Las pruebas de sistema (*System testing*) tienen como objetivo analizar el sistema como un todo. La técnica de *Model Based Testing* puede ser aplicada a todos estos niveles [8].

Otro eje representa las diferentes características que se pueden analizar de una aplicación. Por ejemplo, las pruebas *funcionales* (*Testing Functional* o *Behavioural testing*) que buscan encontrar fallos de funcionalidad del sistema, es decir, si el sistema ha sido construido de acuerdo a sus especificaciones funcionales.

Asimismo, se pueden ver los tipos de *Testing* que permiten analizar propiedades extra-funcionales como son: las pruebas de *robustez* (*Robustness testing*) permiten encontrar fallos ejecutando la aplicación en un entorno para el que no fue desarrollada o con entradas de datos erróneas, las pruebas de *rendimiento* (*Performance testing*) que intentan encontrar fallos utilizando una carga muy alta de datos de entrada, las pruebas de *usabilidad* (*Usability testing*) que tienen como objetivo encontrar problemas en el interfaz de usuario que hacen a la aplicación difícil de utilizar o entender.

El tercer eje muestra la información que se usa para diseñar las pruebas. Pueden ser tests derivados de pruebas de *caja negra* (*Black-box testing*) que analizan las aplicaciones utilizando solamente las entradas y las salidas que se generan. Alternativamente, las pruebas caja blanca (*White-box testing*) utiliza la implementación y la estructura interna de la aplicación para hacer el análisis. *Model Based Testing* es una combinación del *Black-box testing* debido a que las pruebas son generadas a partir del modelo que es construido en función de los requisitos de la aplicación.

4.2. *Model Based Testing*

Model Based Testing se refiere la técnica de *testing* cuyos casos de prueba han sido generados parcial o totalmente a partir de un modelo que describe aspectos mayormente funcionales de un sistema. Un modelo es una abstracción del sistema, teoría o fenómeno, que preserva ciertas propiedades y que se puede utilizar para el análisis del sistema original.

Los modelos de las aplicaciones deben ser pequeños en relación al tamaño de la aplicación que describen. Además, deben ser, por un lado, fáciles de diseñar, aunque, por otro, deben ser lo suficientemente detallados para conservar las propiedades que se quieren analizar. Ejemplos de lenguajes de modelado son los un diagrama de secuencia de *UML* (*Unified Model Language*), o los diagramas de casos de uso. Una vez que se tiene modelada una aplicación, se puede utilizar *Model Based Testing* para generar escenarios de casos de pruebas a partir del modelo.

En la Figura 4.2 se puede ver cómo la técnica de *Testing* es el punto de partida para el *Model Based Testing*. En el *testing* los requisitos (*oráculos*) son la fuente principal para escribir los casos de prueba. Una vez generados, los casos de prueba pueden ejecutados de manera manual o automática (mediante *TestScrips*). Así que, en resumen, el *Model Based Testing* tiene dos entradas principales. Los requisitos y el modelo de la aplicación. Las pruebas se generan a partir del modelo en función de los requisitos (*oráculos*). Las pruebas generadas se transforman en *scripts* de pruebas, que posteriormente son ejecutados sobre la aplicación.

Existen cuatro enfoques de utilización de la técnica, dependiendo de las necesidades de análisis de una aplicación se utilizara un enfoque u otro. A continuación se describen las cuatro utilidades.

La primera utilidad del *Model Based Testing* es la generación de datos de

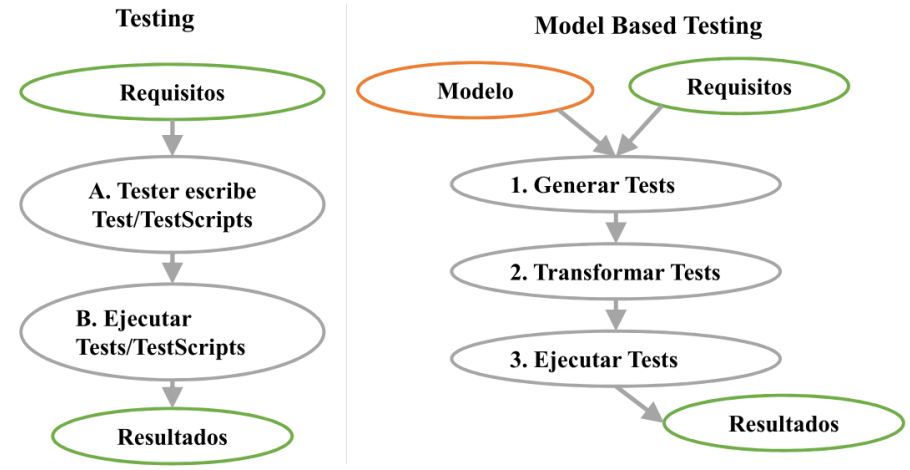


Figura 4.2: *Testing* y *Model Based Testing*

entrada a partir del modelo del dominio de datos de una aplicación. De esta manera, se analiza si alguna secuencia de valores de entrada produce algún tipo de fallo. Para ello, se generan los valores comprendidos en un dominio que deben tomar las variables (como rangos de números enteros entre 1 y 1000, o una selección de una lista de valores posibles). La debilidad de este enfoque radica en que los fallos se encuentran solo cuando la aplicación dispara una excepción, es difícil predecir otro tipo de errores.

Otra utilidad del *Model Based Testing* es la generación de casos de prueba a partir del modelo del entorno de una aplicación de manera que se pueden obtener un conjunto de pruebas que representan posibles llamadas a la aplicación. Al igual que en el caso anterior, la debilidad de este enfoque radica en que es difícil predecir cuándo un conjunto de llamadas produce un error, si el modelo no contiene una secuencia lógica de un comportamiento de una aplicación que permita a predecir los valores *esperados* de salida.

Un tercer enfoque es la generación de casos de prueba a partir del comportamiento de la aplicación para verificar el cumplimiento de los requisitos (*oráculos*). Las pruebas que se generan incluyen información del comportamiento, permite predecir valores *esperados* de salida. En este enfoque se generan, al igual que en el anterior, una serie de casos de prueba que representan las llamadas a la aplicación, pero estos casos de prueba se deben ejecutar de manera ordenada en función de la lógica de comportamiento de una aplicación. De los casos de prueba generados interesa evaluar el resultado solo los que contienen los valores de salida que deben ser evaluados contra los valores *esperados*. Este es el único enfoque que permite un análisis global de la aplicación.

Finalmente, como su cuarto y último enfoque se tiene la generación de *scripts* de pruebas a partir de test abstractos. En este caso, hace falta tener una definición abstracta de los casos de prueba mediante, por ejemplo, diagramas de

secuencia *UML*. El objetivo principal de este tipo de *testing* es transformar un caso de prueba abstracto en *scripts* de pruebas ejecutables.

4.3. Ventajas de la técnica

El testing basado en modelos tiene una serie de ventajas que se describen a continuación [6]. Por un lado, permite la reducción del tiempo y coste del análisis respecto a otras técnicas de *Testing*. Esto se debe a que se emplea más tiempo en el diseño del modelo que en la codificación las de pruebas cuando se añaden o cambian funcionalidades. Además, mejora la calidad del análisis de las aplicaciones porque cuando se dispone del modelo de pruebas se puede hacer el análisis de la aplicación. Por este mismo motivo, también permite la detección de defectos en los requisitos de forma temprana además de trazar un fallo desde el punto del código que falla hasta el requisito funcional que ha generado la prueba que ha detectado el error. Finalmente, la visibilidad que ofrece un modelo de aplicación puede ayudar a mejorar el sistema y a predecir el impacto de nuevos requisitos.

4.4. Limitaciones de la técnica

Una de las mayores limitaciones es determinar si el modelo es fiel a la implementación. Esta misma limitación la tienen otras técnicas del tipo *Model Driven Engineering*. Las diferencias entre el modelo y el sistema limitan la generación de todos los casos de pruebas necesarios para analizar la calidad de una aplicación.

Adicionalmente, existen otros problemas asociados al uso del *Model Based Testing* en proyectos de desarrollo de aplicaciones. En primer lugar, si se proporcionan requisitos que no están actualizados, el modelo construido será incorrecto. En segundo lugar, si los analistas no tienen la formación adecuada, la técnica puede no dar los resultados esperados. Finalmente, se necesita más tiempo para analizar los fallos en las primeras de desarrollo etapas porque deba a una mala generación de pruebas.

Capítulo 5

Model Checking

El *model checking* [1, 2, 3] es una técnica de verificación formal que se basa en la exploración exhaustiva de todo el espacio de estados generado por un modelo de un software (que suele ser concurrente) buscando comportamientos erróneos. Para aplicar la técnica se necesita un modelo M del sistema a analizar (que se describe utilizando un lenguaje de modelado), una propiedad P deseable para el sistema (que se especifica, normalmente, utilizando alguna variante de la lógica temporal), y un algoritmo de *model checking* que enfrenta el modelo y la propiedad y que responde sí, si M satisface P, o no, si se ha encontrado algún comportamiento del modelo que no satisface P. Una de las características de la técnica es que es algorítmica (automática), es decir que, una vez que se tienen M y P, la verificación se realiza sin intervención humana. Además, en el caso de que el modelo no satisfaga la propiedad, la técnica produce la secuencia de instrucciones que han llevado al error (lo que se denomina contraejemplo), por lo que el *model checking* se convierte en una herramienta potente para la depuración de software concurrente.

Aunque hay varias formas de aplicar la técnica, en general, se puede decir que los algoritmos de model checking funcionan creando un *grafo de alcanzabilidad* del sistema que contiene todos sus comportamientos posibles a partir de un estado inicial. Por lo tanto, aunque en la práctica se suelen utilizar distintos métodos de abstracción y compactación del espacio de estados, la técnica tiene implícita lo que se llama *el problema de la explosión de estados* que ocurre cuando el grafo a construir es demasiado grande para los recursos disponibles de la máquina. En los algoritmos de model checking basados en autómatas, la construcción del grafo está guiada por la propiedad que se quiere analizar, de manera que en cuando se detecta que un camino en el grafo satisface la propiedad, se descarta y se continúa la búsqueda por otro camino. Alternativamente, cuando se encuentra una traza errónea, se para la búsqueda y se devuelve el contraejemplo. En la Figura 5.1 se muestra un ejemplo de la exploración exhaustiva de los posibles estados, *backtrack* es el retroceso a un estado anterior para explorar otro posible comportamiento.

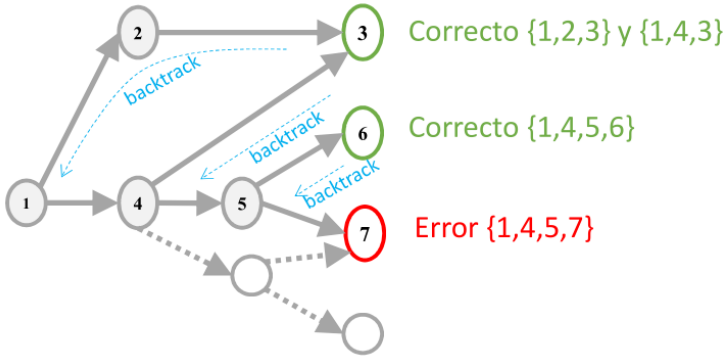


Figura 5.1: *Model cheking*

5.1. Fases de *model checking*

Desde el punto de vista metodológico, la aplicación de la técnica de model checking conlleva la realización de tres fases: modelado, ejecución y análisis. A continuación, se describen brevemente cada una de ellas.

Fase de modelado

En esta fase, se diseña el sistema utilizando un lenguaje de descripción de modelos, como se ha comentado anteriormente. En el contexto de model checking, los lenguajes de modelado realizan una representación abstracta del comportamiento del sistema utilizando, por ejemplo, sistemas de transiciones o autómatas de estados finitos, que modelan la computación utilizando un conjunto finito de estados y transiciones. Los estados muestran una fotografía del sistema antes y después de ejecutar las transiciones (operaciones). Por ejemplo, los estados de un sistema suelen contener los valores actuales de las variables. Las transiciones muestran cómo evoluciona el sistema de un estado a otro.

Por otro lado, las propiedades se escriben utilizando lenguaje de especificación basado, normalmente, en la lógica temporal, lo que permite la descripción de propiedades de seguridad y viveza típicas de los sistemas concurrentes.

En términos de lógica matemática, el model checking comprueba que la descripción del sistema es un modelo de la fórmula de lógica temporal que representa la propiedad. Esto explica el término de *model cheking*. La lógica temporal es básicamente una extensión de una lógica proposicional con operadores modales que permiten describir la evolución de los sistemas en el tiempo.

Fase de ejecución

En esta fase, se utiliza una herramienta de model checking (model checker) para comprobar si el modelo satisface la propiedad descrita, es decir, se utiliza un algoritmo de model checking (que está empotrado en una herramienta) para

analizar si todos los comportamientos posibles del modelo son correctos con respecto a la propiedad.

Fase de análisis

La herramienta de model checking puede dar tres posibles respuestas: (1) la propiedad especificada es válida en el modelo; (2) no es posible analizar el modelo debido al problema de la explosión de estados; (3) la propiedad no se satisface en una traza dada (el contraejemplo). El primer caso es el ideal, ya que implica que el modelo satisface la propiedad especificada. En el segundo caso, hay que intentar reducir el modelo o utilizar alguna técnica adicional de compactación/abstracción y volver a realizar el análisis. Cuando se proporciona un contraejemplo que es el síntoma de error, hay que depurar el sistema y volver a analizar la propiedad.

5.2. Sistemas Concurrentes

En la sección anterior se ha descrito de forma muy general la metodología de uso del *model checking*. En esta sección, se definen los *sistemas de transición* que son la estructura matemática en la que se basan muchos lenguajes de modelado, sobre todo los de los model checkers basados en autómatas. También, se introduce la lógica temporal.

5.2.1. Sistemas de Transición

Los *sistemas de transición/estructuras de Kripke* [20, 2] ayudan a describir comportamientos mediante grafo dirigidos, en los que cada nodo representa un estado, y los arcos son las transiciones entre estados, como se ha comentado anteriormente. Una estructura de Kripke (sistema de transición extendido con proposiciones atómicas) es una tupla $\mathtt{TS} = (\mathtt{S}, \mathtt{Act}, \rightarrow, \mathtt{I}, \mathtt{L})$ donde:

- $\mathtt{S}$ es un conjunto de estados.
- $\mathtt{Act}$ es un conjunto de acciones.
- $\rightarrow \subseteq \mathtt{S} \times \mathtt{Act} \times \mathtt{S}$ es la relación de transición, de ahora en adelante (s, a, s') se escribirá como $s \xrightarrow{a} s'$.
- $\mathtt{I} \subseteq \mathtt{S}$ es el conjunto de estados iniciales.
- $\mathtt{AP}$ es un conjunto de proposiciones atómicas.
- $\mathtt{L}:\mathtt{S} \rightarrow \wp(\mathtt{AP})$ es la función de etiquetado de cada uno de los estados.

$\mathtt{TS}$ es finito si el conjunto de estados $\mathtt{S}$ es finito.

Dado un sistema de transición $\mathtt{TS}$, la semántica basada en trazas $\mathcal{O}(\mathtt{TS}) = \{\pi = s_0 \xrightarrow{a_0} \cdots | s_0 \in \mathtt{I}\}$ es el conjunto de todas posibles trazas de ejecución definidas por el sistema de transición a partir de algún estado inicial.

Los sistemas de transición pueden estar compuestos de procesos cuyas transiciones se componen de forma *síncrona* o *asíncrona*. La composición es síncrona si las acciones de los procesos respectivos obligan a que éstos evolucionen simultáneamente. Por otro lado, la composición es *asíncrona* cuando cada proceso puede evolucionar de forma independiente al resto de los procesos del sistema. La ejecución asíncrona corresponde al modelo de intercalado de procesos (*interleaving*). La relación de transición de un sistema de procesos asíncronos puede expresarse mediante una disyunción de transiciones de cada proceso individual.

5.2.2. Propiedades

Antes de introducir la lógica temporal, en esta sección se describen los distintos tipos de propiedades de un sistema concurrente que suelen analizarse mediante la técnica de model checking.

Propiedades de *bloqueo*

A diferencia de los programas secuenciales, que se supone que deben terminar siempre (lo que se conoce como corrección *total*), la terminación de un sistema concurrente puede ser síntoma de un *bloqueo*. Un bloqueo ocurre cuando el sistema alcanza un estado desde el que no puede avanzar y, al menos, uno de sus procesos no ha terminado. Un escenario común de bloqueo ocurre cuando cada proceso del sistema espera un recurso que está siendo utilizado por otro proceso.

Propiedades de *Seguridad*

Las propiedades de seguridad son requisitos básicos de los sistemas que expresan que el sistema nunca debe estar en un estado de error. El bloqueo o la *exclusión mutua* son ejemplos de propiedades de seguridad. Los *invariantes* son un caso especial de propiedades de *seguridad* porque deben ser ciertos en todos los estados alcanzables del sistema.

Propiedades de *Viveza*

Al contrario de las propiedades de *seguridad*, las propiedades de *viveza* son propiedades de progreso que describen que el sistema estará en un estado deseable *en algún momento*. Desde el punto de vista de la verificación, para analizar una propiedad de *viveza* no basta con analizar un estado o una secuencia finita de estados en cada traza, sino que, normalmente, es necesario analizar la traza completa. Por ejemplo, para el conocido problema de la exclusión mutua, las propiedades "eventualmente, cada proceso entrará en su sección crítica" y "cada proceso entrará en su sección crítica infinitas veces" son típicas propiedades de viveza.

5.2.3. Lógica temporal

La lógica temporal[2] es un formalismo más adecuado para describir propiedades de seguridad y viveza de un sistema concurrente. La lógica temporal extiende la lógica proposicional con operadores modales que permiten representar la evolución temporal de un sistema. La lógica temporal puede ser ramificada (CTL) cuando cada fórmula se evalúa sobre un árbol de ejecución (el grafo de alcanzabilidad expandido) o lineal (LTL), cuando las fórmulas se evalúan sobre trazas de ejecución.A continuación, se introducen brevemente los operadores de la lógica temporal lineal LTL.

Lógica temporal lineal (LTL)

La lógica temporal lineal (LTL) [21, 22] permite describir trazas de ejecución $\pi = s_0 \to \cdots$ de la semántica basada en trazas determinada por un sistema de transición(como por ejemplo los autómatas finitos [23]). Las fórmulas LTL se construyen utilizando proposiciones atómicas de $\mathtt{AP}$, operadores booleanos ($\neg$, $\wedge$) y operadores modales (*en el siguiente estado* $\bigcirc$ y *hasta* U). Cuando las fórmulas no contienen operadores temporales se denominan fórmulas de estado. En otro caso, es decir, si tienen algún operador temporal, la fórmula es de camino. Para dar semántica a los operadores, se supone que se tiene una estructura de Kripke $\mathtt{TS} = (\mathtt{S}, \mathtt{Act}, \to, \mathtt{I}, \mathtt{L})$ sobre cuyos estados y trazas se evalúan las fórmulas. A continuación, se define la semántica habitual de los operadores booleanos, suponiendo que $a \in \mathtt{AP}$ es una proposición atómica, y que α y β son fórmula de estado. Dado un estado $s \in \mathtt{S}$, utilizamos la notación $s \models \alpha$ para indicar que s satisface la fórmula de estado α.

- $s \models a$ sii $a \in \mathtt{L}(a)$, es decir, si la proposición atómica a es una de las etiquetas del estado s según la estructura de Kripke $\mathtt{TS}$.
- Negación: $s \models \neg\alpha$ sii $s \not\models \alpha$
- Disyunción: $s \models \alpha \vee \beta$ sii $s \models \alpha$ o $s \models \beta$

Como es habitual, la semántica de la conjunción y de la implicación se obtiene a partir de las equivalencias $\alpha \wedge \beta \equiv \neg((\neg\alpha) \vee (\neg\beta))$ y $\alpha \to \beta \equiv (\neg\alpha) \vee \beta$.

Las *fórmulas de camino* se evalúan sobre trazas de ejecución del tipo $\pi = s_0 \xrightarrow{a_0} s_1 \cdots$. Suponemos que las trazas son infinitas (si son finitas, repetimos de manera indefinida el último estado para que sean infinitas). A continuación, se describe de manera informal la semántica de las fórmulas LTL sobre una traza $\pi = s_0 \to \cdots$ cuyo estado inicial es s_0, α es una fórmula de estado, además, f y g son fórmulas de estado o camino. En la semántica, usamos $\pi^i = s_i \to \cdots$ para representar la traza sufijo de π que comienza en el estado s_i.

- $\pi \models \alpha$ sii $s_0 \models \alpha$, es decir, una fórmula de estado se satisface sobre una traza, si es cierta sobre el estado inicial.

La siguiente traza satisface α.[1]

$$\alpha \longrightarrow arbitrario \longrightarrow arbitrario \longrightarrow arbitrario \longrightarrow arbitrario \longrightarrow \cdots$$

- $\pi \models \neg f$ sii $\pi \not\models f$.
- $\pi \models f \vee g$ sii $\pi \models f$ o $\pi \models g$.
- $\pi \models \bigcirc f$ sii $\pi^1 \models f$.

  Por ejemplo, dada una fórmula de estado α, la siguiente traza satisface $\bigcirc\alpha$:

  $$arbitrario \longrightarrow \alpha \longrightarrow arbitrario \longrightarrow arbitrario \longrightarrow arbitrario \longrightarrow \cdots$$

- $\pi \models fUg$ sii $\exists i \geq 0$ tal que $\pi^i \models g$ y $\forall 0 \leq j < i.\pi^i \models f$.

  Por ejemplo, si α y β son fórmulas de estado, la siguiente traza satisface la fórmula $\alpha U\beta$.

  $$\alpha \wedge \neg\beta \longrightarrow \alpha \wedge \neg\beta \longrightarrow \alpha \wedge \neg\beta \longrightarrow \beta \longrightarrow arbitrario \longrightarrow \cdots$$

Los operadores $\Diamond$ y $\Box$ pueden definirse a partir de los anteriores. Donde, $\Diamond\alpha$ significa que eventualmente (*eventually*) en un estado futuro α ocurrirá en la traza y $\Box\alpha$ significa que es un hecho que α ocurra siempre (*always*) en todos los estados de la traza .$\Diamond f \equiv trueUf$, que equivale a definir $\pi \models \Diamond f$ sii $\exists i \geq 0$ tal que $\pi^i \models f$. La siguiente traza satisface $\Diamond\alpha$, siendo α una fórmula de estado.

$$\neg\alpha \longrightarrow \neg\alpha \longrightarrow \neg\alpha \longrightarrow \alpha \longrightarrow arbitrario \cdots$$

Similarmente, $\Box f \equiv \neg(\Diamond(\neg f))$, que equivale a definir $\pi \models \Box f$ sii $\forall i \geq 0.\pi^i \models f$. La siguiente traza satisface $\Box\alpha$, siendo α una fórmula de estado.

$$\alpha \longrightarrow \alpha \longrightarrow \alpha \longrightarrow \alpha \longrightarrow \alpha \longrightarrow \cdots$$

Como ejemplo de uso de la lógica temporal, considera el problema de la exclusión mutua para 2 procesos. Los procesos concurrentes P y Q listados en los Códigos 5.1 y 5.2 respectivamente ejecutan, de manera iterativa, una sección no crítica, un pre-protocolo de entrada, la sección crítica, y un post-protocolo de salida.

El requisito de exclusión mutua se puede expresar como un invariante en LTL como $\Box(\neg crit1 \vee \neg crit2)$, es decir, no debe existir ningún estado en el que los procesos P y Q estén simultáneamente en sus respectivas secciones críticas. En concreto, la fórmula lo que dice es que siempre al menos uno de los dos procesos no está en su sección crítica.

Por otro lado, una propiedad de *viveza* que se puede especificar, es que siempre cada proceso, en algún momento, ejecutará su sección critica: $(\Box\Diamond crit1) \wedge (\Box\Diamond crit2)$

[1] Para simplificar el ejemplo, cada estado está representado por la fórmula de estado que satisface. "arbitrario" significa que el estado puede satisfacer cualquier fórmula de estado.

```
process P

    while (true){
          sec-no-crit1
          pre1
          sec1
          post1
    }
```

Código 5.1: Sección crítica proceso P

```
process Q

    while (true){
          sec-no-crit2
          pre2
          sec2
          post2
    }
```

Código 5.2: Sección crítica proceso Q

Dada una estructura de Kripke y una propiedad especificada en lógica temporal, los algoritmos de *model checking* construyen de forma explícita o implícita el llamado grafo de alcanzabilidad generado por el sistema de transiciones buscando trazas que no satisfagan la propiedad [24, 25]. Como se ha mencionado antes, la técnica sufre el problema de la explosión de estados cuando la generación de este grafo de alcanzabilidad supera los recursos de la máquina que está realizando el análisis. Para solucionar este problema, se han desarrollado muchas técnicas de optimización como son la reducción de orden parcial (se eliminan entrelazados entre procesos, cuando solo las instrucciones ejecutadas solo afectan localmente a los procesos), la reducción por simetría (que se aplica cuando el sistema está formado por una serie de procesos que se comportan todos del mismo modo), la abstracción (simplificación de los sistemas aplicando, por ejemplo, interpretación abstracta) y compactación de espacio de estados mediante técnicas de hashing (esta técnica solo permite un generación parcial del espacio de estados), entre otras.

5.3. Reducción de orden parcial

La reducción de orden parcial POR(*Partial Order Reduction*) [1, 26, 2] es una técnica de optimización que intenta de mitigar el problema de la explosión de estados que ocurre cuando se analiza el modelo de un sistema. El método consiste en construir y reducir el grafo de estados que, en muchos casos, puede exceder la capacidad de la memoria del computador donde se verifica el modelo. Hacer una reducción parcial es cuando se tiene uno o más procesos que se ejecutan de manera asíncrona, los eventos de cada proceso se intercalan entre sí pero para la propiedad a analizar el orden de ejecución de estos eventos no es relevante por lo que hay que tener en cuenta un orden de ejecución. POR es esta especialmente diseñada para sistemas asíncronos en los cuales las acciones son intercaladas y no se ejecutan simultáneamente.

En la Figura 5.2 muestra un grafo de ejecución y su versión reducida después de aplicar POR. Se asume que P_1 y P_2 son procesos asíncronos que son ejecutados de manera concurrente $(P_1 \| P_2)$. Los procesos P_1 y P_2 ejecutan una secuencia de acciones $\alpha_1.\alpha_2$ y $\beta_1.\beta_2$ respectivamente. El grafo de la izquierda de la Figura 5.2 muestra todos los intervalos de procesos posibles. La propuesta de POR es examinar solo uno de los intervalos posibles porque el resultado de

ejecutar cualquiera de las secuencias de acciones siempre es el mismo independientemente del orden de sus acciones, por ejemplo, la ruta que se muestra a la derecha de la Figura 5.2 .

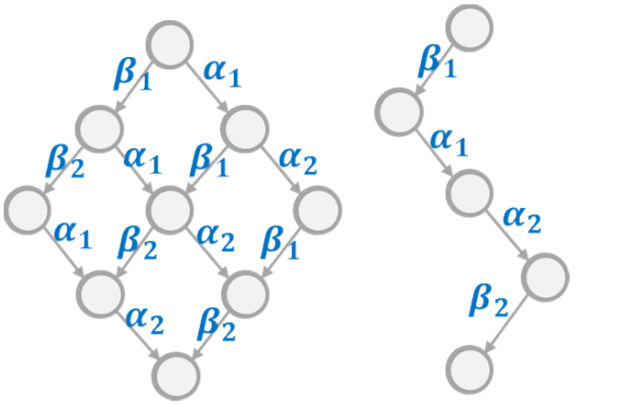


Figura 5.2: Ejecución paralela $(\alpha_1.\alpha_2)\|(\beta_1.\beta_2)$

Cuando se utiliza POR se asume que todos los procesos son todos autónomos, es decir, que no existe ningún proceso de sincronización entre los procesos y, que las propiedades no dependen de los estados intermedios. Su utilización en la verificación de aplicaciones reales es muy difícil dadas estas restricciones, para aplicar la técnica, se identifican fragmentos de todo el grafo de estados los cuales solo se diferencian unos de otros en el orden en el que se ejecutan sus actividades.

5.3.1. Abstracción

La abstracción [27, 28] es una de las técnicas mas frecuentes para reducir el tamaño del espacio de estados. Una aplicación puede ser modelada en varios niveles de abstracción. Los modelos que describen un alto nivel de detalle de implementación son considerados como de bajo nivel de abstracción. Por otro lado, los modelos de alto nivel de abstracción no describen muchos detalles, son pequeños y son más fáciles de verificar, aunque, no son relevantes para analizar propiedades especificas. Existen muchas técnicas de abstracción. Todos ellos derivan de la interpretación de abstracción [29]. Por ejemplo, el cono de influencia (*cone of influence*) [27] es una técnica de abstracción que se basa en remover variables de sistema que no tienen impacto en las variables utilizadas en la propiedad definida para el análisis. Otra técnica es la abstracción de datos (*Data Abstraction*) [27], el cual se centra en establecer una relación entre los valores actuales de una aplicación y un conjunto valores de datos abstractos. De esta manera, cuando se extiende la relación a los estados y transiciones es posible obtener un modelo abstracto de la aplicación que simula a la original.

5.4. Ventajas y desventajas

En este capítulo se han resumido las características y funcionalidades del *model checking*. Se ha explicado cómo la técnica de *model checking* analiza todo el espacio de estados generado por un modelo utilizando propiedades (con lógica temporal) que deben ser especificadas de acuerdo a requisitos específicos de la aplicación. Se ha descrito el problema de explosión de estados y las soluciones planteadas por la técnica de *model checking*, además de las técnica de reducción de orden parcial utilizada para resolver dicho problema.

Una de las ventajas del *Model Checking* es que es una técnica automática. De manera que, en caso de encontrar un error lo devuelve como contra-ejemplo que se compone de una secuencia de estados que se debe seguir para encontrar un error. Esto es muy importante para la depuración de los programas porque dichos errores pueden ser muy difíciles de localizar de manera manual porque, por ejemplo, ocurren cuando los procesos se entrelazan de una forma inesperada. La desventaja principal de ésta técnica, es el problema de la explosión de estados que ocurre cuando el sistema produce más estados de los que la capacidad de la máquina permite almacenar.

Capítulo 6

Runtime Verification

La técnica de verificación dinámica o *Runtime Verification (RV)* [30, 9, 10, 11] tiene como objetivo el análisis de la *ejecución* de las aplicaciones software haciendo uso de lenguajes de especificación (para describir las propiedades que quieren analizarse), algoritmos de análisis dinámicos (que se ejecutan simultáneamente con la aplicación que se está analizando), instrumentación de aplicaciones (para poder monitorizarlas) e incluso se puede apoyar en análisis estáticos. La técnica surgió en el contexto de los métodos formales, pero con el objetivo diferenciado de realizar el análisis del software de forma dinámica. De esta forma, complementa los métodos formales tradicionales que analizan las aplicaciones antes de su ejecución, de manera estática. Asimismo, permitenmejorar las técnicas de monitorización personalizadas que se utilizan en el análisis del rendimiento (*performance*) y en la depuración distribuida. Por lo tanto, es posible identificar varios escenarios de uso del *RV* como son control de calidad, prevención de fallos, detección de intrusiones, reigeniería de programas en función de sus salidas, caracterización de ejecuciones (*Profiling*), visualización de ejecuciones, entre otros.

Es importante notar que la técnica de *RV* es completamente *automática* debido a que se basa en la idea de analizar una aplicación al mismo tiempo que se está ejecutando. Además, se considera como *escalable* debido a que se aplica directamente sobre la ejecución de un programa y, cuando el programa escala su arquitectura, un programa *RV* continuará leyendo la traza de ejecución sin percatarse de los cambios.

La diferencia principal entre *RV* y el *model cheking* es que esta última técnica se aplica antes de que el software está en ejecución. Por otro lado, con respecto a la técnica *model based testing* 4.2 la diferencia es que el veredicto se refiere a cada caso de test generado a partir de un modelo de entrada, las pruebas generalmente son de tipo funcional y de caja negra. En cambio, *Runtime Verification (RV)* emite el veredicto sobre un fallo en la ejecución actual de una aplicación.

6.1. Fases de *Runtime Verification*

La aplicación de la técnica de *RV* sigue las siguientes fases [9, 10], que también se muestran en la Figura 6.1:

1. *Creación de un monitor*: Los monitores se crean típicamente a partir de una propiedad deseable. Su comportamiento consiste en observar si los eventos de la traza actual de ejecución se corresponden con el comportamiento esperado. De este modo, el monitor, de alguna manera, implementa una propiedad que suele estar asociada a los requisitos de corrección exigidos por el usuario.

2. *Instrumentación*: Para que el monitor pueda funcionar, es necesario extraer/capturar y filtrar los eventos de la aplicación que son relevantes para el monitor.

3. *Ejecución*: El sistema se ejecuta, genera eventos que son filtrados por la instrumentación.

4. *Respuestas*: Veredictos del monitor. Existen dos modos de funcionamiento. (1) El monitor produce un veredicto por cada evento consumido indicando el estado de la propiedad que se está evaluando, la secuencia de eventos y los valores de las variables en ese instante. (2) El monitor envía un *feedback* a la aplicación, para que cambie y así evite futuros fallos.

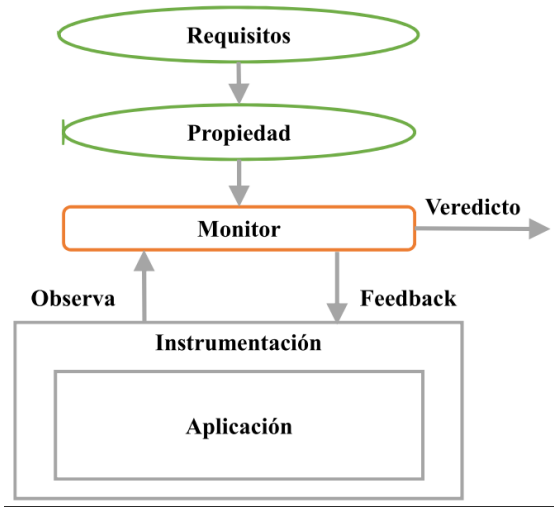


Figura 6.1: *Runtime Verification*

6.2. Eventos, trazas y propiedades

Durante la ejecución de una aplicación es posible capturar una secuencia finita de *eventos*. Los *eventos* muestran lo que ha ocurrido en la aplicación durante la ejecución. Cuando un evento ocurre se guarda un registro del estado actual de las acciones que han tenido lugar. La secuencia de eventos podemos interpretarla como una *traza* de ejecución τ. Una *propiedad* ϕ describe el comportamiento de un conjunto de trazas, que se puede denotar como $L(\phi)$. De este modo, una traza τ satisface la propiedad ϕ sii $\tau \in L(\phi)$, es decir, si τ es una de las trazas admitidas por la propiedad. Como las trazas producidas por una aplicación de software son muy complejas, las trazas analizadas son en realidad una proyección discreta de la ejecución real y pueden estar formada por una secuencia de estados (como es habitual en otras técnicas formales como el *model checking*), una secuencia de eventos que han ocurrido durante la ejecución de la aplicación, o una secuencia mixta tanto de estados como de eventos. Tal y como se muestra Figura 6.2, la traza se puede analizar en cualquier momento de su ejecución. El *presente* representa el evento/estado que acaba de ocurrir. El *pasado* de eventos/estados es la secuencia de eventos/estados registrados y que son conocidos. El *futuro* es desconocido e indeterminista, puede haber muchas evoluciones posibles a partir del estado actual.

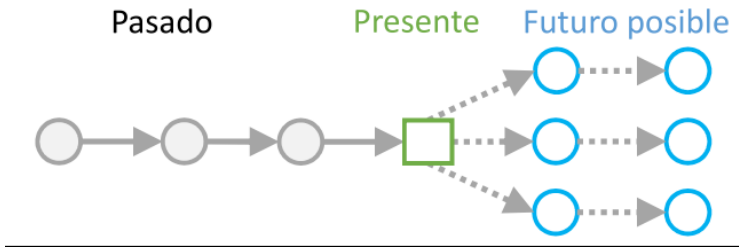


Figura 6.2: Traza de ejecución

En este contexto, una propiedad de *RV* típicamente se describe utilizando un lenguaje especifico de dominio *DSL* (*Domain-Specific Languaje*) [31] para expresar todas las ejecuciones individuales posibles de una aplicación . El DSL seleccionado puede utilizar diferentes formalismos, tales como, expresiones regulares como λ-`calculus`[32], lenguajes orientados a consultas como PQL[33], basados en reglas como Eagle[34] y, finalmente, algoritmos de lógica temporal (por ejemplo LTL)[35, 36], como es el caso de esta tesis . Los *DSL*s se pueden caracterizar de la siguiente manera:

- **Externa:** Cuanto el *DSL* es un lenguaje *stand-alone*, es decir, es independiente al monitor. En este sentido, dependiendo de la implementación del monitor, una propiedad puede ser *compilada* o *interpretada*. En el primer caso, la propiedad se incluye en el monitor previa traducción de la expresión *DSL* al lenguaje de programación. En el segundo caso, las propiedades

son traducidas e incluidas en estructuras de datos que luego el monitor en ejecución interpreta.

- **Interna:** El *DSL* está empotrado en un lenguaje de propósito general *GPL* (General-Purpose Language)[1], es decir, forma parte del monitor. Una propiedad de este tipo puede utilizar todas las características del *GPL*, o bien, puede ser representada como una estructura de datos dentro del *GPL*

6.3. Veredictos y *feedback*

La ventaja de analizar un sistema en tiempo de ejecución es que el sistema puede ejecutar acciones correctivas si se viola la propiedad. Esto significa que se pueden utilizar los resultados de la verificación para dirigir la ejecución de la aplicación hacia comportamientos deseados. Para lograr esto, los monitores se comunican con el sistema a través veredictos y retro-alimentación (*feedback*). Los veredictos pueden informar sobre el estado de la aplicación analizada con respecto a una propiedad y, además, proporcionar un *feedback* con información adicional.

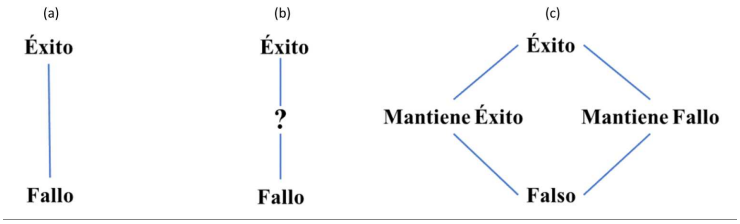


Figura 6.3: Posibles dominios de veredicto

Las respuestas de un sistema *RV* pertenecen a un *Dominio de Veredicto* que se puede denotar como *D*. Los posibles tipos de dominios de veredicto se pueden ver en la Figura 6.3. En el caso (a) de la figura se muestra el caso de dominio más simple, es decir, cuando los veredictos proporcionados por *D* toman un valor de *éxito* o *fallo*. Por otro lado, una gran parte de sistemas *RV* utiliza tres o más valores de respuesta como se observa en los casos (b) y (c) de la figura. En el caso (b) *D* contiene los valores de *éxito*, *fallo* y valores intermedios desconocidos "?" que corresponden a veredictos donde no se falló ni se tuvo éxito aún. Finalmente, el caso (c) extiende el caso (b), de manera que *D* contiene cuatro valores, los típicos *éxito*, *fallo* y dos valores adicionales *Mantiene-éxito*, *Mantiene-fallo* cuyos significados son que el sistema está actualmente teniendo éxito o está fallando

[1]GPL es una forma de denominar a los lenguajes multi-propósito diseñados para ser utilizados en la construcción de aplicaciones sin dominio especifico. De esta manera, una gran mayoría de los lenguajes de programación pertenecen a esta categoría, por ejemplo, Cobol, RPG, Java, Javascript,c#, PHP, entre muchos otros.

respectivamente. Sin embargo, estas respuestas no determinan los veredictos futuros.

6.4. Instrumentación del sistema

El proceso denominado *instrumentación* gestiona, en primer lugar, la manera en que son comunicadas las trazas de las aplicaciones monitorizadas a los sistemas *RV* y, de manera inversa, cómo los veredictos son comunicados a la aplicación monitorizada. De esta manera, la *instrumentación* es una parte fundamental en el proceso de verificación.

La implementación de la *instrumentación* depende de la aplicación que está siendo analizada. Las herramientas basadas en programas JAVA utilizan lenguajes de programación orientados a aspectos, por ejemplo: ASPECTJ [37], SPRING AOP [38], entre otros. Este tipo de lenguajes pueden ser utilizados directamente como un sistema *RV* como se describe en [10]. De otra manera, una *interfaz* debe ser definida entre la herramienta *RV* y la aplicación monitorizada.

6.5. Monitor

Como ya se ha comentado, el monitor es el encargado de analizar los eventos de las trazas de ejecución y proporcionar veredictos. Cada monitor está dedicado al análisis de una única propiedad, por lo que, típicamente hay tantos monitores como propiedades se necesite analizar. Cada monitor almacena los estados y los eventos que le interesan para realizar el análisis de la propiedad que implementa.

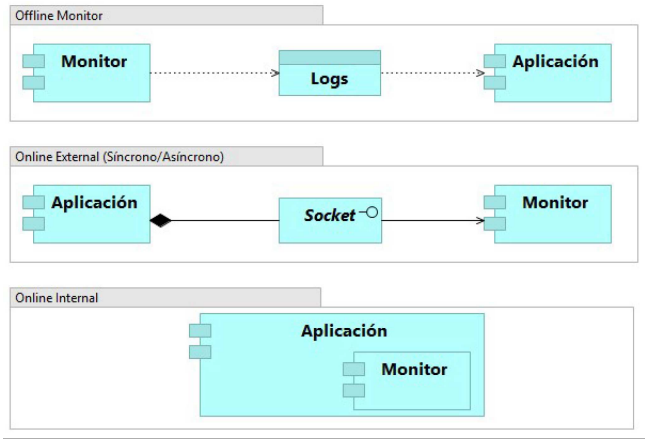


Figura 6.4: Tipos de *Runtime Verification*

Para poder analizar la aplicación, el monitor tiene que estar integrado con ésta de alguna manera. En la Figura 6.4 se muestran dos formas posibles de realizar esta integración. Los monitores *Offline* analizan la traza después de su

ejecución, por ejemplo, utilizando archivos de *logs*. Otra posibilidad es que los monitores actúen *Online*. En este caso, tenemos dos posibilidades también. La técnica *External Online* es utilizada por los monitores que se ejecutan en paralelo con la aplicación y que se comunican con ésta a través de *sockets* síncronos (el sistema espera una respuesta) o asíncronos. Por otro lado, los monitores *Internal Online* son aquéllos que están integrados dentro de la aplicación.

Los monitores pueden construirse de muchas formas. Como se ha comentado antes, lo habitual es que se construyan para encontrar una solución especifica a un problema como pueden ser (*DataRace* o *Deadlock*). La lógica del monitor se puede escribir utilizando un lenguaje de programación, o utilizando el diseño basado en contratos (condiciones pre/post). También es posible que los monitores se escriban utilizando algún lenguaje de modelado/especificación formal como, por ejemplo, máquinas de estados, expresiones regulares, gramáticas o lógica temporal.

6.6. Cobertura

La mayor desventaja de la técnica es que, si se la aplica por si sola, tiene un bajo porcentaje de cobertura. Esto se debe, a que un monitor de *Runtime Verficiation* no interviene en las funcionalidades se ejecutan en la aplicación, por lo tanto, no puede determinar que porcentaje de una aplicación se ha evaluado. Para aumentar el nivel de cobertura, la técnica se puede combinar con otras que si pueden decidir que funcionalidades de una aplicación se ejecutan, como ser, *Model Based Testing* y *Model Cheking*. La combinación con *Model Based Testing* permite que las trazas capturadas sean producto de casos de prueba generados a partir de modelos de una aplicación, de esta manera, el nivel de cobertura dependería de las funcionalidades descritas dicho modelo. La combinación con *Model Checking* permite que las trazas capturadas sean producto de exploración de los todo el espacio de estados de una aplicación, entonces, se asegura el análisis de todas las funcionalidades. En ésta última afirmación, el nivel de cobertura solo se ve restringido por las capacidades de computo del monitor debido a que el espacio de estados puede ser muy grande.

6.7. Resumen

La técnica de *Runtime Verification* permite analizar los sistemas en ejecución. Primero se instrumenta la traza de una aplicación a analizar para que un monitor de *Runtime Verification* pueda tener acceso a la información. Luego, el monitor evalúa si la traza filtrada viola las propiedades de verificación definidas. Mas tarde, el monitor emite un veredicto que indica si ha ocurrido un fallo, o bien, predice si va a ocurrir un fallo en un futuro. Cuando se predicen fallos futuros se puede enviar esa información a la aplicación para que corrija el error antes de que ocurra.

Parte III

Framework de análisis de aplicaciones móviles

Capítulo 7

Propuesta de solución

7.1. *Mobile Verification Engine (MVE)*

Como ya se ha comentado anteriormente, el objetivo principal de esta tesis es proporcionar una metodología formal y una herramienta de análisis para programadores/*testers* que necesitan analizar el comportamiento de aplicaciones móviles con respecto a propiedades extra-funcionales de su interés. Para conseguir esto, hemos diseñado una arquitectura que combina las técnicas de *Model Based Testing* [39], *Model Checking* [1] y, *Runtime Verification* [30]. La arquitectura propuesta, que se muestra en la Figura 7.1, está dividida en dos componentes/capas diferentes. El nivel de usuario (*User level*) describe el comportamiento funcional de la herramienta, desde el punto de vista del usuario. Un programador/*tester*, que quiere analizar una aplicación, lanza el proceso de generación de casos de pruebas (*Test Generation process*) que, en su paso final, ejecuta los casos de prueba sobre los dispositivos *smart*, simultáneamente, el proceso de verificación en tiempo de ejecución (*Runtime Verification process*) analiza y devuelve los resultados sobre los casos de test ya comprobados. El nivel de arquitectura(*Architecture*) contien los principales componentes de *software* de la herramienta. La *Mobile Verification Engine (MVE)* es el núcleo de la herramienta, la máquina de verificación de las *Apps*, mientras que el resto de componentes (que aparecen en la parte inferior de la figura) son herramientas adicionales que proporcionan servicios a *MVE* necesarios para su funcionamiento.

7.1.1. Nivel de usuario (*User Level*)

La metodología propuesta sigue los siguientes pasos. En primer lugar, el usuario *tester* debe construir el *modelo del comportamiento esperado del usuario* (*User behaviour model (UBM)*) utilizando el lenguaje de modelado que se describe en el Capítulo 8.1. Este modelo debe reflejar el modo natural de uso de la(s) *App(s)* bajo análisis. A partir de este modelo, la MVE genera automáticamente conjuntos de casos pruebas, tal y como se explica en el Capítulo 8.1.3. En

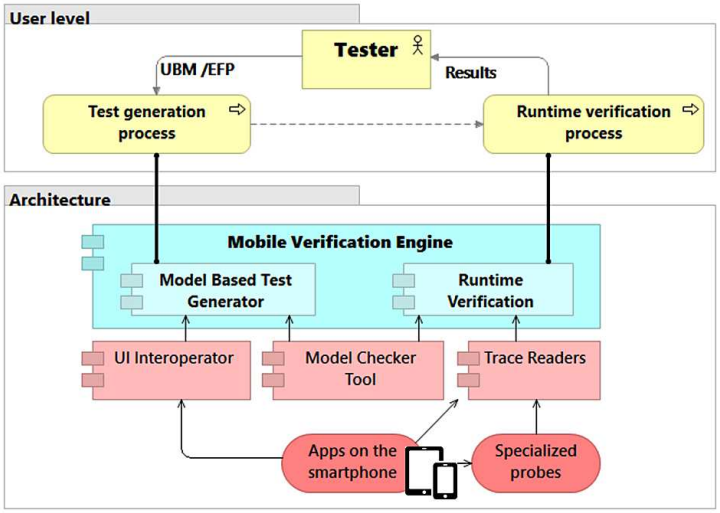


Figura 7.1: Diseño de arquitectura *Mobile Verification Engine*

nuestro marco, es posible construir casos de prueba realistas, que se corresponden con un uso posible del móvil, a diferencia de otras propuestas de generación automática de interacción con aplicaciones móviles, que son completamente aleatorias [40]. Las secuencias que se generan a partir del modelo de comportamiento de usuario representan escenarios típicos de utilización de la(s) aplicación(es) sobre los cuales se puedan analizar propiedades relevantes. También es posible construir modelos de comportamientos maliciosos o intencionalmente erróneos para analizar si las *Apps* son capaces de reaccionar correctamente a eventos inesperados.

Además del modelo, el *tester* debe proporcionar a la herramienta las propiedades de su interés que deben ser analizadas durante la ejecución de las *Apps*. Aunque se pueden analizar propiedades funcionales (requisitos de seguridad o viveza de las *Apps*), en esta tesis nos centramos en las extra-funcionales EFP (*Extra-Functional Properties*) (rendimiento, consumo de energía, seguridad, etc). Para la descripción de propiedades extra-funcionales se utiliza el lenguaje de especificación definido en el Capítulo 9.1.

Con estas dos entradas, el modelo y las propiedades, la *Mobile Verification Engine (MVE)* analiza si ejecuciones generadas a partir de los casos de tests satisfacen las propiedades. A continuación, la herramienta devuleve los resultados de los análisis. En el caso, de que alguna propiedad no se satisfaga, con estos resultados se puede depurar el código de la *App* para encontrar el origen del problema y corregirlo. Como se verá más en detalle en el Capítulo 8.1, el lenguaje de modelado se ha diseñado para que sea modular. Esto significa que a partir de un modelo inicial del comportamiento, se pueden añadir muchos más de manera incremental.

7.1.2. Arquitectura (*Architecture*)

El componente de más alto nivel de la arquitectura de la Figura 7.1 que muestra la herramienta *Mobile Verification Engine* (MVE) está compuesto por dos módulos principales. El primero, llamado *Model Based Test Generator*, es el encargado de generar los casos de prueba a partir de los modelos. El segundo, denominado *Runtime Verification*, es el monitor que analiza las trazas obtenidas del dispositivo mientras éste ejecuta los casos de prueba.

El modelo de comportamiento de usuario (*UBM*), diseñado por el *tester*, contiene nombres *abstractos* de elementos que hacen referencia a los componentes reales de las *Apps* a analizar como son los nombres de controles, eventos, vistas, etc. Por lo tanto, para completar el modelo, el *tester* debe enlazar los nombres abstractos utilizados en el modelo con los reales que corresponden a la aplicación. Para realizar esta acción, que se denomina configuración de la interfaz de usuario (*user interface configuration*), se hace uso de programas que pueden leer la interfaz de usuario de las *Apps* como, por ejemplo, los controles visuales que contiene una vista. En este trabajo, este tipo de programas se denominan (UI Interoperators). En el caso específico de Android, el sistema operativo propociona los programas UIAutomator/UIAutomatorViewer [41].

El modelo del usuario construido por el desarrollador/tester es típicamente indeterminista ya que un usuario puede interactuar con los elementos del dispositivo de muchas formas posibles navegando a través de la *App* sin seguir ningún orden en particular. Para tener en cuenta todos los posibles comportamientos del usuario, el módulo *Model Based Test Generator* utiliza una herramienta de *model checking* para generar automáticamente los casos de prueba. El *model checker* explora todas las posibles secuencias de interacciones permitidas por el modelo y, de esta manera, genera un conjunto de casos de prueba realistas. Cada caso de prueba contiene los eventos de usuario que son ejecutados remotamente sobre un dispositivo móvil. Este componente también hace uso de aplicaciones del tipo *UI Interoperator* para construir los casos de prueba.

Los módulos *Model Based Test Generator* y *Runtime Verification* se ejecutan simultáneamente. Mientras el primero ejecuta los casos de prueba sobre el dispositivo móvil, el segundo construye una traza enriquecida (*enriched trace*) con información *estática* y *dinámica* generada, ésta última, en tiempo de ejecución a partir de los datos del dispositivo y otras fuentes externas. Las tareas de análisis se realizan durante la generación de la traza enriquecida. Esta traza está compuesta por (1) los eventos de usuario ejecutados, (2) los eventos disparados por el dispositivo móvil y (3) información obtenida del dispositivo a través de sensores especializados (*specialized probes*). En este trabajo, se tienen en cuenta cuatro tipos de eventos. En primer lugar, los eventos que se ejecutan cuando cambia el estado del dispositivo (como por ejemplo, un cambio en la conexión del GPS). En segundo lugar, los eventos de grano fino ocurridos durante la ejecución de la aplicación. En tercer lugar, los eventos producidos por la interacción del usuario con el dispositivo como, por ejemplo, cuando se presiona un botón o se rellena un campo de texto. Finalmente, los eventos que provienen de sensores especializados (*specialized probes*) como, por ejemplo, cambios en el

nivel de batería.

En esta tesis, el análisis de las propiedades sobre las trazas construidas es realizado automáticamente por una herramienta de *model checking*. Estas herramientas aceptan, normalmente, propiedades descritas en alguna variante de la lógica temporal como la lógica *Linear Temporal Logic* (LTL). Las trazas enriquecidas sufren un proceso filtrado (se mantiene la información relevante para la propiedad, y se elimina el resto) que las simplifica antes de que el *model checker* las analice tal y como se describe en [42]. Finalmente, el *model checker* proporciona un veredicto, para cada caso de prueba analizado, que se devuelve al *Tester*.

Capítulo 8

Generación de casos de prueba

8.1. Modelando el comportamiento de usuario

En esta sección, consideramos las *Apps* como sistemas reactivos, cuyo comportamiento está dirigido por eventos externos. Una gran parte de este comportamiento viene determinado por las interacciones del usuario con la interfaz de la aplicación bajo análisis. En este sentido, un modelo de una aplicación debe describir secuencias de acciones desde la perspectiva del usuario. Como se comentó en el capítulo anterior, el objetivo del modelo es construir casos de prueba realistas, es decir, secuencias de acciones que se correspondan con el comportamiento natural de un usuario frente a un dispositivo móvil. Los modelos también se pueden diseñar para generar casos de prueba que conduzcan a errores esperados, para conseguir esto, el modelo debe describir solo los comportamientos que generan los casos de prueba que se desean generar. En este capítulo, se describe con detalle el lenguaje para el modelado de las *Apps*.

8.1.1. Elementos de una *App*

Los usuarios que interactúan con una aplicación móvil principalmente utilizan sus elementos gráficos, denominados controles, como los botones, cuadros de texto y listas, entre otros. En una plataforma de interfaz de usuario como la de los dispositivos móviles, cuya entrada principal son pulsos en la pantalla (*touch-based*), los controles pueden utilizarse de distintas maneras. Pueden ser simples gestos como un pulso (*tapping*), u otros más complejos como agrandar imágenes arrastrando dos dedos sobre la pantalla (*pinch-to-zoom*). No todos los controles responden del mismo modo a cada uno de los posibles gestos, sino que cada control define sus propios tipos de gestos. Por ejemplo, un botón reacciona a los pulsos, pero no reacciona al gesto de barrido (*swipe*). Ademas, algunas acciones de usuario dependen del estado de los controles en pantalla para interactuar

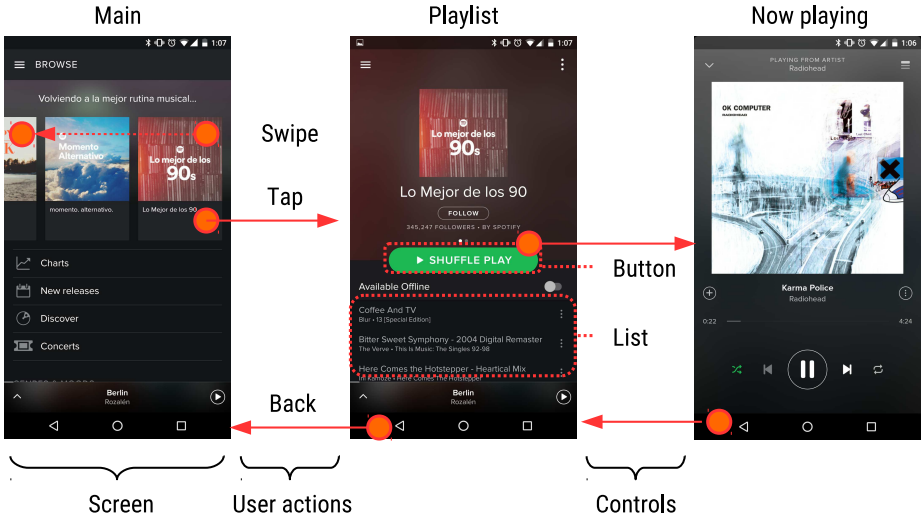


Figura 8.1: Elementos de las *Apps* móviles

con ellos. Por ejemplo, en una aplicación de correo, el botón *enviar correo* aparece *deshabilitado* mientras que no se escribe la *dirección destino*, por lo tanto, ningún gesto que el usuario haga sobre dicho botón tendrá efecto alguno.

Debido a las restricciones debidas al reducido tamaño de las pantalla de los dispositivos móviles, la mayoría de las *Apps* muestran solo una parte de su interfaz gráfica. Los controles que encajan visualmente en el dispositivo conforman una pantalla (*screen*). Por otro lado, al interactuar con la *App*, los contenidos de la pantalla actual pueden cambiar, y ser reemplazados por otros distintos. Por ejemplo, una *App* de correo electrónico puede mostrar una pantalla con la bandeja de entrada pero, cuando se selecciona un correo concreto, se muestra una pantalla distinta con el contenido del mensaje.

La Figura 8.1 muestra tres pantallas (*screens*) de la *App* de *Spotify* (que es uno de los casos de estudio de la presente tesis). *Spotify* es un servicio de transmisión de música por Internet (*music stream*) bien conocido. En cada una de sus pantallas, se identifican diferentes controles. Por ejemplo, en la segunda pantalla se encuentra el botón (*Button*), que reordena la lista de canciones, y la lista de canciones en sí misma (*List*). Al mismo tiempo, la figura muestra las posibles interacciones del usuario en cada pantalla. Por ejemplo, en la primera pantalla se pueden identificar varias: una interacción de barrido (*Swipe*) sobre la lista de controles, que permite visualizar más títulos, un pulso (*Tap*) sobre una canción de la lista, o un pulso sobre el botón atrás (*Back*) que cambia la pantalla actual por otra.

La navegación entre pantallas no es, en general, trivial. Muchas aplicaciones organizan sus pantallas de manera jerárquica. Las nuevas pantallas pueden co-

rresponder a un nivel cada vez más profundo de la jerarquía y el usuario puede navegar hacia atrás para volver a las pantallas por las que navegó anteriormente. La acción de *volver atrás* está soportada directamente por el dispositivo móvil. En el caso de ANDROID es el botón *Back* el que permite volver a pantallas anteriores.

En este trabajo, se utiliza el modelo de navegación de ANDROID, descrito en la Sección 3.3.2, como referencia para definir la navegación entre pantallas. La pantalla que está en la cima de la pila es la que es actualmente visible al usuario. Cuando el usuario navega a una nueva pantalla, ésta se coloca en la cima de la pila. La pantalla actual es eliminada de la pila cuando el usuario retorna (pulsa *Back*). La pantalla inicial del dispositivo (*home screen*), por defecto está en el fondo de la pila.

Por otro lado, se puede acceder a cada pantalla de varias maneras. Cuando los usuarios pulsan *Back* esperan ver la pantalla han visitado inmediatamente antes, pero puede que no sea así, si la pantalla actual accederse desde otras diferentes. Por lo tanto, puede haber varias pantallas previas posibles. Por ejemplo, si un usuario decide compartir una entrada de una *App* de noticias a través del correo electrónico, y mientras está redactando el correo presiona atrás, pueden ocurrir dos cosas: por un lado, si se tenia previamente abierta la aplicación de correo al presionar atrás se mostrara la pantalla con la bandeja de entrada de correos y, por otro lado, si la aplicación de correos estaba cerrada se muestra la pantalla de la *App* de noticias. Algunas veces el historial en la pila puede ser cambiado por decisión del desarrollador. Por ejemplo, cuando se completa información mediante un asistente (*wizard*) es posible que al volver hacia atrás se muestre la pantalla de inicio del asistente.

No es posible describir todos los comportamientos de las *Apps* en función de eventos de usuario ya que muchos de ellos son generados por el propio dispositivo como son, por ejemplo, la recepción de correos, o el despliegue de alertas. Estas acciones, contempladas en el lenguaje de modelado que se presenta a continuación, se denominan eventos de sistema *system events*.

8.1.2. Lenguaje de Modelado

El lenguaje de modelado propuesto está basado en la noción de *máquina de estados* similar a la de *UML* [43] y de Harel *statecharts* [44], incluyendo la notación gráfica. Estas máquinas se han extendido convenientemente de manera que sea posible modelar las *Apps*.

Las máquinas de estados están compuestas por estados conectados a través de transiciones etiquetadas. Un estado se representa con un rectángulo redondeado y las transiciones con flechas directas, ambos con etiquetas personalizables. Las etiquetas de las transiciones comienzan con un carácter *slash* ("/"). Las máquinas de estados son también rectángulos redondeados etiquetados, que internamente contienen estados y transiciones. Existen dos tipos especiales de estados: los iniciales y finales. Los estados iniciales se reprsentan con un círculo pintado no etiquetado, y los finales con un círculo con una cruz dentro.

Mientras que los estados no tienen ninguna relación directa con la interfaz

de la *App* que se está modelando, las transiciones representan acciones sobre los controles. Cada transición está etiquetada con una acción que el usuario puede ejecutar. Las acciones pueden ser, por ejemplo, presionar un botón, o introducir un texto en un recuadro. Las transiciones también pueden etiquetarse con nombres de eventos de sistema que representan un eventos no controlados por el usuario. Esta distinción es muy importante para la generación de casos de prueba ya que, los eventos de sistema no se traducen como acciones a ejecutar sobre la pantalla del dispositivo. Finalmente, el tiempo que un usuario debe esperar entre la ejecución de dos acciones, puede configurarse explícitamente agregando el atributo de tiempo ("{time=x}") en la etiqueta.

Con las transiciones disponibles en la máquina de estados, empezando por el estado inicial, se pueden construir secuencias de acciones de usuario que llamamos flujos (*flows*). Cada flujo representa un posible comportamiento de una aplicación. Como el objetivo principal es representar los comportamientos de usuario más cercanos a la realidad, la máquina de estados debería modelarse respetando ese principio, de tal manera que no se agreguen transiciones innecesarias para el análisis del caso de prueba.

El modelado de la mayoría de las *Apps* con una única máquina de estados no es un buena idea porque daría lugar a estructuras muy complejas. En su lugar, en esta tesis se propone modelar las aplicaciones con varias máquinas que pueden componerse de dos maneras diferentes. El primer tipo de composisición consiste en que una máquina puede "llamar" a otra máquina de estados de forma que cada una describe el comportamiento de un usuario en una pantalla distinta de una aplicación (las máquinas incluso podrían pertenecer a distintas aplicaciones).

El segundo tipo de composición es jerárquico. Las máquinas de estado pueden estar contenidas unas en otras del mismo modo que existe una jerarquía entre los dispositivos, aplicaciones y pantallas. La semántica del lenguaje de modelado presentado tiene en cuenta esta composición entre máquinas que las distingue de otras máquinas de estados más tradicionales. Un dispositivo se puede describir a través de una o más máquinas de estado que representan comportamientos en las pantallas de las *Apps* que contiene (incluido al sistema operativo), a estas máquinas se las denomina como máquinas de estado de dispositivo. Para proporcionar una visualización más estructurada, el lenguaje de modelado propuesto permite agrupar máquinas de estado de forma jerarquizada. Los dispositivos (*Devices*) son las máquinas de más alto nivel. Cada dispositivo contiene una o más *Apps* a analizar. A su vez, cada aplicación puede estar compuesta de una o más *vistas* (*views*) que son las distintas pantallas de la aplicación. Finalmente, cada vista puede contener una o más máquinas de estados, cada una de las cuales define un conjunto de comportamientos de usuario. Este tipo de composición jerárquica permite reutilizar comportamientos modularmente. Los *dispositivos*, *aplicaciones* y *vistas* también se representan como rectángulos redondeados etiquetados en los diagramas.

Las máquinas contienen un tipo especial de estado, denominado *estado de conexión* (*connection state*), que se utiliza para "llamar" a otra máquina de estado en el mismo dispositivo (aunque no necesariamente dentro de la misma

aplicación). Los *estados de conexión* se representan con círculos vacíos no marcados y tienen dos transiciones salientes sin etiquetar: una debe estar conectada a un estado dentro de la misma máquina de estado y, otra se puede conectar con una segunda máquina de estados. Cuando se alcanza un estado de conexión, el flujo continúa con la segunda máquina de estados conectada desde el *estado de conexión*. Una vez que la segunda máquina de estado termina, el flujo continúa con el estado conectado desde el *estado de conexión* de la primera máquina de estados. Todas las maquinas de estados de dispositivo comprenden un conjunto de maquinas de estados anidadas y conectadas a través de estados de conexión.

De esta manera, la ejecución del modelo completo se inicia en una máquina de estados que se alcanza a través de un camino directo de elementos accesibles externamente, es decir, dispositivo, aplicaciones, vistas y máquinas de estados. Esta organización permite que cualquier máquina de estados pueda ser llamada explícitamente desde otra distinta, aunque no se pueda acceder a ella desde el inicio del flujo.

Ejemplo de modelado

En la Figura 8.1, la pantalla "`Playlist`" esta lógicamente ubicada en el nivel más interno que la pantalla "`Main`". Si se pulsa el botón "`Shuffle play`" se muestra una nueva pantalla que reproduce la canción seleccionada. Por otro lado, la pantalla que reproduce canciones puede ser llamada desde otras como, por ejemplo, desde la pantalla "`album`". De esta forma, la pantalla que se muestra cuando pulsamos atrás depende de la secuencia de acciones que el usuario haya realizado antes de llegar a la pantalla actual. El modelo muestra parte del modelo utilizado en uno de los casos de estudio. Todas las vistas tienen una única máquina de estados excepto la vista "`SearchView`", que tiene dos. Estas dos máquinas modelan comportamientos ligeramente distintos de la misma vista.

El modelo contiene la mayoría de los conceptos del lenguaje de modelado. En concreto, en el ejemplo se modela el comportamiento de un usuario que busca escuchar una canción que puede estar en dos listas canciones de diferentes (la de canciones populares y la de no populares). El ejemplo contiene cinco máquinas de estados: "PrincipalStateMachine","SearchPopularStateMachine", etc. Cada una de las máquinas contiene varios estados y transiciones etiquetadas. Por ejemplo, el evento "SWIPE" de la máquina de estados de "PrincipalStateMachine" significa el desplazamiento sobre una lista presentada en la pantalla principal, mientras que el evento "clicConfiguration" está conectado con la acción de pulsar el botón de configuración sobre la pantalla actual. En las máquinas se puede observar también, el uso de los estados inicial y final, y que es posible tener más de una transición hacia un estado. La máquina "SearchPopularStateMachine" contiene varias transiciones configuradas con valores de tiempo que corresponden a las acciones de un usuario que presiona los botones "Play" o "Pause" . Cuando se ejecuta la transición que está a la derecha del estado "S0", se simula un usuario que está escuchando la canción durante 300 segundos, y a continuación, la máquina de estados finaliza. Por otro lado, si se toma otra transición desde "S0", se simula, además, el evento de pulsar el botón "Pause"

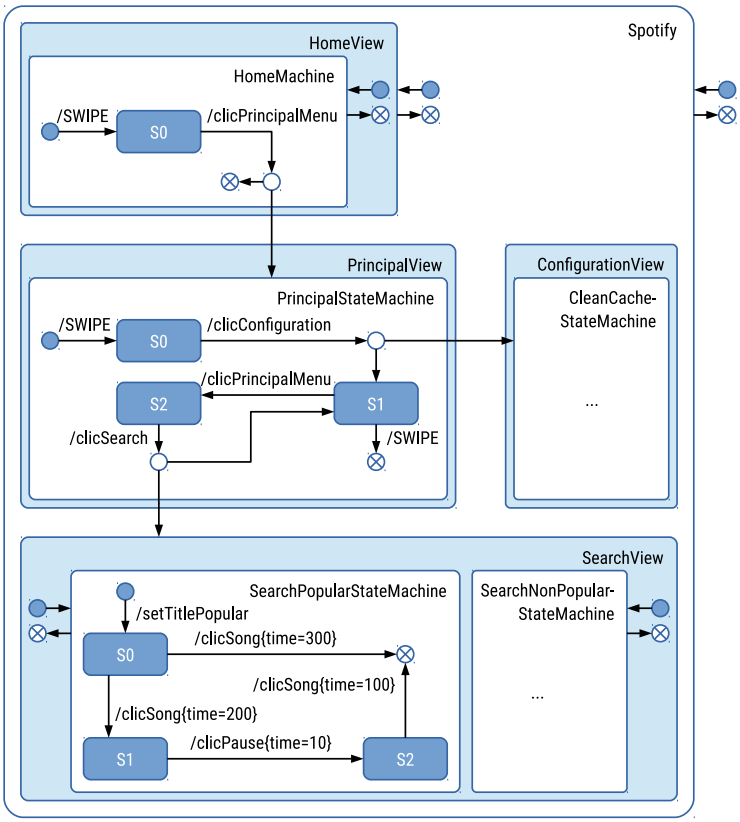


Figura 8.2: Parte del modelo de comportamiento de la *App* de *Spotify*

después de 10 segundos, y luego, la canción se detiene.

La máquina de estados de "`PrincipalStateMachine`" en la Figura 8.2, que representa la pantalla principal de la aplicación, contiene dos estados de conexión. El primero de ellos se alcanza después de ejecutar el evento de usuario "`clicConfiguration`", que simula un pulso en un botón de configuración. Luego, el flujo continúa en la máquina de estados "`CleanCacheStateMachine`", que simula el comportamiento de un usuario limpiando el caché de la aplicación desde la vista de configuración. Cuando esta máquina de estados termina, el flujo continúa por el estado "`S1`" en "`PrincipalStateMachine`", que es el siguiente estado al *estado de conexión* en esta máquina.

Los *estados de conexión* también pueden hacer referencia a una vista en concreto en lugar de una máquina de estados, como es el caso del estado de conexión que se puede alcanzar desde "`S2`" en "`PrincipalStateMachine`". Esto significa que cualquier máquina de estados *accesible externamente* desde la vista puede ejecutarse a continuación. El ser *accesible externamente* es una propiedad de las aplicaciones, vistas y máquinas de estados y se representa enlazando con

el estado de conexión un par de estados inicial y final.

El ejemplo contiene sólo una máquina de estados accesible externamente: "PrincipalStateMachine". El resto de las máquinas de estados sólo son accesibles a través de los estados de conexión. Además, el estado de conexión alcanzado desde "`S2`" apunta a "`SearchView`" (pantalla desde la que se pueden buscar títulos de canciones) una vista que contiene dos máquinas de estado accesibles externamente. En este caso, la transición desde este estado de conexión puede continuar con cualquiera de las dos máquinas de estados.

8.1.3. Semántica formal del lenguaje de modelado

En esta sección, se define formalmente el lenguaje de modelado presentado en la sección anterior, así como la construcción de *flujos de comportamiento de usuario* significativos a partir de un modelo. Se formaliza el uso de las *Apps* a través de la composición de *máquinas de estados* a tres diferentes niveles de abstracción. Las *máquinas de estados de las vistas* están en el nivel más bajo y representan el comportamiento de un usuario sobre una pantalla de un dispositivo. Los usuarios pueden interactuar con la interfaz de una vista en el dispositivo y, como consecuencia, se disparan eventos de usuario. En ocasiones alguno de estos eventos provoca que una vista distinta se muestre en la pantalla del dispositivo. Estas transiciones entre las vistas se han modelado utilizando una *relación de composición binaria* entre las máquinas de estados de vista, a partir de las cuales se construyen las *máquinas de estados de dispositivo*. Las máquinas de estado de vistas utilizan *estados de conexión* para cambiar de la vista actual a otra distinta. En esta formalización no se tiene en cuenta si ambas vistas pertenecen o no a la misma aplicación (debido a que las máquinas de estados de cualquier tipo se relacionan entre sí aunque no pertenezcan a la misma aplicación). Finalmente, el tercer nivel corresponde a la ejecución concurrente de máquinas de estados de dispositivo. En este nivel se asume que diferentes dispositivos móviles se están ejecutando simultáneamente y, además, que interactúan a través de las aplicaciones. En contraste con las máquinas de estado de vistas, las máquinas de estado de dispositivo se componen entrelazando y sincronizando sus transiciones.

View state machines

***Definición* 8.1** *Una* máquina de estados de vista *es un sistema de transición etiquetado* $M = \langle \Sigma, I, \overline{\rightarrow}, E, C, F \rangle$*, donde* Σ *es un conjunto finito de estados,* $I \subseteq \Sigma$ *son estados iniciales,* $C \subseteq \Sigma$ *son los denominados (*estados de conexión*,* $F \subseteq \Sigma$ *es el conjunto de estados finales,* E *es el conjunto de eventos de usuario/internos, y* $\overline{\rightarrow} \subseteq \Sigma \times E \times \Sigma$ *es la relación de transición etiquetada. Los conjuntos* I*,* C *y* F *son mutuamente disjuntos.*

Los *estados finales* son aquéllos desde los que no es posible transitar. Los *estados de conexión* son los que permiten transitar a una máquina de estados diferente. Estos estados son esenciales para modelar el cambio entre las vistas

típico de los dispositivos móviles. Se asume que las máquinas de estados de vista son deterministas en el sentido de que si $s \xrightarrow{e} s_1$ y, $s \xrightarrow{e} s_2$ entonces, $s_1 = s_2$, es decir, la máquina define a lo sumo una transición por cada par estado/evento.

Definimos el conjunto $Flow(M) = \{s_0 \xrightarrow{e_1} \cdots \xrightarrow{e_{n-1}} s_{n-1} \cdots | s_0 \in I, s_{n-1} \in F \cup C\}$ de todas las secuencias *finitas* de estados, definidos por M, que empiezan en un estado inicial, y terminan en un estado final o en un estado de conexión. Los elementos de $Flow(M)$ se denominan *flujos*. La *longitud* de un flujo es el número de estados que lo componen.

***Definición* 8.2** *Dado un flujo de longitud n, $\phi = s_0 \xrightarrow{e_1} \cdots \xrightarrow{e_{n-1}} s_{n-1} \in Flow(M)$, definimos el caso de test $test(\phi)$, determinado por ϕ, como la secuencia de eventos que ocurren en ϕ, es decir, $test(\phi) = e_1 \cdot \cdots \cdot e_n$. El conjunto de casos de tests determinados por M se define como $TC(M) = \{test(\phi) | \phi \in Flow(M)\}$.*

Se asume que el conjunto de eventos E se divide en dos conjuntos disjuntos: el conjunto de *eventos de usuario*, que se denota como E^+, con eventos tales como, pulsar un botón, etc, y el conjunto de *eventos de sistema*, denotados como E^- que incluye, por ejemplo, respuestas del sistema a peticiones del usuario. En adelante, se utiliza e^+, e^- para representar los eventos de usuario y los eventos de sistema, respectivamente. Se utiliza simplemente e cuando no hace falta distinguir el tipo del evento.

De acuerdo a la definición 8.2, los casos de tests son secuencias finitas de eventos de usuario y de sistema. Por ejemplo, una secuencia $e_1^+ \cdot e_2^+ \cdot e_3^- \cdot e_4^+$ representa un caso de prueba donde el usuario primero dispara los eventos e_1^+ y e_2^+, luego el sistema dispara otro evento e_3^- y, finalmente, el usuario dispara el evento e_4^+. Por lo tanto, los eventos de usuario y de sistema tienen un trato similar durante la generación de los casos de test. La principal diferencia entre ellos tiene lugar cuando el caso de prueba es transformado al código que será ejecutado el móvil, tal y como se describe en la Sección 10.1. Los eventos de usuario se transforman en llamadas asíncronas a métodos que simulan la ocurrencia del evento, mientras que los eventos de sistema corresponden con llamadas a métodos síncronos que esperan la llegada del evento del sistema.

Composición de máquinas de estados

En esta sección, se describe la composición de las máquinas de estados de vista, y la construcción de flujos que navegan a través de las distintas vistas. Se utiliza la *relación binaria* $\mathcal{C}$, entre estados de conexión e iniciales para modelar esta navegación. La idea principal de la composión es como sigue. Supón que el flujo de ejecución pertenece a una máquina de estados de vista M_i, y que un estado de conexión cs de M_i se ha alcanzado. Si la relación $\mathcal{C}$ define una transición desde cs a algún estado inicial de otra máquina de estados M_j, el flujo puede saltar desde M_i a M_j, y proceder con la siguiente relación de transición de M_j. Esta transición entre máquinas de estados de vista conlleva el cambio de la pantalla visible (vista) en el dispositivo de M_i a M_j. En lo que sigue, llamamos

activa a la máquina de estados de vista visible en el dispositivo, y llamamos *creada* al resto de las máquinas de estados de vista que se han creada, pero que no son actualmente visibles en el dispositivo.

Sea una familia de n máquinas de estados de vista $M_i = \langle \Sigma_i, I_i, \overline{\rightarrow}_i, E_i, C_i, F_i \rangle$ de manera que ninguna de ellas comparte ningún estado, es decir, que $\forall 1 \leq i, j \leq n.\ i \neq j$ implica $\Sigma_i \cap \Sigma_j = \emptyset$. A partir de estas máquinas se construye la máquina de estados $\bigcup_{i=1}^{n} M_i = \langle \Sigma, I, \overline{\rightarrow}, E, C, F \rangle$, donde:

- $\Sigma = \bigcup_{i=1}^{n} \Sigma_i$, $I = \bigcup_{i=1}^{n} I_i$, $E = \bigcup_{i=1}^{n} E_i$, $C = \bigcup_{i=1}^{n} C_i$, $F = \bigcup_{i=1}^{n} F_i$, y
- $s_i \xrightarrow{e} s_i' \iff \exists 1 \leq i \leq n. s_i \xrightarrow{e}_i s_i'$

Además, denotamos con $\mathcal{E} \subseteq E$ el conjunto de *eventos de llamada* que provocan el cambio entre máquinas de estados de vista.

***Definición* 8.3** *La* conexión *de una familia finita de n máquinas de estados de vista $M_1, \cdots, M_n$ la define una relación $\mathcal{C} \subseteq C \times \mathcal{E} \times I$, que hace posible transitar desde estados de conexión a estados iniciales.*

En lo que sigue, denotamos las ternas (s_i, e, s_j) de $\mathcal{C}$ como $s_i \xrightarrow{e}_c s_j$. Es importante tener en cuenta que las máquinas de origen y el destino i y j de los estados de las ternas pueden coincidir.

Cuando se transita desde una vista para continuar en otra, la vista que llama usualmente se mantiene activa, de manera que la ejecución puede retornar a la vista en el futuro. Para tener en cuenta este comportamiento, se supone que cada estado de conexión $s \in C_i$ tiene asociado un *estado de retorno* $return(s) \in \Sigma_i$ que representa el estado al que se volverá cuando la vista que llama vuelva a estar activa.

El evento que se lanza para crear una nueva vista puede llevar algunos parámetros que determinan cómo debe iniciarse o terminarse. Por ejemplo, si la vista ya existe, el evento debe decidir si quiere reutilizar la vista creada o hay crear una nueva. Además, cuando la nueva vista creada termina su ejecución, la vista que llama puede activarse automáticamente o no. Las funciones booleanas $reuse, auto\_return : \mathcal{E} \rightarrow \{false, true\}$ establecen estos parámetros para los eventos de llamada. Aunque pueden utilizarse muchos otros parámetros en los eventos de llamada, estos dos son suficientes para describir el comportamiento de las *Apps* en un dispositivo móvil.

Máquinas de estados de dispositivo

A continuación, se definen las máquinas de estados de dispositivo (*device state machines*) componiendo máquinas de estados de vista utilizando relaciones de conexión.

***Definición* 8.4** *Dada una familia finita de n máquinas de estados $M_1, \cdots, M_n$ y una relación de conexión de $\mathcal{C}$, una máquina de* estados de dispositivo $\mathcal{D} =$

$\bigcup_{i=1}^{n} M_i \bigoplus \mathcal{C}$ *es un tupla* $\langle \Sigma^* \times \Sigma^* \times \mathcal{E}^*, I, \xrightarrow{-}_d, E, F \rangle$ *donde* Σ^* *y* $\mathcal{E}^*$ *son conjuntos de secuencias finitas de estados de* Σ*, y de eventos de llamada de* $\mathcal{E}$*, respectivamente.*

Llamamos *configuraciones* a los estados de las máquinas de estados de dispositivos. Una configuración es una terna $\langle sh, rh, eh \rangle$ donde la secuencia sh es la lista de estados $s_0 \cdot s_1 \cdots s_n$ que se han visitado hasta ahora en el flujo, siendo s_n el estado actual en la máquina de estados de la vista que está activa. La secuencia rh es una pila de estados $r_1 \cdot r_2 \cdots r_m$ que constituyen la *historia* de las máquinas de estados de vistas que han sido creadas (y que aún no han sido destruidas) en el dispositivo, pero que no están visibles actualmente. Cada r_i de $r_1 \cdot r_2 \cdots r_m$ es el estado de *retorno* de un estado de conexión de una máquina de estados de vista que estuvo activa, pero desde la que se hizo una transición hacia otra vista distinta, por lo que la máquina de estados de vista a la que pertenece r_i dejó de estar activa. Es importante resaltar que tanto $s_0 \cdots s_n$ como $r_1 \cdots r_m$ representan secuencias de estados, es decir, subconjuntos de Σ^*. Finalmente, $eh = e_1 \cdots e_m$ representa la *historia* de los eventos que han provocado el cambio de vistas durante la ejecución actual. Las secuencias rh y eh tienen la misma longitud. Además, si el estado e_i de eh es un evento que disparó una transición a una vista diferente entonces, r_i es el estado al se debe volver cuando el flujo de la máquina de estados de esta vista finalice. En lo que sigue, ϵ representa una secuencia *vacía*, y utilizamos $eh \cdot e_m$ (similarmente, $rh \cdot r_m$) para denotar una pila no vacía con el evento e_m en su cima.

Las siguientes reglas definen la relación $\xrightarrow{-}_d$ que se construye a partir de relaciones de transición de las máquinas de estados de vistas $\xrightarrow{-}_i$, y la relación de conexión binaria $\xrightarrow{-}_c$. En estas reglas, dada una historia de estados $h_1 \cdots h_m$ y un índice j de una máquina de estados de vista M_j, la función $top : \Sigma^* \times \mathbb{N} \to \Sigma \cup \{\bot\}$ devuelve el último estado de la máquina de estados de la vista M_j en la secuencia $r_1 \cdots r_m$. Es decir, $top(r_1 \cdots r_m, j)$ devuelve r_k, si $1 \leq k \leq m$ es el mayor índice tal que $r_k \in \Sigma_j$, o $\bot$, si tal estado entonces no existe.

R1. $$\frac{s \xrightarrow{e}_i s'}{\langle sh \cdot s, rh, eh \rangle \xrightarrow{e}_d \langle sh \cdot s \cdot s', rh, eh \rangle}$$

R2. $$\frac{s \xrightarrow{e}_c s', \neg reuse(e)}{\langle sh \cdot s, rh, eh \rangle \xrightarrow{e}_d \langle sh \cdot s \cdot s', rh \cdot return(s), eh \cdot e \rangle}$$

R3. $$\frac{s \xrightarrow{e}_c s', reuse(e), top(rh, j) = \bot}{\langle sh \cdot s, rh, eh \rangle \xrightarrow{e}_d \langle sh \cdot s \cdot s', rh \cdot return(s), eh \cdot e \rangle}$$

R4. $$\frac{s \xrightarrow{e}_c s', reuse(e), top(r_1 \cdots r_m, j) = r_k}{\langle sh \cdot s, r_1 \cdots r_m, e_1 \cdots e_m \rangle \xrightarrow{e}_d \langle sh \cdot s \cdot r_k, r_1 \cdots r_{k-1}, e_1 \cdots e_{k-1} \rangle}$$

R5. $$\frac{s \in F, auto\_return(e)}{\langle sh \cdot s, rh \cdot r_m, eh \cdot e \rangle \xrightarrow{-}_d \langle sh \cdot s \cdot r_m, rh, eh \rangle}$$

La regla **R1** define que una transición en una misma máquina de estados de

vista M_i es una transición en la máquina de estados de dispositivo. El nuevo estado s' se añade a la lista de estados visitados $sh \cdot s$. Las reglas **R2, R3** modelan una transición desde una máquina a otra nueva (M_j, para algún índice j) cuando tanto el nuevo estado s' como el evento e se concatenan a las pilas de estados y eventos. La regla **R2** se aplica cuando el evento e indica que no se deber reutilizar ninguna vista previamente creada ($reuse(e)$ es falso). **R3** se aplica cuando el evento dice que debe reutilizarse una vista ya creada ($reuse(e)$ es cierto) pero la pila actual de las vistas creadas no contiene ninguna estado de M_j ($top(r_1 \cdots r_m, j) = \perp$). La regla **R4** define una transición a M_j reutilizando una vista creada anteriormente ($reuse(e)$ es verdadero) y hay un estado de M_j que está en rh ($top(r_1 \cdots r_m, j) = r_k$). Finalmente, **R5** define el caso de un flujo que ha llegado a un estado final de una máquina de estados de vista y, la ejecución debe continuar con el estado r_m almacenado en la cima de la secuencia de vistas $rh \cdot r_m$. En otro caso, si $auto\_return(e)$ devuelve falso, la configuración actual $\langle sh, rh, eh \rangle$ no puede evolucionar.

Dadas una *máquina de estados de dispositivo* $\mathcal{D} = \bigcup_{i=1}^{n} M_i \bigoplus \mathcal{C}$, la *semántica basada en trazas* determinada por $\mathcal{D}$ ($\mathcal{O}(\mathcal{D})$) viene dada por el conjunto de secuencias finitas/infinitas de configuraciones (flujos) producidas por la relación de transición $\xrightarrow{-}_d$ desde un estado inicial, es decir, $\mathcal{O}(\mathcal{D}) = \{\langle s_0, \epsilon, \epsilon \rangle \xrightarrow{e_0}_d \langle s_0 \cdot s_1, rh_1, eh_1 \rangle \cdots | s_0 \in I\}$.

***Definición* 8.5** *Dado un flujo* $\phi = c_0 \xrightarrow{e_1}_d c_1 \xrightarrow{e_2}_d c_2 \cdots \in \mathcal{O}(\mathcal{D})$, *el caso de test determinado por* ϕ *es la secuencia de eventos* $test(\phi) = e_1 \cdot e_2 \cdots$. *El conjunto de* casos de tests *determinado por un conjunto de flujos* $\mathcal{T}$ *se denota como* $TC(\mathcal{T}) = \{test(t) | t \in \mathcal{T}\}$.

Así, un flujo $\phi \in \mathcal{O}(\mathcal{D})$ consiste en una secuencia de flujos de máquinas de estados vistas (Definición 8.2) conectados a través de *estados de conexión*. Un flujo ϕ puede terminar en un estado final de alguna máquina de estados de vistas, o puede ser infinito. La *longitud* $|\phi|$ de un flujo ϕ es al numero de estados (configuraciones), si es finito, o ∞, en otro caso. Dado un flujo $\phi = c_0 \xrightarrow{e_1}_d c_1 \xrightarrow{e_2}_d c_2 \cdots \in \mathcal{O}(\mathcal{D})$, el flujo truncado de longitud n, ϕ^n, se define como ϕ si $|\phi| \leq n$ o $\phi^n = c_0 \xrightarrow{e_1}_d c_1 \xrightarrow{e_2}_d c_2 \cdots \xrightarrow{e_{n-1}}_d c_{n-1}$, en otro caso. Considerando esto, se define el conjunto de trazas $\mathcal{O}^n(\mathcal{D})$ como el conjunto de todas los flujos de $\mathcal{O}(\mathcal{D})$ truncadas a la longitud n, es decir, $\mathcal{O}^n(\mathcal{D}) = \{\phi^n | \phi \in \mathcal{O}(\mathcal{D})\}$.

Es importante observar, que el espacio de estados de las máquinas de estado de dispositivo no es finito debido a que las configuraciones incluyen, además del estado actual, las pilas de estados y eventos las que pueden tener longitudes arbitrarias. Además, el espacio de estados también es infinito cuando se construyen todos los flujos definidos por una máquina de estados de dispositivo utilizando un *model checker* explícito. Esto no es sólo debido a las historias de estados y eventos que están en las configuraciones, sino también porque el algoritmo de matching, aplicado durante la generación espacio de estados debe tener en cuenta tanto el estado actual del flujo como la historia de los estados que están en la primera componente de cada configuración. Esto permite que, por ejemplo, los

flujos $\phi_1 = \langle s_0, \epsilon, \epsilon\rangle \xrightarrow{e_1^+}_d \langle s_0 \cdot s_1, \epsilon, \epsilon\rangle \xrightarrow{e_2^+}_d \langle s_0 \cdot s_1 \cdot s_2, \epsilon, \epsilon\rangle \xrightarrow{e_3^+}_d \langle s_0 \cdot s_1 \cdot s_2 \cdot s_3, \epsilon, \epsilon\rangle$ y $\phi_2 = \langle s_0, \epsilon, \epsilon\rangle \xrightarrow{e_4^+}_d \langle s_0 \cdot s_4, \epsilon, \epsilon\rangle \xrightarrow{e_1^+}_d \langle s_0 \cdot s_4 \cdot s_1, \epsilon, \epsilon\rangle \xrightarrow{e_2^+}_d \langle s_0 \cdot s_4 \cdot s_1 \cdot s_2, \epsilon, \epsilon\rangle \xrightarrow{e_3^+}_d \langle s_0 \cdot s_4 \cdot s_1 \cdot s_2 \cdot s_3, \epsilon, \epsilon\rangle$ puedan ambos ser generados por el *model checker*, incluso aunque ambos flujos visitan los estados s_1, s_2 y s_3, el camino recorrido por cada flujo es distinto.

En consecuencia, los modelos de máquinas de estados de dispositivo no son, en general, de estados finitos; esto significa que el proceso de *model checking* no termina. En la implementación actual se ha resuelto este problema limitando la profundidad de la ejecución de los flujos analizados y generando $\mathcal{O}^n(\mathcal{D})$ para algún n fijo.

La cobertura de los casos de test generados depende de la longitud n de los mismos. Si un estado es alcanzable en n transiciones a partir del estado actual, entonces el *model checker* lo visitará. Aumentar el valor de n mejorará la cobertura siempre que no se exceda la capacidad de la máquina.

Componiendo varios dispositivos

La extensión del modelo de máquinas de estados a varios dispositivos se lleva a cabo mediante la composición entrelazada de las máquinas de estados de dispositivos. Así, si $c_0 \xrightarrow{e_1}_d c_1$ y $c_0' \xrightarrow{e_1'}_{d'} c_1'$ son transiciones de los dispositivos $\mathcal{D}$ y $\mathcal{D}'$, respectivamente, entonces $\langle c_0, c_0'\rangle \xrightarrow{e_1}_{d||d'} \langle c_1, c_0'\rangle$ y $\langle c_0, c_0'\rangle \xrightarrow{e_1'}_{d||d'} \langle c_0, c_1'\rangle$ son transiciones de la composición entrelazada de $\mathcal{D}$ y $\mathcal{D}'$.

La comunicación entre ambos dispositivos se modela mediante un evento de usuario enviado por el dispositivo que inicia la comunicación y un evento de sistema en el dispositivo receptor (el que espera el mensaje). Por lo tanto, utilizando el ejemplo anterior, si $e_1 = e_1^+$ es un evento que implica una comunicación desde $\mathcal{D}$ hasta $\mathcal{D}'$, y $e_1' = e_1^-$ es el evento correspondiente que sera recibido por $\mathcal{D}'$ desde $\mathcal{D}$, se generarían los casos de test $e_1^+ \cdot e_1^-$ y $e_1^- \cdot e_1^+$. Debe notarse que en el segundo caso de prueba, el método que implementa la transición de la recepción del evento debe suspender la ejecución de $\mathcal{D}'$ hasta que el evento e_1^+ sea disparado por $\mathcal{D}$.

Cuando se trabaja con más de un dispositivo, se hacen uso de técnicas de *optimización* de model checking tales como la reducción orden parcial [45], que permiten evitar la generación de diferentes casos de prueba que correspondan a una misma interacción entre los dispositivos.

Capítulo 9

Verificación en Tiempo de Ejecución

9.1. Lenguaje de especificación para propiedades extra-funcionales

En esta sección se introduce el lenguaje (basado en la lógica de intervalos) utilizado para la descrición de propiedades extra-funcionales sobre las trazas de ejecución. Por lo general, las técnicas de verificación como el *model checking*, tal y como se explica en el Capítulo 5, evalúan las propiedades sobre las trazas abstrayendo el *tiempo real* en el que ha ocurrido un estado. Esta abstracción es adecuada, por ejemplo, para analizar las propiedades funcionales (seguridad y viveza). Sin embargo, el análisis de algunas propiedades extra-funcionales, como el consumo de energía, requiere tener en cuenta los valores de algunas variables no discretas que *evolucionan con el tiempo*. Por ejemplo, para analizar cuánta energía es consumida por un dispositivo cuando se descarga un archivo, se deben detectar los estados de las trazas en los que comienza y termina de descargar el archivo, para poder medir la energía consumida por el dispositivo durante este período.

En primer lugar, se presentan las tres fórmulas de intervalos que constituyen nuestro simple lenguaje de especificación. En estas fórmulas existe una *sincronización implícita* entre la evolución discreta de las trazas y la evolución continua de los valores de las magnitudes que se evaluan sobre las trazas. Inicialmente, se describe de manera general el significado de cada una de las fórmulas de intervalo, y luego se formalizan. Finalmente, se describe la traducción de estas fórmulas de intervalo al lenguaje (LTL) (descrito en la Sección 5.2.3) de manera que puedan ser evaluar automáticamente sobre *model checkers* como SPIN.

9.2. Formulas de intervalos

Sea $\mathcal{O}(P)$ el conjunto de trazas de ejecución determinado por un sistema de transición $P = \langle \Sigma, \overline{\longmapsto}, \mathcal{L}, s_0 \rangle$. Las trazas son secuencias de estados de la forma $\pi = s_0 \longmapsto s_1 \longmapsto \cdots$[1]. Es necesario notar que, en esta sección, utilizamos la frase *trazas de ejecución*, o simplemente el término *traza*, para denotar secuencias de estados producidos por un sistema de transiciones lo que es suficiente para describir la sintaxis y la semántica de las fórmulas de intervalos. En cualquier caso, en las siguientes secciones, las trazas de ejecución serán secuencias de estados reales generadas a partir de los casos de prueba que se ejecutan sobre los dispositivos móviles.

Sea $\mathcal{F}$ un conjunto de fórmulas de estado que pueden ser evaluados sobre los estados de Σ. La relación $\models \subseteq \Sigma \times \mathcal{F}$ asocia cada estado con las fórmulas de estado que satisface, es decir, $s \in \Sigma$ y $p \in \mathcal{F}$, $s \models p$ sii el estado s satisface la fórmula p. Como es habitual, se supone que las fórmulas de estado están construidas a partir de un conjunto de proposiciones atómicas y operadores *booleanos*.

Sea un conjunto de variables $\mathcal{D}$ que representan magnitudes continuas que pueden ser analizadas sobre las trazas de ejecución. Cada $c \in \mathcal{D}$ es una función real $c : \mathbb{R}_{\geq 0} \to \mathbb{R}$ que define la evolución de c en el tiempo. De esta manera, $c(t) \in \mathbb{R}$ proporciona el valor de c en el instante de tiempo t. Las fórmulas de estados se utilizan para definir intervalos de estados sobre los que se es necesario medir variables continuas. Por ejemplo, las fórmulas de estado $wifi\_on$, $wifi\_off$ pueden servirnos para detectar los estados de las traza en los que se activa y se desactiva el *wifi*.

Con esta idea, se propone un lenguaje sencillo para la especificación de propiedades extra-funcionales que se pueda utilizar para describir los intervalos de estados durante los cuales las variables continuas de interés evolucionan en el tiempo y deben ser monitorizadas.

Dados $p, q \in \mathcal{F}$, $c \in \mathcal{D}$ y $K \in \mathbb{R}_{\geq 0}$, definimos tres fórmulas válidas en el lenguaje de especificación: $[[diff\_c \leq K]]_{[p,q]}$, $\forall[[diff\_c \leq K]]_{[p,q]}$, y $\exists[[diff\_c \leq K]]_{[p,q]}$. La semántica de $\forall[[diff\_c \leq K]]_{[p,q]}$ es la siguiente. Una ejecución de una traza $\pi = s_0 \longmapsto \cdots$ satisface $\forall[[diff\_c \leq K]]_{[p,q]}$ sii para cada par de estados s_i, s_j de π, con $i \leq j$ y $s_i \models p, s_j \models q$, la siguiente condición ($\mathcal{C}ond$) es cierta: "*la diferencia entre dos valores de variable c en los instantes de tiempo cuando s_i y s_j ocurren es menor o igual a K*". Es decir, se utilizan p y q para determinar el intervalo de la traza $s_i \longmapsto \cdots \longmapsto s_j$ sobre el cual la evolución de la variable c tiene que ser observada.

Las otras dos formulas tienen una semántica similar excepto que $\exists[[diff\_c \leq K]]_{[p,q]}/[[diff\_c \leq K]]_{[p,q]}$ se satisface si la condición $\mathcal{C}ond$ es cierta para algún/el primer par de estados s_i, s_j con $i \leq j$, y $s_i \models p, s_j \models q$.

Por ejemplo, si la variable c representa la energía consumida por un dispositivo, la fórmula $\forall[[diff\_c \leq k]]_{[wifi\_on, wifi\_off]}$ es cierta sobre una traza π si cada vez que se activa el *wifi*, la energía consumida es menor o igual a K.

[1] Puesto que no necesitamos utilizar etiquetas, las eliminamos de la relación de transición

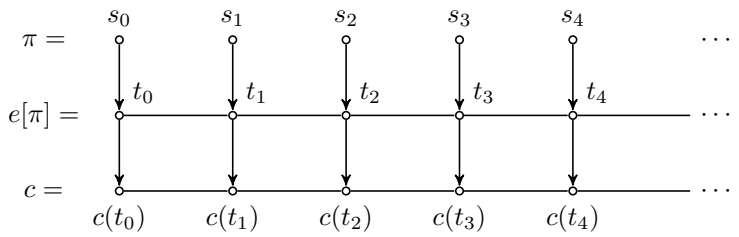


Figura 9.1: Sincronización de la traza π y sus variables continuas c utilizando $e[\pi]$

9.3. Semántica de las formulas de intervalos

Se asume que las trazas $\pi = s_0 \longmapsto \cdots$ se describen com aplicaciones $\pi : \mathbb{N} \to \Sigma$ que asocian cada numero natural con el estado correspondiente de la traza, es decir, $\pi(i) = s_i$. Dado que las trazas generadas como casos de tests son *finitas*, para hacerlas infinitas suponemos que cada traza π tiene un *estado final o* que se repite de manera indefinida. Por lo tanto, dada una traza π existe un numero natural $n > 0$ (la longitud de la traza se denota como $length(\pi)$) tal que (1) $\pi(n-1) \neq o$ [2], y (2) $\forall k \geq n.\pi(k) = o$.

Aunque el tiempo de ejecución se abstrae en la semántica operacional, está claro que la ejecución de cada traza lleva tiempo y que, durante este tiempo, pueden ocurrir muchas otras cosas que influyen o se ven afectadas por la ejecución de las trazas. La siguiente definición hace explícito el tiempo durante el que se ejecutan trazas.

***Definición* 9.1** *Dada una traza* $\pi \in \mathcal{O}(P)$, una ejecución e de π *es una función* $e[\pi] : \mathbb{N} \to \mathbb{R}_{\geq 0}$ *que asocia cada estado* $\pi(i)$ *de* π *con el instante de tiempo* $e[\pi](i) \in \mathbb{R}_{\geq 0}$ *en el que ocurre.*[3][4]

Observa que cada traza π puede tener múltiples ejecuciones $e[\pi]$, cada una de las cuales relaciona cada estado con diferentes instantes de tiempo. Una vez que la ejecución de una traza $e[\pi]$ ha sido fijada, podemos observar los valores de las magnitudes continuas que evolucionan de manera síncrona con la traza. El diagrama en la Figura 9.1 ilustra esta sincronización. Las filas superiores muestran los estados de la traza $\pi = s_0 \longmapsto \cdots$. En la fila del medio se muestra el paso del tiempo, cada estado s_i está asociado mediante $e[\pi]$ con el instante de tiempo en el que ha ocurrido. Finalmente, la fila inferior muestra la evolución de las variables continuas c en el tiempo y su sincronización con $e[\pi]$.

[2] Para simplificar la presentación, se asume que las trazas tienen al menos un estado que no es final s_0

[3] Se asume que $e[\pi](i)$ representa el instante de tiempo en el que se crea el estado s_i

[4] Se asume que si π es una traza de longitud n, $e[\pi]$ asocia a los estados finales de π con el instante de tiempo en el que ocurrió el último estado de la traza, es decir, $\forall k \geq n.e[\pi](k) = e[\pi](n-1)$.

Utilizamos intervalos de estados dentro de las trazas para determinar los períodos de tiempo durante los cuales deben observarse variables continuas. Para lograr esto, hacemos uso del cálculo de intervalos introducido por [46] que nos permite dar semántica formal al lenguaje de las propiedades extra-funcionales presentado en la Sección 9.1. El dominio de la lógica de intervalos es el conjunto de intervalos de tiempo $\mathbb{I}ntv$ definido como $\{[t_1, t_2] | t_1, t_2 \in \mathbb{R}, t_1 \leq t_2\}$. Una *variable de intervalo* v es una función $v : \mathbb{I}ntv \to \mathbb{R}$ que asocia cada intervalo con un número real. Por ejemplo, podemos usar una variable continua $c : \mathbb{R}_{\geq 0} \to \mathbb{R}$ para definir variables de intervalo, como $\mathit{diff}\_c : \mathbb{I}ntv \to \mathbb{R}$ definido como $\mathit{diff}\_c([t_1, t_2]) = c(t_2) - c(t_1)$.

Se pueden construir *expresiones de intervalo* que describen propiedades sobre intervalos utilizando un conjunto de variables de intervalo, operadores relacionales, *booleanos* y constantes de tipo real. Por ejemplo si K es una constante, $\mathit{diff}\_c \leq K : \mathbb{I}ntv \to \{\mathit{true}, \mathit{false}\}$ define la propiedad sobre los intervalos de tiempo $[t_1, t_2]$ que es cierta sii $c(t_2) - c(t_1) \leq K$.

Dada una traza π y un intervalo de números naturales $[i, j]$ denotamos con $\pi \downarrow [i, j]$ la sub-traza de π desde el estado $\pi(i)$ al $\pi(j)$. De forma similar, dada e una ejecución de π, $e[\pi] \downarrow [i, j]$ representa el intervalo de tiempo $[e[\pi](i), e[\pi](j)]$ desde el instante en que se ha creado el estado $\pi(i)$ al instante en el que se ha creado $\pi(j)$ en la ejecución ejecución $e[\pi]$. De este modo, los intervalos de estado y las ejecuciones de las trazas proporcionan intervalos de tiempo sobre los cuales se pueden evaluar expresiones del tipo $\mathit{diff}\_c \leq K$.

A continuación, se explica cómo se utilizan *las formulas de estados* para construir intervalos de estados. Dado que el conjunto de formulas $\mathcal{F}$, definido en la Sección 9.2, llamamos *intervalos de fórmulas* a expresiones de la forma $[p, q]$ donde $p, q \in \mathcal{F}$. Teniendo en cuenta esto, la siguiente definición extiende la relación de satisfacción $\models$ sobre intervalos de estados.

***Definición* 9.2** *Dada una traza π y un intervalo de números naturales $I = [i, j]$ con $i \leq j$, se dice que el intervalo de estados $\pi \downarrow I$ satisface $[p, q]$, y se escribe como $\pi \downarrow I \models [p, q]$, si se satisfacen las siguientes condiciones:*

1. $\pi(i) \models p$
2. $\pi(j) \models q$
3. $\forall i < k < j . \pi(k) \not\models q$

es decir, $[i, j]$ es un intervalo de estados de π tal que $\pi(i)$ satisface p, y $\pi(j)$ es el primer estado después de $\pi(i)$ que satisface q.

A partir de ahora, se asumirá que el estado final de las trazas o no satisface ninguna formula de $\mathcal{F}$, es decir, $\forall p \in \mathcal{F}.\ o \not\models p$.

Dada una traza π y una fórmula de intervalos $[p, q]$, $\pi \Downarrow [p, q]$ denota la secuencia finita de intervalos de estados de π, $I_0 \cdot I_1 \cdots I_{m-1}$, que satisfacen $[p, q]$ en el sentido de la definición anterior, es decir, $\forall 0 \leq i < m.\ \pi \downarrow I_i \models [p, q]$.

Las dos definiciones siguientes muestran cómo se puede construir la secuencia de intervalos $\pi \Downarrow [p, q]$.

***Definición* 9.3** *Dada p una formula de estados, una traza finita π de tamaño n, y $k \geq 0$, $\pi \downarrow_k p$ es el primer estado de π que ocurre después de $\pi(k)$ (incluyéndolo) y que satisface p, si existe, o ∞, en otro caso. Podemos definir $\pi \downarrow_k p$ inductivamente del modo siguiente:*

1. $\pi \downarrow_k p = k$, *sii* $\pi(k) \models p$
2. $\pi \downarrow_k p = \pi \downarrow_{k+1} p$ *sii* $k < n, \pi(k) \not\models p$
3. $\pi \downarrow_k p = \infty$ *sii* $k \geq n$

***Definición* 9.4** *Dada una traza finita π, y dos fórmulas de estado p, q, la secuencia de intervalos de estados determinada por p, q, $\pi \Downarrow [p, q]$ se puede obtener de manera inductiva a partir del operador $\Downarrow_k$ con $k \geq 0$ que se define como:*

1. $\pi \Downarrow_k [p, q] = \epsilon \iff \pi \downarrow_k p = \infty$, *o* $\pi \downarrow_k p = j \wedge \pi \downarrow_{j+1} q = \infty$ [5]
2. $\pi \Downarrow_k [p, q] = [j, l] \cdot (\pi \Downarrow_{l+1} [p, q]) \iff \pi \downarrow_k p = j \wedge \pi \downarrow_{j+1} q = l$

Se define $\pi \Downarrow [p, q]$ como $\pi \Downarrow_0 [p, q]$.

De este modo, dos fórmulas de estados $p, q \in \mathcal{F}$ determinan una secuencia de intervalos de estados $\pi \Downarrow [p, q] = I_0 \cdots I_{m-1}$ de π que satisfacen $[p, q]$. Esta definición se puede extender a ejecuciones e de las trazas π de la forma natural: $e[\pi] \Downarrow [p, q] = e[\pi] \downarrow I_0 \cdots e[\pi] \downarrow I_{m-1}$. Esta definición la utilizamos para dar semánticas a las tres fórmulas del lenguaje de especificación presentado en la Sección 9.2.

La siguiente definición establece cuándo una ejecución e de una traza π satisface una expresión de intervalo Φ del tipo *diff* $\_\ c \leq K$.

***Definición* 9.5** *Sean Φ y $[p, q]$ una expresión de intervalo y un intervalo de fórmula, respectivamente. Sea e una ejecución finita de la traza π. Entonces*

1. *Se dice que $e[\pi]$ satisface Φ sobre el intervalo de tiempo determinado por $[p, q]$, y se denota como $e[\pi] \models [[\Phi]]_{[p,q]}$ sii $e[\pi] \Downarrow [p, q] = T_0 \cdots T_{m-1}$ con $m > 0$ y $\Phi(T_0)$ es cierto.*
2. *Se dice que $e[\pi]$ satisface $\exists\Phi$ sobre el intervalo de tiempo determinado por $[p, q]$, y se denota como $e[\pi] \models \exists[[\Phi]]_{[p,q]}$ sii $e[\pi] \Downarrow [p, q] = T_0 \cdots T_{m-1}$ con $m > 0$ y es cierto que $\exists 0 \leq i < m.\Phi(T_i)$.*
3. *Se dice que $e[\pi]$ satisface $\forall\Phi$ sobre el intervalo de tiempo determinado por $[p, q]$, y se denota como $e[\pi] \models \forall[[\Phi]]_{[p,q]}$ sii $e[\pi] \Downarrow [p, q] = T_0 \cdots T_{m-1}$ con $m > 0$ y se satisface que $\forall 0 \leq i < m.\Phi(T_i)$.*

[5] ϵ representa la secuencia vacía.

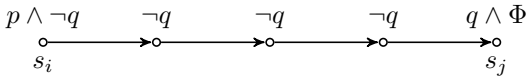
Es decir, una ejecución e de la traza π satisface la formula (1) $[[\Phi]]_{[p,q]}$ sii el primer intervalo de tiempo determinado por $\pi \Downarrow [p,q]$ y e satisface Φ, (2) $\forall[[\Phi]]_{[p,q]}$ sii todos los intervalos de tiempo definidos por $\pi \Downarrow [p,q]$ y e satisfacen Φ; (3) $\exists[[\Phi]]_{[p,q]}$ sii existe un intervalo de tiempo en la secuencia $e[\pi] \Downarrow [p,q]$ que satisface Φ. Debe recordarse que si $I = [i,j]$ es un intervalo de estados de π, $e(\pi)(I)$ es el intervalo de tiempo $[e(\pi)(s_i), e(\pi)(s_j)]$. Por ejemplo, la fórmula $\Phi = diff\_c \leq K$, $[[\Phi]]_{[swifi,ewifi]}$ establece que el intervalo de tiempo determinado por el primer intervalo de estados que satisface $[swifi, ewifi]$ debe satisfacer Φ.

9.3.1. De las propiedades de intervalo a *LTL*

En esta sección, se discute cómo pueden evaluarse de forma práctica las propiedades de intervalo sobre las trazas de ejecución. Cada tipo de propiedad de intervalo puede escribirse como una fórmula que es la que se proporciona al *model checker*. El objetivo de la sección es dar una semántica precisa de la propiedades introducidas a través de la semántica LTL y, al mismo tiempo, mostrar una posible implementación de las mismas utilizando su descripción LTL. Con el propósito de simplificar la representación de las fórmulas, dadas dos fórmulas de estados p y q, se define $\Phi(p,q)$ como:

$$\Phi(p,q) \equiv p \wedge (\neg q \, U \, (q \wedge \Phi))$$

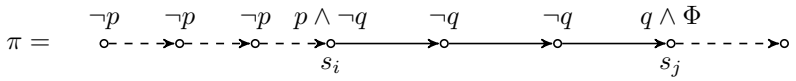
Intuitivamente, $\Phi(p,q)$ es la representación *LTL* de la propiedad: *"p es cierto en el estado actual, q será verdadero en un estado futuro y, en ese momento, el intervalo de tiempo determinado por p y q satisfará Φ"* tal y como ilustra el siguiente diagrama :



Para las propiedades del tipo $[[\Phi]]_{[p,q]}$, utilizamos la especificación *LTL*:

$$[[\Phi]]_{[p,q]} \equiv (\neg p) \; U \; \Phi(p,q) \tag{9.1}$$

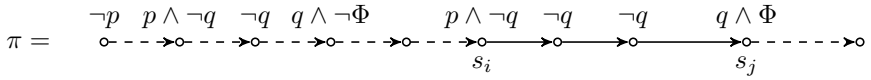
A continuación, se describe el significado de esta fórmula. En primer lugar, se busca el primer estado (s_i) que satisface p; a continuación, se busca por el primer estado que sigue a s_i que satisface q (s_j). Estos dos estados s_i y s_j determinan un intervalo de tiempo. Si Φ es cierto para este intervalo, entonces la fórmula $[[\Phi]]_{[p,q]}$ se satisface. En otro caso, es decir,, si no es posible encontrar ningún estado como s_i o s_j, o si el intervalo de tiempo que determinan no satisface Φ, la formula es falsa. La siguiente secuencia de estados muestra una traza que satisface $[[\Phi]]_{[p,q]}$. En el dibujo, utilizamos flechas sólidas para mostrar el intervalo sobre el que Φ es cierto.



Para la propiedad $\exists\,[[\Phi]]_{[p,q]}$, se utiliza la especificación *LTL*:

$$\exists\,[[\Phi]]_{[p,q]} \equiv \Phi(p,q) \vee \Diamond(\neg p \wedge \circ\,\Phi(p,q)) \tag{9.2}$$

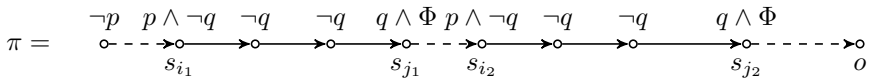
es decir, $\Phi(p,q)$ debe ser cierto o bien en el primer estado o bien en algún otro estado en el futuro. Hay que tener en cuenta que el $\circ$ es el operador "siguiente" (*next*) de la lógica *LTL*. En este caso es seguro utilizar este operador porque no estamos analizando programas concurrentes, sino trazas de ejecución en las que cada estado tiene un único posible estado siguiente. El uso de $\circ$ es necesario para asegurar que la formula $\Phi(p,q)$ se evalúa sobre el intervalo de tiempo mayor determinado por p y q, es decir, el estado que satisface p, si no es el estado inicial, debe estar precedido por un estado que no satisface p. La siguiente secuencia muestra un ejemplo de una traza en la que $\Phi(p,q)$ se satisface sobre el segundo intervalo de tiempo determinado por p and q.



Finalmente, la fórmula *LTL* para $\forall\,[[\Phi]]_{[p,q]}$ viene dada por:

$$\forall\,[[\Phi]]_{[p,q]} \equiv p \rightarrow \Phi(p,q) \wedge \Box((\neg p \wedge \circ p) \rightarrow \circ\,\Phi(p,q)) \tag{9.3}$$

entonces, todos los intervalos maximales determinados por $[p,q]$ deben satisfacer Φ en el instante de tiempo en el que se crea el segundo estado. La siguiente secuencia muestra una traza con dos intervalos de tiempo, determinados por p y q, para los Φ es verdadera. Debe notarse que el último estado está etiquetado con el símbolo o para indicar que la traza no contiene ningún intervalo de estados después de $[s_{i_2}, s_{j_2}]$ que satisface $[p,q]$.



Capítulo 10

Implementación

Esta sección describe la implementación de la propuesta para el análisis de aplicaciones móviles. Aunque la arquitectura de la herramienta desarrollada puede adaptarse a diferentes sistemas operativos para móviles, la implementación actual se ha centrado en el análisis de *Apps* ANDROID. La Figura 10.1 describe los principales componentes la implementación, que como se puede observar, se compone fundamentalmente de dos grandes módulos denominados *Model Based Test Generator* y *Runtime Verfication*.

Como ya se ha discutido en las Secciones 8.1 y 9.1, en la implementación se ha utilizado la técnica de model checking y, en particular, el *model checker* de SPIN, como el método formal subyacente del sistema (ver Figura 10.1): se utiliza, por un lado, para generar los casos de test cuyo comportamiento se analiza y, por otro lado, para realizar la verificación automática de propiedades extra-funcionales, como es el caso del consumo de energía, sobre estos casos de prueba.

10.1. Model-based test generator

La primera parte del proceso de análisis consiste en la generación y ejecución de casos de prueba, partiendo de un modelo del comportamiento del usuario. A continuación, se describen de forma detallada los pasos y componentes individuales de este proceso, que aparecen en el lado izquierdo de la Figura 10.1.

El proceso comienza con un modelo del comportamiento del usuario (*Modeler*) de las *Apps* que se están analizando, el cual es creado por el programador/*tester*. El modelo, que se construye utilizando el lenguaje descrito en la Sección 8.1.2, está compuesto por varias máquinas de estado anidadas, cada una de ellas representando un conjunto de posibles comportamientos del usuario en cada pantalla de la aplicación. El *modelo de comportamiento de usuario*(UBM) se almacena en formato XML (el fichero `model.xml` de la figura) siguiendo el XML Schema que se muestra en la Figura 10.2. Como puede observarse, el esquema permite la definición de varias aplicaciones. Cada aplicación está compuesta

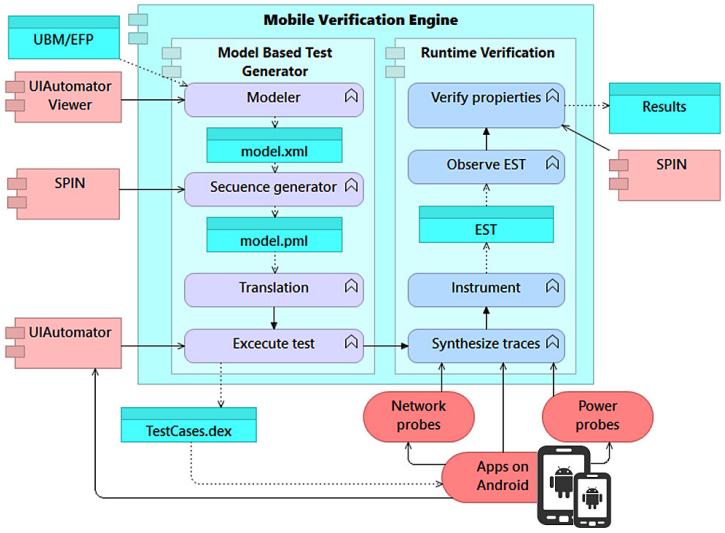


Figura 10.1: Generación de casos de prueba e implementación de verificación en tiempo de ejecución para ANDROID

por varias vistas, que corresponden a sus pantallas, y cada vista puede definir más de una máquina de estado. Las transiciones simples se definen entre estados de la misma máquina de estados, mientras que las transiciones complejas definen estados de conexión, que permiten que el flujo de ejecución transite a otra máquina de estado. Estas transiciones complejas pueden continuar en una máquina de estados concreta, o en cualquiera de las máquinas de estados contenidas en una vista. Finalmente, las transiciones se etiquetan con las *acciones de usuario* que corresponden a las interacciones con los controles presentes de la pantalla que se está modelando, como puede ser, por ejemplo, pulsar un botón. Para relacionar estas etiquetas con los controles reales de una pantalla ANDROID utilizamos, por un lado, la herramienta UIAUTOMATORVIEWER [41], que proporciona una descripción XML de una pantalla determinada del dispositivo. Por otro lado, este archivo XML se enriquece agregándole nuevos atributos que permiten enlazar los controles con las etiquetas en las transiciones. Además, los controles de *entrada de texto* pueden tener atributos adicionales para definir el texto concreto que se introducirá durante la ejecución del caso de prueba. El atributo *time* de una transición indica el tiempo que se necesita para ejecutarla.

El siguiente paso consiste en la exploración exhaustiva del modelo para generar todas las posibles secuencias de eventos del usuario (*Sequence generation*), para lo que utilizamos la potencia del *model checker* SPIN [45]. La herramienta SPIN se puede utilizar para verificar si modelos de software concurrente, descritos en el lenguaje de modelado PROMELA, son correctos con respecto a especificaciones LTL. Aunque esta herramienta se ha utilizado, fundamentalmente, para el diseño y validación de protocolos, también se puede utilizar en otras áreas di-

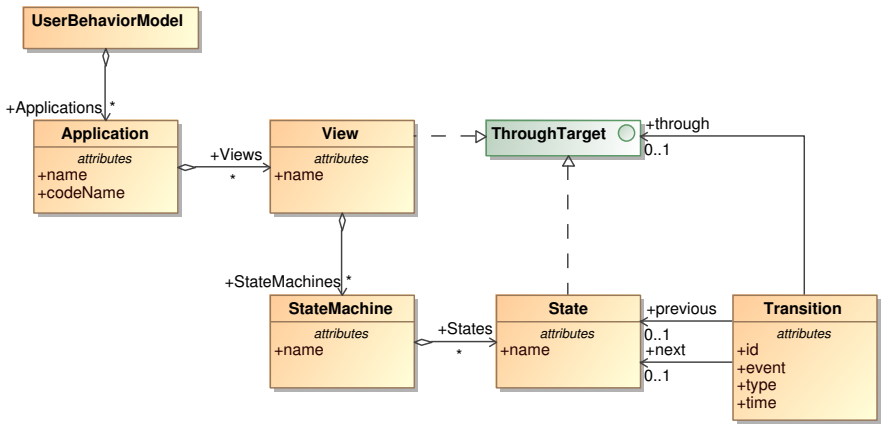


Figura 10.2: Modelo UML del XML Schema para el modelo de comportamiento de usuario

ferentes. SPIN es capaz de analizar la ocurrencia de una propiedad sobre todas las posibles ejecuciones de la especificación de un sistema y, además, proporciona contra-ejemplos cuando encuentra violaciones a la propiedad. En nuestro caso, utilizamos SPIN para realizar una exploración exhaustiva del modelo de las *Apps*, traduciendo la representación XML a una especificación PROMELA. Cada dispositivo se representa por un proceso PROMELA que contiene todas las máquinas de estado de ese dispositivo. Sin embargo, aunque el modelo de la aplicación está compuesto de máquinas de estado anidadas, el código PROMELA genera en un único bucle no determinista para cada dispositivo. Cada rama del bucle se corresponde con una transición en el modelo. Se utiliza una variable global para cada dispositivo de manera que sea posible conocer su estado en cada momento y, de esta forma, se pueda decidir qué transiciones se pueden tomar a continuación.

Para implementar los estados de conexión, cada dispositivo tiene una estructura de pila asociada, denominada *backstack*, que lleva el registro de las transiciones ejecutadas en el pasado que han llevado a estados de conexión. Cuando se termina la ejecución de una máquina de estado, se analiza la cima de la pila *backstack*. Si contiene un estado de conexión, esa información se puede utilizar para transitar al siguiente estado de la máquina de estados a la que pertenece el estado de conexión. Si la pila está vacía, la ejecución del modelo termina.

Este modelo es explorado por SPIN utilizando una búsqueda primero en profundidad, durante la cual se almacena cada transición ejecutada. Cuando se alcanza un estado final válido, la secuencia de transiciones registradas durante la exploración contiene los posibles eventos de un usuario lo que constituye un caso de prueba. Si, en algún momento, se puede tomar más de una transición, SPIN explora primero uno de ellos y luego hace backtracking para explorar el resto

```
1  typedef Backstack{mtype states[MAXB]; Transition trans[MAXB]; short index};
2
3  #define curBackstack    devices[device].backstack
4  #define curState    curBackstack.states[curBackstack.index]
5
6  proctype device_4107a7166c03af9b(int device) {
7      do
8      // Spotify - PlayView - PlayStateMachine
9      :: starting || curState == Spotify_PlayView_PlayStateMachine_init ->
10         pushToBackstack(device, Spotify_PlayView_PlayStateMachine_init);
11         transition(device, VIEW_PlayView, 1);      // Swipe
12         curState = Spotify_PlayView_PlayStateMachine_S0
13     :: !starting && curState == Spotify_PlayView_PlayStateMachine_S0 ->
14         transition(device, VIEW_PlayView, 2);      // clicPrincipalMenu
15         curState = Spotify_PrincipalView_PrincipalStateMachine_init
16     // Spotify - PrincipalView - PrincipalStateMachine
17     :: !starting && curState ==
           Spotify_PrincipalView_PrincipalStateMachine_init ->
18         pushToBackstack(device,
       Spotify_PrincipalView_PrincipalStateMachine_init);
19         transition(device, VIEW_PrincipalView, 1); // Swipe
20         curState = Spotify_PrincipalView_PrincipalStateMachine_S0
21     :: !starting && curState ==
           Spotify_PrincipalView_PrincipalStateMachine_end ->
22         popFromBackstack(device);
23         continueTransition_4107a7166c03af9b(device)
24     od;
25 }
```

Código 10.1: Extracto del modelo PROMELA para la generación de casos de prueba

de las alternativas. De esta forma, el resultado de la exploración es el conjunto de todos los posibles casos de prueba definidos por el modelo.

El Código 10.1 muestra parte de la especificación PROMELA que se ha generado a partir del modelo de la Figura 12.4(a) de la Sección 8.1.2. En el listado se muestran distintos extractos de dos máquinas de estados. El proceso que comienza en la línea 6 modela todas las máquinas de estado contenidas en un dispositivo. La función `transition` (por ejemplo, line 11) registra cada transición ejecutada y actualiza la cima de la pila *backstack*. Para entrar en una nueva máquina de estados desde un estado de conexión, se guarda primero su estado inicial (línea 15) y, a continuación, se agrega un nuevo elemento a la cima de *backstack* (línea 18). Cuando se sale de una máquina de estados, se extrae el estado de la cima de la *backstack* (línea 22), y se analiza para restaurar el estado que sigue de la máquina de estados anterior (line 23).

Durante la fase de traducción (*Translation*), cada secuencia de eventos se traduce a un programa JAVA que puede desplegarse en un dispositivo ANDROID para ejecutar cada caso de prueba. Para esta traducción, se utiliza la API UIAUTOMATOR de ANDROID. Esta API proporciona los medios para acceder al contenido de una pantalla y realizar acciones en sus controles, como pueden ser, por ejemplo, pulsos sobre la pantalla, o desplazamiento sobre controles de listas.

El Código 10.2 muestra parte de la codificación de la clase de JAVA que implementa uno de los casos de prueba que han sido generados para el modelo que se muestra en la Figura 12.4(a). Cada método corresponde a un evento

```
1  // prev  next SO on SearchView
2  public void TestSpotifysetTitlePopular21() throws UiObjectNotFoundException
3  {
4    UiObject control = new UiObject(new UiSelector()
5                                .className("android.widget.EditText")
6                                .index(0).textContains("Search"));
7    control.setText("on top of the world");
8    Log.v("ANDROIDMVE","CONTROL-setTitlePopular");
9  }
10 // prev SO next SO on SearchView
11 public void TestSpotifyclicsong22() throws UiObjectNotFoundException
12 {
13   UiObject control = new UiObject(new UiSelector()
14                                  .className("android.widget.LinearLayout")
15                                  .index(1));
16   control.click();
17   Log.v("ANDROIDMVE","CONTROL-clicsong");
18 }
```

Código 10.2: Extracto de un caso de prueba ANDROID traducido a JAVA donde se muestran dos eventos que se ejecutan sobre la pantalla del dispositivo

del usuario, que se ejecuta utilizando la API UIAUTOMATOR. Cuando el caso de prueba contiene eventos del sistema, el método incluye una condición de espera de manera que la ejecución se bloquea hasta que se recibe el evento del sistema esperado. El primer método encuentra el *campo de texto* llamado "Search" (línea 4), e introduce un título de canción (línea 7). El segundo método selecciona el primer resultado de la búsqueda (imita un pulso sobre el nombre de la canción en la pantalla) (línea 16). Ambos métodos, escriben en el registro del sistema (*log* de ANDROID) la transición que se acaba de ejecutar (líneas 8 y 17). Como se verá más adelante, este registro es útil para hacer un seguimiento del progreso de la ejecución del caso de prueba desde la componente *Runtime Verification*.

Finalmente, la prueba se ejecuta en el dispositivo usando la API de UIAUTOMATOR (paso *Excecute test*). El caso de prueba se compone de una secuencia de eventos de usuario, que hacen uso de información del modelo sobre el tiempo que debe esperarse entre transiciones (si se ha especificado). Si un caso de prueba comprende varios dispositivos, se genera una secuencia de eventos distinta para cada uno de ellos, que ejecutan de manera concurrente. Para este caso, los eventos del sistema se pueden utilizar para sincronizar los dispositivos en puntos específicos de durante su ejecución, como puede ser la recepción de un mensaje.

10.2. *Runtime verification Engine*

La segunda componente de la herramienta, que se muestra a la derecha de la Figura 10.1, está dedicada al análisis de la ejecución del caso de prueba. Este proceso conlleva la extracción de las trazas de ejecución reales sobre las que se analizan las propiedades extra-funcionales dadas (*extra-functional properties EFP*).

Durante la ejecución de cada caso de prueba en el dispositivo ANDROID, se

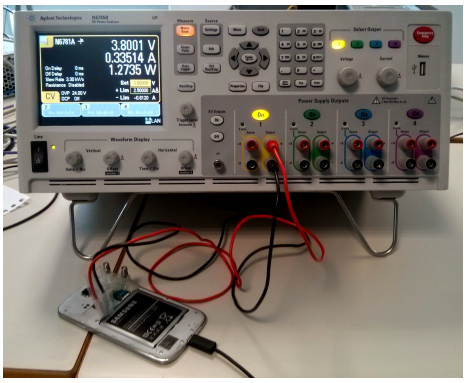


Figura 10.3: Analizador de energía al dispositivo ANDROID

extrae información sobre las trazas de varias fuentes. Claramente, la primera fuente de información es el propio dispositivo, pero también se pueden extraer datos de sensores externos (*specialized probes*). Otra fuente es el LOGCAT [47], que es el lugar del dispositivo donde se registra toda la información (*logs*) producida por el sistema operativo ANDROID y las aplicaciones que están en ejecución. Aunque estos datos no están ni estructurados ni filtrados en la mayoría de los casos, se puede extraer de ahí información relevante sobre la traza que se está ejecutando. Por ejemplo, cada caso de prueba registra información del progreso de su ejecución, como se muestra en el Código 10.2.

Dado que las aplicaciones ANDROID, en su núcleo, son programas JAVA estándares (ver Sección 3.1.5), el depurador de Java (JDI) [48] puede también usarse para extraer información de grano fino sobre la ejecución de las trazas. JDI es la misma API utilizada por los depuradores de JAVA, que permite recorrer el código durante su ejecución y, mostrar información de bajo nivel como, por ejemplo, los valores actuales de las variables del programa.

En la herramienta, se han implementado además dos fuentes de datos útiles para el análisis de las propiedades extra-funcionales como son el *tráfico de red* y el *consumo de energía*. Para el primero, utilizamos la conocida herramienta de línea de comandos TCPDUMP en el dispositivo. TCPDUMP captura la información de paquetes de las interfaces de red de un dispositivo, la filtra y la registra para un análisis posterior.

La información sobre la energía se obtiene utilizando un analizador de potencia externo: una unidad N6705B de *Keysight Technologies* (que se muestra en la Figura 10.3). Este instrumento está conectado al dispositivo ANDROID, en lugar de la batería y, le proporciona energía mientras que, al mismo tiempo, también mide el consumo. El instrumento puede controlarse remotamente usando su interfaz SCPI para, por ejemplo, consultar las mediciones de energía periódicamente.

La componente (*Runtime verification*) de la herramienta recopila información de todas estas fuentes de datos (paso *Synthesize traces*) filtrándola, para

descartar lo irrelevante, y combinándola para construir la llamada traza enriquecida estándar *Enriched Standarized Trace (EST)* (paso *Instrument*). Esta traza es una secuencia de estados discretos que contienen los valores de las variables observadas en determinados instantes de tiempo. Estos estados normalmente corresponden a cambios del sistema en respuesta a eventos relevantes durante la ejecución de un caso de prueba, como puede ser un evento de usuario, o la ejecución de un determinado método JAVA. Además, los valores continuos de variables externas, tales como las mediciones de energía, se incorporan a la traza, incluyendo también la marca de tiempo de cada estado como se describe en la Sección 9.3.

Finalmente, un *observador* analiza la traza (paso *Observe EST*) enriquecida para analizar las propiedades extra-funcionales proporcionadas por el usuario (paso*Observe EST*). El *observador* evalúa cada traza on-the-fly durante la ejecución del caso de prueba y proporciona veredictos tan pronto como sea posible (paso *Verify property*). El observador se implementa utilizando el *model checker* SPIN, que también puede manejar la especificación LTL de las propiedades extra-funcionales, como se describe en la Sección 9.3.1. En lugar de analizar una especificación PROMELA (que permite modelar un sistema), el *observador* tiene que analizar la traza de ejecución proporcionada por el sistema real. Para lograr esto, se utiliza una especificación PROMELA con código `C` incrustado que traduce la traza enriquecida *Enriched Standarized Trace (EST)* a estados de SPIN en tiempo de ejecución [13]. La Figura 10.3 muestra un fragmento simplificado de la especificación PROMELA para uno de nuestros casos de estudio.

La especificación PROMELA contiene variables globales para cada una de las variables relevantes de la EST. El núcleo de esta especificación es un bucle que lee y reconstruye la EST desde una fuente externa (Líneas 20 a 35). Cada iteración del bucle recoge un estado de la traza (línea 27) y actualiza adecuadamente el estado global de SPIN de manera atómica (line 32). Si SPIN tiene que retroceder durante la exploración de la traza, el bucle también puede restaurar correctamente los estados previamente visitados. Desde el punto de vista de SPIN, cada nueva iteración conduce a un nuevo estado a ser explorado. Para analizar los EFPs, la correspondiente fórmula LTL (Ecuaciones 9.1,9.2 y, 9.3) es negada y traducida a un procesos especial de SPIN que se llama *never claim automaton*(que es el autómata Büchi de la fórmula LTL en la notación de SPIN.

Para manejar las variables continuas y propiedades de intervalos, contamos con variables globales adicionales y funciones auxiliares. El valor actual de una variable continua c se almacena en una variable de punto flotante `c`. Estas variables se actualizan en cada iteración del bucle principal. Por ejemplo, en uno de los casos de estudio utilizamos una variable continua denominada `energía` (línea 3). Con el fin de evaluar una fórmula de intervalos ϕ sobre un intervalo $[t_1, t_2]$, los valores inicial y final de la variable continua c en el intervalo, es decir, $c(t_1)$ and $c(t_2)$, deben estar disponibles. Mientras que el primero está almacenado en la variable global `c`, se necesita una nueva variable global `c_t1` que permita acceder al valor del segundo. Estas variables se actualizan automáticamente, mediante llamadas a funciones `update`, que son funciones `C` que calculan los valores de las variables que se extraen de EST.

```
1  c_state "short _interval" "Global"
2  c_state "short testStep" "Global"
3  c_state "double energy" "Global"
4  c_state "double energy_t1" "Global"
5
6  c_code{
7      void update_interval(struct state* newState) {
8          if (!(now.testStep == START) && (newState->testStep == START))
9              newState->_interval = 1;
10         else if (!(now.testStep == END) && (newState->testStep == END)) {
11             newState->_interval = 0;
12     }
13     void update_energy_t1(struct state* newState) {
14         if (!now._interval && newState->_interval)
15             newState->energy_t1 = newState->energy;
16     }
17 }
18
19 init {
20     do
21     :: (running) -> c_code {
22             now.currentState++;
23             if (now.currentState > lastState) {
24                 if (!wasRunning) {
25                     now.running = 0;
26                 } else {
27                     readNewState();
28                     lastState++;
29                     callUpdateFunctions();
30                 }
31             } /* else: backtracked */
32             updateSpinStateFromStateStack();
33         }
34     :: (!running) -> break
35     od
36 }
```

Código 10.3: Fragmento del observador PROMELA para el análisis de la EST

La línea 13 muestra la función de actualización de la variable `energy_t1` en la línea 4. Esta función actualiza solo los valores de `energy_t1` al inicio de un nuevo intervalo. Para detectar esto se ha introducido otra variable `_interval`, que se actualiza automáticamente con otra función (línea 7). Esta función codifica la evaluación de una fórmula de intervalo al principio y al final del mismo. Ambas funciones de actualización se ejecutan dentro del bucle después de haber recuperado las variables para el siguiente estado, en el mismo paso atómico (line 29).

Parte IV

Experimentación

Capítulo 11

Casos de estudio

En este capítulo, se presentan casos de estudio que muestran la capacidad la propuesta para el análisis de propiedades extra-funcionales sobre aplicaciones que se ejecutan sobre dispositivos móviles.

11.1. Construcción del monitor de verificación

En este estudio preliminar se tiene como objetivo principal describir el diseño y implementación del monitor de verificación antes de implementar los casos de estudio. El monitor es extendido/mejorado en los siguientes experimentos. Los resultados se describen en dos trabajos iniciales [49, 42]. Las requerimientos con los que se diseño el monitor son tres. Primeramente, el monitor debe capturar información de un dispositivo ANDROID para pruebas funcionales y/o extra-funcionales. En segundo lugar, la arquitectura del monitor debe seguir los principios de la técnica de *Runtime Verification*. Finalmente, utilizar un lenguaje DSL para evaluar una propiedad simple y observable utilizando la información capturada que permita evaluar el correcto funcionamiento del monitor.

11.1.1. Diseño

El diseño del experimento se puede observar en la Figura 11.1. Uno de los requerimientos indica que se debe hacer la captura de información de los dispositivos para su posterior análisis, esto quiere decir, que el dispositivo *smart* debe ejecutar una aplicación al momento de hacer la captura. Como el objetivo principal es el desarrollo del monitor, para la generación de eventos de usuario se utiliza un método aleatorio que mantenga una APP en ejecución. Para este fin, se ha utilizado un programa propio de ANDROID denominado MONKEY [40], que interactúa dinámicamente con los dispositivo enviándoles eventos de usuario de manera indeterminista, es decir, sin ningún criterio en particular. Los eventos pueden ser clics, toques, o gestos, entre muchos otros. De esta manera, la probabilidad de encontrar una secuencia de eventos de usuario significativa

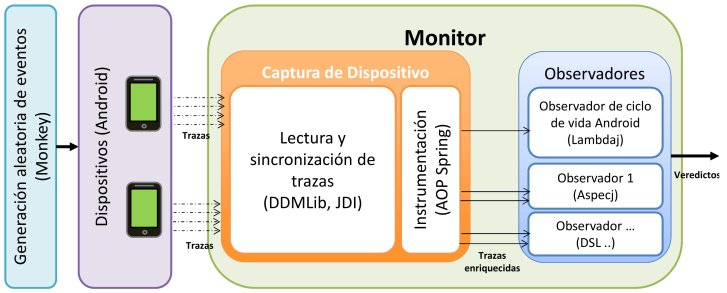


Figura 11.1: Primera definición del monitor en tiempo de ejecución

aumenta mientras más interacciones se produzcan, aunque la mayoría de estas secuencias carezcan de sentido. En la Figura 11.1 este componente aparece a la izquierda.

Como se puede ver en la Figura 11.1, el monitor captura las trazas de los dispositivos y luego las fusiona todas en una sola traza enriquecida. El componente de instrumentación filtra la traza enriquecida para extraer datos relevantes para la verificación. Finalmente, los componentes denominados observadores son los encargados de evaluar una propiedad sobre la traza entregada por el proceso de instrumentación utilizando un *DSL* interno.

Para las capturas se utiliza el DDMLib [50] y JDI [48]. El DDMLib es un programa propio de Android. Es utilizado para extraer diferentes tipos de información del dispositivo y sus aplicaciones como, por ejemplo, información acerca de todos los objetos que ocupan un espacio de memoria (*heap*), los hilos en ejecución, los componentes que se ejecutan en cada hebra e, incluso, puede mostrar la traza de eventos y métodos ejecutados. Por otro lado, JDI (*JavaTM Debug Interface*) es un programa de Java que da información a los depuradores de código (*debugger*) que necesitan acceso (fundamentalmente remoto) al estado actual de la ejecución de una aplicación bajo análisis. La herramienta proporciona acceso al estado actual de las instancias de clases, interfaces, tipos primitivos, ejecuciones y parámetros de métodos, entre otros.

Para la instrumentar las trazas se utiliza el componente que utiliza la programación orientada a aspectos (*AOP*) denominado Spring AOP [38]. Con *AOP*, es posible manejar llamadas a distintas funciones sistemáticamente relacionadas, pero que están en distintos puntos del código. A estas funciones se las denomina *características transversales* (*cross-cutting concerns*) y son utilizadas para definir un aspecto a ser observado.

En este sentido, en Spring AOP un aspecto está compuesto por *joinpoints*,*pointcuts* y *advices*. Un *joinpoint* representa un punto de un programa relacionado con una característica transversal. Cada *joinpoint* esta relacionado con una expresión que describe una característica a observar denominado *pointcut*. Finalmente, un *advice* ejecuta una sección de código antes/durante/-después de que ocurra un *joinpoint*. En este trabajo el *pointcut* se define por el

```
<aop:config  proxy-target-class="true">
 <aop:pointcut id="methodExcecution"
 expression=
 "execution(* com.UMA.RuntimeEngine.AndroidMobile.RunAndroidMonitor.*(String,String,String,*))
 and args(pDevice,pApplicationName,pClassName,pMethod) "/>
 <aop:aspect ref="advice">
     <aop:before pointcut-ref="methodExcecution" method="onNewMethodExcecuted"/>
 </aop:aspect>
</aop:config>
```

Figura 11.2: Configuración de *joinpoints*, *pointcuts* y *advices* en SPRING AOP

```
Lambdaj
List<Person> sortedByAge = sort(people, on(Person.class).getAge());
Without Lambdaj
List<Person> sortedByAge = new ArrayList<Person>(people);
Collections.sort(sortedByAge, new Comparator<Person>() {
        public int compare(Person p1, Person p2) {
            return Integer.valueOf(p1.getAge()).compareTo(p2.getAge());
        }
});
```

Figura 11.3: Ordenar lista de personas por edad con y sin LAMBDAJ

tipo de eventos que se necesita encontrar dentro de la traza, los *joinpoints* se buscarán dentro de la traza en el momento de ser fusionada. Finalmente, mediante los *advices* se enviará a cada observador las trazas filtradas sobre las que necesita evaluar una propiedad. A continuación, en la Figura 11.2 se muestra una sección $xml - section < aop : config >$ en la que el *pointcut* describe que cada vez que se llama a cualquier método de una clase *RunAndriodMonitor* para registrar un evento nuevo en la traza enriquecida, se debe notificar al método $onNewMethodExcecuted$ que llama a los observadores que evalúan las propiedades.

Para la descripción de las propiedades se ha optado por utilizar un lenguaje funcional tal como se hace en estudios similares [51, 34, 52]. En este estudio, se utiliza LAMBDAJ que una la librería basada en JAVA que, a la vez que emplea técnicas de programación funcional, también preserva los tipos estáticos de JAVA. El lenguaje funcional que emplea LAMBDAJ es de tipo `λ-expressions`[32, 53], de esta manera, nos permite definir las reglas de negocio de una aplicación como expresiones que siguen una sintaxis en pseudo inglés. Por ejemplo, en la Figura 11.3 se describe una comparativa de una expresión LAMBDAJ frente a otra escrita en JAVA, la especificación describe una función que ordena una lista de personas por edad.

11.1.2. Experimentación

Para experimentar con el monitor, se hizo el análisis de la correcta ejecución del ciclo de vida de los estados de un *Activity* de una aplicación de ANDROID (Sección 3.2). Con este motivo, se utilizaron APPS propias desarrolladas con errores en esta definición, es decir, en las que el ciclo de vida de los *Activities* de las APPS les hace falta implementar uno de sus estados como, por ejemplo, *OnPause*. Éste no es un caso de mucha relevancia, pero puede ocurrir que la mala

```
public class ActivityInstrumentation
{
    List<ActivityObserver> activityObservers= new ArrayList<ActivityObserver>();
    public void onNewMethodExcecuted(JoinPoint joinPoint,Method pMethod)
    {
        String method= pMethod.getName();
            .....
           for(ActivityObserver observer:activityObservers)
           {
               if(method == "onCreate")
                       observer.onCreate(pDevice, pApplicationName, pClassName);
               else if(method == "onStart")
```

Figura 11.4: Clase Activity Instrumentation

aplicación del ciclo de vida, implique una mala utilización de la memoria, o bien, una mala gestión de las hebras en el dispositivo, mermando el rendimiento de una APP a lo largo del tiempo. La aplicación analizada se llama *SimpleCalculator* que permite calcular operaciones de aritmética simple.

Primeramente, se han filtrado los eventos capturados como se muestra en la Figura 11.2. Después, el método *onNewMethodExcecuted* se encarga de entregar las trazas filtradas a los observadores que evalúan propiedades que tienen que ver con *Activities* como se puede ver en la Figura 11.4. Por lo tanto, después de que un nuevo estado de *Activity* es registrado por en el monitor, el método *onNewMethodExcecuted* es ejecutado.

Para evaluar el ciclo de vida de los *Activities*, se han incluido varios observadores que contienen propiedades descritas en LAMBDAJ que se corresponden con cada uno de los estados del ciclo de vida de un *Activity*. De esta manera, cada propiedad valida que cada estado se haya llamado en el momento correcto. Por ejemplo, una llamada a un evento *OnCreate* es correcta cuando es la primera vez que se crea un *Activity*. También es correcta si, aunque el *Activity* se ha creado antes, luego ha sido destruido (*Destroyed*) o detenido (*Stopped*) antes de invocar a *OnCreate* nuevamente. La Figura 11.5(a) describe el ejemplo mencionado. Como se puede ver, la expresión toma ambas posibilidades:

1. cuando no existan *Activities* (`selectedComponnents.size() ==0`), o
2. cuando el *Activity* ha sido creado (`selectedComponnents.size() !=0`) y un componente (`AndroidComponent.class`) es selecciondo (`select(...)`) con estado

   (`having(... getState())`) es igual a *destroyed* (`equalTo(...Destroyed)`) o es igual a

   stopped (`or(...equalTo(...Stopped)`).

De otra manera, se devuelve un mensaje de error.

En la Figura 11.5(b) se muestra la propiedad utilizada cuando se registra un estado *OnStop* por un *Activity*. En este caso, se debe verificar que un estado *Paused* haya sido previamente ejecutado.

```
@Override
public void onCreate(String DeviceName, String Application, String Activity){
    //..
    Verify(selectedComponents.size() == 0  ||
            (selectedComponents.size() != 0 &&
           (select(selectedComponents,
                   having(on(AndroidComponent.class).getState(),
                           equalTo( ApplicationState.Destroyed.ordinal()))
                    .or(having(on(AndroidComponent.class).getState(),
                           equalTo(ApplicationState.Stopped.ordinal()))))
                .size() == 1))
           , "The activity named "+ Activity + " was already created, " +
                "or the activity was not destroyed or stopped previusly");
```

(a) Verificación del método On Create

```
@Override
public void onStop(String DeviceName, String Application, String Activity) {
    //..
    List<IComponent> selectedActivities =
            select(selectedComponents,
            having(on(AndroidComponent.class).getComponentClass(),
                    equalTo(Activity))
            .and(having(on(AndroidComponent.class).getState(),
                    equalTo(ApplicationState.Paused.ordinal()))));
    boolean  isCorrect =  Verify(selectedActivities.size() == 1
                         , "The activity named "+ Activity
                         + " was stopped without beening paused");
    if(isCorrect )
        selectedActivities.get(0).setState(ApplicationState.Stopped.ordinal());
}
```

(b) Verificación del método On Stop

Figura 11.5: Métodos de verificación del ciclo de vida de ANDROID descritos en LAMBDAJ

11.1.3. Resultados

Una vez que se han configurado los observadores, se puede proceder con el proceso de verificación. Los resultados son expuestos vía consola tal y como se muestra en la Figura 11.6. Los resultados muestran la secuencia de eventos registrados por los observadores, previamente instrumentados. Cuando un observador encuentra que se ha violado una propiedad, se muestra en pantalla una línea de mensaje $***error***$. En el ejemplo, se obtiene este error porque no se ha implementado el método *onPause* en el *Activity* `HelloWorldActivity` de la APP *SimpleCalculator* .

Aunque las propiedades analizadas son simples, este experimento inicial muestra cómo funciona el monitor. Claramente, extendiendo este metodología se pueden construir propiedades mucho más complejas como las que se muestran en los siguientes casos de estudio.

```
A new method is goint to execute from device: BX902UEX3Q Application: SimpleCalculator Activity: android.app.Simpl
====================================================
Wed Apr 24 19:42:45 CEST 2013 Método =onCalculate
====================================================
A new method is goint to execute from device: BX902UEX3Q Application: SimpleCalculator Activity: android.app.Simpl
====================================================
Wed Apr 24 19:42:45 CEST 2013 Método =onShowResults
====================================================
A new method is goint to execute from device: BX902UEX3Q Application: HelloWorld Activity: android.app.Activity.He
***error***The activity named android.app.Activity.HelloWorldActivity was stopped without beening paused
====================================================
Wed Apr 24 19:42:45 CEST 2013 Método =onStop
====================================================
```

Figura 11.6: Resultados de Verificación

11.1.4. Comparativa con otras propuestas

En el momento de desarrollar esta propuesta, se analizaron varias otras propuestas. Por ejemplo, la propuesta [30, 51] sigue una estrategia similar a la nuestra. Ellos instrumentan los datos generados por una serie de trazas de aplicaciones de JAVA utilizando ASPECTJ. Además, ellos utilizan múltiples monitores empotrados dentro de la aplicación, y evalúan propiedades descritas en lenguaje SCALA. Este estudio utiliza SPRING AOP para analizar trazas de ANDROID y utiliza observadores descritos en LAMBDAJ para analizar propiedades. La mayor diferencia entre ambas propuestas, se encuentra en que este estudio mantiene el proceso de verificación de manera externa para las APPS de ANDROID, lo cual permite mayor flexibilidad en el análisis dadas las dificultades inherentes al análisis aplicaciones de dispositivos móviles.

Existen otras propuestas más maduras. Por ejemplo RV-DROID [54] es un *Framework* de verificación basado en *RV*, se caracteriza por la utilización de versiones propias de ASPECTJ para definir propiedades. Cada aspecto corresponde con una propiedad, de manera que, en caso de error, puede ser utilizada para corregir la aplicación en tiempo de ejecución. De todas maneras, ASPECTJ tiene limitaciones en su expresividad al momento de describir propiedades. En el alcance presentado por RV-DROID las propiedades analizan los valores de los parámetros permitidos en los métodos. En contrate, este estudio separa los aspectos de las propiedades, lo cual permite utilizar cualquier tipo de *DSL* ademas del LAMBDAJ para describir las propiedades.

Otros trabajos relacionados han sido desarrollados por *NICTA Software System Research Group* [55]. Sus propuestas, hasta ese entonces, se concentraban en la verificación de propiedades de seguridad. Con este objetivo, ellos han construido su propia versión del *kernel* de ANDROID, que reporta información de violaciones de propiedades de seguridad después de evaluar las trazas de ejecutadas por una APP de ANDROID. Otra propuesta distinta, que también se concentra en definir propiedades de seguridad, es TAINT DROID [56], en este caso, su propuesta consiste en modificar completamente la pila de eventos de ANDROID de esta manera, hacer un seguimiento del flujo de información sensitiva. De esta manera, se puede analizar en tiempo de ejecución si la aplicación viola alguna propiedad de seguridad. Finalmente, SAINT FRAMEWORK [57], modifica los instaladores de las APPS de ANDROID, para analizar, en tiempo de

instalación si la aplicación rompe alguna propiedad de seguridad.

11.1.5. Conclusiones

En este caso de estudio se ha presentado la implementación base del monitor de verificación en tiempo de ejecución para analizar Apps de Android y, sigue la técnica de *Runtime Verification*. Se han analizado trazas provistas por la ejecución de múltiples Apps de Android, utilizando diferentes programas como ser DDMLib y JDI. Luego, el monitor clasifica y filtra las trazas capturas utilizando herramientas como Spring AOP. Finalmente, el diseño de de observadores ha permitido definir propiedades descritas en un lenguaje seudo-funcional (LambdaJ) para evaluar propiedades funcionales, que puede ser adaptado para incluir mas tipos de DSLs como se ve en los casos de estudio. La herramienta generada se ha evaluado mediante un ejemplo que que evalúa propiedades que describen el ciclo de vida de *Activity* de Android.

11.2. Caso de estudio 1: Generación de modelos *Facebook/ Youtube*

Este caso de estudio sirve para describir cómo se modela el comportamiento de las aplicaciones móviles y cómo se generan automáticamente los casos de pruebas a partir de estos modelos. Los resultados de este experimento se presentaron en el trabajo [14]. Como ya se ha discutido anteriormente, el modelo de una aplicación es el punto de partida a partir del cual se construyen los casos de prueba. Así, el modelo diseñado debe permitir la generación de un conjunto significativo de comportamientos de usuario diferentes que produzcan trazas relevantes que puedan ser analizadas por la herramienta en tiempo de ejecución.

Para este caso de estudio se han utilizado las Apps de Android Facebook® y YouTube®. El flujo de ejecución se inicia con Facebook® desde donde se simulan publicaciones de actualización de estados. Algunos estados contienen enlaces a YouTube® que, más tarde, se llaman para ser reproducidos desde bandeja de entrada de Facebook®. Se han utilizado dos dispositivos distintos, uno que comenta el estado con un enlace a YouTube®, y otro que reproduce el vídeo del comentario. En las siguientes secciones, se describe cómo se han diseñado la aplicaciones y los resultados obtenidos. Inicialmente, se construyó un *modelo* de las interacciones de las Apps, usando máquinas de estados. Luego, se generaron los casos de prueba a partir del modelo.

11.2.1. Diseño del caso de estudio 1

Para el escenario previamente descrito que comprende las aplicaciones Facebook® y YouTube® se han construido tres vistas (*HomeView*, *CommentView* y *MovieView*), que describen el comportamiento de un usuario que escribe un enlace a un vídeo de YouTube® en un comentario de Facebook® y posteriormente lo visualiza en YouTube®. El modelo de las maquinas de estados se puede ver en la

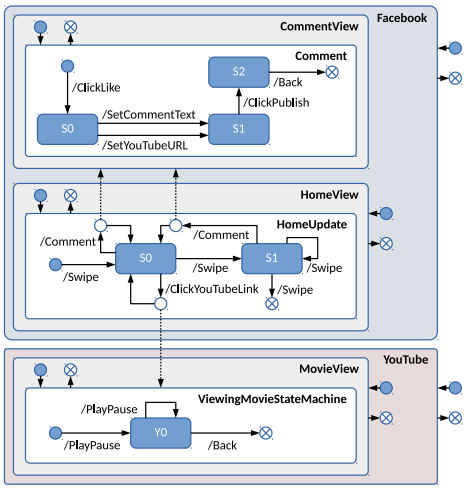


Figura 11.7: Modelo de Facebook® y YouTube®

Figura 11.7. El modelo incluye información adicional necesaria para relacionar las aplicaciones con sus vistas. La definición XML del modelo se ha generado automáticamente a partir de estas máquinas de estado.

El Listado 11.1 muestra parte de la definición XML obtenida[1]. En concreto, el listado contiene la máquina de estados asociada a la vista de *Home* de la *App* Facebook®. La máquina de estados define estados y transiciones, como se describe en la Sección 8.1. Los atributos `type` y `through` identifican el tipo de transición y la máquina de estado a llamar (si existe). El Listado 11.1 muestra ejemplos de transiciones simples, en la misma máquina de estados, (línea 13) como de transiciones complejas que transitan de una máquina a otra diferente a través de un estado de conexión (líneas 12 y 14). Cada transición tiene un nombre `ID` único dentro de la vista que se utiliza para identificar las transiciones, además del evento de usuario/sistema que activa la transición.

Los eventos que desencadenan las transiciones en la Figura 11.7 son las interacciones de usuario sobre los controles colocados en vistas (como se describe en la Sección 8.1.1). Según la definición, las etiquetas de las transiciones se asocian a los controles de las vistas que deben ejecutar la acción. La Figura 11.8 muestra algunos de los controles que se han identificado en la vista *CommentView*. Por ejemplo, la etiqueta de la transición que representa la acción de comentar en Facebook® se ha asociado con los controles marcados con *Comentario* en la imagen. Del mismo modo, la acción de pulsar un enlace de uno de los comentarios de Facebook® se ha asociado con los controles marcados con *clickYouTubeLink* en la imagen. Estas asociaciones se agregan al fichero de definición de controles de la vista en formato XML, que ha sido obtenido con la herramienta `UIAutomatorViewer` [17].

[1] Una versión mas completa de la definición de este modelo se muestra en el apéndice A.1.1

```
<Application name="Facebook" package="com.facebook.android">
  <Views>
    <View name="HomeView" controlsFile="Home.xml" >
      <StateMachines>
        <StateMachine name="HomeUpdate">
          <States>
            <State name="S0"/>
            <State name="S1"/>
          </States>
          <Transitions>
            <Transition ID="1" event="Swipe" prev="" next="S0"
                type="Simple"/>
             <Transition ID="2" event="Comment" prev="S0" next="S0"
                through="CommentView" type="View"/>
             <Transition ID="3" event="Swipe" prev="S0" next="S1"
                type="Simple"/>
             <Transition ID="4" event="ClickYouTubeLink" prev="S0" next="S0"
                through="ViewingMovieStateMachine" type="StateMachine"/>
            <Transition ID="5" event="Swipe" prev="S1" next="S1"
                type="Simple"/>
            <Transition ID="6" event="Comment" prev="S1" next="S0"
                through="CommentView" type="View"/>
            <Transition ID="7" event="Swipe" prev="S1" next=""
                type="Simple"/>
          </Transitions>
        </StateMachine>
      </StateMachines>
    </View>
  </Views>
</Application>
```

Código 11.1: Configuración de máquinas de estados

11.2.2. Resultado del caso de estudio 1

A continuación, se muestra el resultado de la generación exhaustiva de los casos de prueba correspondientes al modelo diseñado. Tal y como se describe en la Sección 10.1 la generación automática la realiza la herramienta SPIN ejecutando el modelo PROMELA obtenido automáticamente a partir del archivo XML del modelo de las máquinas de estados.

El Listado 11.2 muestra un extracto de la especificación PROMELA generada para el ejemplo de Facebook® y YouTube® [2]. La estructura de datos *backstack* se muestra en la línea 2. En el Listado se puede observar cómo se inserta o se extrae un nuevo estado en la pila cuando se entra o se sale de una máquina de estados, respectivamente (líneas 15 y 27). Además, es interesante tener en cuenta que el estado actual (`currentState`) siempre tiene acceso a la cima de esta pila para cada dispositivo.

El código mostrado en el Listado 11.3 contiene parte de un caso de prueba obtenido a partir del modelo de la Figura 11.7. Tal como se explica en la Sección 10.1, el fichero se ha compilado a un formato `.dex` (ANDROID application binary), se ha desplegado en un dispositivo ANDROID ejecutándose mediante la herramienta *adb*.

La tabla 11.1 proporciona algunos resultados cuantitativos del número de

[2] Una versión mas completa de la definición de este modelo se muestra en el apéndice A.1.2

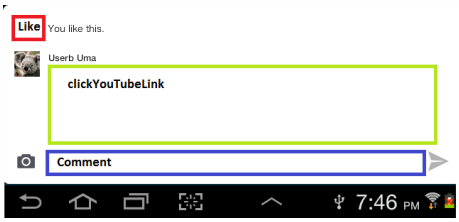


Figura 11.8: Identificación de eventos de transición

```
typedef Backstack { mtype states[MAX_BK]; short index; }
Backstack backstacks[DEVICES];
#define currentBackstack    devices[device].backstack
#define currentState        currentBackstack.states[currentBackstack.index]

active proctype device_219dcac41() {
    if
    true -> app_219dcac41_Facebook(D_219dcac41);
    true -> app_219dcac41_YouTube(D_219dcac41);
    fi;
    devices[D_219dcac41].finished = true
}
inline statemachine_Facebook_HomeView_HomeUpdate(device) {
    currentState = State_Facebook_HomeView_HomeUpdate_init;
    pushToBackstack(device, State_Facebook_HomeView_HomeUpdate_init);
    do
    :: currentState == State_Facebook_HomeView_HomeUpdate_S0 ->
        transition(device, VIEW_HomeView, 2);
        view_Facebook_CommentView(device);
        currentState = State_Facebook_HomeView_HomeUpdate_S0
    :: currentState == State_Facebook_HomeView_HomeUpdate_S0 ->
        transition(device, VIEW_HomeView, 4);
        statemachine_YouTube_MovieView_ViewingMovieStateMachine(device);
        currentState = State_Facebook_HomeView_HomeUpdate_S0
    ...
    od;
    popFromBackstack(device)
}
```

Código 11.2: Especificación PROMELA para Facebook® y YouTube®

casos de prueba generados y el esfuerzo computacional requerido, para varios escenarios, y el promedio en tres ejecuciones. El dispositivo `219dcac4` se dedicado únicamente a la *App* Facebook®, mientras que el dispositivo *219dcac41* se ha dedicado a ambas aplicaciones. La cuarta columna muestra la profundidad máxima que se ha considerado para las transiciones de los caso de prueba generados para un dispositivo. La quinta columna contiene el tiempo total dedicado a la generación de los casos de prueba de los modelos XML. Las tres últimas columnas son estadísticas tomadas a partir de la ejecución de SPIN, es decir, el número de estados generados por la herramienta, el tamaño de cada estado, y la memoria total utilizada, respectivamente. Estos resultados muestran cómo la adición de la *App* YouTube® que, está bastante aislada, tiene poco impacto en los resultados (filas 3 y 4 de datos). Sin embargo, la combinación de los dos dispositivos es costosa, especialmente porque sus casos de prueba son

```
1  public class TestDevice1 extends UiAutomatorTestCase {
2      // Transition  2  previous S0 next S0 on view HomeView
3      public void TestFacebookComment2() throws UiObjectNotFoundException {
4          UiObject control = new UiObject(new UiSelector().
5          className("android.widget.TextView").
6          index(1).textContains("Comment"));
7          control.click();
8      }
9      // Transition  4:  previous S0 next S0 on view HomeView
10     public void TestFacebookclicYouTubeLink27() throws
           UiObjectNotFoundException {
11         UiObject control = new UiObject(new UiSelector().
12         className("android.view.View").index(3));
13         control.click();
14     }
15     // Transition  1:  previous  next Y0 on view MovieView
16     public void TestYouTubeplaypause28() throws UiObjectNotFoundException {
17         UiObject control = new UiObject(new UiSelector().
18         className("android.view.View").index(4));
19         control.click();
20     }
21 }
```

Código 11.3: Casos de prueba generados con *UiAutomator*

Dispositivos		Configuración		Resultados				
219dcac4	219dcac41	*Backstack*	Transición	# Pruebas	Tiempo (s)	# Estados	Tamaño (B)	Memoria (MB)
✓		4	20	5641	1.0	307234	84	156.8
✓		4	26	111317	9.0	6063398	92	728.6
	✓	4	20	5660	1.0	307493	84	156.8
	✓	4	26	111342	9.0	6063735	92	728.6
✓	✓	4	10	1872	7.0	4039337	100	560.3
✓	✓	4	12	12180	52.3	28972472	108	3445.2

Cuadro 11.1: Resultados de caso de prueba

completamente independientes.

Debido a que los test se generan partir del modelo de las APPS utilizando técnicas de *Model Cheking*, los test resultantes representan todo el espacio de estados posible establecido por el modelo. Por este motivo, el nivel se cobertura se definirá por dos variables. El primero, depende del comportamiento de la APP reflejado en el modelo, mientras mayor sea el comportamiento se modelado, mayor sera el numero de funcionalidades que se exploraran. El segundo, depende de la profundidad maximiza con la que se generan las transiciones, mientras mas alto sea el nivel de profundidad, mayor es el nivel de cobertura alcanzado. El mayor problema de éste ultimo enfoque, es que se puede alcanzar secuencias de casos de test muy largos (tal como se ve en la tabla 11.1) cuya ejecución seria muy costosa a nivel computacional.

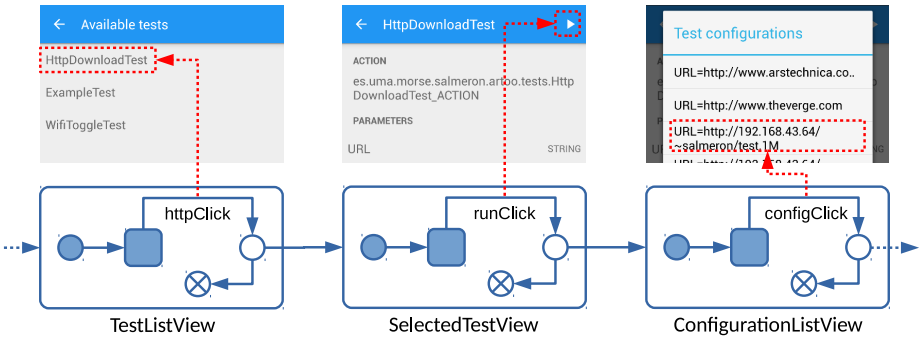


Figura 11.9: Fragmento del modelo para la generación de pruebas

11.3. Caso de estudio 2: Análisis del consumo de energía

En esta sección se describe la aplicación de nuestra propuesta para el análisis del consumo de energía que se presentó en [15]. En este caso de estudio se ha utilizado una *App* simple: una aplicación ANDROID desarrollada internamente para ayudar a caracterizar el perfil energético de tareas comunes. La *App* contiene varios tipos de configuraciones de pruebas que permiten mostrar distintos escenarios de tráfico de red (simulación de trafico de red Http, simulación de Red TCP, etc.). El objetivo principal del caso de estudio es demostrar cómo nuestra propuesta puede ser útil para comprobar el consumo de energía de una cierta secuencia de acciones dentro de un caso de prueba. Además, se a utilizado un trabajo del grupo de investigación que analiza la ejecución de trazas con SPIN [58][13]. Para conseguir esto, utilizamos la siguiente propiedad de intervalo:

$$[energy(t_2) - energy(t1) < P]_{[[testStep=\text{START}, testStep=END]]}$$

Esta propiedad es de la forma $[[\Phi]]_{[p,q]}$, cuya traducción a LTL se presentó en la Sección 9.3.1.

11.3.1. Configuración del caso de estudio 2

Para el experimento se ha utilizado el modelo que se muestra en la Figura 11.9. El interés de este caso de estudio radica en analizar el consumo de energía de uno de los escenarios de pruebas de la *App*, en el que se descarga un archivo de longitud fija desde un servidor local a través de WiFi.

El modelo muestra algunas capturas de pantalla de la aplicación y un fragmento del modelo que, como se ha explicado en anteriores secciones, contiene varias máquinas de estados conectadas por transiciones que se ejecutan cuando el usuario interactúa. Estas máquinas fueron traducidas a PROMELA, y luego a clases de pruebas de JAVA que se ejecutaron sobre el dispositivo, una parte del código generado en JAVA se muestra en Código 11.4, específicamente, se describe

```
1  public void TestSampleApphttpClic1() throws UiObjectNotFoundException {
2      UiObject control = new UiObject(new UiSelector()
           .className("android.widget.TextView").index(0)
           .textContains("HttpDownlo"));
3      control.click();
4      Log.v("TESTCASEGENERATOR","CONTROL-httpClick");
5  }
6  public void TestSampleApprunClic2() throws UiObjectNotFoundException {
7      UiObject control = new UiObject(new UiSelector()
           .className("android.widget.TextView").index(0)
           .descriptionContains("Available tests"));
8      control.click();
9      Log.v("TESTCASEGENERATOR","CONTROL-runClick");
10 }
```

Código 11.4: Parte el caso de prueba generado automáticamente

a un usuario seleccionndo un escenario de prueba HTTP de una lista luego, el test seleccionado se ejecuta contra una de las configuraciones disponibles. En este experimento, la generación de casos de prueba utiliza el mismo enfoque presentado en el caso de estudio de la Sección 11.2.

En este caso de estudio, se ha introducido SPIN como uno de los observadores encargados de evaluar propiedades sobre las trazas capturadas de un dispositivo. En este sentido, se ha utilizado la especificación de PROMELA reconstruir los estados de la traza de ejecución utilizando las trazas enriquecidas que extrae del monitor luego, los estados se entregan a SPIN para su análisis. Esta implementación esta descrita en la Sección 9.3.1. Una parte de la especificación en PROMELA se muestra en la Figura 10.3 que describe un fragmento de la especificación del caso de estudio.

11.3.2. Resultados caso de estudio 2

En este ejemplo, el código PROMELA utiliza una variable continua de energía denominada `energy` que contiene el valor acumulado del consumo de energia del dispositivo en distintos instantes de tiempo. Los casos de prueba obtenidos a partir del modelo se instalaron en un dispositivo móvil Samsung Galaxy S4. Este dispositivo se conectó a un analizador de corriente continua N6705B de Keysight Technologies. Se utilizó la propiedad de intervalo declarada al principio de esta sección que, para esta implementación, sería escrita como:

```
[$energy - $energy_t1 < P] [[$testStep == START, $testStep == END]]
```

Es interesante notar el uso del prefijo `$` en la propiedad para las acceder a las variables internas de los estados en la traza de ejecución. En esta propiedad, el intervalo está dado por las proposiciones `$ testStep == START`, donde `START` es el evento del usuario que se ejecuta cuando se inicia la descarga, y `$ testStep == END` representa el final la descarga. `testStep` es una variable entera incluida en la traza de ejecución que trata SPIN, y `START` y `END` son los eventos de la traza de ejecución que delimitan el intervalo. La propiedad `$ energy - $ energy_t1 <P` se debe satisfacerse los intervalos. La variable continua `energy`

```
1  pLogLevel: 0 pTag: TESTCASEGENERATOR pLogMessage:
       V/TESTCASEGENERATOR(13414): CONTROL-httpClick: 01/05/2015 16:39:28 void
       com.UMA.RuntimeEngine.AndroidMobile.SmartPhoneMonitoring
       .RegisterAndroidLog(String,String,String,String)
2  may 01, 2015 4:39:59 PM
       com.UMA.RuntimeEngine.Observer.Spin.ProtocolClientHelper sendReport
3  measure { id: 2, intValue: 702221648 }
4  measure { id: 5, longValue: 1430498399008 }
5  measure { id: 6, doubleValue: 6.753239800596E13 }
6
7  pLogLevel: 0 pTag: TESTCASEGENERATOR pLogMessage:
       V/TESTCASEGENERATOR(13485): CONTROL-runClick: 01/05/2015 16:39:28 void
       com.UMA.RuntimeEngine.AndroidMobile.SmartPhoneMonitoring
       .RegisterAndroidLog(String,String,String,String)
8  may 01, 2015 4:40:05 PM
       com.UMA.RuntimeEngine.Observer.Spin.ProtocolClientHelper sendReport
9  measure { id: 2, intValue: 114991181 }
10 measure { id: 5, longValue: 1430498405000 }
11 measure { id: 6, doubleValue: 6.433557098917E13 }
```

Código 11.5: Traza de la ejecución del componente de *Runtime Verification*

se actualiza con los datos del analizador de energía. La medición se realizó en intervalos regulares, 1000 veces por segundo, y los valores se introdujeron en la traza enriquecida estandarizada.

El Listado 11.5 muestra parte de la salida de la herramienta mientras se obtiene una traza en tiempo de ejecución. Esta salida incluye los eventos detectados por el monitor en tiempo de ejecución (pulsos sobre la pantalla, etc). La salida muestra cómo se codifican las variables de cada estado para ser enviadas a SPIN. En este caso, el estado incluye los valores actuales de la variable *testStep* (línea 3), la marca de tiempo Unix (4) y la integración de la medida de energía recogida desde el analizador (5).

Con esta configuración, se pudo comprobar que las mediciones de energía reportadas por el analizador de potencia están dentro de los límites esperados, mientras se ejecuta el conjunto de casos de prueba obtenidos a partir del modelo de la *App*. Mientras que los casos de prueba definen una secuencia clara y repetible de eventos de usuario, hay otro tipo de eventos internos que pueden desempeñar un papel importante en el consumo de energía, como es el caso de procesos de fondo inesperados o problemas de conectividad. Una generación exhaustiva de pruebas, que luego es ejecutada y analizada automáticamente, puede ayudar a reconocer estas interacciones inesperadas y orientar a los desarrolladores a optimizar sus aplicaciones en relación con el consumo de energía.

11.3.3. Conclusiones caso de estudio 2

Este caso de prueba muestra la aplicabilidad de la lógica de intervalos para la especificación de las propiedades extra-funcionales, como el consumo de energía, para aplicaciones reales. La lógica propuesta ha permitido demostrar que la lógica de intervalos es una buena para el análisis de este tipo de variables continuas apoyadas en el uso de formulas LTL. Este trabajo ha dado pie al análisis de otras propiedades extra-funcionales en las trazas de ejecución como es, por

ejemplo, el número de paquetes de red recibidos/enviados, tiempo de conexión, etc).

11.4. Caso de estudio 3: Análisis de trafico de red *Spotify*

Esta sección describe un experimento realizado sobre la aplicación *Spotify* para ANDROID. En este caso de estudio se examinaron los patrones de tráfico generados por la aplicación al reproducir una serie de canciones. Se ha utilizado MVE en un sistema operativo de 64-Bits, sobre un procesador de *Intel(R) Core(TM) i5-3337U CPU* con una memoria RAM de 6 GB a 1.8 GHz. Los casos de prueba de *App* se ejecutaron sobre un dispositivo *Samsung Galaxy Tab 10.1* con sistema operativo ANDROID de version 4.1.2 (*Firmware* 3.0.31-805288), conectado a Internet vía *WiFi* a una velocidad de 31.6Mb/s. El total de tiempo de ejecución fue de 8.3 horas. Para modelar la aplicación, se ha diseñado el modelo del comportamiento del usuario de la Figura 8.2.

11.4.1. Configuración del caso de estudio 3

Antes de realizar las tareas de verificación, se ha hecho un estudio del tráfico de red generado por la *App*. En el estudio, se ha encontrado que los servidores de *Spotify* descargan las canciones utilizando conexiones HTTP. Cada canción se descarga utilizando varias solicitudes de tipo *HTTP GET* enviadas justo después de que el usuario pulsa el botón *play*. Cada solicitud es contiene un fragmento de la canción.

En este caso de estudio, se ha introducido la utilización de las propiedades extra-funcionales aplicadas al análisis de trafico de red generado por *Apps media-streaming*. Para esto, se ha utilizado una propiedad extra-funcional del tipo:

$$Z = \frac{|r - \mu|}{\sigma}$$
$$\forall[[Z < 2]]_{[testStep=clickSong, testStep=clicPrincipalMenu]} \tag{11.1}$$

En la fórmula, la variable Z es una medida estándar de una variable de tráfico de red determinada r que sigue la distribución normal, μ es el promedio conocido de la distribución de r-mediciones para las canciones con un comportamiento típico, y σ es la desviación estándar de la misma distribución. El intervalo esta definido entre *clickSong* y *clicPrincipalMenu*, que representan todos los eventos de usuario que inician la reproducción de una canción y los eventos que inician la búsqueda de una nueva canción después de escuchar la anterior, respectivamente. El intervalo de la ecuación 11.1 comprueba que la variable continua se encuentra dentro del intervalo correspondiente a dos desviaciones estándar del promedio de la distribución de valores, es decir, que se identifica con el 95,45 % de las

Variable	Promedio	Desviación estándar
r1	0.252	0.048
r2	0.010	0.002
r3	16.866	0.675

Cuadro 11.2: Promedio y desviación estándar de las variables de tráfico de red

canciones que exhiben un comportamiento típico de tráfico de red. Estos eventos y variables pueden ser identificados en un dispositivo ANDROID en tiempo de ejecución utilizando TCPDUMP remotamente, el tráfico leído se agregó a la traza de ejecución enriquecida.

Las medidas de tráfico de red que fueron monitorizadas durante la reproducción son las siguientes: el número de paquetes enviados y recibidos ($PkgS$ y $PkgR$, respectivamente), el número de bytes enviados y recibidos (BS y BR, respectivamente) y la duración de la conexión en milisegundos (D). Utilizando estas medidas, se establecieron tres variables de evaluación del tráfico de red: $R1$ como cociente entre $PkgS$ vs. $PkgR$ ($r1 = \frac{PkgS}{PkgR}$), $r2$ como el cociente entre BS y BR ($r2 = \frac{BS}{BR}$), y $r3$ como D ($r3 = D$). El promedio y la desviación estandar del comportamiento del trafico de red se utilizan como referencia para definir las propiedades extra-funcionales.

En el experimento, se ha considerado que una canción muestran un comportamiento anormal si la aplicación inicia la descarga de una canción pero la conexión termina repentinamente, por lo que es forzada a establecer una nueva conexión que permita descargar la canción completa. De esta manera, un observador del monitor es el encargado de detectar los comportamientos anormales analizando las variables de tráfico de red $r1$, $r2$ y $r3$ y utilizando la propiedad extra-funcional siguiente:

$$\forall[[(Z1 < 2) \vee (Z2 < 2) \vee (Z3 < 2)]]_{testStep=clickSong, testStep=clicPrincipalMenu} \tag{11.2}$$

La fórmula es un tipo de una propiedad extra-funcional presentada en la Ecuación 11.1, donde $Z1$, $Z2$, y $Z3$ representan los indicadores de $r1$, $r2$, y $r3$ respectivamente. Esta propiedad extra-funcional, junto con el filtro de TCPDUMP para obtener solo los paquetes de audio de *Spotify* se configuran en MVE.

11.4.2. Resultados del caso de estudio 3

La Tabla 11.3 muestra resultados numéricos relacionados con la generación de casos de prueba del modelo XML con tres valores diferentes del limite máximo de profundidad permitido por la exploración del modelo (transiciones). Para cada uno, se presenta un número de transiciones, casos de prueba, tiempo de procesamiento, estados y utilización de memoria por SPIN. Los resultados muestran que el tiempo de ejecución, numero de casos de prueba, estados y utilización de memoria son extremadamente afectados por el número de transiciones. Esto

Configuración	Resultados			
Max. Trans.	# Casos de prueba	Tiempo	# Estados	Memoria
50	199	0.02 s	7,072	129.8 MB
60	815	0.08 s	28,763	133.2 MB
100	226,029	25.3 s	795,9210	1,724.0 MB

Cuadro 11.3: Resultados numéricos de la generación de casos de prueba

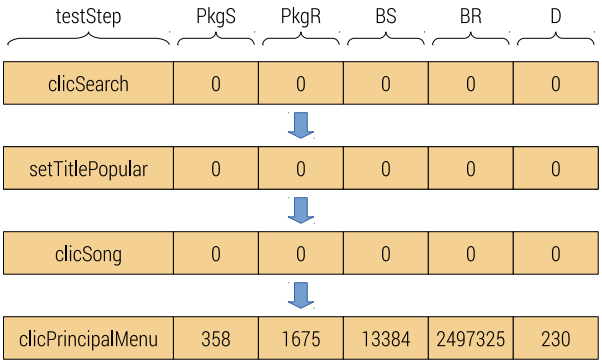


Figura 11.10: Fragmento de la ejecución de la traza

se debe a que cada vez que se suma un valor al número de transiciones se añade un nivel de anidamiento de máquinas de estados llamadas a través de los estados de conexión. Cuando se generen los casos de prueba, es necesario ajustar el numero de transiciones, y el modelo para obtener secuencias de eventos de usuario significativas, en este sentido, se pueden aplicar técnicas para reducir el problema de explosión de estados propios de *Model Cheking*. En consecuencia, esta es la clave para evaluar explorar el espacio de estados que generan los casos de prueba y, de esta manera, disminuir la redundancia entre comportamientos, o encontrar y eliminar las maquinas de estados innecesarias en el modelo de comportamiento origen. Para éste experimento, interesa priorizar la ejecución reiterativa de la máquina de estados que reproduce títulos de canciones después de limpiar el cache de la App.

En la Figura 11.10 se muestra una representación gráfica de un fragmento de una traza de la ejecución de los casos de prueba. La traza esta compuesta de estados discretos, y las transiciones vienen dadas por eventos eventos de usuario. Cada estado contiene el valor de sus variables como, por ejemplo, el número de paquetes enviados y recibidos. El fragmento corresponde a la primera ejecución del caso de prueba *SearchPopularStateMachine*. Las variables de tráfico de red se inicializan con ceros antes de que *testStep* sea igual a *clicPrincipalMenu*. En este último estado, la canción ha sido descargada y reproducida y, en consecuencia, las variables contienen los valores correspondientes.

Los resultados de la evaluación de las propiedades extra-funcionales de la ecuación 11.1 y 11.2) sobre el de tráfico de red se muestran en la Tabla 11.4.

Estadística	$Z1$	$Z2$	$Z3$	$Z1 \vee Z2 \vee Z3$
Positivos	53	52	54	56
Negativos	13	13	13	13
Falsos negativos	3	4	2	0
Sensitividad	95 %	93 %	96 %	100 %
Especificidad	100 %	100 %	100 %	100 %
Exactitud	96 %	95 %	97 %	100 %

Cuadro 11.4: Resultados del proceso de verificación aplicado sobre la *App* de *Spotify* en ANDROID

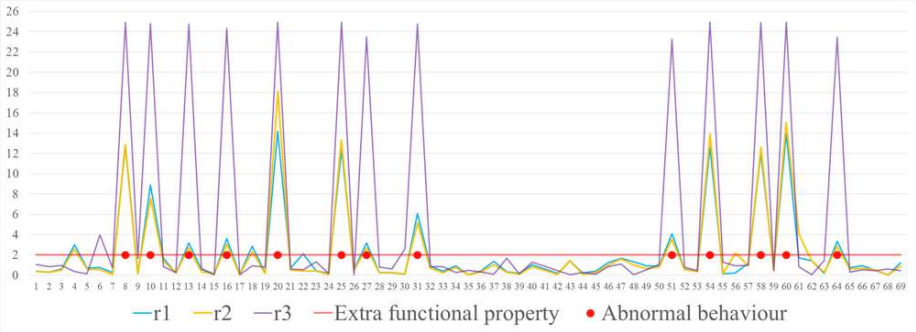


Figura 11.11: Representación gráfica de los indicadores de los cocientes entre los paquetes TCP enviados vs. recibidos (r1), el numero de cocientes de bytes enviados vs. recibidos (r2) y la duración de la conexión en segundos(r3) obtenidos durante la verificación del trafico de red generado por la *App Spotify* para ANDROID

En este caso, se han monitorizado las trazas para obtener los valores de las variables $Z1$, $Z2$ y $Z3$. Las variables se han evaluado utilizando el test de *Shapiro-Wilk*, que ha confirmado la normalidad de la distribución para todos los casos, como resultado de la hipótesis nula que no puede ser rechazada si $\alpha = 0{,}05$. De esta forma, se han evaluado las variables de trafico de red, utilizando las propiedades extra-funcionales descritas en las Ecuaciones 11.1 y 11.2 para cada variable monitorizada de manera separada [3]. La tabla 11.4 contiene el número de trazas que se comportan correcta/incorrectamente en un intervalo $[testStep = clickSong, testStep = clicPrincipalMenu]$ según la formula correspondiente.

Los resultados en la Tabla 11.4 muestran que se han tomado en cuenta todas las variables al mismo tiempo, con el mejor rendimiento (sensitividad = especificidad = exactitud = 100 %), mientras que si se considera las variables de manera separada el rendimiento es menor.

[3] Debe notarse que se ha simplificado la ecuación utilizando $Z1$, $Z2$ y $Z3$ para simplificar la Ecuación 11.1

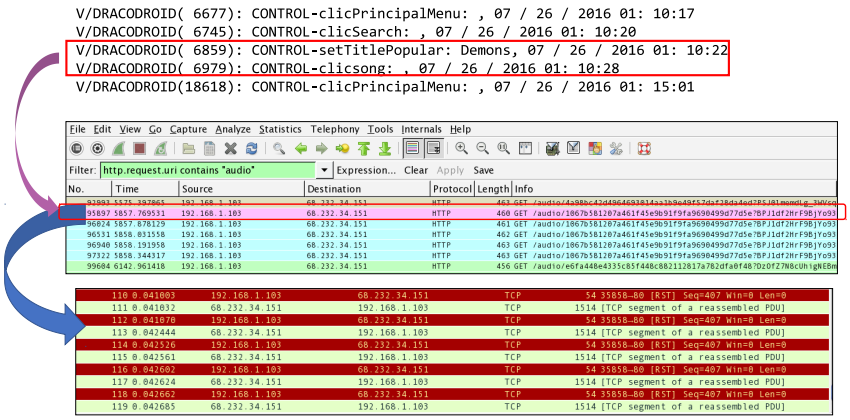


Figura 11.12: Información proporcionada a los desarrolladores para trazar los errores

En la Figura 11.11 se muestra una representación gráfica de la medida de las variables $Z1$, $Z2$ y $Z3$ durante la ejecución del experimento. Es posible ver el tráfico de red anormal correlacionado con los picos altos de los indicadores para todas las variables. Estas son las trazas proporcionadas por MVE como violaciones de la propiedad en la Ecuación 11.2 (negativos en la tabla). Cuando se analizó estas trazas se notó que algunas conexiones HTTP obtenidas no terminaron de manera satisfactoria por varias razones. En la mayoría de los casos, fueron causadas por errores de *timeout* del servidor remoto de *Spotify*. La tercera fila de la Tabla 11.4 muestra el número de violaciones a la propiedad en el intervalo en las que no se detectó *timeout* lo que puede entenderse como un error falso.

Los resultados pueden ser mostrados al desarrollador, con diferentes niveles de detalle. Por ejemplo, la Figura 11.12, muestra los tres niveles de información para la primera anomalia (Canción número 8) de la Figura 11.11.Primero, la canción `Demons` fue seleccionada como parte del caso de prueba, como se muestra en el registro de trazas del monitor en la parte superior de la figura. La acción `clic-song` produce un primer fallo en la solicitud `GET` (se muestra la ventana del WIRESHARK resaltada en color rosa), le siguen varios registros de solicitudes `GET` (se muestran resaltados en cyan). Por otro lado, si se explora internamente el tráfico TCP del primer registro (el fallido) `GET`, entonces se puede ver que desde el cliente se ha cerrado la conexión con la opción `RESET`. Tan pronto como el procedimiento `RESET` se inicia por la APP de Spotify®, el desarrollador descubre que se ha detectado un problema real en la conexión.

11.4.3. Conclusiones del caso de estudio 3

Este caso de estudio presentado muestra la capacidad de la propuesta para identificar los comportamientos de tráfico de red anormales en una aplicación *multimedia* y, cómo las diferentes interacciones de los usuarios pueden influir en los patrones de tráfico inesperados. Primeramente, se ve que cuando un usuario reproduce un determinado título de canción en Spotify®, el tráfico muestra la conexión con la que se descarga el archivo multimedia pero, cuando el usuario reproduce ciertas canciones, el tráfico de red muestra un comportamiento inesperado en el que se produce un cierre abrupto de la conexión al momento de la descarga. Comportamientos, como éstos se pueden valuar, prolongando el tiempo de ejecución de las pruebas para guiarnos a las causas que producen tal comportamiento anormal. Al no tener el código de la aplicación, en este trabajo se ha optado por hacer un análisis estadístico de para determinar la probabilidad de ocurrencia de este comportamiento anormal. De esta misma manera, se puede extraer otro tipo de valores del dispositivo tales como la energía y medir si este comportamiento anormal también tiene algún efecto sobre energía invertida. El caso de estudio también permite mostrar que es posible generar una gran cantidad de casos de prueba de forma automatizada. Las ejecuciones anómalas que tienen desviaciones de los resultados esperados proporcionan información detallada para realizar un análisis post-mortem.

Capítulo 12

Herramienta MVE

12.1. Arquitectura del cliente MVE

En esta sección se describen las principales funcionalidades de la herramienta cliente MVE[1]. El objetivo principal del cliente MVE es proveer a los usuarios una aplicación accesible para analizar sus APPS empleando el *framework* desarrollado en este trabajo, en este capítulo se lo denominará como `framework` MVE. En la Figura 12.1 se observa la arquitectura del componente MVE `Client` que contiene dos funcionalidades principales: *Experiment Definition* y *Verify*. La primera es la funcionalidad que nos permite configurar los parámetros con los que se va a ejecutar un experimento, es decir, el modelo de usuario, la definición de controles de ANDROID, los parámetros de generación de secuencias de casos de prueba a partir del modelo y, finalmente, los parámetros de verificación; tales como la descripción de la propiedad, el tipo de traza que se va a analizar (Tráfico de red, energía, u otro). La segunda funcionalidad ejecuta el experimento con los parámetros definidos anteriormente, muestra el progreso y resultado del análisis en pantalla. Ambas funcionalidades recurren a una tercera funcionalidad llamada *Expermiments Manager* cuyo propósito es gestionar las llamadas al `framework` MVE desde el componente MVE `Client` transformando la información de los experimentos para que sean comprensibles por el `framework` MVE, con en este fin, *Expermiments Manager* almacena todas las configuraciones de experimentos en una base de datos documental `MongoDB` [59].

12.1.1. Configuración del experimento

La Figura 12.2 muestra las funcionalidades de la definición del experimento (*Experiment Definition*) y las relaciones con el módulo *Model Based Test Generator* del `framework` MVE. Para la configuración del experimento, se utiliza el *Model Based Test Generator* para configurar el modelo de comportamiento

[1] Más detalles de la herramienta se pueden encontrar en la página MVE web en `http://www.morse.uma.es/tools/mve`

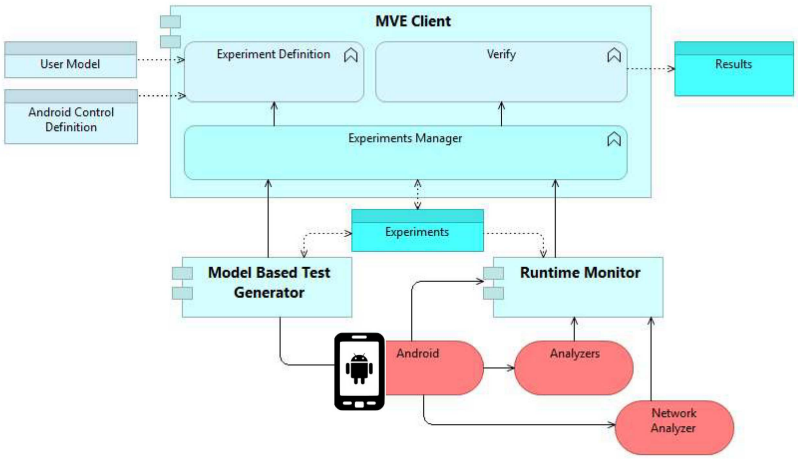


Figura 12.1: MVE `Client`

de usuario (*User Behavior Modeling* en la figura) ademas, para explorar el modelo y generar las secuencias de eventos de usuario (*Sequence Generation* en la figura).

Primeramente, el usuario carga el modelo con las definiciones del comportamiento de usuario (*Upload Model*). A continuación, cada máquina de estados definida en el modelo se configura para que coincidan los *eventos del usuario* (definidos en el modelo) con los controles reales presentes en la pantalla (*Screen Definition Upload* en la Figura). Esto se hace utilizando la herramienta UIAUTOMATORVIEWER del ANDROID SDK que genera archivos de ANDROID *control definition*. Esta funcionalidad se comunica internamente con el componente *Model Based Test Generator* que está encargado de leer el modelo para identificar las máquinas de estados, sus anidamientos y sus relaciones de conexión; luego *Experiments Manager* almacena la información del experimento en la base de datos.

Una vez completada la definición del modelo, se procede a generar las secuencias(*Sequence Generator* en la Figura 12.2 de MVE `Client`). Internamente esta funcionalidad le envía el modelo de comportamiento de usuario al componente de *Model Based Test Generator* para que éste, con la ayuda de SPIN, genere el todo conjunto de secuencias de comportamiento de usuario posible que describe el modelo (tal y como se ve en la Sección 10.1). Esta definición se escribe en formato XML (*Sequence.xml* en la figura). Mas tarde *Experiments Manager* agrega la definición de secuencias la información del experimento y la almacena en la base de datos.

Finalmente se configuran los parámetros de verificación, compuestos por la descripción de la propiedad (descrita en la sección 10.2), y el tipo de traza que se va a capturar para la prueba (tráfico de red, energía o logs). Esta información

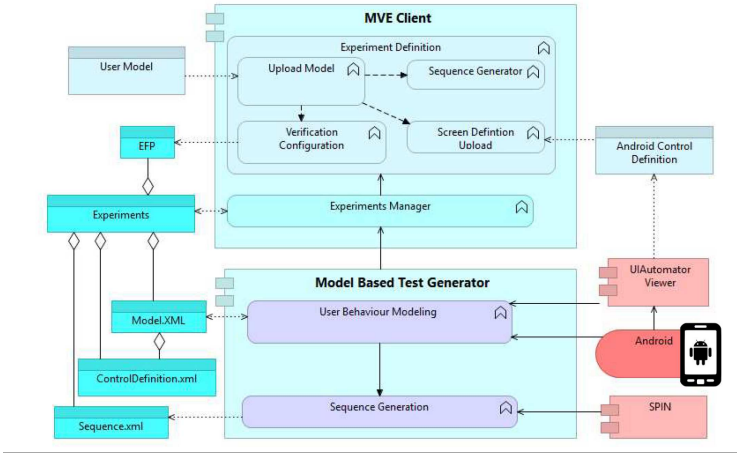


Figura 12.2: Diseño de la funcionalidad de configuración del experimento

de configuración se almacena dentro de la definición del experimento.

La Figura 12.4 muestra la GUI de MVE para crear y configurar experimentos. En la Figura 12.4(a) se pueden ver la siguientes secciones: *User Behaviour Model*, *User Behaviour Sequences*, *Screens*, *Verification Property*, *Verifation Type*. La primera sección despliega la definición del modelo de comportamiento de usuario una vez que se carga. La segunda sección muestra las secuencias generadas a partir del modelo de comportamiento, se puede observar que en la parte superior se configuran los parámetros de máximo/mínimo transiciones y el número máximo de transiciones entre máquinas de estados(*Back Stack Size*). La tercera sección permite configurar los controles definidos en el modelo, para esto se obtiene todas las pantallas(vistas) definidas en el modelo; dado que cada pantalla contiene una o varias una maquinas de estados que a su vez contienen transiciones, a cada transición se le asigna un evento de un control como se ve en la Figura . Finalmente, las dos ultimas secciones permiten la configuración de la verificación de las trazas de ejecución.

12.1.2. Ejecución del experimento, análisis de trazas

Una vez que el análisis está completamente configurado, se puede ejecutar en un dispositivo ANDROID real mediante la funcionalidad de *Verify* descrita en la Figura . Esta funcionalidad ejecuta dos procesos de manera paralela, el primero, la ejecución de los casos de prueba generados sobre el dispositivo (*Test Execution* de *Verify*), el segundo, el análisis de propiedades sobre la traza capturada del dispositivo (*Verify Properties* de *Verify*). A medida que se ejecutan ambos procesos se sincronizan para el despliegue de los resultados, de manera que a cada ejecución de un evento de usuario le corresponde un veredicto tras la evaluación de la propiedad.

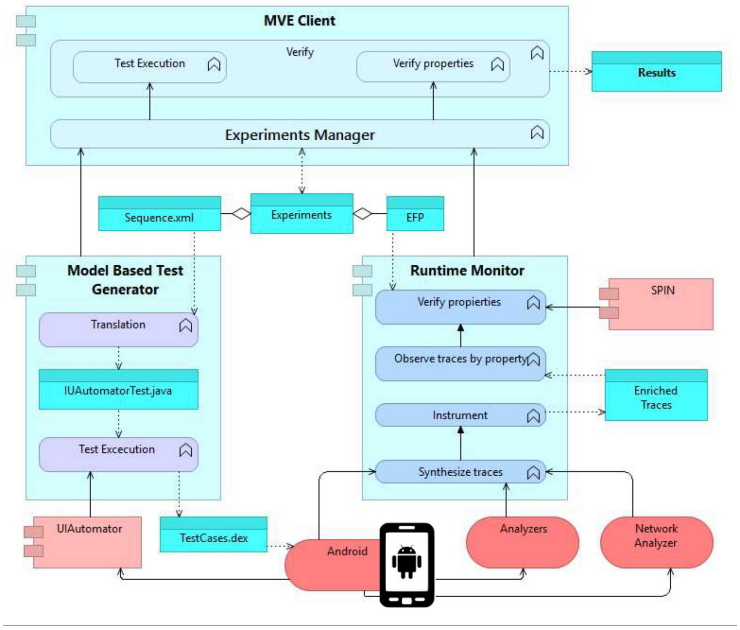


Figura 12.3: Diseño de la funcionalidad de verificación del experimento

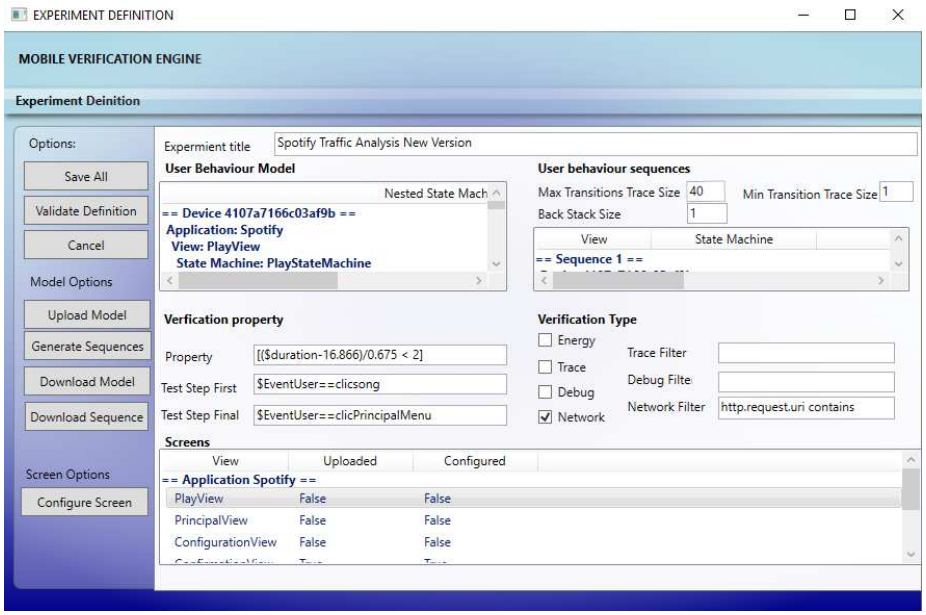
Para la ejecución del proceso de *Test Execution* de *Verify* se utilizan las secuencias generadas en la configuración del experimento como datos de entrada del componente *Model Based Generator* del MVE *framework*. Debe notarse que las secuencias están almacenadas dentro de la configuración del experimento, por lo que en la Figura 12.3 se ve a *Sequence.xml* agregado a *Experiments*. Luego el proceso continua continua a partir del paso de traducción como se explica en la Sección 10.1.

Por otro lado, el proceso de *Verify Properties* de *Verify* llama al componente de *Funtime Monitor* del `framework` MVE para analizar las trazas capturadas. Para este fin utiliza la configuración almacenada dentro del experimento como datos de entrada, la cual indica la propiedad a ejecutar (EFP) y el tipo de traza que necesita para el análisis. Luego, se se ejecuta todo el proceso descrito en la Sección 10.2.

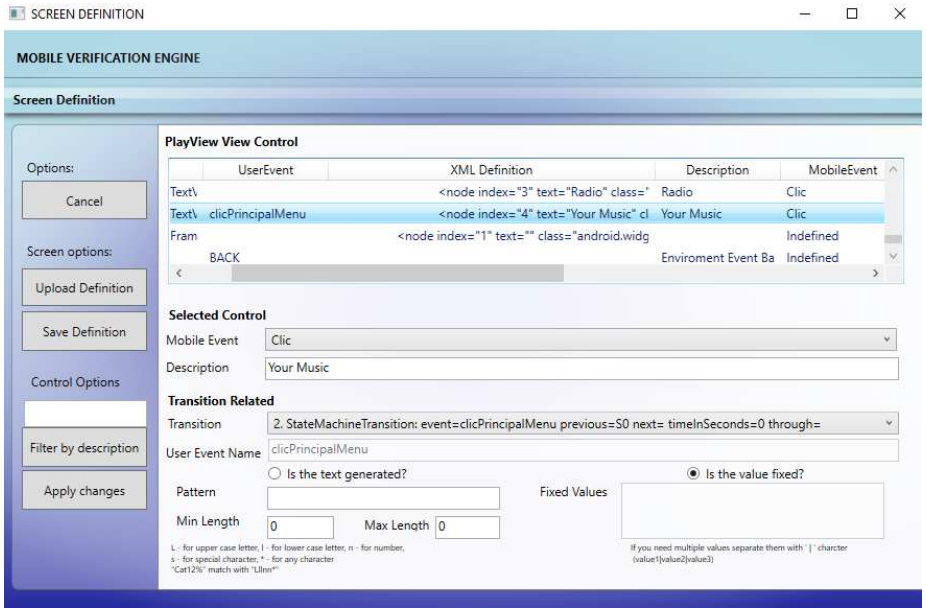
Todo el proceso descrito anteriormente se ejecuta a través de la pantalla *Verify Experiment* que se muestra en la Figura 12.5. Antes de ejecutar el experimento, el usuario debe establecer el número serie del dispositivo que va a utilizar; luego, durante la ejecución del experimento, se muestra el estado de cada caso de prueba y las acciones del usuario enviadas al dispositivo, junto con algunas estadísticas de tráfico de la red.

12.1.3. Conclusiones

En éste capitulo se ha presentado la herramienta MVE `Client`, diseñada para facilitar la utilización del `framework` de análisis de dispositivos móviles *smart* desarrollado en esta tesis. Por un lado permite la configuración de los experimentos de prueba; y por otro lado, permite la evaluación de propiedades extra-funcionales sobre trazas capturadas de tráfico. Queda pendiente una extensión a la herramienta que permita diseñar gráficamente el modelo de comportamiento de un usuario dentro de la herramienta. Además, cabe recalcar que se puede hacer uso de los componentes del `framework` MVE de manera aislada, es decir, que se puede utilizar sin necesidad de un cliente a través de línea de comandos de consola. Lo cual permite escalar cada componente por separado, y utilizar el `framework` a través de nuevos clientes, si el usuario lo considera necesario.



(a) Modelo de comportamiento de usuario, generación de casos de prueba y, verificación de propiedades



(b) Pantalla y eventos de usuario

Figura 12.4: Configuración de las vistas de *App* con la interfaz de usuario

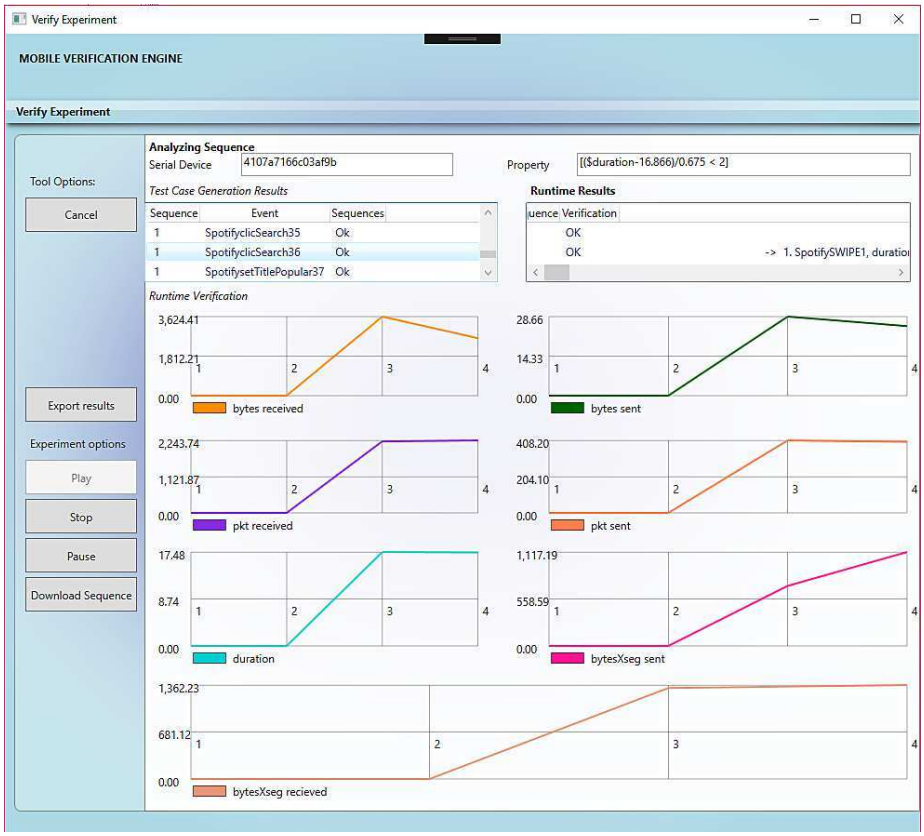


Figura 12.5: Pantalla de verificación del experimento

Parte V

Análisis final

Capítulo 13

Conclusiones y discusiones

13.1. Trabajos Relacionados

Hay algunas propuestas existentes que aplican pruebas basadas en modelos a aplicaciones ANDROID. Algunas de estas propuestas asumen que el proceso de prueba comienza sin un modelo preciso del comportamiento esperado de las aplicaciones y se centran en las técnicas para obtener dicho modelo. El framework *MobiGUITAR* [60] construye automáticamente una máquina de estados de una *App* ejecutando eventos y grabando un árbol con eventos para cada nuevo estado. Los autores utilizan un recorrido de la interfaz gráfica de aplicaciones, que es compatible con aplicaciones de código abierto. De esta manera, no están considerando tener ningún conocimiento sobre la forma de utilizar la *App*, sino que se lleva a cabo una ejecución exhaustiva. Por lo tanto, necesitan algunos criterios para encontrar equivalencias entre estados que permitan prevenir la explosión del estados. La técnica *Swift-Hand* utilizada en [61] construye un modelo aproximado a partir del aprendizaje automático de la *App* durante el proceso de prueba. Su objetivo es cubrir tanto comportamientos como sea posible, obligando a la ejecución a entrar en partes inexploradas del espacio de estados. Otros enfoques también consideran la especificación formal de las *Apps* para iniciar la generación de pruebas. En [62], los autores describen un método basado en la utilización de propiedades mediante la construcción de los modelos en *Alloy*, que es un lenguaje formal basado en lógica de primer orden. En su propuesta, el papel del modelo verificador, que se corresponde con el enfoque de esta tesis, es llevado a cabo por un analizador de *Alloy*, que genera positivos (esperados) y negativos (no deseados) en los casos de prueba.

En comparación con *MobiGUITAR*, en MVE la generación de los casos de prueba y los estados de sus máquinas de estados de alto nivel son limitados y se diferencian por diseño. Por lo tanto, nuestros modelos son más compactos y, por ejemplo, en comparación con *MobiGUITAR* no se necesita realizar trabajos adicionales para eliminar casos de prueba poco realistas. Además, nuestro enfoque permite la generación de casos de prueba para varias *Apps* a la vez que

interactúan usando ANDROID Intents, mientras que la complejidad del proceso de modelado comparado con el las propuestas en tiempo de ejecución de *MobiGUITAR* y *Swift-Hand* las hace más adecuadas para aplicaciones individuales. Como MVE, en [62], los autores usan transformaciones basadas en XML para traducir los casos de prueba a un archivo ejecutable que permite interactuar con las *Apps* que se están analizando. Aparte de las tecnologías internas (verificación de modelos vs. analisis de restricciones), la principal diferencia en ambas propuestas es la forma de obtener el modelo ejecutable refinado. La especificación de *Alloy* en [62] se construye manualmente, mientras que la especificación PROMELA en nuestro trabajo se realiza automáticamente desde el diseño de alto nivel de las máquinas de estados de vistas. Todavía tenemos que trabajar en el mismo caso de estudio para obtener una comparación cuantitativa en el esfuerzo humano y computacional que es requerido en ambos enfoques.

Muchas herramientas comerciales y académicas han sido desarrolladas para monitorizar y analizar algunos tipos de propiedades extra-funcionales en teléfonos móviles. Un primer grupo de herramientas realiza análisis de tráfico. *AntMonitor* [63] es una potente herramienta que supervisa todas las aplicaciones conectadas en el dispositivo ANDROID para producir estadísticas, por ejemplo, sobre cómo cada aplicación contribuye al tráfico total de la red, pero no para eventos específicos o interacciones de usuarios de manera sistemática. *NetworkProfiler* [64] es una herramienta orientada a ayudar a los operadores celulares a identificar el tráfico en sus redes cuando son transportados a través de *HTTP/HTTPS*. *NetworkProfiler* utiliza emuladores de dispositivo y técnicas de aprendizaje automático para crear un identificador de las *Apps* mediante la recopilación de información sobre los servidores a los que se conectan. Dado que la exploración manual de las aplicaciones que se están probando no cubre todos los comportamientos, *NetworkProfiler* genera aleatoriamente eventos de usuario para interactuar con las aplicaciones. Esta estrategia aleatoria también es utilizada por otras herramientas como *Monkey* [40]. Otras herramientas, como *Monkeyrunner* [65], *Robotium* [66] y *Troyd* [67] emplean *scripts* para hacer que las aplicaciones funcionen con una secuencia predefinida de interacciones de usuarios. Sin embargo, ninguna proporciona una metodología automatizada para generar dichos *scripts*. *ProfileDroid* [68] está diseñado para monitorizar de forma sistemática las *Apps* con el fin de detectar inconsistencias o comportamientos no usuales. Se basa en un análisis multicapa de las *Apps*, considerando el análisis estático del código de bytes, las interacciones de los usuarios, las llamadas al sistema operativo y el tráfico de red. La herramienta requiere un usuario real interactuando con el teléfono, pero la secuencia de interacciones se almacena para poder ser reproducida más tarde en un escenario distinto. El enfoque de *Automatic Android App Explorer* (A3E) [69] es utilizar un análisis de flujo de datos estático de la aplicación compilada, para construir un modelo de alto nivel del flujo de control que captura las transiciones entre actividades/pantallas. Luego, se utiliza la exploración para alcanzar una cobertura de 64 por ciento de actividad y un 36 por ciento de cobertura de método en algunas aplicaciones típicas de ANDROID. Este modelo del flujo tiene el mismo papel que los modelos de MVE para representar las interacciones del usuario, y la

búsqueda exhaustiva es similar a nuestro enfoque para la generación de pruebas exhaustivas. *ProfileDroid* y *A3E* son las propuestas más cercanas a nuestro trabajo, y de hecho podrían ser adoptadas en nuestra metodología.

En comparación con *ProfileDroid* y *A3E*, las principales novedades de nuestra metodología son: (a) la cobertura controlada de las interacciones de los usuarios para una o varias aplicaciones debido al enfoque basado en modelos para la generación de pruebas, (b) la capacidad de encontrar automáticamente trazas de ejecución que no cumplan el comportamiento esperado de la aplicación a través de Internet, (c) la inclusión del tiempo en los modelos que permite probar situaciones reales entre las interacciones del usuario, (d) la capacidad de combinar modelos de varias aplicaciones funcionando en paralelo, (e) una técnica para comparar automáticamente el comportamiento real de cada ejecución de las aplicaciones contra el comportamiento esperado. Se podría utilizar *ProfileDroid* para generar los patrones de referencia que se emplean para analizar el comportamiento real de las aplicaciones. También se podría utilizar *A3E* como un método para ayudar en la construcción de los modelos de las aplicaciones.

Adicionalmente, existen referencias a la investigación sobre las propiedades extra-funcionales aplicadas a la estimación del consumo de energía. Phatak et al. (Pathak-HotNets2011) proporcionaron una de las primeras clasificaciones de fallos de energía para hardware y software y propusieron una serie de pasos para desarrollar un *framework* de diagnóstico para tratar estos fallos de energía. Más tarde, Phatak y otros autores presentaron la herramienta **eprof** [70], un generador de energía de grano fino usado para obtener información sobre el uso de energía de los teléfonos inteligentes. Simultáneamente, Yepang Liu [71] también estudió errores de energía (sus tipos y manifestaciones) e identificó patrones comunes. Los investigadores implementaron un analizador de código estático, **PerfChecker** para detectar e identificar patrones de errores. El proyecto **Eloupe** [72] [73] explora una alternativa que mitiga los efectos negativos de una aplicación que requiere mucha energía. El *framework* consiste en un monitor de datos en el móvil que se procesa en la nube para detectar el riesgo de drenar la energía y, de esta manera, producir información para aislar las aplicaciones peligrosas. Las fugas de memoria pueden ser también una razón para el consumo de energía. En [74], los autores diseñan un detector de fugas de memoria que se centra en la fugas de energía de las *activities* y contiene un módulo de priorización que permite decidir la eliminación de las *Apps* que con tienen fugas de energía. De esta manera, se construye un *framework* para detectar fugas de energía utilizando el análisis dinámico. Por último, la energía consumida entre diferentes plataformas móviles se estudia en [75] [76] [77]. El primero compara los errores de energía, el segundo compara la eficiencia energética, y el último, ademas, desarrolló un método de estimación de potencia utilizando trazas de batería.

La propuesta de la herramienta MVE que combina la generación de casos de prueba a partir de modelos con la monitorización del consumo de energía, como la mayoría de las propuestas antes mencionadas. Sin embargo, no se utiliza este resultado directamente, sino como una entrada para un análisis más sofisticado. Se utiliza la lógica de intervalos para representar propiedades de energía para

conducir la identificación de comportamientos incorrectos de las aplicaciones que ejecutan el teléfono inteligente. Como resultado, nuestra técnica de verificación en tiempo de ejecución es útil para detectar de manera muy precisa las *Apps* que drenan energía. Por ejemplo, se proporciona la secuencia de ejecución exacta de una o varias aplicaciones que están causando que el sistema pierda más energía de lo esperado.

Thompson et al. presentan una metodología y una herramienta denominada *SPOT* [78] que permite modelar la arquitectura de una aplicación y emular su consumo de energía durante la fase de diseño. La aplicación se modela utilizando abstracciones configurables de elementos que normalmente contribuyen al consumo de energía, como GPS y conexiones de red, asociadas a actividades o servicios de fondo. La herramienta *SPOT* que se ejecuta en el dispositivo ANDROID utiliza este modelo para emular cada uno de los componentes. El consumo de energía de una emulación se registra utilizando una API ANDROID, y se retroalimenta al programador con los resultados.

Aunque SPOT también utiliza un enfoque basado en modelos, nuestra herramienta está orientada a modelar comportamientos de usuario y utilizar el código de aplicación real, en lugar de una arquitectura simplificada y componentes emulados. Junto con el uso de equipos externos para la medición de energía, se proporciona un análisis más preciso de un perfil de consumo real de la aplicación. Además, podemos correlacionar las mediciones de energía con otras fuentes de tiempo de ejecución reales, como el tráfico de red o la información de depuración, para analizar otras propiedades funcionales o extra-funcionales.

Por último, también se ha aplicado el modelo de verificación con las propiedades extra-funcionales (EFP) para estimar el consumo de energía en redes de sensores inalámbricas. En [79], los autores utilizan un modelo *UPPAAL* de la red que incluye aspectos tales como tiempo, ancho de banda y energía. A continuación, utilizan el modelo de verificación con diferentes escenarios para predecir la influencia de un determinado conjunto de parámetros en el consumo de energía, n particular, los sensores y enrutadores. Nakajima ha introducido recientemente dos nuevos formalismos para tratar este problema, el *consuming power automaton* para representar el sistema bajo análisis y una versión de LTL con cuantificadores para representar el consumo de energía esperado. La presencia de tiempo y energía en los modelos y fórmulas hace que este problema de verificación del modelo sea complejo, y el autor propone varios subconjuntos prácticos para ejecutar esta verificación. Si se compara con estas propuestas, MVE defiende la idea de una lógica específica para representar las propiedades de energía y la mecánica de comprobación de modelos para comprobar esta lógica. Además, en lugar de explorar un modelo de un sistema, tratamos las trazas de ejecución extraídas de dispositivos reales con medidas precisas de consumo de energía para cada período de tiempo.

13.2. Conclusiones

Este documento ha presentado una plataforma completa para verificar propiedades extra-funcionales en aplicaciones móviles (*Apps*) de ANDROID en tiempo de ejecución. El método se basa en dos lenguajes formales para especificar a) las interacciones usuario-aplicación como la entrada para generar automáticamente casos de prueba, y b) las propiedades extra-funcionales esperadas que las ejecuciones de la aplicación deben satisfacer. Tanto la generación de pruebas como la verificación de las propiedades se realizan utilizando la comprobación de modelos. El documento también presenta una implementación y evaluación de la propuesta con aplicaciones reales.

Entre las principales aportaciones cabe notar que se ha definido un lenguaje para modelar el comportamiento de los usuario interactuando con los dispositivos móviles basado en máquinas de estado finitas. Cada una de las máquinas representa, una APP, una vista de una APP, y una secuencia de pasos lógicos dentro de una vista. Ademas, las máquinas se pueden anidar y componer entre ellas para describir comportamientos típicos más complejos de las APPS de un dispositivo móvil; por ejemplo, una máquina de estados que en primera instancia describe a un usuario seleccionando una entrada de Twitter® con un enlace a YouTube®, y por otro lado, una segunda máquina de estados que describe a un usuario que reproduce/pausa/detiene un vídeo de YouTube®; notándose que ambas máquinas se pueden combinar o ejecutar de manera independiente. La definición de este lenguaje se ha formalizado proporcionando una semántica operacional formal para la generación de los casos de prueba, en la cual se describe, la especificación formal de los tipos de estados, los tipos de transiciones posibles entre estados y, finalmente, los efectos de las transiciones en estados futuros. Como resultado, se obtiene un lenguaje desacoplado que permite extender, mantener y reutilizar los comportamientos de una persona con relativa facilidad. Se han modelado varias APPS para probar las capacidades del modelo, entre muchas otras, se ha modelado Spotify®, Facebook®, YouTube®, además de algunas propias del equipo de investigación.

A partir del modelo de comportamiento se han generado automáticamente todas las secuencias de interacciones del usuario-APP posibles. Se ha utilizado el *model checker* SPIN para explorar el modelo previamente traducido a una especificación PROMELA. Luego, el código PROMELA genera bucles por la cantidad de secuencias que encuetre, donde cada bucle representa una secuencia no determinista para cada dispositivo cuyas ramas se corresponden con una transición en el modelo. Cada rama del bucle se ha traducido a código JAVA para ser ejecutado sobre el/los dispositivo(s) utilizando el API UIAUTOMATOR de ANDROID que emulan eventos sobre la interfaz de usuario.

Por otro lado, para evaluar propiedades, se ha utilizado un monitor en tiempo de ejecución que se encarga de capturar las trazas del dispositivo, filtrarlas y evaluarlas. En la captura información se ha utilizado las APIs de DDMLIB y ADB propios de ANDROID, JDI de JAVA, ademas de TCPDUMP para la captura de tráfico. Las trazas se han filtrado utilizando programación orientada a aspectos con la herramienta de SPRING AOP. Finalmente, las propiedades se

han evaluado en observadores que analizan, cada uno, solo una propiedad. Cabe recalcar que en esta tesis muestra como se pueden evaluar propiedades extra-funcionales, aunque aunque se podría extender para el análisis de propiedades funcionales[1].

Como otra de las aportaciones principales de este trabajo se ha introducido un nuevo lenguaje de especificación, que se basa en lógica de intervalos, para describir propiedades extra-funcionales sobre las trazas de ejecución. El lenguaje permite evaluar trazas que contienen variables que evolucionan en el tiempo (la energía consumida, temperatura del dispositivo, paquetes de red descargados, entre otros). Para las variables que evolucionan en el tiempo, se considera la diferencia de sus valores entre los tiempos t_1 y t_2. Los estados t_1 y t_2 son los que determinan el intervalo y se corresponden con los estados que conforman la traza ejecución. Se han presentado tres fórmulas de intervalos que son la base de lenguaje que describen la sincronización implícita entre la evolución discreta de las trazas y la evolución continua de los valores de las magnitudes que se evalúan sobre las trazas. Finalmente, dichas formulas se han formalizado y se han traducido a LTL de manera que pueda ser analizado por el *model checker* de SPIN.

Se ha presentado la arquitectura del *framework* de verificación *Mobile Verification Engine*, que contiene dos módulos principales. El primer módulo, denominado *Model Based Test Generator*, es el encargado emular el comportamiento de usuario sobre un(os) dispositivo(s) móvil(es) *smart*, es decir, genera el modelo de comportamiento de usuario en un archivo XML luego, genera la secuencia de de eventos para traducirlos a casos de prueba de JAVA que son ejecutados sobre el dispositivo. El segundo módulo, denominado *Runtime Verification*, es el encargado del análisis de las propiedades sobre las trazas de ejecución, es decir, sintetiza las trazas de los distintos dispositivos, instrumenta las trazas, y finalmente evalúa las propiedades definidas sobre todas las trazas y emite resultados. Mientras el primer módulo ejecuta los casos de prueba sobre el dispositivo, el segundo módulo analiza las trazas capturadas.

Para evaluar la propuesta se han desarrollado tres casos de estudio principales. En el primer caso de estudio se ha mostrado como se modela el comportamiento de usuario, y como se generan automáticamente los casos de prueba. Se ha descrito un modelo que contiene de la interacción entre dos aplicaciones Facebook® y YouTube® sobre dos dispositivos, de esta manera, se mostraron todas las posibilidades del lenguaje propuesto. El resultado de este estudio indica que el numero de transiciones entre máquinas de estados tiene un alto impacto en la generación de estados, por lo tanto, aunque es mayor el numero de estados explorados, es necesario hacer un análisis del modelo de comportamiento y de la cantidad de transiciones entre máquinas de estados para evitar un posible problema de explosión de estados.

En el segundo caso de estudio se ha evaluado la energía consumida al descargar un archivo de mediante HTTP; para este caso se ha utilizado una aplicación

[1] De hecho, se ha introducido una breve ejemplo de análisis de una propiedad funcional, para analizar el ciclo de vida de ANDROID

propia del grupo de investigación. En este caso de estudio demostrado como nuestra propuesta es utilizó para comprobar el consumo de energía de una secuencia de acciones dentro de un caso de prueba. Para esto se ha determinado una propiedad de intervalo extra-funcional(descrita en nuestro lenguaje de especificación) que evalúa si la energía consumida en un intervalo es menor a un valor específico. Los resultados de este estudio muestran que como el monitor captura las medidas de energía, las evalúa entre los estados de la traza (que representan al usuario interactuando con la interfaz del dispositivo) y detecta cuando el consumo de energía es superior al establecido.

El tercer caso de estudio es un experimento que se ha realizado sobre la aplicación de Spotify® para ANDROID para examinar patrones de tráfico generados por la aplicación al reproducir una serie de canciones. En este experimento demuestra la capacidad de la propuesta para identificar los comportamientos de tráfico de red anormales en una aplicación multimedia y cómo las diferentes interacciones de los usuarios pueden influir en los patrones de tráfico inesperados; tal como el que se describe en el experimento, donde se muestra que reproducir algunos títulos de canciones provoca un cierre abrupto en la conexión.

Como aportación final se presenta una nueva herramienta de software llamada MVE, la cual permite utilizar el *framework* descrito, incluyendo una interfaz de usuario para una interacción más amigable. La herramienta tiene dos funcionalidades principales La primera, permite la configuración de la prueba, es decir, la configuración del modelo, la generación de secuencias, el enlace del modelo con los controles de usuario y, finalmente, la parametrización de la propiedad y el entorno de ejecución. Por otro lado, la segunda funcionalidad contempla la ejecución de los casos de prueba sobre el dispositivo simultáneamente que se evalúan las propiedades sobre las capturas de trazas. La segunda funcionalidad muestra gráficamente el resultado de la ejecución de las secuencias indicando cuales han terminado con errores y cuales no para proporcionar contra ejemplos. A partir de todo lo anterior se muestra que la propuesta es flexible al momento de incluir diversos tipos de dispositivo en las pruebas, contemplando uno o varios dispositivos físicos. En este sentido se ha decido no utilizar emuladores, por que no se puede determinar que las acciones ejecutadas en los dispositivos móviles están libres de errores hasta no analizar cual es su efecto sobre el hardware, por ejemplo, es posible que al ejecutar una acción en un dispositivo no se presenten errores, pero al ejecutar la misma acción en otro dispositivo de distinto hardware si se presenten errores. Por lo tanto, el hardware tampoco ha sido alterado para evitar contaminar los resultados. Los tipos de dispositivos móviles que se han utilizado a lo largo de la investigación en las pruebas son, en orden cronológico, *Sony Ericsson Xperia X10 Mini Pro*, *Samsung Galaxy S2*, *Sony Ericsson Xperia arc* y, los utilizados en los últimos casos de prueba, *Sony Xperia Z2*, *Samsung Galaxy S6*, *Samsung Galaxy Tab 2*

La versión del sistema operativo puede determinar si existe o no un error en la ejecución de una acción, al igual que el hardware puede alterar el resultado de una acción sobre un dispositivo. En este sentido, la implementación es independiente a las versiones ANDROID y se ha adaptado correctamente a su evolución a lo largo de la investigación, las versiones de ANDROID utilizadas son: *Froyo,*

Gingerbread, Ice Cream Sandwich, Jelly Bean, KitKat, Lollipop, Marshmallow.

En conclusión, el conjunto de aportes presentados en este estudio permite analizar propiedades extra-funcionales sobre una o varias APPS al mismo tiempo, que emulan el comportamiento de los usuarios sobre sobre uno o varios dispositivos simultáneamente. De esta manera, se pueden encontrar de forma automatizada los fallos que corresponden a la alta coordinación que requieren estas plataformas, que se usualmente se exponen a medida que un usuario interactua con su dispositivo movil *smart*.

13.3. Trabajo Futuro

- **Análisis de cobertura:**

  Queda pendiente un análisis más profundo sobre el nivel de cobertura que se está consiguiendo con la utilización de cada técnica, tanto de manera asilada así como combinada. De esta manera se propone establecer una formula que permita calcular el nivel de cobertura que estamos alcanzando cuando analizamos una propiedad con un determinado modelo.

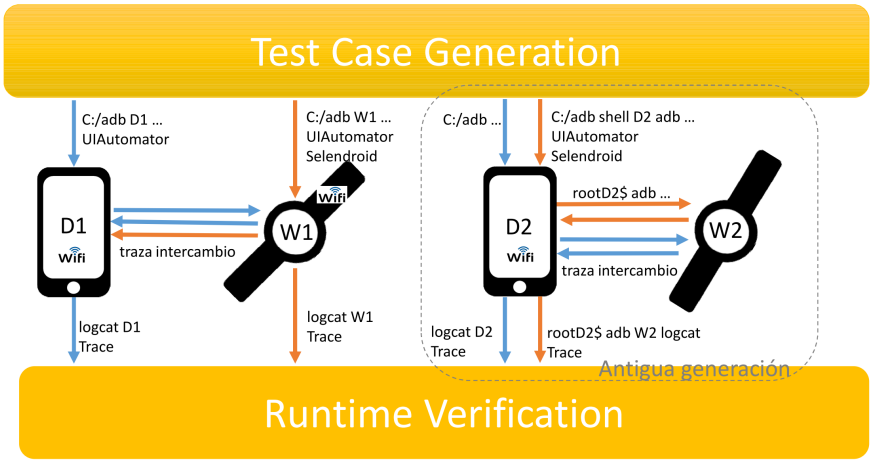
- **Análisis Multiplataforma:**

  Como una línea de investigación abierta por este trabajo se propone adaptar el enfoque presentado en este trabajo a otros sistemas operativos móviles, como *iOS* o *Windows Mobile*. La mayoría de los componentes podrían reutilizarse con cambios menores o sin cambios, como el lenguaje para EFP y los mecanismos para generar casos de prueba. Sin embargo, algunos de ellos requieren un estudio adicional, como el lenguaje de modelado, para describir las interacciones del usuario o el control de las ejecuciones en el teléfono inteligente.

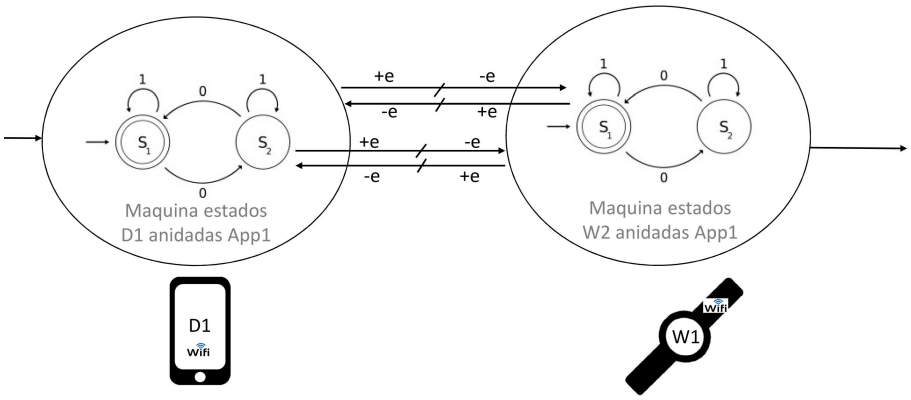
- **Análisis de otro tipo de propiedades:** otra de las líneas que han quedado abiertas es el análisis de propiedades diferentes a las que se han cubierto en este estudio; por ejemplo: propiedades funcionales que nazcan de los requerimientos de la APP, propiedades de seguridad para determinar si una combinación de interacciones de usuario expone una vulnerabilidad de las APPS. De la misma forma, es posible profundizar en el análisis propiedades combinadas, como analizar si un fallo en la conexión de red tiene un efecto en el consumo de energía, o determinar si una APP envía/recibe más tráfico de red debido a un error de la aplicación.

- **Análisis de *Smartwatches***

  Finalmente, una futura línea de investigación derivada de este estudio sería la adaptación de esta propuesta para la verificación de dispositivos *SmartWatches*, que a diferencia de las tabletas y de los *smartphones*, los eventos de usuario se combinan altamente con los estímulos que reciben a través de sus sensores. Además, la mayoría de este tipo de dispositivos



(a) Escenarios de análisis de smartwatches



(b) Sincronización de máquinas de estados para Smartwatches

Figura 13.1: Propuesta de verificación de APPS de *SmartWatches*

están diseñados para funcionar de manera combinada con un *smartphone*. Existen dos tipos de escenarios para analizar este tipo de dispositivos, como se muestra en la Figura 13.1(a), a la izquierda muestra propuesta de verificación de los dispositivos ANDROID a los que se puede acceder a su sistema operativo directamente y, a la derecha se muestra la propuesta para analizar los dispositivos a los que solo se puede acceder a su sistema operativo mediante un *smartphone*. Ambas propuestas son para dispositivos ANDROID y utilizan los componentes de *MVE* de *Test Case Generation* y *Runtime Verification*. Para lo anterior se hace necesario realizar estudios más profundos, tales como: analizar de que manera se deben incluir los eventos de estimulación de sensores en la definición del modelo

de comportamiento de usuario, analizar cómo emular comportamientos de los sensores, y de que manera se sincronizan los eventos entre dispositivos (Figura 13.1(b)). Por otro lado, éste sería un buen un punto de partida para hacer extender el presente trabajo a dispositivos *Wearables* e *IOT*.

Bibliografía

[1] E. M. Clarke, Jr., O. Grumberg, D. A. Peled, Model Checking, MIT Press, Cambridge, USA, 1999.

[2] C. Baier, J.-P. Katoen, Principles of Model Checking (Representation and Mind Series), The MIT Press, 2008.

[3] E. M. Clarke, O. Grumberg, D. E. Long, Model checking and abstraction, ACM Trans. Program. Lang. Syst. 16 (5) (1994) 1512–1542.

[4] G. J. Myers, Art of Software Testing, John Wiley & Sons, Inc., New York, NY, USA, 1979.

[5] S. Ali, T. Yue, Formalizing the iso/iec/ieee 29119 software testing standard, in: 2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS), 2015, pp. 396–405. doi:10.1109/MODELS.2015.7338271.

[6] M. Utting, B. Legeard, Practical Model-Based Testing: A Tools Approach, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[7] A. Pretschner, Model-based testing, in: Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005., 2005, pp. 722–723. doi:10.1109/ICSE.2005.1553582.

[8] A. C. Dias Neto, R. Subramanyan, M. Vieira, G. H. Travassos, A survey on model-based testing approaches: A systematic review, in: Proceedings of the 1st ACM International Workshop on Empirical Assessment of Software Engineering Languages and Technologies: Held in Conjunction with the 22Nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007, WEASELTech '07, ACM, New York, NY, USA, 2007, pp. 31–36. doi:10.1145/1353673.1353681.
URL http://doi.acm.org/10.1145/1353673.1353681

[9] M. Leucker, Teaching Runtime Verification, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 34–48.
URL http://dx.doi.org/10.1007/978-3-642-29860-8_4

[10] Y. Falcone, K. Havelund, G. Reger, A tutorial on runtime verification., Engineering Dependable Software Systems 34 (2013) 141–175.

[11] C. Watterson, D. Heffernan, Runtime verification and monitoring of embedded systems, IET Software 1 (5) (2007) 172–179. doi:10.1049/iet-sen:20060076.

[12] SPIN, Spin reference.
URL `http://spinroot.com/spin/whatispin.html`

[13] A. Salmerón, P. Merino, Integrating model checking and simulation for protocol optimization, Simulation 91 (1) (2015) 3–25. doi:10.1177/0037549714557054.
URL `http://dx.doi.org/10.1177/0037549714557054`

[14] A. R. Espada, M.-M. Gallardo, A. Salmerón, P. Merino, Using model checking to generate test cases for android applications, in: N. Pakulin, A. K. Petrenko, B.-H. Schlingloff (Eds.), Proceedings Tenth Workshop on Model Based Testing, Vol. 180 of Electronic Proceedings in Theoretical Computer Science, Open Publishing Association, 2015, pp. 7–21.

[15] A. R. Espada, M. del Mar Gallardo, A. Salmerón, P. Merino, Runtime verification of expected energy consumption in smartphones, in: B. Fischer, J. Geldenhuys (Eds.), Proceedings of 22nd International Symposium on Model Checking Software (SPIN 2015), Vol. 9232 of Lecture Notes in Computer Science, Springer, Cham, 2015, pp. 132–149. doi:10.1007/978-3-319-23404-5-10.
URL `http://dx.doi.org/10.1007/978-3-319-23404-5-10`

[16] D. MacLean, S. Komatineni, G. Allen, Pro Android 5, 5th Edition, Apress, Berkely, CA, USA, 2015.

[17] AndroidDev, Android developers.
URL `http://developer.android.com/`

[18] AndroidNDK, Android native developers.
URL `https://developer.android.com/ndk/index.html?hl=es-419`

[19] A. Abran, P. Bourque, R. Dupuis, J. W. Moore (Eds.), Guide to the Software Engineering Body of Knowledge - SWEBOK, IEEE Press, Piscataway, NJ, USA, 2001.

[20] K. Schneider, Verification of Reactive Systems: Formal Methods and Algorithms, SpringerVerlag, 2004.

[21] A. Pnueli, The temporal logic of programs, in: 18th Annual Symposium on Foundations of Computer Science (sfcs 1977), 1977, pp. 46–57.

[22] A. P. Sistla, E. M. Clarke, The complexity of propositional linear temporal logics, J. ACM 32 (3) (1985) 733–749.

[23] M. Y. Vardi, An automata-theoretic approach to linear temporal logic, Springer Berlin Heidelberg, Berlin, Heidelberg, 1996, pp. 238–266.

[24] J.-M. Couvreur, On-the-fly Verification of Linear Temporal Logic, Springer Berlin Heidelberg, Berlin, Heidelberg, 1999, pp. 253–271.

[25] L. Tan, O. Sokolsky, I. Lee, Specification-based testing with linear temporal logic, in: Proceedings of the 2004 IEEE International Conference on Information Reuse and Integration, 2004. IRI 2004., 2004, pp. 493–498.

[26] P. Godefroid (Ed.), Concurrent systems and semantics, Springer Berlin Heidelberg, Berlin, Heidelberg, 1996, pp. 19–26.

[27] E. Clarke, O. Grumberg, D. Long, Verification tools for finite-state concurrent systems, Springer Berlin Heidelberg, Berlin, Heidelberg, 1994, pp. 124–175.

[28] D. Dams, R. Gerth, O. Grumberg, Abstract interpretation of reactive systems, ACM Trans. Program. Lang. Syst. 19 (2) (1997) 253–291.

[29] P. Cousot, R. Cousot, Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints, in: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL '77, ACM, New York, NY, USA, 1977, pp. 238–252.

[30] K. Havelund, Implementing runtime monitors. Extended abstract., Invited talk, 2nd TORRENTS Workshop (Time ORiented Reliable Embedded NeTworked Systems (2012).

[31] M. "Mernik, "Domain-Specific Languages: A Systematic Mapping Study", "Springer International Publishing", Çham", "2017", pp. "464–472".
URL "http://dx.doi.org/10.1007/978-3-319-51963-0_36"

[32] D. Leivant, Typing and computational properties of lambda expressions, Theoretical Computer Science 44 (1986) 51 – 68. doi:http://dx.doi.org/10.1016/0304-3975(86)90109-X.
URL http://www.sciencedirect.com/science/article/pii/0304397586901 09X

[33] M. Martin, B. Livshits, M. S. Lam, Finding application errors and security flaws using pql: A program query language, in: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, OOPSLA '05, ACM, New York, NY, USA, 2005, pp. 365–383. doi:10.1145/1094811.1094840.
URL http://doi.acm.org/10.1145/1094811.1094840

[34] H. Barringer, D. Rydeheard, K. Havelund, Rule Systems for Run-Time Monitoring: From Eagle to RuleR, Springer Berlin Heidelberg, Berlin, Heidelberg, 2007, pp. 111–125.

[35] K. Havelund, G. Roşu, Efficient monitoring of safety properties, International Journal on Software Tools for Technology Transfer 6 (2) (2004) 158–173.

[36] A. Bauer, M. Leucker, C. Schallhart, Runtime verification for ltl and tltl, ACM Trans. Softw. Eng. Methodol. 20 (4) (2011) 14:1–14:64. doi:10.1145/2000799.2000800.
URL http://doi.acm.org/10.1145/2000799.2000800

[37] Aspectj.
URL https://eclipse.org/aspectj

[38] Spring aop.
URL https://docs.spring.io/spring/docs/current/spring-framework-reference/html/aop.html

[39] M. Utting, B. Legeard, Practical Model-Based Testing: A Tools Approach, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[40] Monkey.
URL https://developer.android.com/studio/test/monkey.html

[41] A. O. S. Project, UIAutomatorViewer.
URL http://developer.android.com/training/testing/ui-testing/index.html

[42] A. R. Espada, M.-M. Gallardo, D. Adalid, Dragonfly : Encapsulating android for instrumentation, in: Proceedings of the XIII Jornadas de Programación y Lenguajes (PROLE 2013), 2013.

[43] UML 2.4.1 superstructure specification, Tech. rep., Object Management Group (OMG) (aug 2011).
URL http://www.omg.org/spec/UML/2.4.1/

[44] D. Harel, Statecharts: A visual formalism for complex systems, Science of Computer Programming 8 (3) (1987) 231–274. doi:10.1016/0167-6423(87)90035-9.
URL http://dx.doi.org/10.1016/0167-6423(87)90035-9

[45] G. J. Holzmann, The SPIN Model Checker: Primer and Reference Manual, Addison-Wesley Professional, 2003.

[46] Z. Chaochen, M. R. Hansen, Duration Calculus - A Formal Approach to Real-Time Systems, Monographs in Theoretical Computer Science. An EATCS Series, Springer, 2004. doi:10.1007/978-3-662-06784-0.
URL http://dx.doi.org/10.1007/978-3-662-06784-0

[47] A. O. S. Project, logcat.
URL http://developer.android.com/tools/help/logcat.html

[48] Oracle, Java debug interface.
URL `http://docs.oracle.com/javase/6/docs/jdk/api/jpda/jdi/index.html`

[49] A. R. Espada, M.-M. Gallardo, D. Adalid, A runtime verification framework for android applications, in: Proceedings of XXI Jornadas de Concurrencia y Sistemas Distribuidos (JCSD 2013), 2013.

[50] Android, Ddmlib.
URL `https://mvnrepository.com/artifact/com.android.ddmlib/ddmlib`

[51] H. Barringer, K. Havelund, E. Kurklu, R. Morris, Checking flight rules with tracecontract application of a scala dsl for trace analysis (2011).

[52] L. Pike, A. Goodloe, R. Morisset, S. Niller, Copilot: A Hard Real-Time Runtime Monitor, Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 345–359.

[53] S. Katayama, Systematic search for lambda expressions, in: In Proceedings Sixth Symposium on Trends in Functional Programming (TFP2005, 2005.

[54] Y. Falcone, S. Currea, Weave droid: aspect-oriented programming on android devices: fully embedded or in the cloud, in: 2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, 2012, pp. 350–353. doi:10.1145/2351676.2351744.

[55] A. Bauer, J.-C. Küster, G. Vegliach, Runtime Verification Meets Android Security, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 174–180.

[56] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, A. N. Sheth, Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones, in: Proceedings of the 9th USENIX OSDI, OSDI'10, USENIX Association, Berkeley, CA, USA, 2010, pp. 1–6.
URL `http://dl.acm.org/citation.cfm?id=1924943.1924971`

[57] M. Ongtang, S. McLaughlin, W. Enck, P. McDaniel, Semantically rich application-centric security in android, in: Proceedings of the 2009 ACSAC25, ACSAC '09, IEEE Computer Society, Washington, DC, USA, 2009, pp. 340–349. doi:10.1109/ACSAC.2009.39.
URL `http://dx.doi.org/10.1109/ACSAC.2009.39`

[58] P. Merino, A. Salmerón, Combining SPIN with ns-2 for protocol optimization, in: 17th International SPIN Workshop on Model Checking of Software, SPIN 2010, Vol. 6349, Springer, 2010, pp. 40–57.

[59] K. Banker, MongoDB in Action, Manning Publications Co., Greenwich, CT, USA, 2011.

[60] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. Ta, A. Memon, MobiGUITAR – a tool for automated model-based testing of mobile apps, IEEE Software 99 (PrePrints) (2014) 1. doi:http://doi.ieeecomputersociety.org/10.1109/MS.2014.55.

[61] W. Choi, G. Necula, K. Sen, Guided gui testing of android apps with minimal restart and approximate learning, SIGPLAN Not. 48 (10) (2013) 623–640. doi:10.1145/2544173.2509552.
URL `http://doi.acm.org/10.1145/2544173.2509552`

[62] Y. Jing, G. Ahn, H. Hu, Model-based conformance testing for android, in: Advances in Information and Computer Security - 7th International Workshop on Security, IWSEC 2012, Fukuoka, Japan, November 7-9, 2012. Proceedings, 2012, pp. 1–18. doi:10.1007/978-3-642-34117-5-1.
URL `http://dx.doi.org/10.1007/978-3-642-34117-5-1`

[63] A. Shuba, A. Le, M. Gjoka, J. Varmarken, S. Langhoff, A. Markopoulou, Antmonitor: Network traffic monitoring and real-time prevention of privacy leaks in mobile devices, in: Proceedings of the 2015 Workshop on Wireless of the Students, by the Students, and for the Students, S3 '15, ACM, New York, NY, USA, 2015, pp. 25–27. doi:10.1145/2801694.2801707.
URL `http://doi.acm.org/10.1145/2801694.2801707`

[64] S. Dai, A. Tongaonkar, X. Wang, A. Nucci, D. Song, Networkprofiler: Towards automatic fingerprinting of android apps, in: INFOCOM, 2013 Proceedings IEEE, 2013, pp. 809–817. doi:10.1109/INFCOM.2013.6566868.

[65] Monkeyrunner.
URL `https://developer.android.com/studio/test/monkeyrunner/index.html`

[66] Robotium.
URL `https://code.google.com/p/robotium/`

[67] J. Jeon, J. S. Foster, Troyd: Integration testing for android (2012).

[68] X. Wei, L. Gomez, I. Neamtiu, M. Faloutsos, Profiledroid: Multi-layer profiling of android applications, in: Proceedings of the 18th Annual International Conference on Mobile Computing and Networking, Mobicom '12, ACM, New York, NY, USA, 2012, pp. 137–148. doi:10.1145/2348543.2348563.
URL `http://doi.acm.org/10.1145/2348543.2348563`

[69] T. Azim, I. Neamtiu, Targeted and depth-first exploration for systematic testing of android apps, in: Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA '13, ACM, New York, NY, USA, 2013, pp. 641–660. doi:10.1145/2509136.2509549.
URL `http://doi.acm.org/10.1145/2509136.2509549`

[70] A. Pathak, A. Jindal, Y. C. Hu, S. P. Midkiff, What is keeping my phone awake?: Characterizing and detecting no-sleep energy bugs in smartphone apps, in: Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services, MobiSys '12, ACM, New York, NY, USA, 2012, pp. 267–280. doi:10.1145/2307636.2307661.
URL http://doi.acm.org/10.1145/2307636.2307661

[71] Y. Liu, C. Xu, S.-C. Cheung, Characterizing and detecting performance bugs for smartphone applications, in: Proceedings of the 36th International Conference on Software Engineering, ICSE 2014, ACM, New York, NY, USA, 2014, pp. 1013–1024. doi:10.1145/2568225.2568229.
URL http://doi.acm.org/10.1145/2568225.2568229

[72] Microsoft, Eloupe project, http://research.microsoft.com/en-us/projects/eloupe/.

[73] R. Chandra, O. Fatemieh, P. Moinzadeh, C. A. Thekkath, Y. Xie, End-to-End energy management of mobile devices, Tech. Rep. MSR-TR-2013-69, Microsoft (July 2013).
URL http://research.microsoft.com/apps/pubs/default.aspx?id=198163

[74] M. Xia, W. He, X. Liu, J. Liu, Why application errors drain battery easily?: A study of memory leaks in smartphone apps, in: Proceedings of the Workshop on Power-Aware Computing and Systems, HotPower '13, ACM, New York, NY, USA, 2013, pp. 2:1–2:5. doi:10.1145/2525526.2525846.
URL http://doi.acm.org/10.1145/2525526.2525846

[75] G. Metri, W. Shi, M. Brockmeyer, Energy-efficiency comparison of mobile platforms and applications: A quantitative approach, in: Proceedings of the 16th International Workshop on Mobile Computing Systems and Applications, HotMobile '15, ACM, New York, NY, USA, 2015, pp. 39–44. doi:10.1145/2699343.2699358.
URL http://doi.acm.org/10.1145/2699343.2699358

[76] C. Wang, F. Yan, Y. Guo, X. Chen, Power estimation for mobile applications with profile-driven battery traces, in: 2013 IEEE International Symposium on Low Power Electronics and Design (ISLPED), 2013, pp. 120–125. doi:10.1109/ISLPED.2013.6629277.

[77] J. Zhang, A. Musa, W. Le, A comparison of energy bugs for smartphone platforms, in: 2013 1st International Workshop on the Engineering of Mobile-Enabled Systems (MOBS), 2013, pp. 25–30. doi:10.1109/MOBS.2013.6614219.

[78] C. Thompson, D. Schmidt, H. Turner, J. White, Analyzing mobile application software power consumption via model-driven engineering, in: Proceedings of the 1st International Conference on Pervasive and Embedded Computing and Communication Systems, 2011, pp. 101–113.

[79] P. H. Schmitt, F. Werner, Model checking for energy efficient scheduling in wireless sensor networks, in: Tech. report, 2007.

Apéndice A

Apéndice

A.1. Código adicional

Este apéndice contiene fragmentos de código adicionales que completan la documentación de la presente tesis.

A.1.1. Modelo XML

El código A.1 contiene la versión de XML completa del modelo descrito en la Figura 11.7 que fue parcialmente mostrado en el código 11.1.

El modelo XML que se describe en este apéndice incluye la declaración de los dispositivos son utilizados para la ejecución de los casos de prueba, ademas de las aplicaciones, las vistas y sus máquinas de estados.

```
1  <StaticConfiguration>
2    <Applications>
3      <Application name="Facebook" package="com.facebook.android"
           codeName="Facebook" ViewPlace="2" externalAccesability="true">
4        <Views>
5          <View name="HomeView" description="Home View"
               externalAccesability="true" controlsFile="Home.xml">
6            <StateMachines>
7              <StateMachine name="HomeUpdate" description="Home Update"
                   externalAccesability="true">
8                <States>
9                  <State name="S0" description="Principal with news"/>
10                 <State name="S1" description="Pricipal updated"/>
11               </States>
12               <Transitions>
13                 <Transition ID="1" event="Swipe" prev="" next="S0"
                       type="SimpleTransition"/>
14                 <Transition ID="2" event="Comment" prev="S0" next="S0"
                       through="CommentView" type="ViewTransition"/>
15                 <Transition ID="3" event="Swipe" prev="S0" next="S1"
                       type="SimpleTransition"/>
16                 <Transition ID="4" event="ClickYouTubeLink" prev="S0"
                       next="S0" through="ViewingMovieStateMachine"
                       type="StateMachineTransition"/>
17                 <Transition ID="5" event="Swipe" prev="S1" next="S1"
                       type="SimpleTransition"/>
```

```
18                <Transition ID="6" event="Comment" prev="S1" next="S0"
                       through="CommentView" type="ViewTransition"/>
19                <Transition ID="7" event="Swipe" prev="S1" next=""
                       type="SimpleTransition"/>
20              </Transitions>
21            </StateMachine>
22          </StateMachines>
23        </View>
24        <View name="CommentView" description="Comment View"
               externalAccesability="false" controlsFile="Comment.xml">
25          <StateMachines>
26            <StateMachine name="comment" description="Maquina de estados
                   para comentar" externalAccesability="false">
27              <States>
28                <State name="S0" description="Blank Comment View"/>
29                <State name="S1" description="Comented View"/>
30                <State name="S2" description="URL YouTube Selected"/>
31              </States>
32              <Transitions>
33                <Transition ID="1" event="ClickLike" prev="" next="S0"
                       type="SimpleTransition"/>
34                <Transition ID="2" event="SetCommentText" prev="S0"
                       next="S1" type="SimpleTransition"/>
35                <Transition ID="3" event="ClickPublish" prev="S1" next="S2"
                       type="SimpleTransition"/>
36                <Transition ID="4" event="SetYouTubeURL" prev="S0" next="S1"
                       type="SimpleTransition"/>
37                <Transition ID="6" event="Back" prev="S2" next=""
                       type="BackTransition"/>
38              </Transitions>
39            </StateMachine>
40          </StateMachines>
41        </View>
42      </Views>
43    </Application>
44    <Application name="YouTube" package="com.google.android.youtube"
           codeName="YouTube" ViewPlace="3" externalAccesability="true">
45      <Views>
46        <View name="MovieView" description="Movie View"
               externalAccesability="true" controlsFile="YouTubePlayer.xml">
47          <StateMachines>
48            <StateMachine name="ViewingMovieStateMachine"
                   description="ViewingMovie" externalAccesability="true">
49              <States>
50                <State name="Y0" description="Principal with news"/>
51              </States>
52              <Transitions>
53                <Transition ID="1" event="PlayPause" prev="" next="Y0"
                       type="SimpleTransition"/>
54                <Transition ID="2" event="PlayPause" prev="Y0" next="Y0"
                       type="SimpleTransition"/>
55                <Transition ID="3" event="Back" prev="Y0" next=""
                       type="BackTransition"/>
56              </Transitions>
57            </StateMachine>
58          </StateMachines>
59        </View>
60      </Views>
61    </Application>
62  </Applications>
63  <Devices>
64    <Device Port="8080" SerialNumber="219dcac4">
65      <AssignedApplication Code="Facebook" ViewPlace="3">
66        <AssignedView name="HomeView"/>
67      </AssignedApplication>
68    </Device>
69    <Device Port="8081" SerialNumber="219dcac41">
70      <AssignedApplication Code="Facebook" ViewPlace="3">
```

```
71          <AssignedView name="HomeView"/>
72        </AssignedApplication>
73        <AssignedApplication Code="YouTube" ViewPlace="3">
74          <AssignedView name="MovieView">
75            <AssignedStateMachine name="ViewingMovieStateMachine"/>
76          </AssignedView>
77        </AssignedApplication>
78      </Device>
79    </Devices>
80    <DynamicConfigurationFile Path="FacebookDynamicConfiguration.xml"/>
81  </StaticConfiguration>
```

Código A.1: Configuración completa de las máquinas de estados

A.1.2. Especificación PROMELA para la generación de casos de prueba

El código A.2 muestra una gran parte del código de PROMELA generado automáticamente a partir del modelo XML descrito en el código A.1.

Parte de este código se ha descrito en el código 11.2. En este apéndice se muestra la definición completa de los estados globales y sus tipos de datos correspondientes.

```
1  // Constant values ////////////////////////////////////////////////////
2
3  // States (State_<Application>_<View_>_<StateMachine>_<State>)
4  mtype = {
5  // App: Facebook, View: HomeView, StateMachine: HomeUpdate
6      State_Facebook_HomeView_HomeUpdate_init,
7      State_Facebook_HomeView_HomeUpdate_S0,
8      State_Facebook_HomeView_HomeUpdate_S1,
9      State_Facebook_HomeView_HomeUpdate_end,
10 // App: Facebook, View: CommentView, StateMachine: comment
11     State_Facebook_CommentView_comment_init,
12     State_Facebook_CommentView_comment_S0,
13     State_Facebook_CommentView_comment_S1,
14     State_Facebook_CommentView_comment_S2,
15     State_Facebook_CommentView_comment_end,
16 // App: YouTube, View: MovieView, StateMachine: ViewingMovieStateMachine
17     State_YouTube_MovieView_ViewingMovieStateMachine_init,
18     State_YouTube_MovieView_ViewingMovieStateMachine_Y0,
19     State_YouTube_MovieView_ViewingMovieStateMachine_end,
20     state_init
21 }
22
23 c_decl {
24 #include "constants.h"
25 #include "ModelcheckingFacebookTestGenerationConfiguration.xml.ccode.h"
26 }
27
28 // Global state ///////////////////////////////////////////////////////
29
30 typedef BackstackType {
31     mtype states[MAX_BACKSTACK_SIZE];
32     short index = -1
33 };
34
35 typedef TransitionType {
36     byte view;
37     byte id;
38 };
39
```

```
40 typedef HistoryType {
41     TransitionType transitions[MAX_TRANSITION_TRACE_SIZE];
42     short index = -1
43 };
44
45 typedef DeviceType {
46     BackstackType backstack;
47     HistoryType history;
48     bool finished = false
49 };
50
51 DeviceType devices[DEVICES];
52
53 // Inlines and proctypes ////////////////////////////////////////////
54
55 // Transition helpers
56
57 #define currentBackstack    devices[device].backstack
58 #define currentState        currentBackstack.states[currentBackstack.index]
59 #define currentHistory      devices[device].history
60 #define currentTransition   currentHistory.transitions[currentHistory.index]
61
62 inline transition(device, view_, id_) {
63     atomic {
64         if
65         :: (currentHistory.index == MAX_TRANSITION_TRACE_SIZE - 1) ->
66             // Block device, but don't mark it as finished
67 end_transition:     false -> skip
68         :: else ->
69             currentHistory.index++;
70             currentTransition.view = view_;
71             currentTransition.id = id_
72         fi
73     }
74 }
75
76 inline pushToBackstack(device, state) {
77     atomic {
78         if
79         :: currentBackstack.index == MAX_BACKSTACK_SIZE - 1 ->
80             // Block device, but don't mark it as finished
81 end_pushToBackstack:    false -> skip
82         :: else ->
83             currentBackstack.index++;
84             currentState = state
85         fi
86     }
87 }
88
89 inline popFromBackstack(device) {
90     currentBackstack.index--;
91 }
92
93 // Applications (app_<Application>)
94
95 inline app_Facebook(device) {
96     if
97     :: true -> view_Facebook_HomeView(device)
98     :: true -> view_Facebook_CommentView(device)
99     fi
100 }
101
102 inline app_YouTube(device) {
103     if
104     :: true -> view_YouTube_MovieView(device)
105     fi
106 }
107
```

```
108
109 // Views (view_<Application>_<View>)
110
111 inline view_Facebook_HomeView(device) {
112     if
113     :: true -> statemachine_Facebook_HomeView_HomeUpdate(device)
114     fi
115 }
116 inline view_Facebook_CommentView(device) {
117     if
118     :: true -> statemachine_Facebook_CommentView_comment(device)
119     fi
120 }
121
122 inline view_YouTube_MovieView(device) {
123     if
124     :: true ->
            statemachine_YouTube_MovieView_ViewingMovieStateMachine(device)
125     fi
126 }
127
128
129 // State machines (statemachine_<Application>_<View>_<StateMachine>)
130
131 inline statemachine_Facebook_HomeView_HomeUpdate(device) {
132     pushToBackstack(device, State_Facebook_HomeView_HomeUpdate_init);
133
134     do
135     :: currentState == State_Facebook_HomeView_HomeUpdate_init ->
136         // Event: Swipe
137         transition(device, VIEW_HomeView, 1);
138         currentState = State_Facebook_HomeView_HomeUpdate_S0
139     :: currentState == State_Facebook_HomeView_HomeUpdate_S0 ->
140         // Event: Comment
141         transition(device, VIEW_HomeView, 2);
142         view_Facebook_CommentView(device);
143         currentState = State_Facebook_HomeView_HomeUpdate_S0
144     :: currentState == State_Facebook_HomeView_HomeUpdate_S0 ->
145         // Event: Swipe
146         transition(device, VIEW_HomeView, 3);
147         currentState = State_Facebook_HomeView_HomeUpdate_S1
148     :: currentState == State_Facebook_HomeView_HomeUpdate_S0 ->
149         // Event: clicYouTubeLink
150         transition(device, VIEW_HomeView, 4);
151         statemachine_YouTube_MovieView_ViewingMovieStateMachine(device);
152         currentState = State_Facebook_HomeView_HomeUpdate_S0
153     :: currentState == State_Facebook_HomeView_HomeUpdate_S1 ->
154         // Event: Swipe
155         transition(device, VIEW_HomeView, 5);
156         currentState = State_Facebook_HomeView_HomeUpdate_S1
157     :: currentState == State_Facebook_HomeView_HomeUpdate_S1 ->
158         // Event: Comment
159         transition(device, VIEW_HomeView, 6);
160         view_Facebook_CommentView(device);
161         currentState = State_Facebook_HomeView_HomeUpdate_S0
162     :: currentState == State_Facebook_HomeView_HomeUpdate_S1 ->
163         // Event: Swipe
164         transition(device, VIEW_HomeView, 7);
165         currentState = State_Facebook_HomeView_HomeUpdate_end
166     :: currentState == State_Facebook_HomeView_HomeUpdate_end ->
167         break
168     od;
169
170     popFromBackstack(device)
171 }
172
173 inline statemachine_Facebook_CommentView_comment(device) {
174     pushToBackstack(device, State_Facebook_CommentView_comment_init);
```

```
175
176     do
177     :: currentState == State_Facebook_CommentView_comment_init ->
178         // Event: clicLike
179         transition(device, VIEW_CommentView, 1);
180         currentState = State_Facebook_CommentView_comment_S0
181     :: currentState == State_Facebook_CommentView_comment_S0 ->
182         // Event: setCommentText
183         transition(device, VIEW_CommentView, 2);
184         currentState = State_Facebook_CommentView_comment_S1
185     :: currentState == State_Facebook_CommentView_comment_S1 ->
186         // Event: clicPublish
187         transition(device, VIEW_CommentView, 3);
188         currentState = State_Facebook_CommentView_comment_S2
189     :: currentState == State_Facebook_CommentView_comment_S0 ->
190         // Event: setYouTubeURL
191         transition(device, VIEW_CommentView, 4);
192         currentState = State_Facebook_CommentView_comment_S1
193     :: currentState == State_Facebook_CommentView_comment_S1 ->
194         // Event: clicPublish
195         transition(device, VIEW_CommentView, 5);
196         currentState = State_Facebook_CommentView_comment_S2
197     :: currentState == State_Facebook_CommentView_comment_S2 ->
198         // Event: Back
199         transition(device, VIEW_CommentView, 6);
200         currentState = State_Facebook_CommentView_comment_end
201     :: currentState == State_Facebook_CommentView_comment_end ->
202         break
203     od;
204
205     popFromBackstack(device)
206 }
207
208 inline statemachine_YouTube_MovieView_ViewingMovieStateMachine(device) {
209     pushToBackstack(device,
            State_YouTube_MovieView_ViewingMovieStateMachine_init);
210
211     do
212     :: currentState == State_YouTube_MovieView_ViewingMovieStateMachine_init
            ->
213         // Event: playpause
214         transition(device, VIEW_MovieView, 1);
215         currentState = State_YouTube_MovieView_ViewingMovieStateMachine_Y0
216     :: currentState == State_YouTube_MovieView_ViewingMovieStateMachine_Y0 ->
217         // Event: playpause
218         transition(device, VIEW_MovieView, 2);
219         currentState = State_YouTube_MovieView_ViewingMovieStateMachine_Y0
220     :: currentState == State_YouTube_MovieView_ViewingMovieStateMachine_Y0 ->
221         // Event: Back
222         transition(device, VIEW_MovieView, 3);
223         currentState = State_YouTube_MovieView_ViewingMovieStateMachine_end
224     :: currentState == State_YouTube_MovieView_ViewingMovieStateMachine_end
            ->
225         break
226     od;
227
228     popFromBackstack(device)
229 }
230
231
232 // Devices
233
234 // Device 219dcac4
235
236 inline app_219dcac4_Facebook(device) {
237     if
238     :: true -> view_219dcac4_Facebook_HomeView(device)
239     fi
```

```
240 }
241
242 inline view_219dcac4_Facebook_HomeView(device) {
243     if
244     :: true -> statemachine_Facebook_HomeView_HomeUpdate(device)
245     fi
246 }
247
248 proctype device_219dcac4(int device) {
249     if
250     :: true -> app_219dcac4_Facebook(device)
251     fi;
252
253     devices[device].finished = true
254 }
255
256 // Device 219dcac41
257
258 inline app_219dcac41_Facebook(device) {
259     if
260     :: true -> view_219dcac41_Facebook_HomeView(device)
261     fi
262 }
263
264 inline view_219dcac41_Facebook_HomeView(device) {
265     if
266     :: true -> statemachine_Facebook_HomeView_HomeUpdate(device)
267     fi
268 }
269
270 inline app_219dcac41_YouTube(device) {
271     if
272     :: true -> view_219dcac41_YouTube_MovieView(device)
273     fi
274 }
275
276 inline view_219dcac41_YouTube_MovieView(device) {
277     if
278     :: true ->
            statemachine_YouTube_MovieView_ViewingMovieStateMachine(device)
279     fi
280 }
281
282 proctype device_219dcac41(int device) {
283     if
284     :: true -> app_219dcac41_Facebook(device)
285     :: true -> app_219dcac41_YouTube(device)
286     fi;
287
288     devices[device].finished = true
289 }
290
291
292 proctype traceCloser() provided (devices[0].finished && devices[1].finished)
     {
293 end_tc: c_code { printTransitionTrace(); }
294 }
295
296 init {
297     c_code { printTransitionTraceStart(); };
298
299     c_code { setDeviceSerialNumber(0, DEVICE_STRING_219dcac4); };
300     devices[0].history.index = -1;
301     c_code { setDeviceSerialNumber(1, DEVICE_STRING_219dcac41); };
302     devices[1].history.index = -1;
303
304     run device_219dcac4(0);
305     run device_219dcac41(1);
```

```
306
307     run traceCloser()
308 }
```

Código A.2: Especificación PROMELA para Facebook y YouTube