Formalización del protocolo de comunicación entre aplicaciones para dispositivos móviles Java

José Armando Forte

<bibaldo@gmail.com>

Legajo: F-1804/0

Universidad Nacional de Rosario.

Facultad de Ciencias Exactas, Ingeniería y Agrimensura.

Tesina de Grado - Licenciatura en Ciencias de la Computación

Director : Carlos Luna

<cluna@fing.edu.uy>

Co-director : Jesús Mauricio Martín Chimento

<checholcc@gmail.com>

Septiembre 2013

Resumen

En los últimos años se ha incrementado de manera significativa el uso de dispositivos móviles, tales como teléfonos celulares y smartphones. Este rápido crecimiento ha provocado que los usuarios integren el uso de aplicaciones en su rutina diaria. A su vez, este fenómeno ha generado grandes cambios a la hora de considerar la seguridad e integridad de la información que éstos manejan.

En la Plataforma Java Micro Edition, el Perfil para Dispositivos de Información Móviles (MIDP) provee el ambiente de ejecución estándar para teléfonos móviles y asistentes de datos personales. La tercera y más reciente versión del perfil introduce, en particular, una nueva dimensión en el modelo de seguridad de MIDP: la seguridad a nivel de aplicación. En esta nueva versión las MIDlets que quieren compartir datos entre ellas utilizan un protocolo de comunicación denominado *InterMIDlet Communication* (IMC). El presente trabajo analiza formalmente el nuevo modelo de seguridad y formaliza este nuevo protocolo. Concretamente, el trabajo extiende una especificación formal desarrollada en el Cálculo de Construcciones Inductivas, usando el asistente de pruebas Coq. Se formalizan los eventos relacionados con el protocolo mencionado anteriormente, se demuestra que la extensión es conservativa y se analizan propiedades relevantes de seguridad relativas a la extensión. Finalmente, se refina la especificación junto con la extensión propuesta y se certifican en Coq dos algoritmos como implementaciones de los nuevos eventos.

Agradecimientos

A mi familia, por enseñarme los valores que hoy día me convirtieron en la persona que soy. Por siempre empujarme y motivarme a alcanzar lo que uno se proponga.

A mis amigos, los que conozco de toda la vida, compañeros de la facultad y amigos que fui conociendo en este camino que sigo transitando día a día.

A mi director Carlos por su ayuda, motivación y guía en este trabajo. Y a mi codirector Mauricio, por su colaboración y participación, y por siempre estar dispuesto a dar una mano o un consejo cuando lo necesitaba.

Y especialmente a Rosi, porque sin su apoyo, confianza y compañerismo esto no hubiese sido posible. Gracias por estar siempre al lado mio en todos los momentos y hacerme feliz, sos una compañera incondicional.

Índice general

Capítulo 1

Motivación y Objetivos

1.1. Motivación

En los últimos años el uso de dispositivos móviles, tales como teléfonos celulares, smartphones y tablets, se ha popularizado a escala mundial. Este tipo de dispositivos tienen fines y características que son en esencia diferentes de los que tienen, por ejemplo, las computadoras portátiles o de escritorio.

En trabajos anteriores [1], [2] y [3] se ha propuesto una especificación formal del modelo de seguridad definido por el perfil *Mobile Information Device Profile (MIDP)* [4]. Esta especificación que ha sido formalizada en Coq [5], es una base formal para la verificación del modelo de seguridad y la comprensión de su complejidad.

Esta tesina propone formalizar el protocolo de comunicación entre aplicaciones y el análisis de propiedades de seguridad relevantes sobre el mismo. Con la introducción de dos nuevos eventos relacionados a este protocolo, se definen dos algoritmos y luego se los certifica respecto de la especificación presentada.

1.2. Objetivos

Como objetivo general, se propone desarrollar una especificación formal del protocolo de comunicación entre aplicaciones, *InterMIDlet Communication* (IMC), no presente en versiones anteriores.

En este trabajo se extiende la especificación formal actual de MIDP para incorporar los

nuevos conceptos propios de esta investigación y se analizan ciertas propiedades de seguridad relacionadas con el modelo.

Asimismo, en este proyecto se refina la especificación construyendo un modelo concreto en base al modelo abstracto presentado y se certifican dos algoritmos escritos en Coq como implementaciones de los eventos mencionados con anterioridad.

La extensión de la formalización está implementada usando el asistente de pruebas Coq. Se utilizó para compilar los archivos la versión 8.4pl2 del mismo. La versión completa del código fuente está disponible en [6].

1.3. Organización de la tesina

En el capítulo 2, para que este trabajo sea autocontenido, damos una introducción de los conceptos y aspectos fundamentales de las diferentes versiones de MIDP, junto con una descripción de algunas de las características innovadoras presentes en la nueva versión.

En el capítulo 3 describimos la especificación formal propuesta por Zanella, Betarte, Luna y Mazeikis [7] del modelo de seguridad de MIDP. En dicha especificación se plantean y verifican algunas propiedades deseables del modelo de seguridad.

En el capítulo 4 extendemos la especificación formal haciendo hincapié en el nuevo protocolo de comunicación entre MIDlets (IMC). Presentamos dos nuevos eventos relacionados con este nuevo protocolo, y probamos que los mismos son correctos según su especificación. Extendemos entre otras cosas el estado del dispositivo, junto con las propiedades de validez del mismo. Probamos que luego de la ejecución de cualquier evento el sistema debe estar asociado a un estado válido, por lo que en la especificación introducimos el concepto de estado válido y el concepto de ejecución válida.

Luego, en el capítulo 5 presentamos una representación concreta para el protocolo IMC junto con dos algoritmos que modelan los eventos previamente mencionados. Asimismo demostramos su correctitud.

Finalmente, en el capítulo 6 realizamos un análisis comparativo con el Modelo de Seguridad de Android, exhibimos las conclusiones y proponemos los futuros trabajos que se pueden dar a partir de éste.

Capítulo 2

Introducción

En la actualidad, la gran mayoría de las plataformas tecnológicas para los dispositivos móviles incorporan la tecnología Java para sistemas embebidos (embedded systems) denominada *Java Micro Edition* (Java ME) [8]. La característica fundamental de esta edición, al igual que en toda tecnología Java, es el uso de una máquina virtual (JVM), especialmente diseñada y adaptada para aprovechar al máximo los escasos recursos con los que cuentan los dispositivos portátiles. La tecnología Java ME proporciona mecanismos integrales para garantizar propiedades de seguridad de los dispositivos; en particular, para dispositivos móviles define un modelo de seguridad que restringe el acceso de las aplicaciones a funciones consideradas potencialmente peligrosas.

2.1. Arquitectura de Java Micro Edition

La plataforma Java ME proporciona un entorno robusto y flexible para aplicaciones que se ejecutan en dispositivos embebidos móviles y otros como teléfonos móviles, PDAs, TDT, reproductores Blu-ray, dispositivos multimedia digitales, impresoras y demás. Ésta provee de una colección certificada de APIs de desarrollo de software para dispositivos con recursos restringidos.

Java ME fue desarrollado mediante el *Java Community Process* [9] bajo la especificación JSR 68.

La tecnología Java ME se creó originalmente para enfrentar las limitaciones de memoria, visualización y potencia asociadas a la creación de aplicaciones para pequeños dispositivos.

Java ME [1] plantea una arquitectura basada en dos capas por encima de la máquina virtual:

[1] http://codewurd.wordpress.com/category/j2me-java-2-micro-edition/

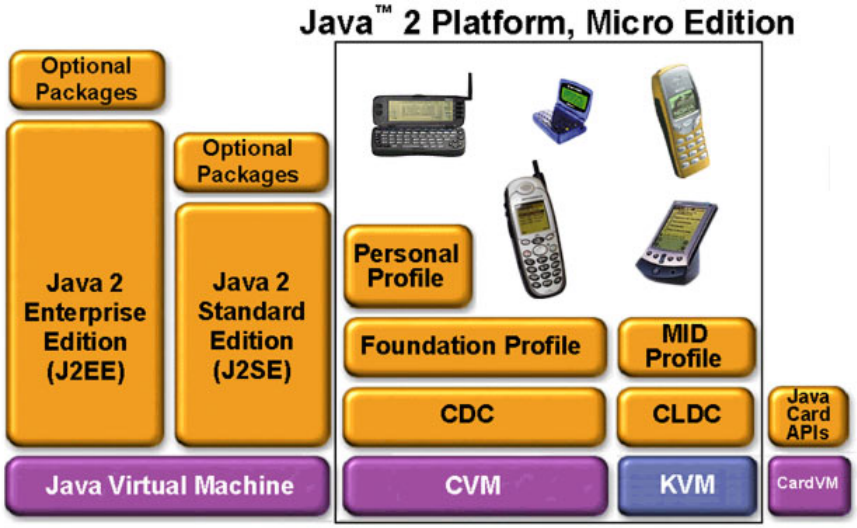

Figura 2.1: La plataforma Java

la capa *configuración* y la capa *perfil*.

La configuración *Connected Limited Device Configuration* (CLDC)[10] es un conjunto mínimo de bibliotecas de clases para ser utilizadas en dispositivos con procesadores poco potentes, memoria de trabajo limitada y capacidad para establecer comunicaciones de bajo ancho de banda. A su vez, si hablamos de dispositivos con mayor capacidad entonces hacemos referencia a la configuración *Connected Device Configuration* [11].[2]

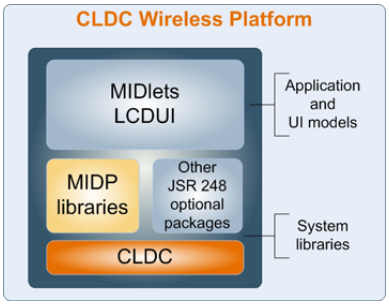

Figura 2.2: Connected Limited Device Configuration

A la configuración CLDC se la complementa con el perfil *Mobile Information Device Profile* (MIDP) para obtener un entorno de ejecución adaptado a dispositivos portátiles como teléfonos

[2]http://www.oracle.com/technetwork/java/javame/java-me-overview-402920.html

móviles y asistentes digitales personales. Un ejemplo ampliamente adoptado consiste en combinar la CLDC con el Mobile Information Device Profile (MIDP) para proporcionar un completo entorno de aplicaciones Java para teléfonos móviles y otros dispositivos con capacidades similares. Estas aplicaciones son conocidas como *MIDlets*.

2.2. Mobile Information Device Profile (MIDP)

En la Plataforma *Java ME*, el *Perfil para Dispositivos de Información Móviles* (MIDP) define, entre otras cosas, un modelo de seguridad para aplicaciones, el ciclo de vida de las mismas, los mecanismos de comunicación de red y una biblioteca para el acceso a funciones propias de la clase de dispositivos para la que fue concebido.

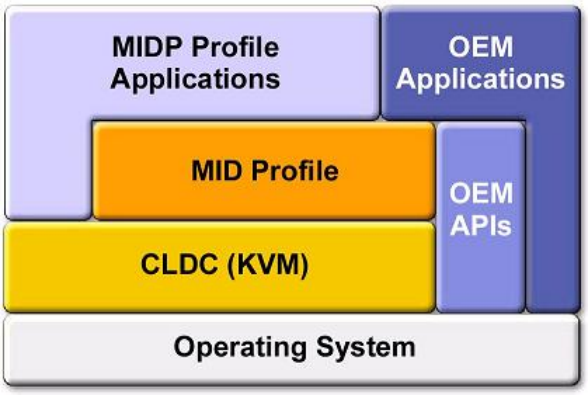

Figura 2.3: Arquitectura MIDP

CLDC y MIDP proporcionan la funcionalidad de aplicación básica que requieren las aplicaciones móviles, en la forma de un entorno de ejecución Java estándar y un rico conjunto de APIs Java. Los desarrolladores que utilizan MIDP pueden escribir aplicaciones una vez y luego implementarlas rápidamente sobre una amplia variedad de dispositivos móviles.

MIDP ha sido ampliamente adoptado como la plataforma de elección para las aplicaciones móviles. Existen varias versiones de MIDP, las cuales describiremos a continuación.

2.2.1. MIDP 1.0

Esta es la versión original [12], que proporcionaba la funcionalidad básica requerida por las aplicaciones móviles, incluyendo la interfaz de usuario básica y seguridad de la red.

En esta versión el modelo de seguridad se basaba en la seguridad a nivel de plataforma y definía que toda aplicación que no estaba preinstalada en el dispositivo (e.g. por el fabricante), se ejecutaba en un ambiente controlado denominado *sandbox*. Una aplicación no confiable solo podía tener acceso a las funciones que son consideradas potencialmente peligrosas con autorización explícita del usuario.

Esta versión tuvo muchas limitaciones que luego fueron subsanadas con una nueva versión.

2.2.2. MIDP 2.0

MIDP 2.0 [13, 14] es una extensión y mejora de la versión anterior. La mayoría de los teléfonos que utilizamos hoy en día utilizan este perfil. A su vez también soportan aplicaciones y juegos de la versión anterior.

Esta versión mejora la experiencia final del usuario, mejora la portabilidad de las aplicaciones y proporciona una mayor extensibilidad. Es compatible con MIDP 1.0, y en general, la API ofrece clases más orientadas hacia el desarrollo de juegos.

Algunas de las mejoras con respecto a la versión anterior son las siguientes:

- Seguridad
- Solicitud de firma
- Soporta la reproducción de audio
- Soporta modo pantalla completa en LCDUI
- Incluye una API específica para la programación de juegos
- Soporta HTTPS para conexiones seguras
- Se puede trabajar mejor con imágenes RGB
- Posee nuevos controles y características para la interfaz de usuario de alto nivel
- Posee una Arquitectura Push que permite lanzar las MIDlets en respuesta a conexiones de red entrantes.
- La Provisión a-través-del-aire [15] que permite descargar e instalar contenido a través de una red inalámbrica ahora es un requisito para la especificación de MIDP 2.0.

- Los Almacenes de Registros (record stores) pueden ser compartidos entre MIDlets.

En otras palabras, MIDP 1.0 estableció un entorno de Java estándar para pequeños dispositivos con conectividad de red inalámbrica. MIDP 2.0 se expande considerablemente en la versión original, otorgando mayor soporte para las interfaces de usuario avanzadas, multimedia, seguridad de red HTTP, y muchas otras funciones útiles.

2.2.3. MIDP 3.0

Esta versión está basada en la especificación de MIDP 2.0 y proporciona compatibilidad de forma tal que las MIDlets desarrolladas para versiones anteriores puedan ejecutarse en entornos de MIDP 3.0 [16]. Es por esto que la mayoría de las características presentes en versiones anteriores se mantienen de una versión a otra.

Está diseñada para operar en el nivel superior de la Connected Limited Device Configuration (CLDC), pero también opera por encima de la CDC. Se prevé que la mayoría de las implementaciones de MIDP 3.0 estarán basadas en la CLDC.

MIDP 3.0 introduce una serie de nuevas características que se basan en el éxito anterior de MIDP. Estas nuevas características permiten una amplia gama de aplicaciones y servicios, y además mejoran la estabilidad y la previsibilidad de las implementaciones de MIDP. De estas nuevas características nos concentraremos en las siguientes:

- Concurrencia
- InterMIDlet Communication (IMC)
- RMS Data Provisioning & RMS Interchange File Format
- Componentes compartidos (LIBlets)

2.3. Características de MIDP 3.0

De aquí en adelante al hacer mención a MIDP haremos referencia a la última versión de MIDP, la 3.0.

En esta sección describimos las principales características de MIDP en relación con las versiones anteriores.

2.3.1. Concurrencia

Las versiones anteriores de MIDP permitían la concurrencia pero no eran muy explícitas a la hora de detallar el comportamiento esperado de las MIDlets que corren en forma concurrente. Además, tampoco proveían los mecanismos necesarios para permitir a las MIDlets detectar y manejar cambios de estado.

Los dispositivos que implementen la última especificación deben soportar la ejecución concurrente de una o más MIDlets. La especificación debe definir los comportamientos esperados sobre los problemas básicos de la concurrencia de tal forma que las implementaciones se comporten de la manera más consistente sin imponer decisiones de implementación.

La primer regla a tener en cuenta es pensar que la MIDlet está ejecutándose en forma independiente. Para lograr esto, los conjuntos de datos entre las MIDlets deben ser aislados, la contención de recursos debe ser lo más transparente posible, los errores no fatales deben manejarse de forma silenciosa y la planificación debe tener en cuenta la experiencia del usuario.

Existen diferentes casos de uso de concurrencia que pueden ser resueltos implementando una solución basada en detener la ejecución de la aplicación mientras otra aplicación está ejecutándose *(Ejemplo: un juego)*. Sin embargo hay otros casos en el que se desea que la aplicación siga ejecutándose mientras está en segundo plano *(Ejemplo: reproductor de música)*.

Por lo tanto, MIDP debe proveer los mecanismos para que cada aplicación pueda elegir su propio comportamiento cuando esté ejecutándose en segundo plano.

Aislamiento de los datos

Se debe asegurar que los datos accedidos por una aplicación deben estar aislados de los datos visibles por otras aplicaciones en otro ámbito (contexto).

Las clases y los datos estáticos no deben ser compartidos entre MIDlets, incluso aunque pertenezcan a la misma MIDlet Suite.

Sin embargo, los datos que hacen a la definición de una MIDlet Suite tales como los permisos, accesos RMS (Record Management System) y recursos, permanecen por Suite.

Manejo de errores

Cuando se ejecuta una sola MIDlet, se manejan los errores reinicializándola o apagando la máquina virtual. Sin embargo, cuando varias MIDlets se ejecutan concurrentemente, puede ser

que un error que afecta a una MIDlet no esté afectando a otra MIDlet y por lo tanto no sea necesario apagarla.

Planificación

La granularidad de los detalles y los algoritmos utilizados en la planificación de las aplicaciones va a depender de las plataformas. Debido a las diferencias entre las implementaciones y las plataformas es impráctico determinar una política específica para todos los dispositivos. Sin embargo deben tener características similares.

MIDP debe implementar una política que evite la inanición.

Ejecución de más de una instancia de una MIDlet

El Application Management Software (AMS) no debe lanzar una segunda instancia de una MIDlet. Lo que se hace en el caso de recibir un pedido así, es enviarle un evento (relaunch system event) a la MIDlet.

2.3.2. Componentes Compartidos (LIBlets)

Una LIBlet es un componente de software compartido, el cual una o varias MIDlets pueden utilizar en tiempo de ejecución. La ventaja de que múltiples aplicaciones compartan un mismo componente permite una mayor flexibilidad y escalabilidad a la plataforma.

Las LIBlets permiten que diferentes MIDlets compartan el mismo código sin empaquetarlo en forma redundante. Esto conlleva a reducciones de tamaño en aplicaciones individuales.

De esta forma, reducen de manera significativa el tiempo de descarga de las aplicaciones con dependencias en componentes compartidas. También reducen el costo de desarrollo e incrementan el reuso de código.

Las LIBlets por sí solas son fragmentos de código sin "sentido" (inertes); no tienen un contexto de ejecución de sí mismas pero son percibidas como partes integradas de la MIDlet a la que pertenecen (Figura 2.4). Es por ello que las clases contenidas dentro de ellas no se deben ejecutar fuera del contexto de ejecución que existe para esa MIDlet.

Una LIBlet está empaquetada como un JAD y un JAR, de forma similar que una MIDlet Suite. El archivo JAR contiene:

1. un manifiesto que describe el contenido

2. archivos de clases

3. archivos de recursos (archivos de texto, imágenes, audio, video, etc)

4. RMS Interchange Format

En estos archivos existen unos atributos específicos para las LIBlets.

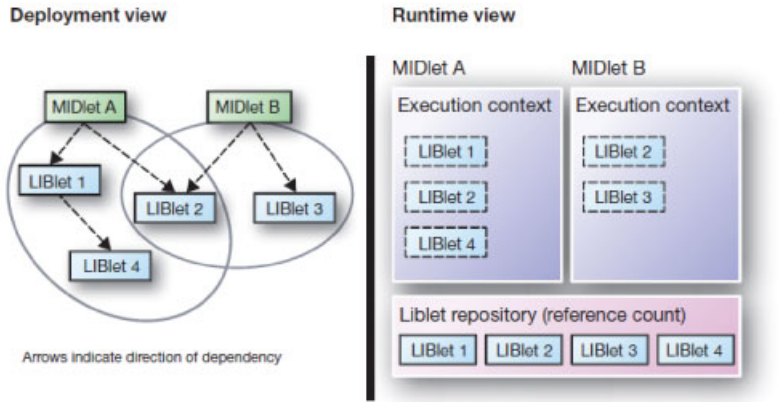

Figura 2.4: MIDP LIBlets

El manejo de dependencias es un problema común asociado con bibliotecas de código compartido. De hecho, muchas implementaciones de bibliotecas compartidas en otros entornos sufren ampliamente de problemas relacionados con las dependencias de software. Por lo tanto, se ha tomado gran cuidado en MIDP para prevenir que esto ocurra. Eso es especialmente importante en dispositivos tales como teléfonos móviles, donde no se puede esperar que los usuarios administren manualmente las configuraciones de la aplicación.

Las LIBlets se pueden distribuir por su cuenta o como parte de un conjunto de MIDlets. De cualquier manera, pueden ser reutilizadas por cualquier MIDlet que la requiera en forma particular, como dijimos anteriormente. Este enfoque ayuda a reducir la huella de memoria, es decir la cantidad de memoria principal que utiliza o referencia un programa mientras se ejecuta.

2.3.3. RMS Data Provisioning & RMS Interchange File Format

Un dispositivo móvil debe ofrecer a los usuarios mecanismos para identificar las MIDlet suites que pueden ser descargadas. Este mecanismo es una base de datos (o *record store*) llamada *RMS*

(Record Management System), que permite a las MIDlets almacenar datos de forma persistente y recuperarlos más tarde.

MIDP 2.0 no tenía un mecanismo donde el Record Store pudiera ser empaquetado junto a los archivos JAR. Usualmente en este tipo de situación, los archivos de datos eran almacenados en el mismo directorio del archivo JAR y luego insertados en el Record Store cuando se usaba por primera vez la aplicación.

MIDP ahora provee de una nueva funcionalidad al RMS. De esta forma los archivos RMS son empaquetados con el formato *RMS Interchange File Format*, y pueden ser creados de forma offline y empaquetados en archivos separados o embebidos en el JAR de la aplicación. Las LIBlets también pueden utilizar estos archivos.

El tipo MIME de este archivo es .rms. El RMS Interchange File Format no es un simple archivo de texto; sino que es un archivo encriptado con sus respectivas políticas de seguridad. Con una misma aplicación se pueden suministrar más de un archivo rms, cada uno con un record store diferente (el nombre es único dentro del contexto de la MIDlet).

Para las MIDlets Suites, estos record stores pueden ser privados o compartidos. En cambio para las LIBlets, deben ser exclusivamente privados.

Record Tags

En MIDP 2.0 no había un mecanismo para categorizar los registros en el RMS. Por lo tanto si se deseaba obtener algún grupo particular de registros se debían implementar algoritmos que cumplan tal fin. Esto llevaba a demorar más y tener un mayor procesamiento.

En cambio ahora MIDP incorpora *Record Tags*, el cual permite etiquetar registros en el RMS a una categoría determinada.

2.3.4. InterMIDlet Communication (IMC)

En la versión de MIDP 2.0 ya se proporcionaba la capacidad de que las MIDlets se comuniquen indirectamente mediante un RMS compartido. Esto se lograba registrando las aplicaciones en ejecución para que escuchasen cambios en el RMS. Sin embargo la comunicación indirecta a través de un repositorio compartido no es tan flexible como la comunicación directa con una aplicación que está corriendo en tiempo real, lo cual IMC sí permite.

Mediante el nuevo protocolo de comunicación *InterMIDlet Communication* (IMC), una aplicación puede acceder a un componente compartido por otra aplicación que está en un ambiente de ejecución diferente. Este protocolo es parecido al protocolo de comunicación *Socket*. Las MIDlets usan este protocolo cliente-servidor para crear una conexión de tipo servidor, donde otras MIDlets pueden conectarse (como clientes). Para restringir el acceso, la MIDlet servidor debe listar el conjunto de MIDlets que están permitidas conectarse. Esta lista es entonces usada durante el proceso de autorización de acceso en el nivel de aplicación.

Una de las ventajas del protocolo IMC de MIDP es que el servidor no tiene necesidad de estar en ejecución antes de que algún cliente haga un pedido. Lo mencionado es interesante porque las aplicaciones móviles tienen una fracción del espacio de pila disponible para ellas en comparación con las aplicaciones de servidor tradicional. Debido a esta falta de recursos de sistema, los servidores IMC no pueden siempre estar en ejecución en el dispositivo móvil.

Si bien el protocolo IMC ofrece numerosas ventajas, no funciona bien cuando el servidor necesita enviar datos a varios clientes al mismo tiempo. La solución a este problema son los llamados *eventos* de MIDP. El *framework* de eventos de MIDP es una arquitectura de publicación y suscripción que se adecua bien cuando se necesita enviar el mismo conjunto de datos a múltiples aplicaciones MIDP (clientes). Éste permite a las MIDlets enviar datos a todos los suscriptores autorizados. Además de esto, el *framework* de eventos permite a las MIDlets suscribirse a eventos específicos del sistema como nivel de batería, estado de la red, y estado del sistema.

Mediante este nuevo protocolo el acceso a algún recurso compartido puede suponer el acceso a información privilegiada. Esto implica que la aplicación que usufructúa el recurso deba contar con la autorización de acceso correspondiente.

Capítulo 3

Especificación Formal de MIDP 3.0

En este capítulo describimos los principales elementos de la especificación formal del modelo de seguridad para MIDP, para su posterior referencia en la extensión propuesta por este trabajo de tesina.

Con ese objetivo presentamos la notación utilizada, las definiciones de aplicación, política de seguridad, estado del dispositivo, los eventos relacionados con la seguridad, y la noción de sesión de ejecución, definida como una secuencia de eventos.

3.1. Notación

La notación presentada a continuación es la misma que la utilizada en [1]. Para las expresiones lógicas son utilizados los operadores estándar $(\wedge, \vee, \neg, \rightarrow, \forall, \exists)$.

Los predicados anónimos son introducidos usando la notación lambda, por ejemplo $(\lambda x.x = 0)$ es un predicado que aplicado a n devuelve *true* si y sólo si n es cero.

Los tipos *records* son introducidos de la siguiente forma:

$$R \stackrel{\text{def}}{=} [\![\ field_0 : A_0, ..., field_n : A_n\ ]\!] \tag{3.1}$$

lo cual genera un tipo inductivo no recursivo con un único constructor, llamado mkR y n funciones de proyección $field_i : R \rightarrow A_i$.

Las aplicaciones de las funciones de proyección están abreviadas utilizando la notación *punto* (ej $(field_i\ r = r.field_i)$).

A su vez se utilizan los tipos inductivos paramétricos $option\,T$ con los constructores $None$: $option\,T$ y $Some$: $T \to option\,T$. Definimos las secuencias finitias con $seq\,T$, la secuencia vacía con [ ] y la secuencia que agrega un elemento s al final de una secuencia ss con $ss \cap s$. La concatenación de secuencias se denota con $\oplus$.

Una relación inductiva I se define a través de reglas de introducción de la forma:

$$\frac{P_1 ... P_n}{I\ x_1 ... x_m}(nombre\_regla) \tag{3.2}$$

donde las variables libres se consideran universalmente cuantificadas e I puede usarse en notación infija (en el caso de relaciones binarias).

3.2. Formalización de los componentes

En esta sección repasamos brevemente las componentes del modelo respecto a la especificación anterior [3].

Como primer punto definimos algunos tipos que utilizaremos en el resto de la especificación. Denotamos al conjunto de los dominios de protección del dispositivo con *Domain*, al conjunto de los identificadores válidos para MIDlet suites con *SuiteID*, a los nombres de vendedores de suites con *Vendor*, al conjunto de los certificados de clave pública con *Certificate* y a la firma del archivo de aplicación con *Signature*.

Un tipo registro *Suite* representa una suite instalada, con sus respectivos campos para su identificador, su dominio de protección asociado y su descriptor.

$$Suite \stackrel{\text{def}}{=} [\![\ id : SuiteID, domain : Domain, descriptor : Descriptor\ ]\!] \tag{3.3}$$

El conjunto de los permisos definidos por cada dominio de protección para cada API o función protegida del dispositivo se denota *Permission*. Los permisos requeridos por una suite para su normal funcionamiento se declaran en el descriptor con el predicado *required*, mientras que aquellos considerados opcionales se representan con el predicado *optional*. El predicado *domainAuthorization* denota la declaración de autorización de acceso por dominio de protección. La autorización de acceso para un conjunto de suites firmadas con un certificado particular se establece con el predicado *signerAuthorization*. El predicado *vendorUnsignedAuthorization* hace lo propio para el conjunto de suites sin firmar de un vendedor, mientras que *vendorSignedAuthorization* denota la autorización de acceso al conjunto de suites, firmadas con un certificado

específico de un vendedor. Cada descriptor de una suite ha sido modelado como un registro compuesto por estos predicados.

$$
\begin{aligned}
Descriptor \stackrel{\text{def}}{=} [\![\\
& required, optional : Permission \to Prop, \\
& vendor : Vendor, \\
& jarSignature : option\ Signature, \\
& signerCertificate : option\ Certificate, \\
& domainAuthorization : Domain \to Prop, \\
& signerAuthorization : Certificate \to Prop, \\
& vendorUnsignedAuthorization : Vendor \to Prop, \\
& vendorSignedAuthorization : Vendor \to Certificate \to Prop\ ]\!]
\end{aligned}
\tag{3.4}
$$

Los modos de interacción con un usuario se representan como un tipo enumerado con tres elementos:

$$Mode \stackrel{\text{def}}{=} oneshot \mid session \mid blanket \tag{3.5}$$

y los definimos de la siguiente forma:

- blanket, el permiso otorgado es válido mientras la aplicación está instalada;
- session, el permiso otorgado es válido mientras la aplicación está ejecutándose;
- oneshot, el permiso otorgado es válido sólo para ese uso

La política de seguridad es una constante de tipo *Policy*.

$$
\begin{aligned}
Policy \stackrel{\text{def}}{=} [\![\\
& allow : Domain \to Permission \to Prop, \\
& user : Domain \to Permission \to Mode \to Prop\ ]\!]
\end{aligned}
\tag{3.6}
$$

3.3. Estado del dispositivo

El estado del dispositivo se modela con un registro que contiene:

- el conjunto de suites instaladas,

- la suite activa,
- los permisos otorgados o revocados a todas las suites,
- y las autorizaciones de acceso concedidas y rechazadas por una suite a otra.

$$
\begin{aligned}
State \stackrel{\text{def}}{=} [\![& \\
& suite : SuiteID \to Prop, \\
& session : optionT\ SessionInfo, \\
& granted, revoked : SuiteID \to Permission \to Prop, \\
& authorized, unauthorized : SuiteID \to SuiteID \to Prop\ ]\!]
\end{aligned}
\tag{3.7}
$$

La suite activa, junto con los permisos otorgados o revocados a ella en la sesión, se representa con el registro *SessionInfo*.

$$
\begin{aligned}
SessionInfo \stackrel{\text{def}}{=} [\![& \\
& id : SuiteID, \\
& granted, revoked : Permission \to Prop\ ]\!]
\end{aligned}
\tag{3.8}
$$

3.3.1. Propiedades de Validez del Estado

No todo elemento de tipo *State* es un estado válido, para ello es necesario que sus componentes verifiquen ciertas propiedades. Estas propiedades involucran a componentes del estado y al contexto, y se definen con el predicado *Valid*.

$$
\begin{aligned}
Valid \stackrel{\text{def}}{=} & \ SuiteCompatible \land UniqueSuiteID \land CurrentInstalled \\
& \land ValidSessionGranted \land ValidGranted \land ValidAuthorization
\end{aligned}
\tag{3.9}
$$

La condición *SuiteCompatible* define que toda suite instalada debe ser compatible con su dominio de protección asociado.

La condición *UniqueSuiteID* garantiza la unicidad de los identificadores de suites instaladas, mientras que la condición *CurrentInstalled* garantiza que el identificador de la suite en la sesión actual corresponde al de una suite instalada.

La condición *ValidSessionGranted* asegura que los permisos otorgados en la sesión están contenidos en los permisos declarados en el descriptor de la suite activa, y que el dominio de protección permite otorgarlos en modo *session* por lo menos.

La condición *ValidGranted* establece que los permisos otorgados a una suite en forma permanente están contenidos en los permisos declarados en su descriptor y que el dominio de protección permite otorgarlos en modo *blanket*.

Finalmente, con la condición *ValidAuthorization* definida en [3] se establece que una suite no puede estar autorizada y desautorizada, al mismo tiempo, por otra suite.

3.4. Eventos

Los eventos relacionados con la seguridad se modelan como constructores del tipo *Event*. La siguiente tabla refleja todos los eventos:

Nombre	**Descripción**	**Tipo**
install	Instala una suite	$SuiteID \to Descriptor \to Domain \to Event$
remove	Remueve una suite	$SuiteID \to Event$
start	Inicia una sesión	$SuiteID \to Event$
terminate	Finaliza una sesión	$Event$
request	Solicita un permiso	$Permission \to option\, UserAnswer \to Event$
authorization	Solicita autorización de acceso	$SuiteID \to Event$

Eventos relacionados con la seguridad

En el evento *request*, la respuesta del usuario se representa con el tipo *UserAnswer*, donde *ua_allow* denota la aceptación y *ua_deny* el rechazo.

$$ua\_allow, ua\_deny : Mode \to UserAnswer \tag{3.10}$$

El comportamiento de los eventos se especifica con una precondición *Pre* y una poscondición *Pos*. Las precondiciones indican qué restricciones deben cumplirse en el estado. Las poscondiciones en cambio indican las características que deben cumplirse en el estado obtenido luego de haberse ejecutado un evento. Éstas se definen en términos del estado anterior, el estado posterior y una respuesta opcional del dispositivo. La respuesta del dispositivo se representa con el tipo *Response*, donde el valor *allowed* denota la aceptación y *denied* el rechazo.

$$Response : Set \stackrel{\text{def}}{=} allowed \,|\, denied \tag{3.11}$$

Formalmente podemos definir lo anterior de la siguiente manera:

$$\begin{aligned} &Pre : State \to Event \to Prop \\ &Pos : State \to State \to option\ Response \to Event \to Prop \end{aligned} \tag{3.12}$$

La semántica de los eventos da lugar a una *relación de ejecución en un paso* ($\Rightarrow$), con *(npre)* se establece que el estado va a permanecer inalterado si la precondición no se cumple, y con *(pre)* se establece que el estado va a cambiar si la precondición se satisface.

$$\frac{\neg\ Pre\ s\ e}{s\ \Rightarrow^{e/None}\ s}\quad (npre) \qquad \frac{Pre\ s\ e \quad Pos\ s\ s'\ r\ e}{s\ \Rightarrow^{e/r}\ s'}\quad (pre) \tag{3.13}$$

La notación $s\ \Rightarrow^{e/r}\ s'$ puede leerse como "a partir del estado s, la ejecución del evento e da lugar al estado s' y a la respuesta r".

La relación de ejecución conduce al concepto de *sesión de ejecución* determinada por un estado válido inicial del dispositivo s_0 y una secuencia de pasos $(< e_i, s_i, r_i >, i = 1, ..., n)$ que comienza con el evento *start* y finaliza con el evento *terminate*.

3.5. Invariancia de la validez del estado

La validez del estado del dispositivo (3.9) es una propiedad que se demuestra invariante con respecto a la ejecución en un paso de cualquier evento, y también con respecto a una sesión de ejecución.

A partir de la invariancia de *Valid*, el cumplimiento de varias propiedades deseables del modelo de seguridad también fue demostrado [1].

- la aplicación activa es una aplicación instalada.
- las aplicaciones instaladas disponen de permisos compatibles con los del dominio de protección al que están ligadas.
- los permisos otorgados a una aplicación durante una sesión son compatibles con los permisos declarados y con los del dominio de protección.
- los permisos otorgados en forma permanente a una aplicación son compatibles con los permisos declarados y con el dominio de protección.

- una aplicación no puede estar autorizada y desautorizada al mismo tiempo.

Una propiedad invariante en una sesión de ejecución es la revocación de un permiso. A partir de este resultado se prueba que si un permiso es revocado por el resto de una sesión, cualquier solicitud posterior del mismo permiso será también rechazada.

Otra propiedad invariante en una sesión de ejecución es la desautorización de acceso demostrada en [3]. A partir de este resultado se demuestra que si una suite ha sido desautorizada en una sesión de ejecución, cualquier solicitud posterior de autorización de acceso de la misma suite seguirá siendo rechazada.

Capítulo 4

Especificación Formal de InterMIDlet Communication

En este capítulo extendemos la formalización descrita en el capítulo anterior con los nuevos conceptos introducidos en MIDP 3.0 en base a la intercomunicación entre MIDlets. Se detallan los tipos de datos, dos nuevas condiciones de validez del estado y dos nuevos eventos:

- **IMCConnection**, modela la solicitud de conexión a la suite de la sesión actual por parte de otra suite instalada.

- **IMCServerConnection**, modela el registro como servidor IMC por parte de la suite activa.

Asimismo, analizamos la preservación de las propiedades de seguridad demostradas en los trabajos anteriores y mencionamos nuevas propiedades relacionadas a la extensión.

4.1. Formalización

Se introduce un nuevo tipo de dato: *Version* que denota las versiones de las suites.

La versión de un servidor, junto con las conexiones no autorizadas hacia éste y el predicado

que decide si una suite es aceptada o no se representa con el registro *ServerInfo*.

$$\begin{aligned} ServerInfo \stackrel{\text{def}}{=} [\![\\ & version : Version, \\ & unauthorizedIMCConnection : SuiteID \to Prop, \\ & suiteAcceptance : SuiteID \to Prop \,]\!] \end{aligned} \tag{4.1}$$

El descriptor de la suite se extiende para representar si un servidor se ha registrado o no como tal.

$$\begin{aligned} Descriptor \stackrel{\text{def}}{=} [\![\\ & required, optional : Permission \to Prop, \\ & vendor : Vendor, \\ & jarSignature : option\ Signature, \\ & signerCertificate : option\ Certificate, \\ & domainAuthorization : Domain \to Prop, \\ & signerAuthorization : Certificate \to Prop, \\ & vendorUnsignedAuthorization : Vendor \to Prop, \\ & vendorSignedAuthorization : Vendor \to Certificate \to Prop \\ & server : optionT\ ServerInfo \,]\!] \end{aligned} \tag{4.2}$$

4.2. Nuevo Estado del Dispositivo

El estado del dispositivo se extiende para representar el conjunto de servidores registrados y el registro de conexiones IMC activas.

$$\begin{aligned} State \stackrel{\text{def}}{=} [\![\\ & suite : SuiteID \to Prop, \\ & session : optionT\ SessionInfo, \\ & granted, revoked : SuiteID \to Permission \to Prop, \\ & authorized, unauthorized : SuiteID \to SuiteID \to Prop \\ & server : SuiteID \to Prop, \\ & IMCConnectionRegistry : SuiteID \to SuiteID \to Prop \,]\!] \end{aligned} \tag{4.3}$$

La validez del estado se complementa con dos condiciones; una sobre los servidores registrados y otra sobre el registro de conexiones activas.

El predicado *ValidServer* sobre un elemento *s: State* establece que todo servidor registrado debe ser una suite instalada en el dispositivo.

$$ValidServer \stackrel{\text{def}}{=} \forall\, idReq : SuiteID, \\ s.server\ idReq \rightarrow s.suite\ idReq \tag{4.4}$$

El predicado *ValidConnection* establece que en toda conexión activa la segunda componente debe ser un servidor registrado.

$$ValidConnection \stackrel{\text{def}}{=} \forall\, idGrn,\ idReq : SuiteID, \\ s.IMCConnectionRegistry\ idGrn\ idReq \rightarrow s.server\ idReq \tag{4.5}$$

4.3. Eventos

En esta sección hacemos referencia a los dos nuevos eventos que presentamos al principio del capítulo. Dividimos las precondiciones y poscondiciones en varias condiciones que se deben cumplir simultánea (mediante conjunción) o alternativamente (mediante disyunción).

4.3.1. IMCConnection

En primera instancia presentamos al evento IMCConnection que modela la solicitud de conexión a la suite de la sesión actual por parte de otra suite instalada. Para poder realizar esta conexión entre suites en cierto tipo de situaciones es requerido el atributo *Vendor* de una suite.

$$IMCConnection : SuiteID \rightarrow option\ Vendor \rightarrow Event \tag{4.6}$$

El evento tiene como precondición que en el estado s exista una suite activa, la suite activa debe ser un servidor, el identificador de la suite solicitante *idReq* debe corresponder con el identificador de una suite instalada en el dispositivo, la suite solicitante no debe ser la suite activa, no debe existir una conexión activa entre la suite solicitante y la suite activa, y por último *idReq* no debe pertenecer al conjunto de conexiones no autorizadas del servidor.

Analizaremos la precondición del evento en el caso en que se solicite o no al momento de la

conexión el *Vendor* de la suite.

$$
\begin{aligned}
&Pre\ s\ (IMCConnection\ idReq) \stackrel{\text{def}}{=} \exists\ ses : SessionInfo,\\
&\quad s.session\ =\ SomeT\ ses \wedge\\
&\quad (idReq\ <>\ ses.id \wedge\\
&\qquad s.server\ (ses.id) \wedge\\
&\qquad (\exists\ msReq1 :\ MIDletSuite,\ s.suite\ msReq1\ \wedge\ msReq1.id\ =\ idReq\\
&\qquad \wedge\ \neg\ s.IMCConnectionRegistry\ (msReq1.id)\ (ses.id)\\
&\qquad \wedge\ (\exists\ (msReq2 : MIDletSuite)\ (sinfo : ServerInfo),\ s.suite\ msReq2\\
&\qquad \wedge\ msReq2.id\ =\ ses.id\\
&\qquad \wedge\ (s.descriptor\ msReq2).server\ =\ SomeT\ sinfo\\
&\qquad \wedge\ \neg\ (sinfo.unauthorizedIMCConnection)\ sidReq)))
\end{aligned}
$$

$$
\begin{aligned}
&Pre\ s\ (IMCConnection\ idReq\ v) \stackrel{\text{def}}{=} \exists\ ses : SessionInfo,\\
&\quad s.session\ =\ SomeT\ ses \wedge\\
&\quad (idReq\ <>\ ses.id \wedge\\
&\qquad s.server\ (ses.id) \wedge\\
&\qquad (\exists\ msReq1 :\ MIDletSuite,\ s.suite\ msReq1\ \wedge\ msReq1.id\ =\ idReq\\
&\qquad \wedge\ \neg\ s.IMCConnectionRegistry\ (msReq1.id)\ (ses.id)\\
&\qquad \wedge\ (\exists\ (msReq2 : MIDletSuite)\ (sinfo : ServerInfo),\ s.suite\ msReq2\\
&\qquad \wedge\ msReq2.id\ =\ ses.id\\
&\qquad \wedge\ (s.descriptor\ msReq2).server\ =\ SomeT\ sinfo\\
&\qquad \wedge\ \neg\ (sinfo.unauthorizedIMCConnection)\ sidReq\\
&\qquad \wedge\ \mathbf{v = (s.descriptor\ msReq2).vendor})))
\end{aligned}
$$

La poscondición establece las posibles respuestas del dispositivo y la relación entre el estado previo (s) y posterior (s') a la ejecución del evento. Se analiza por casos, según exista o no una conexión activa, o en caso de no haber un antecedente, según el resultado obtenido. Asimismo se tiene en cuenta el caso cuando la suite activa no es un servidor registrado.

Si la suite activa no es un servidor, el estado del dispositivo permanece inalterado.

$$
\begin{aligned}
&Pos\ s\ s'\ r\ (IMCConnection) \stackrel{\text{def}}{=} \\
&\quad s\ =\ s' \wedge \\
&\quad \exists\, ses : SessionInfo,\ s.session\ =\ SomeT\ ses \\
&\quad \wedge\ \neg\, s.server\ (ses.id)
\end{aligned}
$$

Si ya existe una conexión activa, entonces el estado del dispositivo permanece inalterado.

$$
\begin{aligned}
&Pos\ s\ s'\ r\ (IMCConnection\ sidReq) \stackrel{\text{def}}{=} \\
&\quad s\ =\ s' \wedge \\
&\quad \exists\, ses : SessionInfo,\ s.session\ =\ SomeT\ ses\ \rightarrow \\
&\quad s.IMCConnectionRegistry\ sidReq\ (ses.id)
\end{aligned}
$$

Si no existe una conexión activa y el servidor acepta la conexión por parte de la suite, entonces el estado posterior se modifica para esa suite y se agrega al registro de conexiones activas.

$$
\begin{aligned}
&Pos\ s\ s'\ r\ (IMCConnection\ sidReq) \stackrel{\text{def}}{=} \\
&\quad equiv\_IMCConnectionRegistry\ s\ s' \wedge \\
&\quad \forall\, ses : SessionInfo,\ s.session\ =\ SomeT\ ses\ \rightarrow \\
&\quad \neg\, s.IMCConnectionRegistrysidReq(ses.id) \wedge \\
&\quad s.server\ (ses.id) \\
&\quad \wedge\ \exists\, (msReq : MIDletSuite)\ (sinfo : ServerInfo),\ s.suite\ msReq \\
&\quad \wedge\ msReq.id\ =\ ses.id \\
&\quad \wedge\ (s.descriptor\ msReq).server\ =\ SomeT\ sinfo \\
&\quad \wedge\ (sinfo.suiteAcceptance\ sidReq)
\end{aligned}
$$

$\wedge\ s'.IMCConnectionRegistry\ sidReq\ (ses.id)$

$\wedge\ (\forall\ sid1\ sid2 :\ SuiteID,$

$s.IMCConnectionRegistry\ sid1\ sid2 \rightarrow s'.IMCConnectionRegistry\ sid1\ sid2)$

$\wedge\ (\forall\ sid1 :\ SuiteID,\ sid <>\ sid1\ \rightarrow\ s'.IMCConnectionRegistry\ sid1\ sid \rightarrow$

$s.IMCConnectionRegistry\ sid1\ sid)$

$\wedge\ (\forall\ sid1 :\ SuiteID,\ sid <>\ sid1\ \rightarrow\ ses.id <>\ sid1\ \rightarrow$

$s'.IMCConnectionRegistry\ sid\ sid1 \rightarrow\ s.IMCConnectionRegistry\ sid\ sid1)$

$\wedge\ (\forall\ sid1\ sid2 :\ SuiteID,\ sid <>\ sid1\ \rightarrow\ sid <>\ sid2\ \rightarrow$

$s'.IMCConnectionRegistry\ sid1\ sid2 \rightarrow s.IMCConnectionRegistry\ sid1\ sid2)$

Por último analizamos el caso en el que no existe una conexión activa y el servidor no acepta la conexión por parte de la suite, entonces el estado se modifica para así agregar la suite al conjunto de conexiones no autorizadas por el servidor dentro de la información del servidor.

$Pos\ s\ s'\ r\ (IMCConnection\ sidReq) \stackrel{\text{def}}{=}$

$equiv\_suite\ s\ s' \wedge$

$(\forall\ ses : SessionInfo,\ s.session\ =\ SomeT\ ses\ \rightarrow$

$\neg\ s.IMCConnectionRegistrysidReq(ses.id) \wedge$

$s.server\ (ses.id)$

$\wedge\ (\exists\ (msReq : MIDletSuite)\ (sinfo : ServerInfo),\ s.suite\ msReq$

$\wedge\ msReq.id\ =\ ses.id$

$\wedge\ (s.descriptor\ msReq).server\ =\ SomeT\ sinfo$

$\wedge\ \neg\ (sinfo.suiteAcceptance\ sidReq)$

$\wedge\ (\exists\ (msReq2 : MIDletSuite)\ (sinfo2 : ServerInfo),\ s'.suite\ msReq2$

$\wedge\ msReq2.id\ =\ ses.id$

$\wedge\ (s.descriptor\ msReq2).server\ =\ SomeT\ sinfo2$

$\wedge\ (sinfo2.unauthorizedIMCConnection\ sidReq)$

$\wedge\ msReq.domain = msReq2.domain$

$\wedge\ descriptor\_server\_eq\ msReq\ msReq2))$

$\wedge\ (\forall\, ms :\ MIDletSuite,\ ms <>\ msReq\ \rightarrow\ s'.suite\ ms\ \rightarrow\ s.suite\ ms)$

$\wedge\ (\forall\, ms :\ MIDletSuite,\ ms <>\ msReq\ \rightarrow\ s.suite\ ms\ \rightarrow\ s'.suite\ ms)))$

Podemos ahora definir la poscondición del evento IMCConnection como la disyunción de las poscondiciones enunciadas previamente.

$$Pos\ s\ s'\ r\ (IMCConnection\ sidReq) \stackrel{\text{def}}{=}$$
$$r\ =\ None \wedge$$
$$(Pos\_IMCConnection\_no\_server\ \vee\ Pos\_IMCConnection\_existent\ sid \vee$$
$$Pos\_IMCConnection\_allowed\ sid\ \vee\ Pos\_IMCConnection\_denied\ sid)$$

4.3.2. IMCServerConnection

Introducimos a continuación el evento IMCServerConnection que modela el registro como servidor IMC por parte de la suite activa.

$$IMCServerConnection : Event \tag{4.7}$$

El evento tiene como precondición que en el estado s exista una suite activa, y que la suite activa no sea un servidor registrado.

$$Pre\ s\ (IMCServerConnection) \stackrel{\text{def}}{=} \forall\, ses : SessionInfo,$$
$$s.session\ =\ SomeT\ ses\ \wedge\ \neg\, s.server\ (ses.id)$$

La poscondición del evento se analiza por casos, según el caso si la suite activa ya pertenece al conjunto de servidores registrados o no.

Si la suite activa pertenece al conjunto de servidores registrados, entonces el estado del dispositivo permanece inalterado.

$$Pos\ s\ s'\ r\ (IMCServerConnection) \stackrel{\text{def}}{=}$$
$$s\ =\ s' \wedge$$
$$\wedge\ \exists\, ses : SessionInfo,\ s.session\ =\ SomeT\ ses\ \rightarrow$$
$$s.server\ (ses.id)$$

Si la suite activa no es un servidor registrado, el estado posterior del dispositivo se modifica y la suite activa se agrega al conjunto de servidores registrados.

$$
\begin{aligned}
&Pos\ s\ s'\ r\ (IMCServerConnection) \stackrel{\text{def}}{=} \\
&\quad r\ =\ None \wedge \\
&\quad equiv\_server\ s\ s' \wedge \\
&\quad (\forall\ sid :\ SuiteID,\ s.server\ sid\ \rightarrow s'.serversid) \\
&\quad \wedge\ (\forall\ ses : SessionInfo,\ s.session\ =\ SomeT\ ses\ \rightarrow \\
&\quad s'.server\ (ses.id) \wedge \\
&\quad (\forall\ sid :\ SuiteID,\ sid\ <>\ (ses.id)\ \rightarrow\ s'.server\ sid\ \rightarrow \\
&\quad s.server\ sid))
\end{aligned}
$$

Por lo tanto, podemos definir la poscondición del evento IMCServerConnection como la disyunción de las poscondiciones enunciadas con anterioridad.

$$
\begin{aligned}
&Pos\ s\ s'\ r\ (IMCServerConnection) \stackrel{\text{def}}{=} \\
&\quad (Pos\_IMCServerConnection\_not\_server \vee \\
&\quad Pos\_IMCServerConnection\_already\_server)
\end{aligned}
$$

4.4. Invariancia de Estado Válido

Con la introducción del conjunto de MIDlet suites servidores y el registro de conexiones IMC activas, la aparición de dos nuevos eventos y dos nuevas condiciones para la validez del estado del dispositivo, los invariantes demostrados anteriormente se ven afectados.

Por lo tanto, demostramos que la validez del estado del dispositivo (extendida) sigue invariante como consecuencia de la ejecución en un paso de cualquier evento, incluido los dos nuevos eventos. Formalmente, esta propiedad queda definida de la siguiente manera:

Teorema 1: ***Invarianza de Estado Válido.*** Sea *ExtValid* el predicado sobre *State* definido como la conjunción de los predicados *Valid*, *ValidConnection* y *ValidServer*. Para todo *s s': State, r: Response* y *e: Event*, si se cumplen *ExtValid s* y $s \Rightarrow^{e/r} s'$, entonces también se cumple *ExtValid s'*.

Demostración.

La prueba se sigue por análisis de casos sobre $s \Rightarrow^{e/r} s'$.

- La regla *npre* aplica cuando *Pre s e* no se verifica, y en dicho caso *s = s'*.

  De esta igualdad y de *ExtValid s* se obtiene la conclusión del teorema, *ExtValid s'*

- En el otro caso, la regla de introducción *pre* establece: *Pos s s' r e*.

La demostración continúa por análisis por casos pero esta vez sobre el evento *e*. La prueba puede encontrarse de forma completa en [6].

Asimismo demostramos que la propiedad de validez del estado se demuestra también invariante con respecto a una sesión de ejecución. El siguiente teorema establece dicho resultado.

Teorema 2: ***Invarianza de Estado Válido en Sesiones.*** Sea *ss* una sesión de ejecución a partir de un estado inicial válido. Todos los estados de *ss* son válidos:
$all(\lambda(step : StepSession).ExtValidstep.s)ss.$

La demostración por inducción estructural en la definición de sesión de ejecución puede consultarse en [1].

4.4.1. Invariancia de InterMIDlet Communication

Las propiedades mencionadas anteriormente resultan útiles para analizar otras propiedades interesantes inherentes a una sesión de ejecución. Una de estas propiedades demuestra que una vez que un servidor se registra como tal en una sesión parcial, el registro es un invariante para el resto de la sesión. El siguiente lema formaliza esta propiedad:

Lema 1: ***Invariancia de Registro de Servidores en Sesiones.*** Dados un servidor *sid* registrado y una sesión *ss*, generada a partir de un estado inicial válido, el siguiente predicado representa un invariante de sesión:

$$all\ (\lambda\ (step :\ StepSession)\ .\ \exists\ ses :\ SessionInfo,$$
$$(step.s.session\ =\ Some\ ses\ \wedge\ step.s.server\ ses.id))\ ss$$

La demostración la realizamos por inducción en la secuencia de pasos *ss*, y luego hacemos análisis de casos según el evento. La prueba completa puede encontrarse en el módulo Coq *SessionInvariants.v* dentro de [6].

Capítulo 5

Certificación de los algoritmos para IMC

En este capítulo presentamos los algoritmos relacionados al protocolo de comunicación entre MIDlets (IMC), así como sus demostraciones de corrección, mediante pruebas formales en Coq.

Para conseguir esto introducimos una representación concreta del modelo presentado en los capítulos anteriores.

5.1. Representación Concreta

El refinamiento que aplicamos a continuación sigue los lineamientos utilizados en [1] y [3].

La formalización del protocolo *IMC* presentada en los capítulos anteriores es una especificación de alto orden. En particular, observamos que se utilizan predicados para representar los componentes del estado y que éstos tienen tipo *Prop*. El inconveniente que genera esto es que la especificación resultante no es ejecutable. Con esto queremos decir que no podemos usar *Coq* para extraer un programa a partir de una especificación. Lo que podríamos haber hecho desde un principio es plantear la especificación de una manera más concreta, pero de esta forma las propiedades formales de la especificación del modelo resultarían menos generales.

Se puede entonces desde una especificación abstracta obtener una especificación ejecutable.

Para lograr esto la metodología que se adopta es la siguiente:

- de cada componente se elige una representación concreta

- cada predicado P que define una colección de elementos se representa como una lista l de elementos de tipo *Set* que satisfacen el predicado P
- los predicados P que definen alguna propiedad sobre un conjunto finito A se representan como una función F de A en *bool*

5.2. Estado Concreto del Dispositivo y Eventos

Las representaciones concretas planteadas en la sección anterior pueden ser usadas para obtener un modelo concreto del estado del dispositivo, de los nuevos eventos y de las diferentes componentes planteadas en los capítulos precedentes. Para diferenciar los tipos de datos de su representación de alto nivel se prefijan con el carácter C.

En *ServerInfo* las conexiones no autorizadas por el servidor y el predicado que decide si una suite es aceptada o no quedan representados como predicados sobre booleanos.

$$\begin{aligned} CServerInfo \stackrel{\text{def}}{=} [\![\\ & version : Version, \\ & unauthorizedIMCConnection : SuiteID \to bool, \\ & suiteAcceptance : SuiteID \to bool \,]\!] \end{aligned} \tag{5.1}$$

En el descriptor de la suite, los servidores quedan formalizados de la siguiente forma:

$$\begin{aligned} CDescriptor \stackrel{\text{def}}{=} [\![\\ & required, optional : Permission \to bool, \\ & vendor : Vendor, \\ & jarSignature : option\ Signature, \\ & signerCertificate : option\ Certificate, \\ & domainAuthorization : Domain \to bool, \\ & signerAuthorization : Certificate \to bool, \\ & vendorUnsignedAuthorization : Vendor \to bool, \\ & vendorSignedAuthorization : Vendor \to Certificate \to bool \\ & server : optionT\ CServerInfo \,]\!] \end{aligned} \tag{5.2}$$

En el estado del dispositivo, el conjunto de servidores registrados se representa con una lista.

$$
\begin{aligned}
CState \stackrel{\text{def}}{=} [\![& \\
& suite : list\ CMIDletSuite, \\
& session : optionT\ CSessionInfo, \\
& granted, revoked : SuiteID \to Permission \to bool, \\
& authorized, unauthorized : SuiteID \to SuiteID \to bool \\
& server : list\ SuiteID, \\
& IMCConnectionRegistry : SuiteID \to SuiteID \to bool\ ]\!]
\end{aligned}
\tag{5.3}
$$

La precondición del evento *IMCConnection* establece la presencia de la suite solicitante en la lista de suites instaladas y la presencia de la suite activa en la lista de servidores registrados.

También establece que la suite solicitante no debe coincidir con la suite activa, no debe existir una conexión activa entre la suite solicitante y el servidor, y que no debe existir ningún registro de conexión rechazada por parte del servidor a la suite solicitante.

$$
\begin{aligned}
& CPre\_imcconnection\_none\_vendor\ cst\ (IMCConnection\ idReq)\ := \\
& \quad \forall\ cses :\ CSessionInfo,\ cst.session\ =\ SomeT\ cses\ \wedge \\
& \quad (sidReq\ <>\ csession\_suite\ (getSession\ cst)\ \wedge \\
& \quad isServer\ cst\ (csession\_suite\ (getSession\ cst))\ =\ true\ \wedge \\
& \quad isMIDletSuite\ sidReq\ (cst\_suite\ cst)\ \wedge \\
& \quad isIMCConnectionRegistry\ cst\ sidReq \\
& \qquad (csession\_suite\ (getSession\ cst))\ =\ false\ \wedge \\
& \quad getUnauthorizedIMCConnection\ (\\
& \qquad\quad getMIDletSuite\ (csession\_suite\ (getSession\ cst)) \\
& \qquad\quad (cst\_suite\ cst))\ sidReq\ =\ false)
\end{aligned}
$$

En el caso en que se solicite el *Vendor* para autenticar la conexión, éste deberá coincidir con el *Vendor* de la suite solicitante.

$$
\begin{aligned}
&CPre\_imcconnection\_vendor\ cst\ (IMCConnection\ idReq)\ (v :\ Vendor)\ :=\\
&\quad \forall\, cses :\ CSessionInfo,\ cst.session\ =\ SomeT\ cses \wedge\\
&\quad (sidReq\ <>\ csession\_suite\ (getSession\ cst) \wedge\\
&\quad isServer\ cst\ (csession\_suite\ (getSession\ cst))\ =\ true \wedge\\
&\quad isMIDletSuite\ sidReq\ (cst\_suite\ cst) \wedge\\
&\quad isIMCConnectionRegistry\ cst\ sidReq\\
&\qquad\qquad (csession\_suite\ (getSession\ cst))\ =\ false \wedge\\
&\quad getUnauthorizedIMCConnection\ (\\
&\qquad\qquad getMIDletSuite\ (csession\_suite\ (getSession\ cst))\\
&\qquad\qquad\quad (cst\_suite\ cst))\ sidReq\ =\ false \wedge\\
&\quad isEqVendor\ (getVendor\ (getMIDletSuite\ (\\
&\qquad\qquad csession\_suite\ (getSession\ cst))\\
&\qquad\qquad\quad (cst\_suite\ cst)))\ v\ =\ true)
\end{aligned}
$$

A modo de ejemplo, detallamos una de las cuatro poscondiciones del evento. Este es el caso en el que no existe una conexión activa y el servidor acepta la conexión por parte de la suite solicitante, entonces el estado posterior se modifica para esa suite y se agrega al registro de

conexiones activas.

$$
\begin{aligned}
&CPos\_imcconnection\_allowed\ cst\ (IMCConnection\ idReq)\ :=\\
&\quad cequiv\_IMCConnectionRegistry\ cst\ cst' \wedge\\
&\quad (\forall\ cses :\ CSessionInfo,\ cst.session\ =\ SomeT\ cses\ \rightarrow\\
&\quad isIMCConnectionRegistry\ cst\ sidReq\ (csession\_suite\ (getSession\ cst))\ =\ false \wedge\\
&\quad isServer\ cst\ (csession\_suite\ (getSession\ cst))\ =\ true \wedge\\
&\quad getSuiteAcceptance\ (getMIDletSuite\ (csession\_suite\ (getSession\ cst))\\
&\qquad (cst\_suite\ cst))\ sidReq\ =\ true \wedge\\
&\quad isIMCConnectionRegistry\ cst'\ sidReq\ (csession\_suite\ (getSession\ cst))\ =\ true \wedge\\
&\quad \wedge\ (\forall\ sid1\ sid2 :\ SuiteID,\ isIMCConnectionRegistry\ cst\ sid1\ sid2\ =\ true\\
&\qquad \rightarrow\ isIMCConnectionRegistry\ cst'\ sid1\ sid2\ =\ true)\\
&\quad \wedge\ (\forall\ sid1 :\ SuiteID,\ sidReq\ <>\ sid1\ \rightarrow\\
&\qquad isIMCConnectionRegistry\ cst'\ sid1\ sidReq\ =\ true\\
&\qquad \rightarrow\ isIMCConnectionRegistry\ cst\ sid1\ sidReq\ =\ true)\\
&\quad \wedge\ (\forall\ sid1 :\ SuiteID,\ sidReq\ <>\ sid1\ \rightarrow\ (csession\_suite\ (getSession\ cst)\ <>\ sid1\\
&\qquad \rightarrow\ isIMCConnectionRegistry\ cst'\ sidReq\ sid1\ =\ true\\
&\qquad\quad \rightarrow\ isIMCConnectionRegistry\ cst\ sidReq\ sid1\ =\ true)\\
&\quad \wedge\ (\forall\ sid1\ sid2 :\ SuiteID,\ sidReq\ <>\ sid1\ \rightarrow\ sidReq\ <>\ sid2\\
&\qquad \rightarrow\ isIMCConnectionRegistry\ cst'\ sid1\ sid2\ =\ true\\
&\qquad\quad \rightarrow\ isIMCConnectionRegistry\ cst\ sid1\ sid2\ =\ true))
\end{aligned}
$$

La precondición del evento *IMCServerConnection* establece que la suite activa no se encuentre en la lista de servidores registrados.

$$
\begin{aligned}
&CPre\_imcserverconnection\ cst\ (IMCServerConnection)\ :=\ \forall\ cses :\ CSessionInfo,\\
&\quad cst.session\ =\ SomeT\ cses \wedge\\
&\quad isServer\ cst\ (csession\_suite\ (getSession\ cst))\ =\ false
\end{aligned}
$$

La poscondición queda detallada de la siguiente forma:

$$
\begin{aligned}
&CPos\_imcserverconnection\ cst\ (IMCServerConnection)\ :=\\
&\quad r\ =\ None\ \wedge\\
&\quad cequiv\_server\ cst\ cst'\ \wedge\\
&\quad (\forall\ sid:\ SuiteID,\ isServer\ cst\ sid\ =\ true\ \rightarrow\ isServer\ cst'\ sid\ =\ true)\ \wedge\\
&\quad \forall\ cses:\ CSessionInfo, cst.session\ =\ SomeT\ cses\ \rightarrow\\
&\quad isServer\ cst'\ (csession\_suite\ (getSession\ cst))\ =\ true\ \wedge\\
&\quad (\forall\ sid:\ SuiteID,\ sid\ <>\ (csession\_suite\ (getSession\ cst))\ \rightarrow\\
&\quad isServer\ cst'\ sid\ =\ true\ \rightarrow\ isServer\ cst\ sid\ =\ true)
\end{aligned}
$$

5.3. Algoritmos para InterMIDlet Communication

En esta sección proponemos algoritmos para los eventos IMCConnection e IMCServerConnection.

El algoritmo de solicitud de conexión **CAlg_imcconnection** recibe como parámetros el identificador de la suite solicitante *(sidReq)* y el estado concreto actual *(cst)*, y retorna una pareja formada por un estado, eventualmente modificado, y una respuesta.

A partir del estado actual el algoritmo obtiene el ID de la suite activa (idGrn), que es la que va a actuar de servidor para la conexión.

Con la obtención de la suite solicitante y la suite servidor que participan en esta solicitud de conexión, el algoritmo procede a realizar una serie de comparaciones. Este proceso se descompone en los siguiente casos anidados:

- Caso 0 : la suite solicitante no es la suite activa.
- Caso 1 : la suite activa es un servidor registrado. Si se da el caso en que la suite activa no es un servidor registrado entonces se retorna el estado original sin modificaciones y una respuesta nula. Si por el contrario, la suite activa es un servidor registrado se procede al siguiente caso.
- Caso 2 : existe un registro de conexión activa entre la suite solicitante y la suite activa. Si este es el caso, entonces se retorna el estado original sin modificaciones y una respuesta

nula; caso contrario se procede a comparar si la suite solicitante pertenece al conjunto de suites no autorizadas por el servidor.

- Caso 3 : la suite solicitante pertenece al conjunto de suites no autorizadas. Si se da el caso, se retorna el estado original sin modificaciones y una respuesta nula; caso contrario se procede a comparar si la suite solicitante es aceptada por el servidor a establecer una conexión.

- Caso 4 : la suite no es aceptada por el servidor. Si este es el caso, entonces el estado original se modifica, ya que la suite solicitante pasa a formar parte del conjunto de suites no autorizadas dentro de la información propia del servidor. Por el contrario, si el servidor autoriza la conexión entonces el estado original también se modifica, ya que la suite solicitante pasa a formar parte del conjunto de conexiones IMC activas.

El código *Coq* del algoritmo CAlg_imcconnection se presenta a continuación.

```
CAlg_imcconnection(sidReq : SuiteID)(cst : CState) : CState * option Response :=
let idGrn := (csession_suite(getSession cst)) in
let msGrn := (getMIDletSuite(csession_suite(getSession cst))(cst_suite cst)) in
  match (sid_eq_dec sidReq idGrn) with
    | left => (cst, None)(** caso 0: la suite solicitante no es la suite activa *)
    | right =>
      if (isServer cst idGrn) then
        if (isIMCConnectionRegistry cst sidReq idGrn) then
        (cst, None)(** caso 2 : existe un registro de conexion activa *)
        else if (getUnauthorizedIMCConnection msGrn sidReq) then
          (cst, None)(** caso 3: pertenece al conjunto de suite no autorizadas*)
          else if (getsuiteAcceptance (getMIDletSuite idGrn (cst_suite cst)) sidReq) then
          (** caso 5 : la suite SI es aceptada por el servidor *)
            (mkCState (cst_suite cst)(cst_session cst)(cst_granted cst)(cst_revoked cst)
              (cst_authorized cst)(cst_unauthorized cst)(cst_server cst)
```

$(addMIDletSuiteIMCConnectionRegistry\ cst\ sidReq\ idGrn),\ None)$

$else$**(** caso 4 : la suite no es aceptada por el servidor *)**

$(mkCState\ (addMIDletSuiteUnauthorizedIMCConnection\ cst\ idGrn\ sidReq)$

$(cst\_session\ cst)(cst\_granted\ cst)(cst\_revoked\ cst)(cst\_authorized\ cst)$

$(cst\_unauthorized\ cst)(cst\_server\ cst)(cst\_IMCConnectionRegistry\ cst),\ None)$

$else\ (cst,\ None)$**(** caso 1 : la suite activa no es un servidor registrado *)**

$end.$

Para el caso del algoritmo de registro como servidor IMC, **CAlg_imcserverconnection**, recibe como parámetros el estado concreto actual *(cst)*, y retorna una pareja formada por un estado, eventualmente modificado, y una respuesta.

En este caso también a partir del estado actual se obtiene el ID de la suite activa (idGrn), que es la que va a registrarse como servidor.

El código *Coq* del algoritmo *CAlg_imcserverconnection* se presenta a continuación.

CAlg_imcserverconnection$(cst : CState) : CState * option\ Response :=$

$let\ idGrn := (csession\_suite(getSession\ cst))\ in$

$if\ (isServer\ cst\ idGrn)\ then$

$(cst, None)$**(** caso 1: la suite activa ya es un servidor registrado *)**

$else\ (mkCState$ **(** caso 2: la suite activa no es un servidor registrado *)**

$(cst\_suite\ cst)(cst\_session\ cst)(cst\_granted\ cst)(cst\_revoked\ cst)$

$(cst\_authorized\ cst)(cst\_unauthorized\ cst)(addMIDletSuiteServer\ cst\ idGrn)$

$(cst\_unauthorized\ cst)(cst\_server\ cst)(cst\_IMCConnectionRegistry\ cst)$

$(cst\_IMCConnectionRegistry\ cst),\ None)$

5.4. Certificación de los Algoritmos

Diremos que un algoritmo está certificado si partiendo de un estado inicial válido, en el cual se cumplen las precondiciones del evento, se obtiene por la ejecución del mismo un estado resultante que satisface la poscondición de dicho evento. En otras palabras, un algoritmo está certificado cuando cumple con la especificación.

A continuación formalizamos la certificación de los algoritmos presentados en la sección anterior a través de dos teoremas.

Teorema IMCConnectionCorrectness.
Para toda pareja de estados concretos *(cst cst': CState)*, suite solicitante *(msReq: CMIDletSuite)*, vendor *(option Vendor)* y respuesta opcional *(r: option Response)* a la solicitud de conexión IMC; si en el estado *cst* se cumple *CPre_imcconnection cst (csuite_id msReq) v*, la precondición del evento IMCConnection, entonces en el estado concreto resultante del algoritmo de IMCConnection *(cst',r) = CAlgIMCConnection (csuite_id msReq) cst* se cumple *CPos_imcconnection cst cst' r (csuite_id msReq) v*, la poscondición del evento IMCConnection.

Demostración.

La demostración sigue los lineamientos utilizados en [3]. Ésta se realiza mediante el análisis de casos de las condiciones evaluadas por el algoritmo. Dado que en la representación concreta los predicados están expresados como funciones cuyo recorrido son los booleanos, se inyectan como hipótesis la disyunción de cada una de las condiciones evaluadas en verdadero y falso:

$$\begin{aligned} isServer\ cst\ (csession\_suite\ (getSession\ cst)) \ &= \ false \vee \\ isServer\ cst\ (csession\_suite\ (getSession\ cst)) \ &= \ true \end{aligned} \tag{5.4}$$

$$\begin{aligned} isIMCConnectionRegistry\ cst\ (csuite\_id\ msReq)(csession\_suite(getSession\ cst)) \ &= \ false \vee \\ isIMCConnectionRegistry\ cst\ (csuite\_id\ msReq)(csession\_suite(getSession\ cst)) \ &= \ true \end{aligned} \tag{5.5}$$

$$\begin{aligned} &getUnauthorizedIMCConnection(getMIDletSuite(csession\_suite(getSession\ cst)) \\ &\quad (cst\_suite\ cst))(csuite\_id\ msReq) \ = \ false \vee \\ &\quad getUnauthorizedIMCConnection(getMIDletSuite(csession\_suite(getSession\ cst)) \\ &\quad (cst\_suite\ cst))(csuite\_id\ msReq) \ = \ true \end{aligned} \tag{5.6}$$

$$\begin{aligned} &getsuiteAcceptance(getMIDletSuite(csession\_suite(getSession\ cst))(cst\_suite\ cst)) \\ &\quad (csuite\_id\ msReq) \ = \ false \vee \\ &getsuiteAcceptance(getMIDletSuite(csession\_suite(getSession\ cst))(cst\_suite\ cst)) \\ &\quad (csuite\_id\ msReq) \ = \ true \end{aligned} \tag{5.7}$$

De esta forma se cubren todos los casos detallados en el algoritmo.

Para ilustrar la demostración se pone énfasis en el caso 1, correspondiente a la solicitud de conexión IMC hacia un servidor no registrado.

De la inyección de las condiciones se agrega la siguiente hipótesis sobre la suite activa:

- $isServer\ cst(csession\_suite(getSession\ cst)) = false$, la suite activa no es un servidor registrado.

A continuación, se verifica la poscondición cuando la suite activa no es un servidor registrado.

De la precondición del evento, se obtiene una serie de hipótesis que luego se sustituyen en el algoritmo de imcconnection. Se obtiene la salida del algoritmo, formada por el estado resultante y la respuesta del evento, que se incorporan como nuevas hipótesis.

De las poscondiciones del evento imcconnection, solamente aplica la correspondiente a cuando la suite activa no es un servidor registrado y cuando el estado permanece inalterado.

Al verificar cada uno de los predicados, demostramos entonces que la poscondición del evento IMCConnection se cumple.

La demostración completa, formulada y verificada con el asistente de pruebas *Coq*, se puede consultar en [6].

Teorema IMCServerConnectionCorrectness.
Para toda pareja de estados concretos *(cst cst': CState)* y respuesta opcional *(r: option Response)* al registro como servidor IMC por parte de la suite activa; si en el estado *cst* se cumple *CPre_imcserverconnection cst*, la precondición del evento IMCServerConnection, entonces en el estado concreto resultante del algoritmo de IMCServerConnection *(cst',r) = CAlgIMCServerConnection cst* se cumple *CPos_imcserverconnection cst cst' r*, la poscondición del evento IMCServerConnection.

Demostración.

De forma similar a la demostración anterior, se introduce como hipótesis la disyunción de la condición que establece si una suite es servidor evaluada en verdadero y falso:

$$\begin{aligned} isServer\ cst\ (csession\_suite\ (getSession\ cst)) &= false \vee \\ isServer\ cst\ (csession\_suite\ (getSession\ cst)) &= true \end{aligned} \tag{5.8}$$

Para ilustrarla se detalla el caso 2, correspondiente al registro como servidor de una suite que ya es servidor.

De la inyección de la condición se agrega la siguiente hipótesis sobre la suite activa:

$$isServer\ cst\ (csession\_suite\ (getSessiocn\ cst))\ =\ true$$

De la precondición del evento obtenemos la sesión activa y que la suite activa no es un servidor registrado. Luego, haciendo una reescritura en las hipótesis obtenemos que $true = false$; y mediante esto podemos probar nuestro goal.

Se demuestra entonces que la poscondición del evento IMCServerconnection se cumple.

La demostración completa, formulada y verificada con el asistente de pruebas *Coq*, se puede consultar en [6].

Capítulo 6

Trabajo relacionado, Conclusiones y Trabajo futuro

6.1. Trabajo Relacionado

6.1.1. El Modelo de Seguridad de Android

En esta sección describimos las principales características del modelo de seguridad de Android [17]. Asimismo realizamos una comparación con el modelo que presenta MIDP.

Una de las características más innovadora de Android es su *openness*. Esto lleva aparejado inquietudes respecto de la seguridad que se ofrece a la hora de instalar una aplicación. Android aborda estos problemas de seguridad mediante la aplicación de una política de seguridad basada en permisos en cada dispositivo [18]. Estos permisos son cadenas de caracteres únicas y distinguibles unas de otras.

Un componente (o API) protegido mediante permisos puede ser accedido sólo por aplicaciones que sean propietarias de estos permisos. La autorización de los permisos se hace en el momento en que la aplicación es instalada, con el consentimiento del usuario, es decir, la aplicación solicita el conjunto de permisos requeridos para completar la tarea cuando es instalada en el dispositivo. Android provee una serie de permisos por defecto y también permite definir nuevos permisos. Los nuevos permisos deben ser declarados dentro de la aplicación e introducidos en el sistema una vez que la aplicación es instalada.

Existen diferentes niveles de protección que poseen los permisos. El nivel de protección es un

atributo propio de un permiso, y determina cómo un permiso puede ser otorgado. Estos niveles de protección pueden definirse como: normal, peligroso, firmado y firmadoOSistema.

Android [19] utiliza el enfoque *"sandbox"* para evitar que una aplicación ejerza intenciones maliciosas en otras aplicaciones o en el sistema. Básicamente, este concepto apunta a que una aplicación se ejecuta en un *sandbox* (arenero) seguro; por lo que no se puede acceder a recursos a menos que esté explícitamente autorizada a hacerlo. El *principio de seguridad principal* de Android enuncia que "las aplicaciones pueden realizar operaciones que afecten a otras partes del sistema sólo cuando se les permite hacerlo". El mecanismo sandbox es soportado por mecanismos de bajo nivel.

Al igual que muchas de las características de seguridad, el sandbox de aplicaciones no es inviolable. Sin embargo, para hacerlo, uno debe comprometer la seguridad del kernel de Linux. A diferencia de éste, en la primera versión de MIDP, el modelo de seguridad se basaba en la seguridad a nivel de plataforma y definía también un *sandbox* que prohibía el acceso a aquellas funciones sensibles del dispositivo.

En MIDP 2.0 se refina el modelo sandbox en uno menos restrictivo, basado en el concepto de *dominio de protección*. Cada aplicación instalada está ligada a un único dominio de protección. La política de seguridad es una matriz de control de acceso, donde los sujetos son las aplicaciones asociadas a cada dominio de protección y los objetos son las funciones del dispositivo que deben protegerse.

En la versión más reciente de MIDP, el modelo de seguridad se basa en la seguridad a nivel de aplicación. Se permite que aplicaciones de diferentes dominios de seguridad compartan datos en formato RMS y se comuniquen en tiempo de ejecución mediante eventos de aplicación, o mediante el protocolo IMC. La seguridad a nivel de aplicación, basada en el concepto de autorización de acceso, es complementaria de la seguridad a nivel de plataforma, basada en el concepto de permiso. Para que una aplicación comparta recursos debe declarar autorizaciones de acceso, que se diferencian por las credenciales exigidas a las aplicaciones solicitantes. Cuando otra aplicación pretende acceder a recursos compartidos, el dispositivo aplica un conjunto de reglas para determinar si está o no autorizada.

Por lo tanto, notamos que en el modelo de seguridad que presenta Android, el conjunto abstracto de operaciones no incluye operaciones de solicitud/revocamiento de permisos, mientras que la especificación de MIDP sí lo hace. Estas operaciones son representadas en las primitivas

y se llevan a cabo cuando las aplicaciones son instaladas/desinstaladas.

Intercomunicación entre procesos

Los procesos pueden comunicarse entre ellos utilizando cualquiera de los mecanismos de comunicación provistos por UNIX. Algunos ejemplos pueden ser el sistema de archivos, sockets locales, o señales.

Android provee de algunos mecanismos de comunicación entre procesos:

- *Binders*; son implementados usando un driver de Linux a medida.
- *Servicios*; pueden proveer interfaces directamente accesibles mediante binders.
- *Intents*; es un simple objeto mensaje que representa una "intención" de hacer algo. Los intents también pueden ser utilizados para transmitir eventos interesantes (tales como notificaciones) hacia todo el sistema.
- *Proveedores de contenidos*; un claro ejemplo de éstos es el que se usa para acceder a la lista de contactos del usuario. Una aplicación puede acceder a datos que otra aplicación haya expuesto mediante estos proveedores de contenidos, y además una aplicación puede definir su propio proveedor de contenidos para exponer sus datos.

La API de Android define métodos que aceptan *intents* y usan la información provista por ellos para inicializar actividades, servicios y transmitir mensajes. La invocación de estos métodos le informa al *framework* de Android que comience a ejecutar código en la aplicación destino.

Este proceso de intercomunicación entre componentes (ICC) se denomina acción. En muchos sentidos, el ICC es análogo a la intercomunicación entre procesos de sistemas basados en UNIX. Las funciones que utiliza el ICC hacen caso omiso a si el destino está o no en la misma aplicación, a excepción de las reglas de seguridad. La mediación del ICC es la base del framework de seguridad de Android. En términos generales, cada aplicación se ejecuta como si fuese un único usuario, lo cual permite a Android limitar daños potenciales debido a fallas en la programación. Para especificar el dominio de protección de una aplicación se asignan etiquetas de permisos a la aplicación y para especificar una política de acceso para proteger sus recursos se asignan permisos a los componentes de una aplicación. El conjunto de etiquetas de permisos es seteado

en el momento de la instalación, como se ha dicho anteriormente, y no puede ser cambiado hasta que la aplicación no sea reinstalada.

El grupo de desarrolladores de Google ha incorporado una serie de refinamientos al modelo de seguridad. Uno de estos refinamientos se basa en el hecho de tener *componentes privadas y públicas*. Algunas aplicaciones contienen componentes que nunca deberían ser accedidas por alguna otra aplicación. Así que en vez de definir un permiso de acceso para esta componente, lo que se hace es convertir esta componente en una componente privada.

Como se expresó previamente, en MIDP 3.0 se define un protocolo de comunicación (*IMC*) que permite a las MIDlets (aplicaciones) que residen en un mismo dispositivo móvil, que se comuniquen y compartan datos entre ellas. A diferencia de Android que proporciona una serie de mecanismos de comunicación entre procesos, MIDP utiliza el protocolo IMC que tiene semejanzas con el protocolo de comunicación *Socket*.

Actualizaciones en Android

Existen dos formas de actualizar el código en la mayoría de los dispositivos con Android: over-the-air (OTA) o side-loaded. Las actualizaciones side-loaded son provistas por una locación central, mediante un archivo zip hacia una computadora personal o directamente en el dispositivo. Una vez que la actualización es copiada o descargada en la tarjeta SD del dispositivo, Android reconoce la actualización, verifica su integridad y autenticidad, y automáticamente actualiza el dispositivo.

En MIDP 2.0 se especifica la provisión a través del aire (OTA) como requisito para descargar e instalar contenido a través de una red inalámbrica. En la versión más reciente de MIDP se utiliza el RMS para la provisión de datos hacia las aplicaciones. La diferencia recae en el formato en el que estos archivos son formateados.

6.2. Conclusiones

Se presentó una especificación formal del modelo de seguridad de MIDP en base a los trabajos [1],[20]. Se formalizaron las componentes básicas que sirven para modelar un estado válido de un dispositivo; se presentaron los eventos relacionados con la seguridad; se definió un predicado que establece la validez del estado del dispositivo; se presentaron invariantes de los eventos y de

las sesiones, que establecen propiedades de seguridad básicas del modelo, a partir de las cuales pueden demostrarse otras propiedades relevantes.

A raíz de este modelo se extiende la especificación precedentemente aludida para así formalizar el protocolo de comunicación entre MIDlets (IMC). Se incorporan al modelo nuevas componentes para describir los dos nuevos eventos que constituyen el protocolo; la solicitud de conexión entre una suite y un servidor activo, y el registro como servidor por parte de una suite activa. Se conservan las propiedades verificadas en la formalización anterior, por lo cual podemos decir que la extensión es conservativa, y se plantean nuevas propiedades relacionadas con el protocolo IMC. En base a esto, se verifica que todo servidor registrado debe ser una suite instalada en el dispositivo, y que en toda conexión activa una de las suites involucradas debe ser un servidor registrado. Asimismo se demuestra que una vez que un servidor se registra en una sesión parcial, el registro es un invariante para el resto de la sesión.

Finalmente, refinamos la especificación construyendo un modelo concreto en base al modelo abstracto presentado y definimos dos algoritmos escritos en Coq como implementaciones de los eventos mencionados con anterioridad. Uno que corresponde al registro de servidores IMC y el otro que corresponde a la solicitud de conexión de una MIDlet suite con un servidor válido. Luego, verificamos la corrección de estos dos algoritmos respecto a sus pre y pos condiciones.

En la siguiente guía se detallan los diferentes módulos (archivos) Coq que integran la especificación desarrollada en esta investigación, junto con las diversas modificaciones realizadas a los mismos en este trabajo.

- Datatypes: se definen los tipos de datos utilizados en la especificación.

- Prelude: están definidos los permisos, el dominio de protección, el conjunto de identificadores válidos de MIDlet Suite, el descriptor de aplicación, las MIDlet Suite, los diferentes modos de los permisos y la política de seguridad. En este trabajo de tesina se agregan las versiones de las suites, el registro *ServerInfo* con la información de los servidores, se extiende el descriptor de la suite para representar si un servidor se ha registrado o no como tal y se agregan axiomas de igualdad entre MIDlets.

- State: están definidas las componentes básicas del estado del dispositivo y las propiedades de validez del mismo. Aquí se extiende el estado para representar el conjunto de servidores registrados y un registro de conexiones activas. Además la validez del estado se comple-

menta con dos condiciones; una sobre los servidores registrados (*ValidServer*) y otra sobre el registro de conexiones activas (*ValidConnection*). Se agregan algunas relacionan útiles entre los estados y se modifican las ya definidas.

- Events: se modelan los eventos relacionados con la seguridad. Aquí se agregan los dos nuevos eventos del protocolo IMC; *IMCConnection* e *IMCServerConnection*.

- Behaviour: se especifican las pre y pos condiciones de los eventos. Por lo tanto se agregan las mismas para los dos nuevos eventos y se realizan modificaciones para conservar las propiedades verificadas en la formalización anterior.

- StepInvariants: están definidos los invariantes de los eventos. Se realizaron modificaciones en las definiciones y demostraciones de los invariantes ya definidos, como consecuencia de la adición de los nuevos eventos y las condiciones de validez del estado. Además se agregaron invariantes para los eventos IMCConnection e IMCServerConnection.

- Session: se define la noción de sesión para una Suite.

- SessionInvariants: se definen los invariantes de sesión, es decir, propiedades de un paso que se cumplen por el resto de la sesión una vez que se establecen en un paso determinado. También se realizan modificaciones producto de los nuevos eventos y se agrega un invariante que plantea que una vez que un servidor se registra como tal en una sesión parcial, el registro es un invariante para el resto de la sesión.

- Concrete: contiene la representación concreta del modelo de seguridad para obtener un prototipo ejecutable. Se extiende esta representación con los componentes y definiciones inherentes a nuestra especificación. A su vez se definen los dos algoritmos que competen al protocolo de *InterMIDlet Communication*.

- Correctness: contiene la certificación de los algoritmos respecto a sus pre y pos condiciones. En nuestro caso, se plantean los teoremas que sirven para certificar los dos nuevos algoritmos.

Archivo	Formalización previa (LOC)	Extensión desarrollada (LOC)
Datatypes.v	83	83
Prelude.v	138	170
State.v	210	283
Events.v	55	66
Behaviour.v	538	728
StepInvariants.v	2142	4241
Session.v	88	88
SessionInvariants.v	1003	1638
Concrete.v	377	746
Correctness.v	501	1708
Total	**5135**	**9753**

Tabla comparativa entre la especificación anterior y la presente

En el cuadro anterior se puede observar que la especificación completa comprende alrededor de 9700 líneas de código *Coq*, dónde más de 4600 corresponden a la extensión desarrollada en el marco de este trabajo.

6.3. Trabajo futuro

Como trabajo futuro se propone utilizar la especificación certificada realizada con el asistente de pruebas Coq para así extraer un prototipo correcto por construcción. También se propone formalizar los eventos inherentes a la desconexión de suites y al desregistro de servidores. Esta extensión debería darse en forma trivial respecto a los eventos complementarios propuestos en el presente trabajo.

Además se propone trabajar en la formalización del modelo de seguridad de Android y el análisis formal del mismo en relación al de MIDP.

Otra posible línea de trabajo es la formalización de características de MIDP 3.0 no abordadas en el presente trabajo (concurrencia, RMS Data Provisioning & RMS Interchange File Format, LIBlets).

Bibliografía

[1] S. Zanella, G. Betarte, and C. Luna. A formal specification of the midp 2.0 security model, 2006. http://www.fing.edu.uy/inco/pedeciba/bibliote/reptec/TR0609.pdf Último acceso, junio 2013.

[2] S. Zanella, G. Betarte, and C. Luna, 2006. A Formal Specification of the MIDP 2.0 Security Model. Formal Aspects in Security and Trust 2006: 220-234, Hamilton, Ontario, Canada.

[3] G. Mazeikis, G. Betarte, and C. Luna. Formalización y Análisis del Modelo de Seguridad de MIDP 3.0, 2009. http://www.fing.edu.uy/inco/grupos/gsi/proyectos/STEVE/Doc/Formacion/Grado/Mazeikis/tesisMazeikis2009-midp3-0.pdf Último acceso, junio 2013.

[4] Java. Mobile information device profile. http://jcp.org/aboutJava/communityprocess/final/jsr271/index.html Último acceso, junio 2013.

[5] The Coq Development Team. The coq proof assistant reference manual. version 8.4pl2, abril 2013.

[6] José Forte. Código coq de la especificación formal de midp 3.0. http://www.fing.edu.uy/~cluna/Forte_tesina.zip Último acceso, septiembre 2013.

[7] G. Mazeikis, G. Betarte, and C. Luna, 2009. Autorización de Acceso en MIDP 3.0, Congreso Iberoamericano de Seguridad Informática, CIBSI'09, Uruguay, Noviembre de 2009.

[8] Oracle. The java micro edition platform, 2011. http://www.oracle.com/technetwork/java/javame/about-java-me-395899.html Último acceso, junio 2013.

[9] Oracle. Java community process. http://jcp.org/en/introduction/overview Último acceso, junio 2013.

[10] Sun Microsystems Inc. Connected limited device configuration 1.1 (jsr 139). http://jcp.org/jsr/detail/139.jsp Último acceso, junio 2013.

[11] Sun Microsystems Inc. Connected device configuration (jsr 218) 1.1. http://jcp.org/jsr/detail/218.jsp Último acceso, junio 2013.

[12] Nokia. Mobile information device profile 1.0. http://www.developer.nokia.com/Community/Wiki/MIDP_1.0 Último acceso, junio 2013.

[13] Nokia. Mobile information device profile 2.0. http://www.developer.nokia.com/Community/Wiki/MIDP_2.0 Último acceso, junio 2013.

[14] Oracle. What's new in midp 2.0. http://www.oracle.com/technetwork/systems/midp20-156411.html Último acceso junio 2013.

[15] Sun Microsystems Inc. Over the air provisioning. http://www.oracle.com/technetwork/systems/ota-156595.html Último acceso, junio 2013.

[16] Nokia. Mobile information device profile 3.0. http://www.developer.nokia.com/Community/Wiki/MIDP_3.0 Último acceso, junio 2013.

[17] Google Inc. What is android? http://developer.android.com/guide/components/index.html Último acceso, junio 2013.

[18] William Enck, Machigar Ongtang, and Patrick McDaniel. Understanding android security. http://dx.doi.org/10.1109/MSP.2009.26 Último acceso, junio 2013.

[19] Wook Shin, Shinsaku Kiyomoto, Kazuhide Fukushima, and Toshiaki Tanaka. A formal model to analyze the permission authorization and enforcement in the android framework. *Symposium on Automatic Program Verification, Universidad Nacional de Río Cuarto*, 2010. http://dx.doi.org/10.1109/SocialCom.2010.140 Último acceso, junio 2013.

[20] G. Mazeikis, G. Betarte, and C. Luna, 2009. Formal Specification and Analysis of the MIDP 3.0 Security Model, sccc, pp.59-66, 2009 International Conference of the Chilean Computer Science Society.