UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

PORTANDO AMBIENTTALK A DISPOSITIVOS MÓVILES LIVIANOS

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

ESTEBAN ARMANDO ALLENDE PRIETO

PROFESOR GUÍA:
SR. ÉRIC TANTER

MIEMBROS DE LA COMISIÓN:
SR. PABLO BARCELÓ BAEZA
SR. LUIS MATEU BRULE

SANTIAGO DE CHILE
ABRIL 2010

RESUMEN DE LA MEMORIA
PARA OPTAR AL TITULO DE
INGENIERO CIVIL COMPUTACIÓN
POR: ESTEBAN ALLENDE P.
FECHA: 09/03/2010
PROF. GUIA: SR. ÉRIC TANTER

"PORTANDO AMBIENTTALK A DISPOSITIVOS MÓVILES LIVIANOS"

El objetivo general del presente trabajo es poder interpretar un subconjunto del lenguaje de programación AmbientTalk en un dispositivo móvil liviano que sea capaz de interactuar con otros programas desarrollados en AmbientTalk alojados en otros dispositivos que no sean necesariamente del mismo tipo de equipo. El dispositivo móvil liviano usado para esta memoria es un Sun SPOT.

El intérprete oficial de AmbientTalk esta desarrollado para Java ME CDC, mientras que los Sun SPOT poseen como plataforma de desarrollo Java ME CLDC. Una de las diferencias importantes entre ambos es que tanto reflexión como serialización no están presentes en CLDC, mientras que si lo están en CDC. Ambas son características muy usadas y muy imprescindibles para el intérprete oficial, por lo que se hace necesario replantear un nuevo intérprete para los Sun SPOT.

Debido a las capacidades de cómputo limitadas de un Sun SPOT, se decidió separar la plataforma en dos aplicaciones que corren en máquinas distintas: un compilador que lea código fuente AmbientTalk y que genere un archivo binario, y un intérprete AmbientTalk que lea ese archivo binario e interprete el programa almacenado en él en un Sun SPOT. A cada una de las aplicaciones se le realizó un diseño de arquitectura lógica, separando los componentes en módulos semi desacoplados.

Luego se realizó una validación al intérprete, realizando para tal efecto una aplicación ejemplo de programación distribuida consistente en un sistema controlador de luces remotas. Se realizó esta aplicación tanto para Java ME CLDC, como para AmbientTalk, mostrando que mientras la aplicación en Java ME CLDC es más eficiente, la aplicación en AmbientTalk fue más simple de desarrollar.

Finalmente se concluye que esta plataforma es un paso para tener una implementación de AmbientTalk en toda la gama de dispositivos con Java y que AmbientTalk permite disminuir el tiempo de programación para aplicaciones distribuidas en un Sun SPOT.

Índice General

Capítulo 1

Introducción

1.1. Programación orientada a ambientes

El desarrollo de las tecnologías computacionales está tendiendo a la computación pervasiva [23]. Esta consiste en un ejército de dispositivos móviles, integrados a objetos usados a diario, para proveer facilidades al usuario, sin que éste se dé cuenta que hay un computador apoyando esa tarea.

Una característica necesaria para el desarrollo de la computación pervasiva es una infraestructura de red, requerida para la comunicación entre los dispositivos. Debido a que esta tecnología es transparente para el usuario y los objetos se trasladan de posición, tal infraestructura requiere que los nodos espontáneamente y continuamente desaparezcan y reaparezcan. Son por estos motivos que la computación pervasiva va a necesitar de las redes móviles ad-hoc. Las redes móviles ad-hoc permiten a los dispositivos comunicarse inalámbricamente y de forma espontánea cuando se encuentran dentro del rango de comunicación.

La mayoría de las herramientas actualmente disponibles no son capaces de abstraer a un desarrollador de las características inherentes de las redes móviles ad-hoc, teniendo que gastar parte del tiempo de desarrollo en adaptar estas herramientas.

Es esta el área que quiere solucionar la programación orientada a ambientes [5, 7], la cual se basa en los siguientes principios:

1. **Objetos sin clases**: en redes móviles ad-hoc es demasiado estricto pedir que se mantenga sincronizado entre todos los dispositivos una estructura de clases. Esto hace que

en dos máquinas una misma clase se comporte de distinta forma, invalidando así el concepto de clase. Al hacer que un objeto sea autosuficiente, se evita este problema [11].

2. **Primitivas de comunicación no bloqueantes**: En redes de comunicación, una respuesta puede tardar bastante tiempo, produciendo retrasos inaceptables en la comunicación. En redes móviles, este problema es más frecuente, debido a la movilidad de los nodos, con la posibilidad de quedar temporalmente no accesibles. La solución a este problema, son primitivas de comunicación no bloqueantes, o sea, primitivas que respondan inmediatamente, inclusive si no tienen todavía la respuesta. En el caso que no tenga la respuesta, la comunicarán de alguna forma más tarde.

3. **Reificación de trazas de comunicación**: Como las primitivas de comunicación ya no son bloqueantes, se requiere de un sistema de sincronización. Si se hace explícito los mensajes que han sido recibidos exitosamente y los mensajes a enviar, es posible implementar un sistema de sincronización.

4. **Manejo de recurso de ambientes**: Ya que no hay infraestructura fija en redes móviles ad-hoc, es necesario que se pueda identificar los recursos que se encuentran disponibles en la vecindad del dispositivo de forma dinámica, tanto los servicios que tienen nombre como los anónimos.

1.2. AmbientTalk

AmbientTalk [6, 8] es un lenguaje de programación orientado a ambientes. Como tal cumple con los principios expuestos en la sección anterior. Es así como para no poseer clases, este lenguaje usa objetos basados en prototipos, mientras que las primitivas de comunicación no bloqueantes están basadas en mensajes asíncronos entre objetos. La reificación de las trazas de comunicación se logra a través del sistema de reflexión de AmbientTalk. Finalmente, el manejo de recurso de ambientes se logra usando un sistema de publicación y subscripción de servicios disponibles.

Aparte de poseer todas las características necesarias para ser llamado como tal, además posee clausuras, gramática de primera clase, reflexión, continuaciones y futuros, entre otras cosas. El intérprete oficial está desarrollado en Java SE y Java ME CDC [2].

1.3. Sun SPOT

Los Sun SPOT [20, 24] son dispositivos portátiles que poseen sensores de movimiento, temperatura y luz; 8 LEDs de colores rojo, verde y azul, posibilitando mostrar una gama bastante amplia de colores; dos botones, comunicación inalámbrica y 4 MB de memoria flash interna.

Todas estas características hacen que se pueda usar en aplicaciones bastante interesantes, como poder manejar la luz o temperatura de la pieza en que alguien se encuentra, llevar información médica y comunicarla transparentemente al médico, o inclusive, llevar un registro de la actividad física, entre otros ejemplos.

Los Sun SPOT permiten que sean programados en Java. Para esto, traen incorporado una maquina virtual Java, la Squawk JVM, y una API base, Java ME CLDC 1.1. Esta API es un subconjunto muy pequeño de las funcionalidades de la API estándar de Java. Dos características importantes que le faltan a esta API, para efectos de esta memoria, es reflexión y serialización. Reflexión es necesario para implementar la integración entre AmbientTalk y Java, mientras que la serialización, si estuviera, podría ser usado como reemplazo al parser del lenguaje.

1.4. Motivación

Aunque los Sun SPOTs se programan en Java, realizar cualquiera de estas aplicaciones todavía es una tarea difícil. Manejar las desconexiones y reconexiones puede demandar buena parte del desarrollo de éstas. Como se vio en la Sección 1.1, un lenguaje de programación orientado a ambientes es una buena alternativa para desarrollar estas aplicaciones.

Sin embargo, no es posible ocupar el intérprete oficial de AmbientTalk en un SPOT por dos razones. La primera es que aunque AmbientTalk está desarrollado en Java, ocupa clases y métodos que no están disponible en el SPOT. La segunda razón es que un SPOT no cumple el requisito de memoria necesario para ejecutar este intérprete.

Una característica que falta de Java SE en el Sun SPOT es el sistema de reflexión. Sería posible implementarlo, pero el trabajo escapa del ámbito de esta memoria por su complejidad. Sin reflexión, es imposible implementar la interactividad con Java que posee AmbientTalk. Es por esto que se necesita desarrollar una interfaz para las características del SPOT para

que sean usables dentro de AmbientTalk.

1.5. Objetivos

1.5.1. Objetivo general

Poder interpretar un subconjunto del lenguaje de programación AmbientTalk en una máquina Sun SPOT que sea capaz de interactuar con otros programas desarrollados en AmbientTalk alojados en otros dispositivos que no sean necesariamente un equipo Sun SPOT, y proveer de una interfaz para el acceso a los sensores que posee esta máquina.

1.5.2. Objetivos específicos

1. Entregar un intérprete de AmbientTalk liviano, con respecto al consumo de memoria RAM.
2. Distribuir una aplicación desarrollada en AmbientTalk a los SPOTs de forma simple.
3. Ofrecer un sistema de control de recursos de un SPOT, para así permitir una coexistencia simultánea de varios programas AmbientTalk.
4. Proveer al usuario una interfaz simple de acceso a los sensores.

1.6. Alcance

Para el trabajo de esta memoria, el intérprete debe ser capaz de:

- Soportar todo el ámbito de programación procedural y funcional que posee AmbientTalk.
- Proveer el soporte a objetos y herencia.
- Poseer todos los objetos primitivos básicos de AmbientTalk.
- Comunicarse entre varios Sun SPOTs ocupando el sistema propuesto por AmbientTalk.
- Acceder al uso de los botones y LEDs de los Sun SPOT.

1.7. Esquema del documento

Esta memoria está estructurada de la siguiente forma:

- En la parte **Antecedentes** se explican conceptos y conocimientos esenciales para el entendimiento del resto de la memoria.
  - En el capítulo *AmbientTalk*, se muestra la sintaxis y funcionalidades importantes del lenguaje de programación AmbientTalk.
  - En el capítulo *Sun SPOT* se explica la plataforma de desarrollo Java ME y algunas de las características particulares de esta plataforma con respecto a un Sun SPOT.
- En la parte **Diseño e Implementación** se detalla la arquitectura física y lógica de las aplicaciones que conforman la plataforma.
  - En el capitulo *Diseño General* se detalla la arquitectura física y lógica de la plataforma, además de mostrar la arquitectura lógica de cada aplicación de manera superficial.
  - En el capitulo *Compilador AmbientTalk* se muestra una descripción más profunda de los componentes que conforman el compilador de la plataforma.
  - En el capitulo *Intérprete AmbientTalk* se muestra una descripción más profunda de los componentes que conforman el interprete de la plataforma.
- En la parte **Validación y Conclusión** se valida la memoria y se realiza una conclusión de lo alcanzado en esta.
  - En el capitulo *Validación* se valida la realización de esta memoria mostrando las ventajas de programar en AmbientTalk en vez de Java ME.
  - En el capitulo *Conclusiones* se concluye mostrando lo logrado, lo contribuido y lo que falta por hacer del trabajo realizado en esta memoria.
- En los **Anexos** hay información específica correspondiente a la implementación y los ejemplos de validación.

Parte I

Antecedentes

Capítulo 2

AmbientTalk

En este capítulo se realizará una pincelada al lenguaje de programación AmbientTalk. Se verá que es AmbientTalk, su sintaxis, el modelo de concurrencia usado por AmbientTalk y su sistema de reflexión.

2.1. Introducción

El paradigma de Programación Orientada a Ambientes (AmOP [1]) [7] resuelve a nivel del lenguaje de programación los problemas de la computación distribuida en el contexto de los dispositivos móviles. Esta reconoce que los dispositivos móviles interconectados son diferentes que los sistemas distribuidos tradicionales:

- **Conexiones volátiles**: Los dispositivos móviles se comunican usando tecnología inalámbrica, que es afectado por frecuentes interferencias y limitaciones de rango tal que las fallas son la norma en este tipo de comunicaciones, en vez de excepciones.
- **Cero Infraestructura**: Se espera que los dispositivos móviles operen autónomamente en distintos ambientes. Esos ambientes pueden ofrecer infraestructura para soportar interacciones. Sin embargo, esta infraestructura no siempre está disponible y un modelo de programación debería soportar software que no dependa de alguna infraestructura.

El paradigma AmOP define un número de criterios para los modelos de programación distribuida para resolver las siguientes características:

[1] Del ingles Ambient-Oriented Programming

- **Desacoplamiento temporal**: implica que no es necesario que ambos participantes de la comunicación estén conectados al mismo tiempo. Este criterio soporta la comunicación entre dispositivos móviles que están temporalmente incomunicados debido a la naturaleza volátil de la comunicación.

- **Desacoplamiento de sincronización**: implica que el flujo de ejecución de los participantes no se bloquea esperando recibir o enviar un mensaje. Este criterio asegura la disponibilidad de los recursos compartidos, que pueden estar bloqueados en un hilo de control, que no depende de la disponibilidad de otros recursos en la red.

- **Desacoplamiento espacial**: implica que los participantes de la comunicación no necesitan conocer la dirección del otro antes de iniciarse la comunicación. Este criterio es requerido para soportar comunicación que no depende de la infraestructura.

Para poder soportar estos criterios, ha sido desarrollado un lenguaje de programación distribuida llamado AmbientTalk [6]. AmbientTalk es un lenguaje de programación orientado a objetos distribuido diseñado específicamente para reunir objetos de servicio en redes móviles ad-hoc. AmbientTalk hereda la mayoría de sus características de los lenguajes Scheme [21], Self [22] y Smalltalk [10]. De Scheme, hereda la noción de clausuras[2] con alcance léxico. De Self y Smalltalk, hereda la expresiva sintaxis de clausura de bloque, la representación de clausuras como objetos y el uso de clausuras de bloques para definir las estructuras de control. El modelo de objetos de AmbientTalk también es derivado de Self: sin clases, objetos basados en slots usando delegación como mecanismo de reúso de clases. El lenguaje también soporta reflexión. En lo restante de este capítulo se dará un resumen de la sintaxis y modelo de objetos de AmbientTalk.

2.2. Objetos en AmbientTalk

AmbientTalk es un completo lenguaje orientado a objetos con soporte de tipos dinámicos. Su modelo de objetos está basando en el modelo de prototipos de Self [22]. Para explicar cómo los objetos funcionan en AmbientTalk, consideremos la definición de un objeto Printer realizado en el código 2.1.

[2]Una clausura es una función sin nombre que captura el scope léxico

Código 2.1: Definición de un objeto Printer en AmbientTalk

```
1   def Printer := object : {
2       def dpi;
3       def queue;
4       def init(dpi){
5           self.dpi := dpi;
6           self.queue := Queue.new(10);
7       };
8       def addJob(aJob){ queue.add(aJob); };
9       def getQueueSize() { queue.length(); };
10      def print() {
11          queue.foreach: {|doc|
12              doc.print();
13              queue.remove(doc);
14          };
15      };
16  };
```

En el extracto del Código 2.1, se define un objeto ex-nihilo [12] `Printer` con el constructo `object:`. Este objeto tiene dos campos: la resolución de la impresora `dpi` y una cola interna para almacenar los trabajos entrantes. El método `init` es usado para inicializar nuevos objetos cuando el método `new` es invocado. En vez de crear una nueva instancia como lo hacen los lenguajes orientados a objetos que usan clases, el mensaje `new` retorna un clon del objeto que lo recibe, antes llamando al método `init` para inicializar el objeto a valores coherentes para todos los campos. Hay tres métodos definidos en el objeto `Printer` para manipular el estado interno del mismo: `addJob`, `getQueueSize` y `print`. El método `print` ilustra el uso de las clausuras y keywords. El objeto `queue` tiene un método `foreach:` que toma una clausura como su argumento. Una clausura es sintácticamente creada con `{ |arg1 ... argN| exp1; ... expN }`. A diferencia del resto de métodos, el método `foreach:` no está en forma canónica. En cambio, éste está basado en keywords, que fue por primera vez introducido por SmallTalk [10]. Los keywords se reconocen por los dos puntos que siguen al identificador y pueden tomar varios argumentos. Por ejemplo, una iteración esta expresada por `1.to: 10 do: { |i| system.println(i); }`, donde el keyword `to:do:` representa una función que toma dos argumentos, un entero y una clausura. En el Código 2.1 la clausura es usada en el método `print` para iterar (usando `foreach:`) en la cola de la impresora. La clausura posee un argumento `doc`, que imprime el documento y lo remueve de la cola. En otras palabras, el método `print` itera sobre los documentos de la cola, imprimiéndolos y luego removiéndolos.

2.3. Programación concurrente

El modelo de concurrencia de AmbientTalk está basado en el modelo de ciclos de eventos de comunicación del lenguaje de programación E [14], que es una extensión del modelo de actores [1]. El modelo de E combina actores y objetos en un modelo de concurrencia unificado. A diferencia de otros lenguajes que poseen actores como Act1 [13], ABCL [25] y Actalk [4], un actor es un contenedor de objetos regulares, en la cual cada objeto individualmente se le pueden enviar mensajes asíncronos. Así, a diferencia de los lenguajes con actores tradicionales en la cual solo los actores pueden ser capaces de recibir mensajes, ahora objetos regulares pueden recibir mensajes. El actor es solo un contenedor, responsable de ejecutar esos mensajes uno a la vez. Para poder comprender cómo se comportan los actores de AmbientTalk, primero necesitamos describir las propiedades fundamentales del modelo de concurrencia de ciclos de eventos.

Concurrencia en ciclos de eventos. El ciclo de eventos de comunicación del lenguaje E es un modelo de concurrencia basado en eventos. En este modelo, el ciclo de eventos es un hilo que permanentemente procesa eventos de su cola de eventos, en la cual invoca al manejador del evento correspondiente. Además, un ciclo de eventos puede forzar las siguientes tres propiedades de control de concurrencia:

- **Ejecución secuencial**: Un ciclo de eventos procesa eventos entrantes en su cola de eventos uno por uno, en un orden estrictamente secuencial.
- **Comunicación no bloqueante**: Un ciclo de eventos nunca suspende su ejecución para esperar a otro ciclo de eventos para finalizar su procesamiento. En vez de eso, todas las comunicaciones entre ciclos de eventos suceden en forma de notificaciones asíncronas de eventos.
- **Acceso exclusivo al estado**: Los manejadores de eventos y sus estados asociados pertenecen a un solo ciclo de eventos. En otras palabras, un ciclo de eventos tiene acceso exclusivo a su estado mutable.

Actores. En AmbientTalk los ciclos de eventos concurrentes son creados usando actores. Los actores representan una cola de eventos usando una cola de mensajes. Esto significa que los eventos son representados como mensajes, las notificaciones de eventos como mensajes

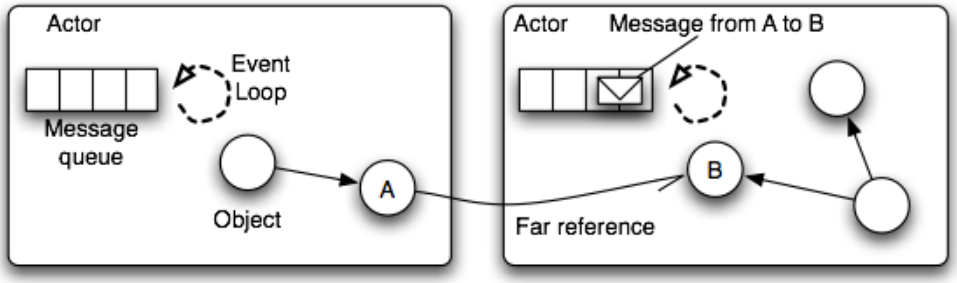


Figura 2.1: Actores de AmbientTalk como ciclos de eventos comunicantes

asíncronos y los manejadores de eventos son representados por métodos o clausuras. El ciclo de eventos del hilo de un actor está permanentemente despachando los mensajes de la cola de mensajes hacia el correspondiente método en el objeto receptor del mensaje. Mensajes en una cola de eventos son procesados secuencialmente para evitar data races en el estado compartido por los objetos. Por defecto, todos los objetos que son creados pertenecen al actor de la máquina virtual. Otros actores pueden ser creados también como es mostrado en el Código 2.2.

Código 2.2: Creación de actores en AmbientTalk

```
1        def anActor := actor: { |arg1, arg2, ... , argN|
2                ...
3        };
```

En el Código 2.2 un nuevo actor `anActor` es definido con el constructo `actor:`. Un actor es aislado de su scope léxico para evitar compartir estados con el actor que lo creo. De todas formas, se puede especificar un conjunto de argumentos `|arg1, arg2, ... , argN|` que serán copiados al scope léxico del recién creado actor.

Comunicación. Cada objeto pertenece a un y solo un actor. Si los objetos deben pertenecer a diferentes actores, la comunicación entre ellos debe ser de forma asíncrona por medio de una referencia lejana. Una referencia lejana es un proxy de un objeto que le pertenece a otro actor. La Figura 2.1 ilustra a los actores de AmbientTalk como ciclos de eventos comunicándose. Las líneas punteadas representan los hilos del ciclo de eventos de los actores en la que están tomando mensajes de la cola de mensajes del actor y ejecutando de forma sincrónica el correspondiente método de uno de los objetos pertenecientes al actor. Si dos objetos pertenecen al mismo actor, ellos pueden comunicarse tanto asíncronamente o usando un envió de mensaje secuencial.

Un mensaje asíncrono es encolado en la cola de mensajes del actor que actúa como contenedor del objeto destino. AmbientTalk distingue entre un envío de mensaje secuencial usando el operador punto `(obj.m());` y envío de mensajes asíncronos usando el operador flecha `(obj<-m());`. Los mensajes asíncronos pueden devolver futuros. Un futuro es un reemplazo para el resultado real de una llamada asíncrona.

Código 2.3: Llamadas a métodos asíncronos en AmbientTalk

```
1        def printerQueueSizeFut := printer<-getQueueSize();
```

La variable `printerQueueSizeFut` está ligada a un futuro (Código 2.3), en este caso el reemplazo para el valor de la cola de la impresora. Después que la impresora ha finalmente procesado el mensaje `getQueueSize`, el futuro es resuelto con el valor retornado por la invocación de `getQueueSize`(Código 2.4).

Código 2.4: Resolución de futuros en AmbientTalk

```
1        when: printerQueueSizeFut becomes: { |queueSize|
2                system.print( "El largo de la cola es: " + queueSize ) ;
3        }
```

La función `when:becomes:` toma un futuro y una clausura como sus argumentos, y registra a la clausura como observador del futuro. Cuando el futuro es resuelto con un valor, la clausura es ejecutada con el valor resuelto del futuro como su parámetro.

2.4. Programación distribuida

En la sección anterior se discutió el modelo de concurrencia de AmbientTalk. Cada actor encapsula en un hilo una colección de objetos que son serialmente accesibles a través de envíos de mensajes asíncronos. Desde un punto de vista de programación distribuida, los lenguajes de actores tradicionales también usan actores como unidad de distribución y éste también es el caso en AmbientTalk. Sin embargo, en contraste con los modelos de actores tradicionales, el modelo de AmbientTalk hace posible que se puedan referenciar remotamente objetos regulares usando referencias lejanas como canal de comunicación. Esto significa que, debido a que las referencias lejanas solo soportan invocaciones asíncronas a métodos, toda forma de comunicación remota es asíncrona también. De aquí, que el concepto de referencias

lejanas es usado tanto para computaciones locales como para computaciones concurrentes distribuidas remotamente. AmbientTalk también provee de abstracciones de programación para enfrentarse con fallas parciales.[CMB+07].

2.5. Reflexión

Reflexión es el proceso en el cual un programa computacional puede observarse y modificar su propia estructura y comportamiento. Reflexión es un tipo particular de metaprogramación. Reflexión es una parte integral del lenguaje de programación AmbientTalk. Usando reflexión, el núcleo del lenguaje puede ser extendido con soporte de programación como también nuevos constructos del lenguaje. Ambos requieren distintos tipos de acceso reflexivo.

AmbientTalk soporta mirrors que recuerdan a los usados en Self y Strongtalk, en la cual pueden ser usados para realizar introspección y automodificación [3]. AmbientTalk llama a tales mirrors, mirrors explícitos. La novedad de la arquitectura de metanivel de AmbientTalk son sus mirrors implícitos, que pueden ser usados adicionalmente como intermediarios.

2.6. Resumen

En este capítulo se realizó una pincelada al lenguaje de programación AmbientTalk. Se vio que es AmbientTalk, su sintaxis, el modelo de concurrencia usado por AmbientTalk y su sistema de reflexión.

Capítulo 3

Sun SPOT

En este capítulo se mostrará la plataforma de desarrollo para un Sun SPOT, Java ME. Se verá que es Java ME, la configuración y perfil usados en un SPOT, algunas URI especiales de un SPOT y cómo usar la API de sensores de un SPOT.

3.1. Introducción

Un Sun SPOT[1] es un dispositivo pequeño, de dimensiones 41 x 23 x 70 mm y que pesa 54 gramos [20,24]. Posee un procesador ARM de 180 MHz, una memoria RAM de 512 KiB y una memoria Flash para almacenamiento permanente de 4 MiB. Posee una radio que cumple la especificación IEEE 802.15.4 que permite que se pueda conectar a una red ZigBee. Sin embargo, un SPOT no soporta este estándar, aunque si se desea, se puede desarrollar el stack ZigBee faltante. Los SPOTs además posee sensores de luz, temperatura y de aceleración.

Por las características de procesamiento y tamaño, un SPOT entra en la categoría de "mote". Sin embargo, a diferencia de la mayoría de estos dispositivos en la cual se programa en C, en un SPOT se programa en Java ME.

3.2. Java ME

Java Micro Edition (Java ME) [9] es una plataforma Java diseñada para ejecutarse en dispositivos móviles y sistemas embebidos. Esta plataforma provee solo un subconjunto de la

[1]Del ingles Small Programmable Object Technology

API de Java Standard Edition (Java SE)[2], debido a que estos dispositivos tienen una capacidad de cómputo limitada. Java ME posee los conceptos de configuración y perfil producto a que la capacidad de cómputo de estos dispositivos es también muy diversa.

Una configuración es la base de la plataforma Java ME, ya que especifica que versión de Java SE se basa y cuáles son sus diferencias con respecto a ésta. Estas diferencias son tanto a nivel de la JVM[3], como del lenguaje Java. Además, especifica qué parte de la API base de Java SE es usada, agregando APIs adicionales obligatorias de la configuración.

Un perfil extiende una configuración, añadiendo tanto APIs obligatorias, como restricciones extras a la de la configuración. Los dispositivos están obligados a implementar un perfil.

Los Sun SPOTs usan la configuración CLDC 1.1 y el perfil IMP 1.0, que se explican a continuación.

3.2.1. Connected Limited Device Configuration (CLDC)

La configuración CLDC [19] es una de las dos configuraciones disponibles actualmente en Java ME[4]. Esta configuración es usada principalmente en celulares.

La versión 1.1 de esta configuración usa la versión 1.3 de Java SE y establece las siguientes tipos de clases disponibles para un usuario:

- **Clases fundamentales**: Son clases que están íntimamente ligadas a la maquina virtual o son usadas por el compilador. Ej: `java.lang.Object`, `java.lang.String`, `java.lang.Thread`, `java.lang.StringBuffer`
- **Clases de tipos primitivos**: Son clases usadas como envoltorio para datos primitivos. Ej: `java.lang.Integer`, `java.lang.Float`
- **Colecciones**: Permite almacenar objetos, para luego ser recuperados. Son solo 3 clases: `java.util.Hashtable`, `java.util.Vector` y `java.util.Stack`
- **Entrada y salida**: Permite leer y escribir datos. Ej: `java.io.InputStream`, `java.io.Writer`, `java.io.DataInputStream`, `java.io.ByteArrayOutputStream`

[2]Esta es la edición usada para aplicaciones de escritorio
[3]Java Virtual Machine
[4]La otra configuración es la Connected Device Configuration (CDC)

- **Fecha y hora**: Permite el manejo de fecha y horas. Son solo 3 clases: `java.util.Date`, `java.util.Calendar`, `java.util.TimeZone`
- **Excepciones y errores**: Lanzados por las librerías o la JVM para señalar que ha ocurrido una excepción. Ej: `java.lang.NullPointerException`, `java.lang.OutOfMemoryError`
- **Referencias débiles**: Permite tener una referencia a un objeto, pero que es ignorada por el recolector de basura para establecer si un objeto es alcanzable o no. Son 2 clases: `java.lang.ref.Reference` y `java.lang.ref.WeakReference`
- **Utilitarios extras**: Permite realizar cálculos y obtener valores pseudoaleatorios. Son solo 2 clases: `java.util.Math`, `java.util.Random`
- **Generic Connection Framework (GCF)** : Permite realizar conexiones con equipos remotos. Ej: `javax.microedition.io.Connector`

Las clases antes mencionadas no necesariamente implementan todos los métodos disponibles en Java SE. Por ejemplo, `Object` no posee el método protegido `clone()` o `finalize()`, `Class` no tiene ningún método para consultar sobre la estructura de la clase que representa o `System` no tiene la variable estática `in`, que corresponde a la entrada estándar.

El GCF es un framework que permite conectarse a equipos remotos(o locales) usando una URI[5]. Un ejemplo es el Código 3.1 que se conecta a un equipo vía socket.

Código 3.1: Abriendo y cerrando conexiones usando GCF

```
1  String uri="socket://129.144.111.222:2800";
2  StreamConnection sc=(StreamConnection) Connector.open(uri);
3  InputStream is=sc.openInputStream();
4  OutputStream os=sc.openOutputStream();
5  //Aquí se envía los datos
6  is.close(); os.close(); sc.close();
```

CLDC especifica 6 subtipos de conexiones, de los cuales 5 están para manejos de stream, tanto unidireccionales como bidireccionales, y una para transmisión de paquetes. Sin embargo, CLDC no establece ningún protocolo que esté disponible para todos los dispositivos CLDC.

[5]Uniform Resource Identifier

3.2.2. Information Module Profile (IMP)

IMP [16] es un perfil basado en MIDP[6] 1.0 [18], el perfil usado por la gran mayoría de celulares. Aunque hay muchos celulares con MIDP 2.0 o 2.1, estas versiones son también compatibles con aplicaciones desarrolladas en MIDP 1.0. IMP está diseñado para dispositivos móviles que no poseen una interfaz gráfica o ésta es muy primitiva para MIDP. En resumen, IMP 1.0 es MIDP 1.0 pero sin las clases de interfaz gráfica. IMP extiende CLDC con las siguientes funcionalidades:

- **Timers**: Agrega la posibilidad de programar tareas. Corresponde a las clases `java.util.Timer` y `java.util.TimerTask` de Java SE
- **Protocolo HTTP en GCF**: Permite usar el protocolo `http` en el framework GCF.
- **Record Management System (RMS)**: Framework que permite almacenar persistentemente datos en el dispositivo.
- **MIDlets**[7]: Permite manejar el ciclo de vida de la aplicación. Es obligatorio su uso en un dispositivo que usa este perfil.

Un MIDlet es una clase controlada por el contenedor de MIDlets y es un reemplazo a la tradicional forma de ejecución de aplicaciones Java[8]. Esto es debido a que los dispositivos móviles pueden recibir un evento que obliga a detener la aplicación por un tiempo, debido al escaso poder de cómputo. Un ejemplo es cuando un celular recibe una llamada, en la cual esta tiene prioridad versus la aplicación.

Un MIDlet puede estar en tres estados de ejecución: iniciado, pausado o destruido. El contenedor puede avisarle al MIDlet que va a cambiarlo de estado, llamando a los métodos `startApp()`, `pauseApp()` y `destroyApp()` del MIDlet respectivamente. También el MIDlet puede avisarle al contenedor que quiere cambiar de estado, aunque no es necesario que el contenedor responda la petición.

RMS es la API de persistencia de Java ME. Esta API se centra en RecordStores y Records. Un Record es un arreglo de bytes que es identificado por un entero dentro de un RecordStore.

[6] Mobile Information Device Profile

[7] En IMP se llaman IMlets, pero salvo por como lo llama la especificación, son idénticos en todo sentido con los MIDlets de MIDP así que se ha mantenido este último nombre en esta memoria

[8] El método de clase público main, que es mantenido en CLDC

Un RecordStore mantiene un conjunto de Records en común y está identificado por un nombre. Haciendo analogías con una base de datos relacional, un RecordStore sería una tabla, un Record sería una fila de esa tabla y el identificador de ese Record sería la llave primaria de esa fila. Los RecordStore sólo pueden ser vistos por el MIDlet que los creó, así pueden coexistir sin problemas dos RecordStore que tienen el mismo nombre pero creados en distintos MIDlet.

El Código 3.2 añade a un log un nuevo error mientras que el Código 3.3 imprime los errores anotados en el log[9].

Código 3.2: Agregando un Record

```
1  RecordStore rstore=RecordStore.openRecordStore("log",true);
2  String errorInfo="Acceso no autorizado: "+(new Date());
3  byte[] bytes=errorInfo.getBytes();
4  rstore.addRecord(bytes,0,bytes.length);
5  rstore.closeRecordStore();
```

Código 3.3: Imprimiendo los Records en un RecordStore

```
1  RecordStore rstore=RecordStore.openRecordStore("log",true);
2  RecordEnumeration renum=rstore.enumerateRecords(null,null,false);
3  while(renum.hasNextElement()){
4    String error=new String(renum.nextRecord());
5    out.println(error);
6  }
7  renum.destroy();
8  rstore.closeRecordStore();
```

3.3. Squawk y los Sun SPOT

Squawk [17] es la JVM usada por los SPOT. A diferencia de la gran mayoría de JVM, esta JVM tiene un micro núcleo desarrollado en C, mientras que el resto está hecho en Java. Otra diferencia es el hecho que Squawk es prácticamente el sistema operativo de los SPOTs, teniendo que manejar toda la memoria, el scheduling de procesos o la entrada y salida de datos. Para esta última misión tiene una amplia API de entrada y salida de bajo nivel.

Los SPOTs incluyen una gran variedad de protocolos usables en GCF. La tabla 3.1 los ilustra. Los protocolos radiogram y socket son los únicos que permite esperar una conexión

[9]Por simplicidad en el código, el orden de los elementos no es determinado

URI	Descripción
`memory://`*direccionDecimal*	Permite leer una ubicación arbitraria de la memoria
`multicast:`*URI1;...;URIn*	Escribe a varias salidas a la vez
`radiogram://`[*direccion*][`:`*puerto*]	Envía y recibe paquetes a un SPOT
`radiostream://`*IEEEAddress*`:`*puerto*	Conexión fluida bidireccional entre SPOTs
`serial://`	Envía y recibe datos a través del puerto USB
`socket://`*IP*[`:`*puerto*]	Simula una conexión IP a un equipo. Necesita un socket-proxy

Cuadro 3.1: Protocolos GCF en un SPOT

anónima, o enviar paquetes a todos (broadcast). Todas las conexiones salvo conexiones con envíos broadcast, se realizan de forma confiable.

3.4. API de sensores y de E/S

Los SPOTs vienen con un set de sensores por defecto[10], entre ellos un acelerómetro, un sensor de luz y de temperatura. Además, incluye varios pines de entrada y salida, tanto analógicos como digitales, 8 LEDs RGB[11] y 2 botones.

La API de los SPOTs tiene agrupado a todos estos dispositivos en la clase `EDemoBoard`. Así, por ejemplo, para obtener el sensor de luz, se realiza usando `EDemoBoard.getInstance().getADCInstance()`.

Cada uno de los sensores (luz, temperatura y aceleración) permite obtener el valor que está registrando[12] y registrar observadores (Listeners en Java). El sensor posee un valor límite modificable tanto superior como inferior. Cuando el valor registrado sale del rango especificado, todos los observadores son notificados de este evento. Debido a que el acelerómetro vale por 3 sensores, uno por cada eje cartesiano, se puede registrar un observador independiente en cada eje.

[10]Los SPOTs permiten que el sector adonde estén los sensores sea desacoplado para añadir otra tarjeta. En esta memoria supondremos que esta tarjeta es la que venía con el Sun SPOT

[11]Cada uno de estos "LED" está formado de un LED rojo, otro verde y uno azul. Aunque a simple vista se note el truco, por la cercanía entre ellos permite mostrar cualquier color, pero con algunas distorsiones

[12]La temperatura se puede obtener tanto en Celsius como en Fahrenheit, mientras que la aceleración esta medido en Gs. El sensor de luz no especifica en que unidad se obtiene.

3.5. Resumen

En este capítulo se mostró la plataforma de desarrollo para un Sun SPOT, Java ME. Se vio que es Java ME, la configuración y perfil usados en un SPOT, algunas URI especiales de un SPOT y como se debe usar la API de sensores de un SPOT.

Parte II

Diseño e Implementación

Capítulo 4

Diseño General

En este capítulo se explicará el diseño de la plataforma de software a desarrollar en esta memoria. Primero se verán las decisiones de diseño que afectan a la plataforma en general. Luego, se revisará la arquitectura física de esta plataforma. Finalmente, se revisará la arquitectura lógica de alto nivel tanto de la plataforma, como de las aplicaciones que la conforman.

4.1. Introducción

El objetivo general de esta memoria es poder interpretar un subconjunto del lenguaje AmbientTalk en un SPOT. Un lenguaje interpretado, en general, posee una fase de parseo del código fuente, que lo transforma a un AST, y una fase de interpretación de ese AST.

Una dificultad al pensar en implementar un intérprete en Java ME CLDC es que no es posible usar directamente las herramientas que se ocupan generalmente para la fase de parseo, conocidas como "compiler compiler", debido a que faltan clases o métodos que sí están disponibles en Java SE. La solución planteada aquí es simplificar el código fuente a interpretar en el SPOT, aprovechando el hecho que todo programa AmbientTalk debe provenir originalmente de otro equipo que no es un SPOT.

Por esta razón, es que en esta memoria se ha desarrollado dos aplicaciones complementarias entre sí: un compilador que simplifique el código fuente, y un intérprete que ejecute el código fuente simplificado.

4.2. Decisiones de diseño

Hay decisiones que afectan al diseño arquitectónico en si de la plataforma. Estas son:

DECISIÓN I **Separar la plataforma en dos aplicaciones:** El problema de ejecutar un programa AmbientTalk en un SPOT se puede abordar de 3 maneras. Estas son:

- **Interprete:** Cargar en el SPOT un programa Java ME capaz de leer un código fuente AmbientTalk y ejecutarlo sin necesidad de otro paso.
- **Compilador:** En un computador host se transforma el código fuente AmbientTalk a clases Java compatibles con Java ME y son estas clases las que se cargan al SPOT, sin que este sepa que se está ejecutando un programa AmbientTalk.
- **Híbrido:** Cargar en el SPOT un programa Java ME capaz de leer un archivo intermedio y ejecutarlo. Este archivo es generado por un programa en un computador host usando código fuente AmbientTalk.

De las tres alternativas, el intérprete es la única alternativa que no es viable para esta memoria, debido a la falta de herramientas de parseo en Java ME de archivos fuentes. Sin estas herramientas, es difícil poder implementar este intérprete en el tiempo que dura esta memoria.

De las otras dos opciones, se eligió la opción hibrida debido a que el lenguaje a interpretar presenta grandes diferencias con respecto a Java. Una de estas, es que Java es un lenguaje de programación orientado a objetos basado en clases fuertemente tipado, mientras que AmbientTalk está orientado a objetos prototipados sin tipos. Con la falta de reflexión, al final una solución tipo compilador terminaría siendo un intérprete embebido en el código.

DECISIÓN II **Usar AmbientTalk Binary XML(ABX) como formato de archivo de intercambio:** Para el archivo de intercambio, una característica fundamental es que debe ser fácil de parsear y de ejecutar por el intérprete. Los bytecodes cumplen la característica de ser fácilmente parseables, ya que no es necesario observar más caracteres para saber qué hacer con el que se está leyendo ahora. La JVM posee un bytecode propio y sería una buena alternativa como bytecode de intercambio si es que

la JVM lo ejecutara sin necesidad de un interpretador. Sin embargo, Java ME CLDC prohíbe introducir código en tiempo de ejecución de forma programática, por lo que no está disponible esa opción.

Al final se eligió un formato de archivo de intercambio que permite conservar la estructura del AST, característica que permite simplificar una posible implementación de gramáticas de primer orden, ya que sería el mismo archivo el que diría la estructura de los nodos.

El formato de archivo ABX[1] es un formato parecido a un XML tradicional, salvo que los tags del XML se representan con números en vez de texto, y se codifica en forma binaria. Este formato posee tanto la característica de mantener la estructura del AST, ya que un archivo XML permite representar sin pérdida de información la estructura de un árbol, que es la que posee un AST, como la de ser fácilmente parseable ya que el formato es un tipo de bytecode. En la Subseccion 4.4.1 se ve en más detalle en qué consiste este formato.

Decisión III **Minimización de cantidad de opcodes :** Se ha tratado de minimizar los opcodes (tags de un archivo ABX) necesarios. Esto es para reducir el código necesario para implementar el intérprete. Con esto se logra reducir el tiempo necesario de codificación y disminuir la cantidad de errores. La lista de opcodes se detalla en el Anexo A.

4.3. Arquitectura física

Hay dos escenarios físicos a las cuales la plataforma se enfrentará en momentos distintos: al cargar un programa AmbientTalk en un SPOT y cuando se ejecuta este programa.

4.3.1. Carga del programa

Cuando se carga un programa, el equipo host compila el programa AmbientTalk a un archivo ABX usando el compilador. Luego, usando el mismo compilador, se envía el archivo a un SPOT usando el puerto USB de éste, usando comunicación serial, al intérprete, que lo

[1] ABX es un formato de archivo creado para esta memoria.

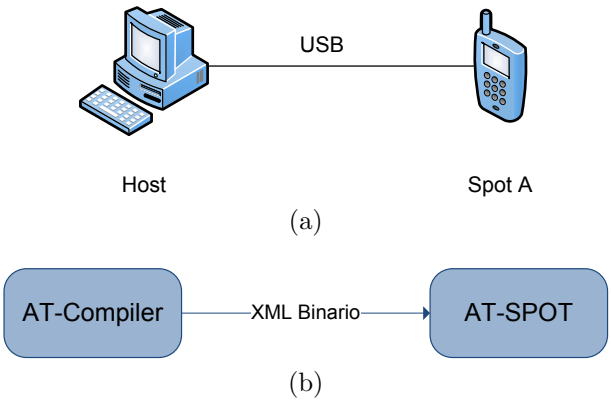


Figura 4.1: Arquitectura física y lógica cuando se carga un programa AmbientTalk

almacena en la memoria flash del dispositivo. En la Figura 4.1a se muestra la arquitectura física cuando se está realizando este proceso.

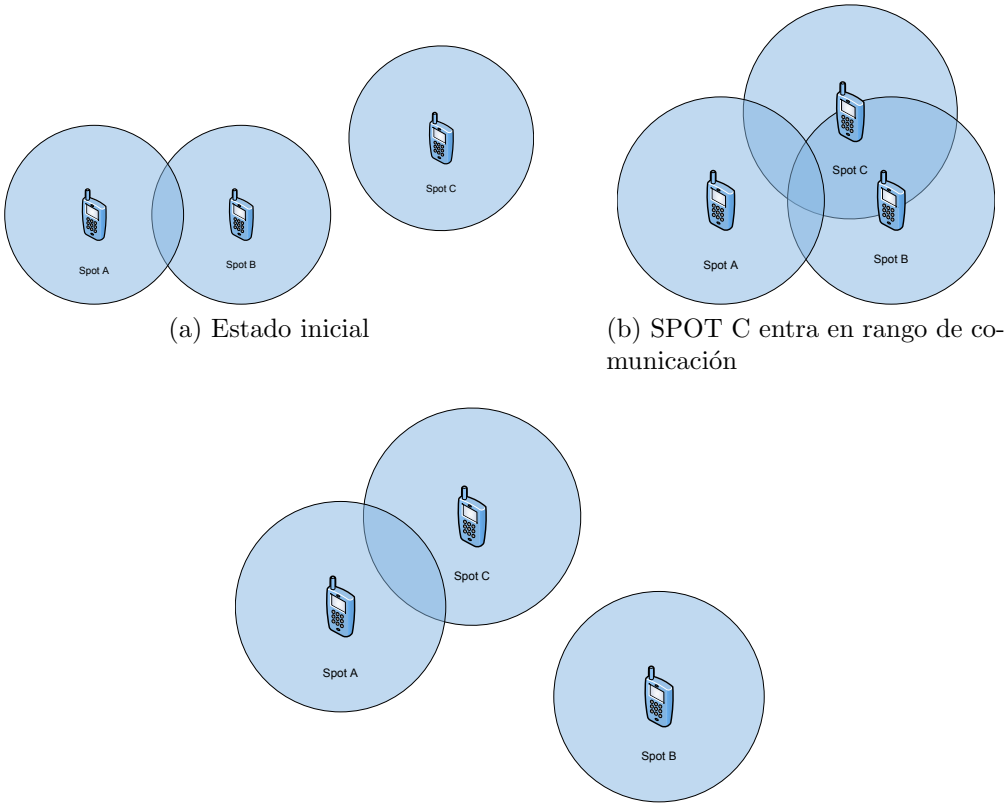
4.3.2. Ejecución del programa

En la Figura 4.2a se muestra a tres SPOTs, dos de los cuales, A y B, están en rango de comunicación. Como es mostrado en la Figura 4.2b, es posible que en el medio de la ejecución del programa AmbientTalk, un nuevo SPOT(SPOT C) entre en el rango de comunicación de A y B. En ese caso, el programa AmbientTalk en A debe ver a este nuevo SPOT. También es posible que un SPOT salga del rango de comunicación, como el SPOT B mostrado en 4.2c, ante lo cual el programa AmbientTalk debe ser tolerante a esta desconexión, y en caso de que B entre nuevamente en rango, enviarle lo que no se pudo enviar en el tiempo de desconexión.

4.4. Arquitectura lógica

La plataforma se separa en dos aplicaciones: ATCompiler y ATSpot, por los motivos explicados en Decisión 1. ATCompiler es el encargado de simplificar el código fuente, transformándolo en un archivo ABX, para luego enviarlo a un SPOT. ATSpot es el encargado de interpretar este ABX en un SPOT. La Figura 4.1b muestra la relación entre ATCompiler y ATSpot.

A continuación, se detallará tanto la arquitectura lógica de cada aplicación como el formato ABX usado como intercambio entre ambos.



(a) Estado inicial

(b) SPOT C entra en rango de comunicación

(c) SPOT B sale del rango de comunicación

Figura 4.2: Arquitectura física ejecución AmbientTalk

4.4.1. AmbientTalk Binary XML

AmbientTalk Binary XML(ABX) es una representación binaria de un archivo XML, en el cual los tags de inicio y fin son representados por un byte. Los tags de inicio son siempre positivos, mientras que los de fin son siempre negativos. El tag de fin de un elemento siempre cumple que es el inverso aditivo del tag de inicio. Existen 5 tipos de tags: vacío, entero, real, cadena y contenedor. Solo el elemento vacío no tiene tag de fin, ya que representa a un elemento que siempre es vacío. En el Código 4.2 se muestra el archivo ABX que le correspondería al archivo XML mostrado en el Código 4.1

Representar un ejecutable AmbientTalk usando el formato de archivo ABX posee los siguientes beneficios:

- Permite representar un AST de manera íntegra, debido a que un documento XML representa naturalmente un árbol.
- Señala el inicio y fin de un nodo de manera explícita, permitiendo guardar o saltarse

Código 4.1: Archivo XML de listas de personas

```
<personas>
        <persona>
                <nombre>Juan Perez</nombre>
                <edad>24</edad>
        </persona>
        <persona>
                <nombre>Pedro Gonzalez</nombre>
                <edad>17</edad>
        </persona>
        <persona>
                <edad>37</edad>
                <nombre>Maria Alarcon</nombre>
        </persona>
</personas>
```

Código 4.2: Archivo ABX de listas de personas

```
1
        2
                3 ''Juan Perez'' -3
                4 24 -4
        -2
        2
                3 ''Pedro Gonzalez'' -3
                4 17 -4
        -2
        2
                4 37 -4
                3 ''Maria Alarcon'' -3
        -2
-1
```

un nodo arbitrariamente. Ambas marcas permiten saber cuando un nodo del mismo tipo existe dentro de él, para así no determinar el fin del nodo prematuramente.

- Se puede al menos parsear y ejecutar con un gasto de 20 bytes de memoria RAM extra por archivo ABX. Este es el consumo teórico de memoria que realiza la implementación del parser de archivos ABX usado por ATSpot.
- La ejecución de un nodo de un ejecutable AmbientTalk no necesita saber de los nodos siguientes o anteriores, por lo que no se requiere que esté todo el código en memoria

4.4.2. ATCompiler

La misión de ATCompiler es transformar el código fuente AmbientTalk en un archivo ABX y enviarlo a un SPOT. La Figura 4.3 muestra el flujo de datos del compilador desde el código fuente hasta que se almacena en un SPOT

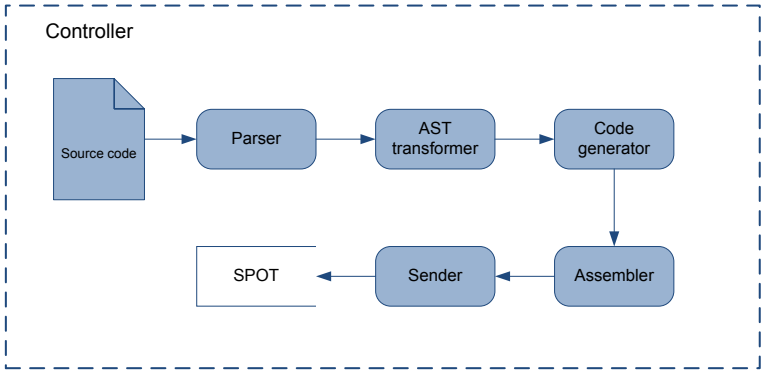


Figura 4.3: Arquitectura Lógica Compilador

Ahora se va a explicar la función de cada paso del proceso señalado en la Figura 4.3.

1. **Parser:** es el encargado de leer el código fuente y transformarlo en un AST. Para esta memoria se ocupó el parser del intérprete oficial.

2. **AST transformer:** transforma el AST recibido del parser (AST interprete) a un AST usable por el compilador que permita generar código. Este paso es necesario debido a que el código del parser y del AST que entrega éste es externo al programa.

3. **Code generator:** genera un documento XML intermedio que representa un código ejecutable AmbientTalk. Este documento XML intermedio es generado, como es explicado en Decision 5, para ser usado tanto para evitar bugs del compilador, como para realizar debugueo al código generado por el compilador.

4. **Assembler:** transforma el documento XML intermedio a un archivo ABX.

5. **Sender:** envía el archivo ABX a un SPOT.

6. **Controller:** encargado de manejar el proceso de compilación.

4.4.3. ATSpot

La misión de ATSpot es recibir y ejecutar el código AmbientTalk en formato ABX recibido del compilador, y proveer la infraestructura y librerías AmbientTalk necesarias para poder realizar esta labor. La Figura 4.4 muestra los módulos de ATSpot y las dependencias entre módulos entre éstos, con una flecha señalando que un módulo depende de otro.

A continuación, se explica cada sistema y los módulos asociados del intérprete de AmbientTalk en un SPOT.

Core: Provee lo necesario para la ejecución de un programa AmbientTalk. Esto consiste en abrir un archivo ABX de una memoria no volátil, parsearlo e interpretarlo de una manera correcta y completa. Los módulos que integran este sistema son:

- **Core structs:** Provee definiciones y objetos fundamentales al intérprete AmbientTalk, incluida una parte del intérprete mismo

- **SBX:** Permite leer un ejecutable AmbientTalk en formato ABX, e interpretarlo.

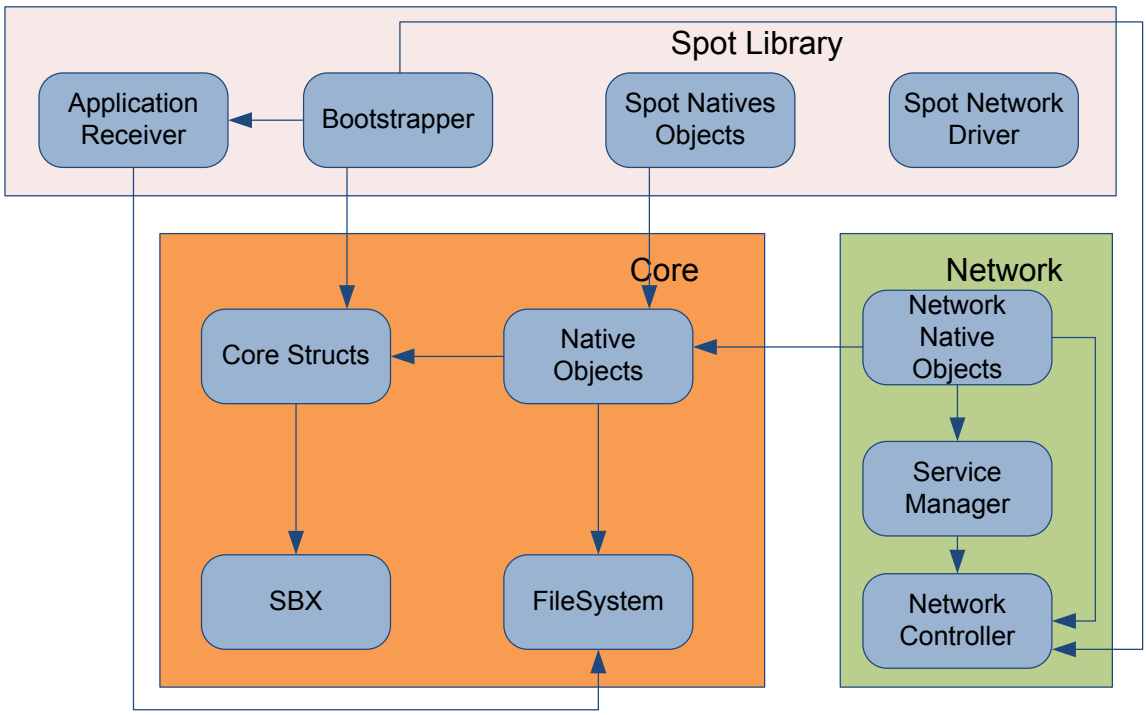


Figura 4.4: Arquitectura Lógica Interprete

- **Filesystem:** Provee de un sistema de archivos a un dispositivo con Java ME CLDC usando RecordStores. Con esto, provee una implementación de java.io.File, java.io.FileInputStream y java.io.FileOutputStream.

- **Native objects:** Implementación de los objetos básicos de AmbientTalk.

Network: Provee el soporte de red que brinda AmbientTalk. Permite usar referencia lejanas remotas, y el servicio de descubrimiento y uso de servicios AmbientTalk. Los módulos que integran este sistema son:

- **Network controller:** Maneja las referencias lejanas remotas, tanto de objetos locales como remotos

- **Service manager:** Maneja los servicios AmbientTalk proveídos por este SPOT y requeridos por un programa AmbientTalk.

- **Network native objects:** Implementación de los objetos de red visibles dentro de AmbientTalk.

SPOT libraries: Provee los objetos AmbientTalk y Java que corresponden a la plataforma Sun SPOT. Los módulos que integran este sistema son:

- **Application receiver:** Permite recibir un ejecutable AmbientTalk a través del puerto USB de un SPOT
- **Bootstrapper:** Inicia y configura el intérprete de AmbientTalk al momento de encender o reiniciar un SPOT
- **SPOT native objects:** Implementación de objetos AmbientTalk que permite manejar los sensores, botones y leds de un SPOT
- **Network driver:** Provee la implementación de red dependiente de la plataforma SPOT

4.5. Resumen

En este capítulo se explicó el diseño de la plataforma de software a desarrollar en esta memoria. Primero se vieron las decisiones de diseño que afectan a la plataforma en general. Luego se revisó la arquitectura física de esta plataforma. Finalmente se revisó la arquitectura lógica de alto nivel tanto de la plataforma, como de las aplicaciones que la conforman.

Capítulo 5

Compilador AmbientTalk

En este capítulo se detallará la arquitectura lógica del compilador. Primero se veran las decisiones de diseño para la parte del compilador. Después se verá la arquitectura lógica del compilador. Finalmente se verá en profundidad cada componente del compilador.

5.1. Introducción

El código fuente de un lenguaje de programación debe ser procesado para generar el AST correspondiente. Este es usado por el intérprete para ejecutar las instrucciones provistas por el programador. Este procesamiento, para gramáticas libres de contexto en general, es realizado en dos fases. La primera fase es un análisis léxico, que reconoce tokens en la entrada, y los entrega a la siguiente fase. La segunda fase es un análisis sintáctico, que dado unos tokens de entrada, reconoce qué regla gramatical le corresponde, y de éste genera el AST correspondiente.

La función del compilador AmbientTalk es generar un código más simple de procesar por el intérprete AmbientTalk, realizando el análisis léxico y análisis sintáctico del código fuente AmbientTalk y luego generando un ejecutable en la cual estas dos fases son más simples de realizar para una máquina de bajo poder de cómputo, que es un Sun SPOT.

5.2. Decisiones de diseño

Hay decisiones que se tomaron que afectan la arquitectura lógica, la implementación, o ambas, del compilador AmbientTalk. Estas son:

DECISIÓN IV **No modificar el código del parser de AmbientTalk:** Se podría haber agregado en el código del parser del intérprete oficial de AmbientTalk, el método que permite generar el XML necesario. Sin embargo, se decidió no tocar el código del parser para que así se pueda mantener el desarrollo independiente del parser, y que después se pueda actualizar el parser usado por el compilador de forma sencilla.

DECISIÓN V **Generar un XML tradicional:** Se decidió añadir el paso intermedio de generación de un XML tradicional entre el AST y el archivo ABX. Una de las razones para este paso intermedio es que permite implementar una validación al XML generado por el compilador, usando para esto herramientas de validación disponibles para XML, permitiendo atrapar algunos bugs del compilador. Otra razón es que es mucho más sencillo revisar visualmente un archivo XML tradicional, que un archivo ABX para efectos de debuguear el compilador.

DECISIÓN VI **No usar una tabla de símbolos:** Aunque agregar una tabla de símbolos podría reducir el tamaño del archivo ABX, se necesitaría que esta tabla de símbolos esté en memoria RAM durante toda la ejecución del archivo, además del fragmento de archivo que se está ejecutando actualmente. Sin la tabla de símbolos solo es necesario lo último, ahorrando memoria RAM, pero gastando más espacio en la memoria flash, recurso que es menos escaso que la memoria RAM.

DECISIÓN VII **Enviar código a los SPOTs a través del puerto USB:** Hay dos formas que se podría enviar un archivo ABX a un SPOT - a través del puerto USB o vía red inalámbrica. Sin embargo, la segunda opción requiere que el computador transmisor esté conectado a un SPOT que funcione como basestation[1]. Es así que se prefirió implementar la transmisión vía USB antes que la vía inalámbrica, para así no necesitar un SPOT de intermediario.

[1]Un SPOT basestation permite a cualquier computador que este directamente conectado a su puerto USB, comunicarse a una red inalámbrica IEEE 802.15.4 (La capa MAC de ZigBee)

5.3. Arquitectura lógica

La Figura 5.1 muestra el proceso de transformación de un código fuente AmbientTalk almacenado en el computador host, a un archivo ABX almacenado en el SPOT. Cada paso del proceso señalado es un módulo del compilador. Se agrega un módulo controlador que actúa como pegamento de los módulos.

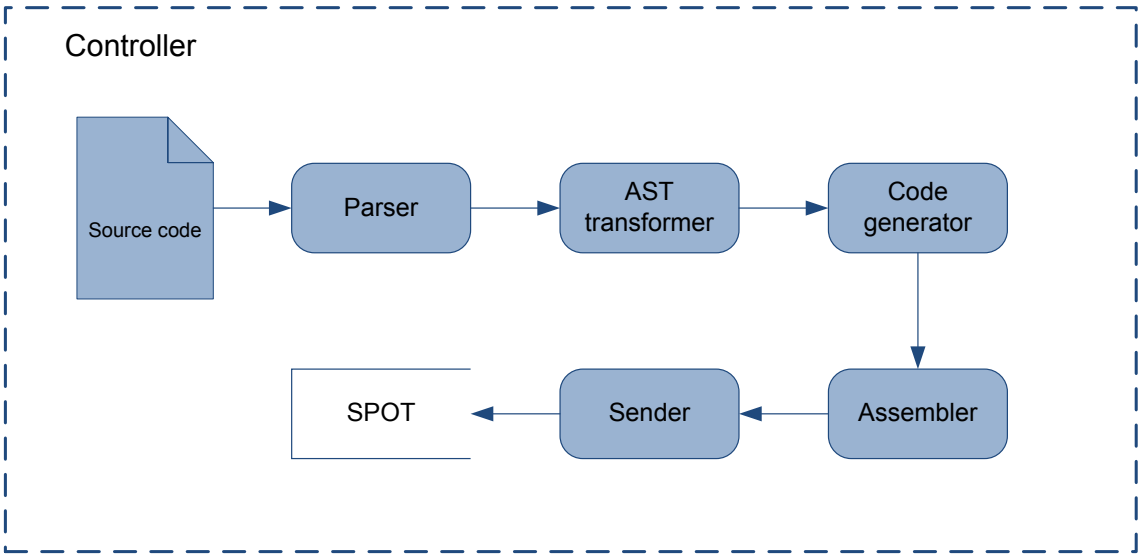


Figura 5.1: Arquitectura Lógica Compilador

5.4. Parser

El módulo Parser tiene como función principal la tarea de leer la entrada y entregar el AST correspondiente. Se prefirió reutilizar el parser usado por el intérprete oficial, debido principalmente a que la referencia de la gramática del lenguaje AmbientTalk es la que reconoce este parser.

5.5. AST transformer

Debido a que el AST obtenido en el paso anterior, el Parser, es un módulo que no se tiene acceso al código fuente (por Decisión 4), es necesario un módulo que traduzca este AST en otro AST, función que es realizada por el AST transformer. Esta transformación realiza una navegación en profundidad del árbol, transformando los nodos padres luego de transformar

los nodos hijos. En el anexo B, se detalla una tabla de traducción de AST origen a AST destino.

La transformación tiene dos objetivos: cambiar el AST original por un AST capaz de generar código y reducir la cantidad de tipos de nodos del AST distintos. El segundo objetivo se debe a que cada tipo de nodo distinto va a resultar en al menos un opcode distinto en el archivo ABX , por lo que reducirlo logra que se reduzca la cantidad de opcodes, y por Decisión 3 se está tratando de tener la menor cantidad posible de opcodes. Sin embargo, estas transformaciones deben dejar el significado semántico del programa intacto. Eso quiere decir, que ambos programas deben de realizar lo mismo.

Para lograr eliminar nodos se ha usado lo siguiente: reducir la azúcar sintáctica más de lo que hace el parser del intérprete oficial de AmbientTalk, introducir un objeto que permite acceder al scope léxico, sin necesidad de nodos especiales, o reemplazar un nodo por varios nodos equivalentes.

Estas transformaciones pueden producir que una futura implementación de gramática de primer orden en ATSpot tenga resultados distintos entre el intérprete oficial y el intérprete implementado en esta memoria. No obstante, en el momento en que se realice tal implementación, se pueden agregar tales opcodes sin ningún problema.

5.6. Code generator

El módulo Code generator es el encargado de generar un archivo XML tradicional que corresponde al AST generado por AST transformer. Para generar este archivo, se procede primero a construir la representación DOM del archivo XML y luego generar el XML usando esta representación. Siempre un nodo del AST se transforma en un subárbol del XML y no en un subbosque de éste.

5.7. Assembler

El módulo Assembler tiene dos funciones: validar que el archivo XML este correcto y transformar un archivo XML tradicional en un archivo ABX. Para realizar la validación y debido a la naturaleza de la transformación, se decidió usar un parser SAX[2] (JAXP-SAX)

[2]Simple API for XML

para leer el archivo XML generado por el Code generator.

Un parser SAX funciona generando eventos producidos al leer secuencialmente la entrada, tales como inicio de tag, fin de tag, o contenido de texto. Estos eventos son manejados por un handler que es capaz de comprender y realizar acciones relacionadas con estos. En el caso del Assembler, el handler consiste en buscar el opcode que le corresponde a un cierto tag XML usando un diccionario cuando encuentra un evento de inicio o fin de tag. También traduce el contenido, si corresponde, dependiendo del tipo.

El diccionario mencionado es un diccionario que asocia namespace y nombre de elemento con el número de opcode y tipo de elemento ABX. Es así como sólo es necesario modificar la asociación del diccionario para agregar, modificar o eliminar opcodes a generar por el assembler.

5.8. Sender

La función del módulo Sender es leer un archivo ABX y enviarlo a un SPOT a través del puerto serial (USB). Para realizar la transmisión por el puerto serial se ocupa la librería RXTXcomm, que permite ocupar este tipo de puertos en Java usando Streams al igual que el usado por otros sistemas de entrada en Java como consola, archivos o comunicación vía sockets.

Sin embargo, es necesario en primer lugar saber el nombre del puerto serial a abrir, que es dependiente del sistema operativo en que corre el compilador[3]. Es por esto, que este módulo ofrece un componente para que el usuario elija el puerto serial a transmitir, dentro de una lista de puertos seriales conectados.

Los streams que ofrece esta librería no dan ninguna garantía de transmisión, debido a la naturaleza del puerto serial. Es así, como al mandar datos a un puerto que no está escuchando, el programa no se dará cuenta y los datos se habrán perdido. También, si el que envía cierra la conexión, el que escucha no sabrá que el que envía cerró la conexión. Otro problema es que si el puerto no es cerrado, al menos en Windows, el puerto serial ya no podrá ser ocupado hasta que se reinicie el sistema operativo.

Es así como este componente implementa un cerrado "automático" del puerto cuando

[3]En sistemas operativos Windows, el nombre del puerto ha sido COMx. En linux, el nombre de un puerto serial es ttySx. En ambos casos, la x es un entero asignado por el SO

se cierran todos los streams abiertos del puerto, como un "protocolo" de comunicación. Sin embargo, el problema de la fragilidad en la comunicación no se ha resuelto en esta memoria debido a la baja prioridad del problema y del tiempo limitado para realizar ésta.

5.9. Controller

El módulo Controlador es el encargado de realizar la conexión entre la salida de un módulo y la entrada del otro. La existencia de este módulo permite que si es necesario se pueda agregar o quitar pasos del proceso de compilación.

En el proyecto existen dos componentes controladores `ATCompiler` y `AmbientTalk`. El componente `ATCompiler` permite compilar un solo archivo a formato ABX, o todo un directorio en profundidad, y guardar los archivos resultantes en otro directorio, manteniendo la estructura de directorio. El componente `AmbientTalk` permite compilar un solo archivo en memoria y enviarlo a un SPOT a través de un puerto serial seleccionado vía interactiva.

5.10. Resumen

En este capítulo se detalló la arquitectura lógica del compilador. Primero se vio las decisiones de diseño para la parte del compilador. Después se vio la arquitectura lógica del compilador. Finalmente se vio en profundidad cada componente del compilador.

Capítulo 6

Intérprete AmbientTalk

En este capítulo se detallará la arquitectura lógica del intérprete. Primero se veran las decisiones de diseño para la parte del intérprete. Después se verá la arquitectura lógica del intérprete. Finalmente se verá en profundidad cada componente del intérprete.

6.1. Introducción

El intérprete AmbientTalk permite ejecutar código AmbientTalk en una plataforma Java ME CLDC. El intérprete puede ejecutar un subconjunto de las características del lenguaje AmbientTalk.

6.2. Decisiones de diseño

Hay decisiones que se tomaron que afectan la arquitectura lógica, la implementación, o ambas, del intérprete AmbientTalk. Estas son:

General

Decisión VIII **Priorizar portabilidad:** Aunque este intérprete está enfocado en correr sobre un SPOT, se desea que pueda ser llevado a otros dispositivos con Java ME CLDC. Es por esto que se prefirió usar librerías estándar de Java ME CLDC o propias, en vez de librerías específicas del SPOT. En los casos en que la implementación es dependiente de la arquitectura, se prefirió tenerlo en módulos independientes, tal que se puedan

reemplazar al cambiar de arquitectura. En el aspecto de la portabilidad, se consideran también máquinas capaces de correr Java SE.

Decisión IX **Optimizar uso de memoria RAM:** Un SPOT, y en general, los dispositivos que corren máquinas virtuales compatibles con Java ME CLDC poseen una pequeña cantidad de memoria RAM disponible para los programas. El intérprete debe usar una cierta cantidad para el mismo, y otra para los objetos en sí. Es por esto que es importante reducir el consumo de memoria, usando como trade-off tiempo de ejecución y tamaño del código.

Decisión X **Optimización en el tamaño del código:** El código también es cargado en memoria RAM, en un área reservada de la RAM para ese propósito. Aunque tiene menor prioridad que ahorrar memoria RAM o permitir portabilización, es importante optimizar el tamaño para que sea posible agregar nuevas funcionalidades al intérprete o evitar que el intérprete no funcione por superar el límite del tamaño de código. Este límite es más estricto en celulares, que superando el límite superior que poseen, el intérprete no se puede ni siquiera guardar en este.

Core

Decisión XI **Separación entre el handler y parser SBX (SAX for Binary XML):** El handler y parser SAX pueden ser realizados monolíticamente o tenerlos separados. La primera opción posee como beneficio un ahorro en memoria RAM, mientras que tenerlos separados permite una mayor flexibilidad en la interpretación de opcodes, permitiendo que una futura implementación de mirages no produzca un impacto en performance en el código que no lo ocupa, además de poseer una mayor modularidad, aumentando la mantenibilidad. Se decidió que era preferible la mantenibilidad y flexibilidad versus el ahorro de memoria RAM, por lo que se eligió una implementación por separado.

Decisión XII **SBX como parser pasivo:** Se decidió que el parser avance de un solo evento a la vez, en vez de todos los eventos hasta el fin de archivo o fin de nodo. Esto permite que sea la aplicación la que controle el flujo de ejecución, permitiendo terminar anticipadamente el parseo, o dejarlo temporalmente suspendido.

Decisión XIII **Proveer la API de Java SE para manejo de archivos:** Se decidió implementar la API de manejo de archivos en Java SE, que corresponde a las clases `File`, `FileInputStream` y `FileOutputStream`, en vez de una API especial para esta memoria, debido a que por Decisión 8, este intérprete también debe correr en máquinas con Java SE. Así, ocupando la API de Java SE, al portar a Java SE, solo es necesario quitar el módulo Filesystem.

Network

Decisión XIV **Publicar los requisitos de servicios:** El sistema de publicación y subscripción requiere, cuando hay cero arquitectura, que los dispositivos comuniquen a todos los demás dispositivos cercanos que servicios requieren o que servicios proveen. Se decidió publicar los servicios requeridos debido a que se necesita comunicar menos información que publicar los servicios proveídos, ya que los proveedores no solo ponen a disposición un tipo de servicio, sino también los supertipos de ese tipo a disposición.

6.3. Arquitectura lógica

La misión de ATSpot es recibir y ejecutar el código AmbientTalk en formato ABX recibido del compilador, y proveer la infraestructura y librerías AmbientTalk necesarias para poder realizar esta labor. La Figura 6.1 muestra los módulos de ATSpot y las dependencias entre módulos entre estos, con una flecha señalando que un modulo depende de otro.

A continuación, se explica cada sistema y los módulos asociados del intérprete de AmbientTalk en un SPOT.

6.4. Core

El sistema Core provee lo necesario para la ejecución de un programa AmbientTalk. Esto consiste en abrir un archivo ABX de una memoria no volátil, parsearlo e interpretarlo de una manera correcta y completa. A continuación, se detalla los módulos que integran este sistema.

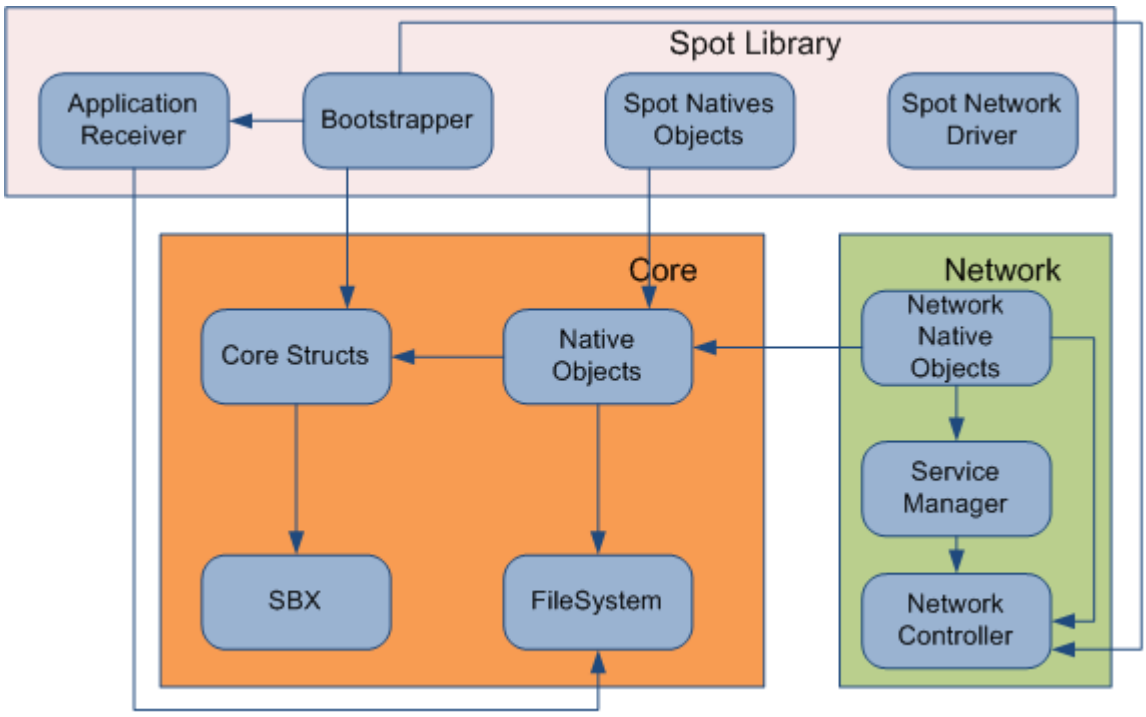


Figura 6.1: Arquitectura Lógica Interprete

6.4.1. Filesystem

El modulo Filesystem es el encargado de proveer de un sistema de archivos a la plataforma Java ME CLDC. En Java SE o Java ME CDC, el acceso a archivos y directorios está provisto por el SDK de Java usando las clases: `File` que permite navegar por la jerarquía de directorios, `FileInputStream` que permite abrir un archivo para leer y `FileOutputStream` que permite abrir o crear un archivo para escribir.

En Java ME CLDC estas clases no están disponibles, poniendo como reemplazo de almacenamiento permanente el framework RMS. Lo que hace este módulo es proveer las tres clases de manejo de archivos del SDK de Java usando dos RecordStores: uno para mantener la meta información de los archivos como su tamaño o si son directorios, mientras que el otro es usado para almacenar el contenido en sí de los archivos.

Debido a la forma en que está implementado este módulo, el tamaño de los archivos no influye en el gasto de memoria al tener abierto un archivo.

6.4.2. SBX

El módulo SBX (SAX for Binary XML) es el módulo que se encarga de leer un archivo ABX y de generar eventos SAX que son manejados por un handler SAX. Al igual que un parser SAX, genera eventos de inicio y fin de tag. Sin embargo, estos eventos son generados solo para nodos contenedores. Para el resto se genera eventos de tags de entero, real, string o vacío, dependiendo del tipo del nodo.

Otra diferencia con un parser SAX es que el parser solo genera el siguiente evento, en vez de generar todos los eventos, como lo realizaría un parser SAX comúnmente. Sin embargo, esta característica permite dejar suspendida la generación de eventos, que es ocupado al invocar un método o también para pedirle al parser que omita ciertos eventos.

6.4.3. Core structs

El módulo Core structs es el encargado de entregar todas las clases e interfaces necesarias para la ejecución del intérprete. Este modulo está compuesto de: estructuras de datos básicas, actores, mensajes, handlers SAX, y las interfaces que definen un objeto AmbientTalk.

Los actores son tareas que ejecutan código AmbientTalk y se podría decir que son el núcleo del intérprete. Ellos son los que contienen todas las estructuras necesarias para la ejecución: el código a ejecutar, el parser y handler SAX actual, la pila de operandos [1], el environment[2] y el objeto self actual.

Además, poseen una cola de mensajes que les permite recibir solicitudes para ejecutar código AmbientTalk. La implementación de esta cola es la de una cola bloqueante. Una cola bloqueante es aquella en la que bloquea a una tarea si esta intenta desencolar un elemento mientras la cola está vacía. Si otra tarea inserta un elemento y había tareas bloqueadas, las desbloquea.

Los mensajes que puede recibir un actor son de 5 tipos. Estos son:

- **Mensaje de invocación asíncrona:** Solicita invocar un método de un objeto asociado con este actor. Corresponden a la invocación de mensajes asíncronos en AmbientTalk. Este es el tipo de mensaje más frecuente que recibe un actor.

[1] La pila de operandos es adonde se leen los argumentos para evaluar un opcode y adonde se deja el resultado de tal evaluación

[2] El environment es adonde se guardan las variables que se encuentran dentro del scope léxico del código en ejecución

- **Mensaje de invocación asíncrona con callback:** Es parecido a un mensaje asíncrono, pero con una petición de avisar a otro actor, mediante un mensaje asíncrono, cuando la ejecución de este mensaje haya concluido.

- **Mensaje de término:** Le señala al actor que deje de procesar mensajes y termine su ejecución.

- **Mensaje de carga de modulo:** Solicita ejecutar código correspondiente a un archivo ABX. Permite cargar un módulo AmbientTalk que no es cargado por una closure u otro modulo AmbientTalk. Este tipo de mensaje es usado por el Bootstrapper para cargar el módulo AmbientTalk inicial.

- **Mensaje de inicialización de actor:** Solicita ejecutar un lambda que permita definir los métodos y campos del objeto del actor AmbientTalk. Este tipo de mensaje es el primer mensaje ejecutado por todo actor, salvo el creado por el modulo Bootstrapper.

Aunque es el actor el que ejecuta el código AmbientTalk, son los Executors los que le dan un significado a los opcodes de un archivo ABX. Un Executor es el nombre dado a un handler SAX que puede interpretar un archivo ABX que corresponde a código AmbientTalk. Actualmente hay dos Executors: el por defecto y el que evalúa los parámetros de un lambda.

Este módulo también define la interfaz que debe implementar todo objeto AmbientTalk nativo implementado en Java. Algunos métodos corresponden a acciones del lenguaje como definir métodos o campos, realizar una invocación o asignación, o realizar clonar. Otros métodos corresponden a información que requiere un actor, como cual es su executor o su actor asociado. Finalmente, hay métodos para realizar invocación en el scope léxico, como saber si la definición del método es superficial o conocer cuál es el objeto contenedor.

Con respecto a la invocación, el proceso de invocación se puede separar en dos partes: la obtención del código del método y la evaluación de este código. Cuando se pide a un objeto obtener el método o campo de un objeto, este tiene tres maneras de contestar. Estas son:

- **Método:** Señala que el resultado es un código en formato ABX, y que debe ser evaluado antes de entregar el resultado.

- **Campo:** Señala que el resultado es un objeto AmbientTalk. Como tal, no es necesario realizar una evaluación, salvo casos especiales.

- **Nativo:** Señala que el resultado de obtener el método es un objeto AmbientTalk que corresponde a la evaluación de un método nativo. Como tal, no se debe realizar una evaluación.

Los actores soportan dos tipos de evaluaciones: las evaluaciones implícitas, que son realizadas por los actores después de que un executor pide invocar un método, y las evaluaciones explícitas, que permiten a un método nativo en Java ejecutar mensajes síncronos y quedar suspendido.

6.4.4. Native objects

El modulo Native objects es el encargado de implementar todos los objetos AmbientTalk que son necesarios por el núcleo y que son implementados en Java. Los objetos nativos se pueden separar en tres tipos:

- **Primitivos:** Son los objetos nativos que corresponden a objetos básicos suministrados por el lenguaje. Corresponde a los enteros, reales, booleanos, cadenas, tablas, lambdas, mensajes, type tags y el objeto nil.

- **Puentes:** Son los objetos nativos que permiten que una implementación de un objeto sea hecha en AmbientTalk. Actualmente solo existe un objeto nativo puente, que corresponde a un objeto genérico, aunque es posible que en trabajos futuros hayan mas objetos nativos puentes.

- **Esenciales:** Son los objetos nativos que realizan funciones especiales al intérprete AmbientTalk y que el usuario no maneja el objeto en sí. Corresponde a los objetos Lexical Root, scope léxico, environment y lobby.

De los objetos nativos, los esenciales merecen una explicación más detallada. El objeto Lexical Root implementa métodos disponibles en cualquier scope. Sin embargo, este objeto es encapsulado por otras capas que agregan más métodos al lexical root. El objeto scope permite invocar métodos, obtener el valor de campos y asignar valores a ellos que se encuentren dentro del scope léxico del código en ejecución. En jerga AmbientTalk, correspondería a invocar funciones y obtener el valor de variables. El objeto environment corresponde a los objetos usados exclusivamente como contenedores. Aunque un objeto genérico también serviría para

	1	2	3	4	5
booleans		X		X	X
environment	X	?	X		
objeto genérico	X		X		
integer		X		X	
lambda		X			
lexical root		X	X	X	X
lobby		X		X	
messages		X		X	
nil		X		X	X
real		X		X	
scope		X	X		X
string		X		X	
table	X			X	
type tag		X		X	

Cuadro 6.1: Características de los objetos nativos

realizar este trabajo, un environment esta optimizado para ocupar la menor cantidad de memoria posible. El objeto lobby permite cargar módulos. A diferencia del campo lobby del lexical root de AmbientTalk, este objeto representa una ruta intermedia de acceso al modulo.

Los objetos nativos pueden poseer las siguientes características:

1. **Mutable:** Un objeto nativo es mutable si se pueden definir campos o cambiar el estado de un objeto.

2. **Único:** Un objeto nativo es único, si al clonarlo se obtiene el mismo objeto.

3. **Contenedor:** Un objeto nativo es contenedor, si es capaz de ser contenedor de otro objeto.

4. **Isolado:** Un objeto nativo es isolado, si no posee actor, y por ende puede ser ejecutado en cualquier actor, incluso los mensajes asíncronos.

5. **Controlado:** Un objeto nativo es controlado si la cantidad de instanciaciones está limitado por alguna variable.

La Tabla 6.1 señala con una X que características posee un cierto objeto nativo. Un ? indica que no es posible definir si pertenece o no.[3]

[3]Un environment no se puede clonar

Muchas de estas características corresponden a las mismas propiedades que poseen estos objetos en AmbientTalk. Sin embargo, estas características permiten realizar optimizaciones en la implementación. Es así como los objetos inmutables no poseen campos ni métodos no nativos (incluso las tablas tampoco los poseen) y los objetos que no son contenedores, no soportan saber si un método está definido, ya que esta información solo es necesaria para cuando son contenedores. Ambas optimizaciones tienen el objetivo de reducir el código innecesario, ahorrando tiempo de codificación, como espacio en memoria para las clases Java.

Si se observa la tabla, se puede ver que en general un objeto nativo no es mutable, es único, no es contenedor, es isolado y no es controlado. Usando esta información, se puede también optimizar en tamaño de código al crear una clase base que posea estas características y luego por herencia se vaya cambiando de acuerdo a las desviaciones de ese objeto.

6.5. Network

El sistema Network provee el soporte de red que brinda AmbientTalk. Permite usar referencia lejanas remotas, y el servicio de descubrimiento y uso de servicios AmbientTalk. A continuación, se detalla los módulos que integran este sistema.

6.5.1. Network controller

El modulo Network controller se encarga de manejar las referencias lejanas, tanto para el envió de mensajes, como la recepción de estos. Todo objeto que desea ser invocado por otra máquina, debe de realizar los siguientes pasos:

1. Registrar el objeto en el modulo network controller, Al realizar esto se obtiene un id de referencia para ese objeto.

2. Enviar a la máquina destino el id de referencia y el id de la maquina origen.

3. En la máquina destino se le pide al network controller crear una referencia remota que apunte al id de máquina y referencia obtenidos.

Así, al mandar un mensaje a la referencia remota, network controller se encarga de enviar ese mensaje a la máquina remota, que lo invocara de manera asíncrona en el objeto registrado.

Para realizar tal envió, ocupa un driver de red, en la cual este modulo especifica la interfaz de esta, pero no provee ninguna implementación.

Este módulo se encarga de registrar y llamar a los listeners de eventos de desconexión y reconexión. Para esto mantiene el número de envíos fallidos consecutivos. Si este número supera un límite, llama a los listeners de desconexión correspondientes. Aunque este módulo no es capaz de saber cuándo se recupera la conexión, el service manager si, por lo que avisa al network controller y este llama a los listeners de reconexión correspondiente.

6.5.2. Service manager

El módulo Service manager se encarga del sistema de publicación y subscripción usado por AmbientTalk. Este sistema permite realizar de forma transparente al programador el envió de id de referencia e id de la maquina requerido por el network controller para comunicarse con objetos remotos.

Este módulo realiza periódicamente un envió de forma broadcast[4] el tipo de los servicios que requiere. Esto hace que las otras maquinas reciban este mensaje y vean si poseen un objeto exportado que satisface ese tipo. Si es así, le envía al que solicitó, el servicio que pueden satisfacer, el id de la máquina y el id de referencia del objeto. Con esta información, la máquina solicitadora pide al network controller crear una referencia remota, y llama al listener de descubrimiento que tiene registrado para ese tipo de servicio.

6.5.3. Native network objects

El módulo Native network objects se encarga de implementar los objetos AmbientTalk nativos que permiten interactuar con el sistema de red. El más importante de estos objetos es el Network Lexical Root que extiende el lexical root con métodos de suscripción y publicación de servicio, o registrar eventos de desconexión o reconexión. Otro objeto nativo AmbientTalk es la referencia remota, en la que todas las invocaciones que este objeto recibe las envía a la referencia remota entregada por el network controller. El resto de los objetos son las implementaciones nativas de objetos que se disponen en AmbientTalk, como network que permite iniciar o parar la comunicaciones de red, o network publication, que permite cancelar una exportación.

[4]Un envió en broadcast es un envió que tiene como destinatario cualquiera que pueda recibir el mensaje

6.6. SPOT libraries

El sistema SPOT libraries provee los objetos AmbientTalk y Java que corresponden a la plataforma Sun SPOT. A continuación, se detalla los módulos que integran este sistema.

6.6.1. Application receiver

El módulo Application receiver permite recibir archivos ABX del compilador. Actualmente, este módulo posee funcionalidades básicas de recibimiento, que permite grabar solo al archivo "init.at".

6.6.2. Bootstrapper

El módulo bootstrapper es el encargado de inicializar al intérprete. Las obligaciones que debe realizar son:

- Definir el objeto AmbientTalk que va a corresponder al lexical root.
- Asociar un driver de red al network controller.
- Definir el directorio raíz de los módulos AmbientTalk.
- Crear un actor y pedirle que ejecute un módulo. Actualmente el módulo a cargar es "init.at"
- Crear e inicializar el módulo application receiver.

6.6.3. SPOT native objects

El módulo SPOT native objects define los objetos AmbientTalk disponibles en un SPOT y que permiten ocupar características especiales de estos dispositivos. Actualmente se pueden ocupar los 8 leds del SPOT, además de los 2 botones de usuario. Los leds se encuentran agrupados en una tabla, permitiendo ocupar métodos de AmbientTalk para recorrer los leds. Para los botones, es posible registrar listeners a los eventos de presionar y liberar botón. Finalmente, este módulo provee un nuevo lexical root que incorpora estos elementos.

6.6.4. SPOT network driver

El módulo SPOT network driver es una implementación de un network driver requerido por network controller y service manager para funcionar.

6.7. Resumen

En este capítulo se detallo la arquitectura lógica del intérprete. Primero se vio las decisiones de diseño para la parte del intérprete. Después se vio la arquitectura lógica del intérprete. Finalmente se vio en profundidad cada componente del intérprete.

Parte III

Validación y Conclusión

Capítulo 7

Validación

En este capítulo se verá una breve introducción que explica los motivos para realizar una validación, se explicará el ejemplo que va a ser resuelto, se mostrarán las implementaciones de este ejemplo, tanto en AmbientTalk como directamente en Java ME, y se realizará una comparación entre ambas soluciones.

7.1. Introducción

Los Sun SPOTs poseen un SDK[1] de desarrollo de por si bastante completo. Este SDK permite programar en Java ME, con lo cual se posee manejo de tareas, escritura a memoria persistente y comunicación en redes diversas. Además, el dispositivo incluye librerías para usar los sensores, las luces y los botones que éste posee.

Es por este motivo que se puede dudar de que programar en AmbientTalk posea algún beneficio a un desarrollador con respecto a programar la misma aplicación directamente en el SDK de los SPOTs. Aunque en la Sección 1.1 se dan las razones teóricas, es mejor ver las diferencias con un ejemplo implementado usando el SDK de los SPOTs, y el mismo ejemplo implementado en un programa AmbientTalk que corre usando el intérprete realizado en esta memoria. El código de todos los ejemplos se encuentra en el Anexo D.

[1] Kit de desarrollo de software

7.2. Controlador de luz remoto con un Spot: un ejemplo de validación

El ejemplo a desarrollar es un control remoto de luz usando dos SPOTs: uno de ellos corresponde al control remoto en sí, y el otro SPOT corresponde a la luz que va a ser controlada. El control remoto posee dos botones: una para encender la luz, y otro para apagarla. La luz que maneja es la primera que encuentra al encenderse el control, inclusive si después pierde la comunicación. El control remoto posee un indicador de si puede comunicarse con la luz. Cuando no hay problemas de comunicación muestra una sola luz verde, mientras que si hay problemas, muestra tantas luces rojas como comandos que no ha podido enviar desde antes que se perdiera la comunicación.

A continuación se explica la implementación del ejemplo usando Java ME y después usando AmbientTalk. Ambas soluciones se pueden separar en dos aplicaciones: una para el SPOT que funciona como luz, que de ahora en adelante lo llamaremos controlador de luz, y otro para el SPOT que funciona como control remoto.

7.2.1. Solución en Java ME

La solución en el lado del controlador de luz se resuelve con dos tareas. La tarea principal espera un mensaje del control remoto y dependiendo de este, prende o apaga la luz. La tarea secundaria manda cada cierto tiempo un mensaje al ambiente especificando su dirección.

En el lado del control remoto, primero trata de encontrar un mensaje con la dirección de un controlador. Cuando lo encuentra, recuerda esta dirección, activa los botones y la tarea principal. Al presionar un botón, le avisa a la tarea principal que quiere enviar un mensaje. El hilo principal recuerda el último mensaje y cuantas veces le han avisado. Cuando tiene un mensaje, intenta enviar ese mensaje. Si el envío es exitoso, resetea el número de mensajes a enviar y espera un nuevo mensaje. Si el envío falla, activa tantas luces rojas como mensajes pendientes había y espera un mensaje de dirección de la luz. Cuando éste llega, vuelve a dejar solo una luz verde encendida y vuelve a intentar mandar el mensaje.

7.2.2. Aproximación AmbientTalk

La solución en el lado del controlador de luz se resuelve exportando un objeto de tipo `Light` que posee un método que permite encender o apagar la luz.

En el lado del control remoto, cuando encuentra un objeto de tipo `Light`, activa los botones y los eventos de desconexión de ese objeto remoto. Cuando se presiona un botón, se envía un mensaje al objeto remoto indicando si se debe encender o apagar la luz. Cuando se pierde la conexión con el objeto remoto, se obtienen los mensajes no enviados, para que después se cuenten, se reenvíen y luego se enciende tantas luces rojas como mensajes no enviados habían. Cuando se vuelve a obtener la conexión, se vuelve a dejar solo una luz verde encendida.

7.3. Comparación

Ambas soluciones son casi idénticas para el usuario final del control remoto, sin embargo presenta ventajas y desventajas para el desarrollador de la aplicación. Las ventajas de la aplicación Java ME con respecto a la de AmbientTalk son:

- **Mayor eficiencia en el uso de la red:** Permite mayor granularidad en el envío de mensajes, permitiendo realizar optimizaciones adhoc, y así reducir la congestión de la red inalámbrica.
- **Mayor eficiencia en el uso de los recursos del SPOT:** La aplicación Java ME ocupa menos memoria RAM, posee menos tareas y ocupa menos memoria flash que la aplicación AmbientTalk en conjunto con el intérprete.

Las ventajas de la aplicación AmbientTalk con respecto a la de Java ME son:

- **Menos código:** La aplicación AmbientTalk necesitó menos líneas de código que la misma aplicación Java ME.
- **No posee ninguna sincronización explícita:** En el programa AmbientTalk, todas las sincronizaciones son manejadas implícitamente, mientras que en la aplicación Java ME, es el desarrollador el que debe encargarse de que no haya *dataraces*.

- **Manejo de alto nivel en red:** En Java ME para poder enviar un mensaje se debe saber: la dirección del SPOT a comunicarse, el puerto, el protocolo que debe usarse y el formato de los mensajes. En AmbientTalk, lo único que hay que saber es el nombre del tipo de objeto exportado y los mensajes que soporta.

- **Manejo de errores implícito:** En AmbientTalk el manejo de errores de comunicación es implícito, en otras palabras, es el intérprete el que reenvía los mensajes cuando se restablece la comunicación. En cambio, en Java ME, es el desarrollador el que debe preocuparse de reenviar los mensajes y de manejar los errores. No solo eso, sino que hay bugs que pueden saltar sólo cuando hay fallas en la comunicación, dificultando esta tarea.

De las ventajas y desventajas se puede concluir que programar en AmbientTalk requiere menos trabajo para el desarrollador que programar en Java ME directamente, para el caso de aplicaciones distribuidas en red. Sin embargo, Java ME sigue siendo útil en este ámbito para aplicaciones en que la eficiencia en el uso de la red, o el uso de los recursos del SPOTs sean primordiales.

7.4. Resumen

En este capítulo se vio una breve introducción que explicó los motivos para realizar una validación, se explicó el ejemplo que va a ser resuelto, se mostró las implementaciones de este ejemplo, tanto en AmbientTalk como directamente en Java ME, y se realizó una comparación entre ambas soluciones.

Capítulo 8

Conclusiones

Aunque el trabajo de esta memoria no llega a cumplir a cabalidad el objetivo general y todos los objetivos específicos planteados al inicio de la memoria, sí se logró cumplir los más importantes de éstos y dejar una base fuerte para lograr cumplir lo que falta por realizar.

8.1. Contribuciones

El trabajo realizado contribuye en los siguientes aspectos:

- Permite programar en un lenguaje que logra disminuir el tiempo ocupado en programación en un Sun SPOT en el caso de aplicaciones distribuidas.
- Permite ocupar AmbientTalk en toda la gama de dispositivos que soportan tecnología Java.
- Permite probar AmbientTalk en dispositivos afines a su finalidad.

8.2. Trabajo futuro

El trabajo futuro se puede separar en 3 tipos de trabajos: nuevas funcionalidades, portar y mejoras.

Nuevas funcionalidades

El trabajo futuro con respecto a agregar nuevas funcionalidades al intérprete AmbientTalk son:

- **Mirage y mirrors:** AmbientTalk tiene un sistema de reflexión basado en mirrors y mirages [15]. Ambos no están actualmente implementados.
- **Gramática de primer orden:** En AmbientTalk, a una instrucción se le puede pedir que en vez de ejecutarse, devuelva el AST que le corresponde, usable desde AmbientTalk. Esta característica no está implementada tanto en el intérprete como en el compilador AmbientTalk.
- **Type tags:** Aunque esta implementado los type tags en si, no está implementado marcar con type tags tanto mensajes ni métodos.
- **Excepciones:** Implementar tanto el sistema de manejo de excepciones, como el envió correcto de excepciones AmbientTalk en caso de error.
- **Sensores SPOT:** Permitir ocupar los sensores de luz, de temperatura o acelerómetro en el intérprete AmbientTalk.

Portar

El trabajo futuro con respecto a portar el intérprete a otras plataformas son:

- **Portar a Java SE:** Crear una versión del intérprete AmbientTalk que permita correr a éste en Java SE. Aunque el desarrollo estuvo pensado en ser portable, falta implementar algunos detalles e implementar características que no están disponibles en un dispositivo móvil liviano, como por ejemplo, una consola. Teniendo esta implementación, se va a completar el último aspecto que falta por completar del objetivo general, que es la capacidad de hablar con otras máquinas que no sean Sun SPOT o celulares.
- **Portar a un celular:** Crear una versión del intérprete AmbientTalk que permita correr en un celular. En particular, que pueda comunicarse vía bluetooth o vía OTA[1] con otros celulares, y que permita a un programa AmbientTalk interactuar con un usuario.

[1] Over The Air

Mejoras

El trabajo futuro con respecto a mejorar el intérprete AmbientTalk son:

- **Usar símbolos predefinidos:** Hay mucho símbolos frecuentemente usados en AmbientTalk como: `at_put`, `at`, `foreach:`, etc. Se puede crear un opcode que permita asociar un id numérico a uno de estos símbolos, disminuyendo el tamaño del código ejecutable.

- **Reemplazar driver de red de Sun SPOT:** El driver de red actualmente implementado no permite enviar grandes cantidades de datos en una sola vez. Para programas pequeños esto no es problema, pero si para programas grandes.

- **Máquina Virtual para AmbientTalk:** En vez de tener un intérprete AmbientTalk hecho en Java, crear una maquina virtual que pueda ejecutar aplicaciones AmbientTalk, basándose en los opcodes y formato ABX obtenidos en esta memoria. Esto permitirá tener una mejor eficiencia al correr aplicaciones AmbientTalk.

Anexos

Anexo A

Opcodes AmbientTalk

Número	Nombre	Tipo	Descripción
0x01	NIL	Vacio	Devuelve el objeto *nil*
0x02	SELF	Vacio	Devuelve el objeto *self* actual
0x03	SCOPE	Vacio	Devuelve el objeto que representa el scope léxico
0x04	TRUE	Vacio	Devuelve el objeto booleano que representa a *true*
0x05	FALSE	Vacio	Devuelve el objeto booleano que representa a *false*
0x06	INTEGER	Entero	Devuelve el objeto entero correspondiente al valor expresado
0x07	REAL	Real	Devuelve el objeto de número real correspondiente al valor expresado
0x08	STRING	Cadena	Devuelve el objeto string correspondiente a la cadena expresada
0x09	SYMBOL	Cadena	Devuelve un símbolo correspondiente a la cadena expresada
0x0A	LAMBDA	Contenedor	Construye un objeto lambda
0x0B	TYPE	Contenedor	Construye un objeto type tag
0x0C	MSG	Contenedor	Construye un objeto de mensaje normal

Número	Nombre	Tipo	Descripción
0x0D	FIELD_MSG	Contenedor	Construye un objeto de mensaje de campo
0x0E	ASYNC_MSG	Contenedor	Construye un objeto de mensaje asíncrono
0x0F	DELEGATE_MSG	Contenedor	Construye un objeto de mensaje delegado
0x11	TABLE	Contenedor	Construye un objeto de tabla (arreglo)
0x12	EXPRS	Contenedor	Evalúa todos los "nodos" y devuelve el valor del último evaluado
0x13	DEFINE	Cadena	Define un campo en el environment actual
0x14	METHOD	Contenedor	Define un método en el environment actual
0x15	ASSIGN	Contenedor	Asigna un valor a un "campo"
0x16	MULTI_ASSIGN	Contenedor	Asigna un valor distinto a varios "campo", usando una tabla
0x17	GET_METHOD	Contenedor	Obtiene un lambda equivalente a invocar el método
0x18	SEND	Contenedor	Envía un mensaje a un objeto
0x19	SPLICE	Contenedor	Vierte el contenido de una tabla dentro de otra
0x1A	IMPORT_SCOPE	Contenedor	Incluye las definiciones de un environment dentro de otro
0x1B	PARAM	Contenedor	Define los parámetros de un lambda
0x1C	DEFAULT	Contenedor	Define el valor por defecto de un parámetro
0x1D	REST	Cadena	Define un parámetro que contiene el resto de los parámetros
0x1E	ASSIGN_TABLE	Contenedor	Define las variables a asignar en un multi-assign
0x1F	ALIAS	Contenedor	Define un alias a una de las definiciones a importar
0x20	EXCLUDE	Contenedor	Establece que definiciones se deben excluir al importar
0x21	INIT_TABLE	Contenedor	Devuelve una tabla inicializada usando un lambda

Número	Nombre	Tipo	Descripción
0x22	QUOTE	Contenedor	Devuelve una gramática de primer orden que representa a lo contenido
0x23	UNQUOTE	Contenedor	Evalúa el contenido y lo devuelve como elemento de una gramática
0x24	RECEIVE	Sintético	Usado para la recepción de mensajes asíncronos
0x25	RECEIVE_ACTOR	Sintético	Usado por un actor para la recepción de mensajes asíncronos
0x26	SEND_ACTOR	Sintético	Usado por un actor para el envió de mensajes
0x27	PROCESS	Sintético	Usado por un actor para procesar un mensaje
0x30	MODULE	Contenedor	Inicializa un modulo. También es el elemento raíz de un ejecutable AmbientTalk

Anexo B

Transformación del AST

A continuación se detallan las transformaciones realizadas al AST entregado por el modulo Parser

Nodo origen	**Nodo destino**	**Comentarios**
`ATApplication(function, args)`	`SEND(a, MSG(b, args)`	Si `function` es un símbolo, `a=Scope` y `b=function`. Si no, `a=function` y `b='apply'`
`ATAssignField(obj,field,rvalue)`	`ASSIGN(obj,field,rvalue)`	
`ATAssignTable(t=table,i,v)`	`SEND(t,MSG('at_put',TABLE(i,v)))`	
`ATAssignVariable(l=left,r=right)`	`ASSIGN(SCOPE,l,r)`	
`ATAsyncMC` [1]`(name,args,ann)`	`AYNC_MSG(name,args,ann)`	
`ATBegin(ATTAble(exprs))`	`EXPRS(exprs)`	
`ATClosureLiteral(args,body)`	`LAMBDA(args,body)`	
`ATDefExternalField(...)`	`NIL`	Nodo origen deprecado
`ATDefExternalMethod(...)`	`NIL`	Nodo origen deprecado
`ATDefField(n=name,v=value)`	`EXPRS(DEF[n],ASSIGN(SCOPE,n,v))`	
`ATDefMethod(n=name,a=args,b=body)`	`METHOD(n,a,b)`	
`ATDefTable(n=name,l=length,i=init)`	`EXPRS(DEF[n],ASSIGN(SCOPE,n,a))`	`a = INIT\_TABLE(l,i)`
`ATDefType(n=name,p=parents)`	`EXPRS(DEF[n],a)`	`a = ASSIGN(SCOPE,n,TYPE(n,p))`
`AGDelegationCreation(sel,args,ann)`	`DELEGATE_MSG(sel,args,ann)`	
`ATFieldSC`[2]`(name,ann)`	`FIELD_MSG(name)`	Si `ann` no es una tabla vacía, el compilador arroja un error.
`NATFraction(valor)`	`REAL[valor]`	
`ATImport(...)`	`IMPORT_SCOPE(...)`	La transformación solo encapsula los alias y los exclude con `ALIAS` y `EXCLUDE`
`ATLookup(name)`	`GET_METHOD(SCOPE,name)`	
`ATMessageSend(msg)`	`SEND(msg)`	

[1] ATSyncMessageCreation
[2] ATFieldSelectionCreation

Nodo origen	Nodo destino	Comentarios
ATMethodIC[3](sel,args,ann)	MSG(sel,args,ann)	
ATMultiAssignment(left,right)	MULTI_ASSIGN(right,a)	a = ASSIGN_TABLE(left)
ATMultiDefinition(left,right)	EXPRS(...)	Igual que ATMultiAssignment, salvo que se define antes con DEF las variables
NATNumber(valor)	INTEGER[valor]	
ATQuote(ast)	QUOTE(ast)	
ATSelection(obj,sel)	GET_METHOD(obj,sel)	
ATSplice(expr)	SPLICE(expr)	
ATSymbol(name)	a	Si name es 'nil', 'true' o 'false', a=NIL, TRUE o FALSE respectivamente. Si no, a=SEND(SCOPE,FIELD_MSG(name))
NATTable(expr*)	TABLE(expr*)	
ATTabulation(table,i)	SEND(table,MSG(át',TABLE(i)))	
NATText(string)	STRING[string]	
ATUnquote(ast)	NIL	No implementado
ATUnquoteSplice(ast)	NIL	No implementado

[3]ATMethodInvocationCreation

Anexo C

Schema del XML AmbientTalk

```
<?xml version="1.0" encoding="UTF-8"?>
<schema targetNamespace="http://pleiad.dcc.uchile.cl/ATSchema"
     elementFormDefault="qualified"
     xmlns="http://www.w3.org/2001/XMLSchema"
     xmlns:at="http://pleiad.dcc.uchile.cl/ATSchema">

  <element name="module" type="at:Expresion" />

  <complexType name="Expresion">
    <choice>
      <group ref="at:Expresions" />
    </choice>
  </complexType>

  <complexType name="Expresions">
    <choice minOccurs="0" maxOccurs="unbounded" >
      <group ref="at:Expresions" />
    </choice>
  </complexType>

  <group name="Expresions">
    <choice>
      <element name="nil" type="at:Empty" />
      <element name="self" type="at:Empty" />
      <element name="scope" type="at:Empty" />
      <element name="true" type="at:Empty" />
      <element name="false" type="at:Empty" />
      <element name="integer" type="int" />
      <element name="real" type="double" />
      <element name="string" type="string" />
      <element name="exprs" type="at:Expresions" />
      <element name="send" type="at:Send" />
```

```
    <element name="assign" type="at:Assign" />
    <element name="define" type="string" />
    <element name="multi-assign" type="at:MultiAssign" />
    <element name="table" type="at:Expresions" />
    <element name="init-table" type="at:InitTable" />
    <element name="splice" type="at:Expresion" />
    <element name="lambda" type="at:Lambda" />
    <element name="method" type="at:Method" />
    <element name="get-method" type="at:GetMethod" />
    <choice>
      <group ref="at:Messages" />
    </choice>
    <element name="import-scope" type="at:Import" />
    <element name="type" type="at:Type" />
    <element name="quote" type="at:Expresion" />
  </choice>
</group>

<group name="Messages">
  <choice>
    <element name="msg" type="at:Message" />
    <element name="async-msg" type="at:Message" />
    <element name="delegate-msg" type="at:Message" />
    <element name="field-msg" type="at:FieldMessage" />
  </choice>
</group>


<complexType name="Empty" />

<complexType name="Send">
  <sequence>
    <!-- Receiver -->
    <choice>
      <group ref="at:Expresions" />
    </choice>
    <!-- Message -->
    <choice>
      <group ref="at:Expresions" />
    </choice>
  </sequence>
</complexType>

<complexType name="Assign">
  <sequence>
    <!-- Left value: object and field name-->
    <choice>
```

```
      <group ref="at:Expresions" />
    </choice>
    <element name="symbol" type="string" />
    <!-- Right value -->
    <choice>
      <group ref="at:Expresions" />
    </choice>
  </sequence>
</complexType>

<complexType name="MultiAssign">
  <sequence>
    <!-- Right value -->
    <choice>
      <group ref="at:Expresions" />
    </choice>
    <!-- Left values-->
    <element name="assign-table" type="at:AssignTable" />
  </sequence>
</complexType>

<complexType name="AssignTable">
  <sequence>
    <element name="symbol" type="string" minOccurs="0" maxOccurs="unbounded" />
    <element name="rest" type="string" minOccurs="0" maxOccurs="1" />
  </sequence>
</complexType>

  <complexType name="InitTable">
  <!-- 1.Table size, 2.Init closure -->
    <choice minOccurs="2" maxOccurs="2">
      <group ref="at:Expresions" />
  </choice>
  </complexType>

<complexType name="Lambda">
  <sequence>
    <!-- Parameters -->
    <element name="param" type="at:Param" />
    <!-- Body -->
    <choice>
      <group ref="at:Expresions" />
    </choice>
  </sequence>
</complexType>

<complexType name="Param">
```

```
  <sequence>
    <sequence minOccurs="0" maxOccurs="unbounded">
      <element name="symbol" type="string" />
      <element name="default" type="at:Expresion" minOccurs="0" maxOccurs="1" />
    </sequence>
    <sequence minOccurs="0" maxOccurs="1">
      <element name="rest" type="string" />
      <element name="default" type="at:Expresion" minOccurs="0" maxOccurs="1" />
    </sequence>
  </sequence>
</complexType>

<complexType name="Method">
  <sequence>
    <!-- Name -->
    <element name="symbol" type="string" />
    <!-- Params and body -->
    <element name="lambda" type="at:Lambda" />
    <!-- Type tag -->
    <choice>
      <group ref="at:Expresions" />
    </choice>
  </sequence>
</complexType>

<complexType name="GetMethod">
  <sequence>
    <choice>
      <group ref="at:Expresions" />
    </choice>
    <element name="symbol" type="string" minOccurs="1" maxOccurs="unbounded" />
  </sequence>
</complexType>

<complexType name="Message">
  <sequence>
    <!-- Selector -->
    <element name="symbol" type="string" />
    <!-- Arguments -->
    <element name="table" type="at:Expresions" />
    <!-- Type tag -->
    <choice minOccurs="0" maxOccurs="1">
      <group ref="at:Expresions" />
    </choice>
  </sequence>
</complexType>
```

```
  <complexType name="FieldMessage">
    <sequence>
      <element name="symbol" type="string" />
    </sequence>
  </complexType>

  <complexType name="Import">
    <sequence>
      <choice>
        <group ref="at:Expresions" />
      </choice>
      <element name="alias" type="at:Alias" />
      <element name="exclude" type="at:Exclude" />
    </sequence>
  </complexType>

  <complexType name="Alias">
    <sequence minOccurs="0" maxOccurs="unbounded">
      <sequence minOccurs="2" maxOccurs="2">
        <element name="symbol" type="string" />
      </sequence>
    </sequence>
  </complexType>

  <complexType name="Exclude">
    <sequence minOccurs="0" maxOccurs="unbounded">
      <element name="symbol" type="string" />
    </sequence>
  </complexType>

  <complexType name="Type">
    <sequence>
      <element name="symbol" type="string"></element>
      <sequence minOccurs="1" maxOccurs="1">
        <choice>
          <group ref="at:Expresions"></group></choice>
      </sequence>
    </sequence>
  </complexType>
</schema>
```

Anexo D

Código de los ejemplos de validación

D.1. Solución en Java ME

D.1.1. Control remoto

```
package example;

import java.io.IOException;

import javax.microedition.io.Connector;
import javax.microedition.io.Datagram;
import javax.microedition.io.DatagramConnection;
import javax.microedition.midlet.MIDlet;
import javax.microedition.midlet.MIDletStateChangeException;

import com.sun.spot.peripheral.ISleepManager;
import com.sun.spot.peripheral.Spot;
import com.sun.spot.sensorboard.EDemoBoard;
import com.sun.spot.sensorboard.peripheral.ISwitch;
import com.sun.spot.sensorboard.peripheral.ISwitchListener;
import com.sun.spot.sensorboard.peripheral.ITriColorLED;
import com.sun.spot.util.IEEEAddress;

public class RemoteControlMidlet extends MIDlet {

  private class LightSender extends Thread{
    boolean toSend;
    int msgCount;
    ITriColorLED[] leds;

    public LightSender(){
      toSend=false;
```

```
    msgCount=0;
    leds=EDemoBoard.getInstance().getLEDs();
  }
  public void run(){
    int packetSize=ControllerLightMidlet.PACKET_SIZE;
    int spotPacketSize=ControllerLightMidlet.SPOT_PACKET_SIZE;
    for(int i=0;i<leds.length;i++){
      leds[i].setOff();
      leds[i].setRGB(255, 0, 0);
    }
    leds[0].setRGB(0, 255, 0);
    leds[0].setOn();
    while(true){
      DatagramConnection dc=null;
      waitMsg();
      try {
        String conName="radiogram://"
                    +IEEEAddress.toDottedHex(address)+":80";
        dc=(DatagramConnection) Connector.open(conName);
        Datagram d=dc.newDatagram(packetSize);
        d.writeBoolean(getToSend());
        dc.send(d);
        reduce();
      } catch (IOException e) {
        int actualMsgCount=getMsgCount();
        e.printStackTrace();
        leds[0].setRGB(255, 0, 0);
        for(int i=0;i<actualMsgCount;i++){
          leds[i].setOn();
        }
        DatagramConnection dc2=null;
        long recAddress=0;
        while(recAddress!=address){
          try {
            dc2=(DatagramConnection) Connector.open("radiogram://:81");
            Datagram d2=dc2.newDatagram(spotPacketSize);
            dc2.receive(d2);
            recAddress=d2.readLong();
          } catch (IOException e1) {
            e1.printStackTrace();
          }finally{
            if(dc2!=null){
              try {
                dc2.close();
              } catch (IOException e1) {
                e1.printStackTrace();
              }
```

```
            }
          }
        }
        for(int i=0;i<actualMsgCount;i++){
          leds[i].setOff();
        }
        leds[0].setRGB(0, 255, 0);
        leds[0].setOn();
      }finally{
        if(dc!=null){
          try {
            dc.close();
          } catch (IOException e) {
            e.printStackTrace();
          }
        }
      }
    }
  }

  public synchronized boolean getToSend(){
    return toSend;
  }

  public synchronized void setToSend(boolean toSend){
    this.toSend=toSend;
    msgCount++;
    this.notifyAll();
  }
  public synchronized int getMsgCount(){
    return msgCount;
  }

  public synchronized void waitMsg(){
    while(msgCount==0){
      try {
        this.wait();
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
  }

  public synchronized void reduce(){
    msgCount--;
    if(msgCount>1) msgCount=1;
  }
```

```
  public synchronized void clean(){
    msgCount=0;
  }
}

long address;

public RemoteControlMidlet() {
  address=0;
}

protected void destroyApp(boolean unconditional)
    throws MIDletStateChangeException {
}

protected void pauseApp() {
}

protected void startApp() throws MIDletStateChangeException {
  int spotPacketSize=ControllerLightMidlet.SPOT_PACKET_SIZE;
  ISleepManager sleepManager = Spot.getInstance().getSleepManager();
  sleepManager.disableDeepSleep();
  EDemoBoard.getInstance().getSwitches()[0].waitForChange();
  DatagramConnection dc=null;
  final LightSender lightSender=new LightSender();
  try{
    while(address==0){
      dc=(DatagramConnection) Connector.open("radiogram://:81");
      Datagram d=dc.newDatagram(spotPacketSize);
      dc.receive(d);
      address=d.readLong();
    }
  }catch(IOException e){
    e.printStackTrace();
  }finally{
    if(dc!=null){
      try {
        dc.close();
      } catch (IOException e) {
        e.printStackTrace();
      }
    }
  }
  lightSender.start();
  ISwitch[] switches=EDemoBoard.getInstance().getSwitches();
  switches[0].addISwitchListener(new ISwitchListener(){
```

```
      public void switchPressed(ISwitch sw) {
        lightSender.setToSend(true);
      }

      public void switchReleased(ISwitch sw) {}
    });
    switches[1].addISwitchListener(new ISwitchListener(){
      public void switchPressed(ISwitch sw) {
        lightSender.setToSend(false);
      }

      public void switchReleased(ISwitch sw) {}
    });
  }

}
```

D.1.2. Controlador de luz

```
package example;

import java.io.IOException;

import javax.microedition.io.Connector;
import javax.microedition.io.Datagram;
import javax.microedition.io.DatagramConnection;
import javax.microedition.midlet.MIDlet;
import javax.microedition.midlet.MIDletStateChangeException;

import com.sun.spot.peripheral.ISleepManager;
import com.sun.spot.peripheral.Spot;
import com.sun.spot.peripheral.radio.RadioFactory;
import com.sun.spot.sensorboard.EDemoBoard;

public class ControllerLightMidlet extends MIDlet {

  public static final int PACKET_SIZE= 1;
  public static final int SPOT_PACKET_SIZE= 8;
  private static final int BROADCAST_SLEEP = 1000;

  public ControllerLightMidlet() {
  }

  protected void destroyApp(boolean arg0)
          throws MIDletStateChangeException {
  }
```

```
protected void pauseApp() {
}

protected void startApp() throws MIDletStateChangeException {
  ISleepManager sleepManager = Spot.getInstance().getSleepManager();
  sleepManager.disableDeepSleep();
  EDemoBoard.getInstance().getSwitches()[0].waitForChange();
  EDemoBoard.getInstance().getLEDs()[0].setRGB(0, 0, 255);
  Thread lightListener=new Thread(){
    public void run(){
      while(true){
        DatagramConnection dc=null;
        try{
          dc=(DatagramConnection) Connector.open("radiogram://:80");
          Datagram d=dc.newDatagram(PACKET_SIZE);
          dc.receive(d);
          boolean b=d.readBoolean();
          EDemoBoard.getInstance().getLEDs()[0].setOn(b);
        }catch(IOException e){
          e.printStackTrace();
        }finally{
          if(dc!=null){
            try {
              dc.close();
            } catch (IOException e) {
              e.printStackTrace();
            }
          }
        }
      }
    }
  };
  lightListener.start();

  Thread spotListener=new Thread(){
    public void run(){
      while(true){
        DatagramConnection dc=null;
        long myAddress=RadioFactory.getRadioPolicyManager()
                .getIEEEAddress();
        try{
          String conName="radiogram://broadcast:81";
          dc=(DatagramConnection) Connector.open(conName);
          Datagram d=dc.newDatagram(SPOT_PACKET_SIZE);

          d.writeLong(myAddress);
          dc.send(d);
```

```
          }catch(IOException e){
            e.printStackTrace();
          }finally{
            if(dc!=null){
              try {
                dc.close();
              } catch (IOException e) {
                e.printStackTrace();
              }
            }
          }
          try {
            Thread.sleep(BROADCAST_SLEEP);
          } catch (InterruptedException e) {
            e.printStackTrace();
          }
        }
      }
    };
    spotListener.start();
  }

}
```

D.2. Solución en AmbientTalk

D.2.1. Control remoto

```
network.online();

deftype Light;

when: Light discovered: { |light|
  foreach: {|led|
    led.on := false;
    led.setRGB(255,0,0);
  } in: leds;

  leds[1].setRGB(0,255,0);
  (leds[1]).on := true;

  buttons[1].whenPressed: {
    light<-setLight(true);
    true;
  };
```

```
  buttons[2].whenPressed: {
    light<-setLight(false);
    true;
  };

  whenever: light disconnected: {
    def msgs := retract: light;
    def lmsg := msgs.length() ;
    foreach: {|msg| light <+ msg;} in: msgs;
    1.to: (lmsg + 1) do:{ |i|
      (leds[i]).on := true;
    };
    leds[1].setRGB(255,0,0);
  };

  whenever: light reconnected: {
    foreach: {|led| led.on := false} in: leds;
    (leds[1]).on := true;
    leds[1].setRGB(0,255,0);
  };
};
```

D.2.2. Controlador de luz

```
network.online();

leds[1].setRGB(0,0,255);
deftype Light;

def lightObject := object: {
  def setLight(status){
    (leds[1]).on := status;
  };
};

export: lightObject as: Light;
```

Referencias

[1] G. Agha. *ACTORS: a model of concurrent computation in distributed systems*. The MIT Press: Cambridge, MA, 1986.

[2] Página web de AmbientTalk. (inglés) `http://prog.vub.ac.be/amop/`, 2 de Julio 2009.

[3] G. Bracha and D. Ungar. Mirrors: Design principles for meta-level facilities of object-oriented programming languages. In *Proceedings of the 19th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2004)*, pages 331–344, Vancouver, British Columbia, Canada, Oct. 2004. ACM Press. ACM SIGPLAN Notices, 39(11).

[4] J.-P. Briot. From objects to actors: study of a limited symbiosis in smalltalk-80. In *Proceedings of the 1988 ACM SIGPLAN workshop on Object-based concurrent programming*, pages 69–72, New York, NY, USA, 1988. ACM Press.

[5] T. V. Cutsem. *Ambient References: Object Designation in Mobile Ad Hoc Networks*. PhD thesis, Vrije Universiteit Brussel, 2008. (inglés) `http://prog.vub.ac.be/~tvcutsem/publications/phd_tom_van_cutsem.pdf`.

[6] T. V. Cutsem, S. Mostinckx, E. G. Boix, J. Dedecker, and W. D. Meuter. AmbientTalk: Object-oriented event-driven programming in mobile ad hoc networks. In H. Astudillo and É. Tanter, editors, *Proceedings of the XXVI International Conference of the Chilean Computer Science Society*, pages 3–12, Iquique, Chile, Nov. 2007. IEEE Computer Society.

[7] J. Dedecker, S. Mostinckx, T. Van Cutsem, W. De Meuter, and T. D'Hondt. Ambient-oriented programming. In *OOPSLA 2005 Onward! Track*, Oct. 2005.

[8] J. Dedecker, T. Van Cutsem, S. Mostinckx, T. D'Hondt, and W. De Meuter. Ambient-oriented programming in AmbientTalk. In D. Thomas, editor, *Proceedings of the 20th European Conference on Object-oriented Programming (ECOOP 2006)*, number 4067 in Lecture Notes in Computer Science, pages 230–254, Nantes, France, July 2006. Springer-Verlag.

[9] Página web de Java ME. (inglés) `http://java.sun.com/javame/index.jsp`, 7 de Julio 2009.

[10] A. C. Kay. The early history of smalltalk. *SIGPLAN Not.*, 28(3):69–95, 1993.

[11] H. Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In N. Meyrowitz, editor, *Proceedings of the 1st International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 86)*, pages 214–223, Portland, Oregon, USA, Oct. 1986. ACM Press. ACM SIGPLAN Notices, 21(11).

[12] H. Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. *SIGPLAN Not.*, 21(11):214–223, 1986.

[13] H. Lieberman. Concurrent object-oriented programming in ACT 1. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 9–36. MIT Press, 1987.

[14] M. Miller, E. D. Tribble, and J. Shapiro. Concurrency among strangers: Programming in E as plan coordination. In R. De Nicola and D. Sangiorgi, editors, *Symposium on Trustworthy Global Computing*, volume 3705 of *LNCS*, pages 195–229. Springer, April 2005.

[15] S. Mostinckx, T. Van Cutsem, S. Timbermont, E. Gonzalez Boix, É. Tanter, and W. De Meuter. Mirror-based reflection in AmbientTalk. *Software—Practice and Experience*, 39(7):661–699, May 2009.

[16] Information Module Profile (IMP) specification, 2003. Version 1.0, JSR 195.

[17] Página web de Squawk JVM. (inglés) `https://squawk.dev.java.net/`, 7 de Julio 2009.

[18] Mobile Information Device Profile (MIDP) specification, 2000. Version 1.0, JSR 37.

[19] Connected Limited Device Configuration (CLDC) specification, 2003. Version 1.1, JSR 139.

[20] Página web de SUN SPOT. (inglés) `http://www.sunspotworld.com`, 2 de Julio 2009.

[21] G. J. Sussman and G. L. Steele. Scheme: An interpreter for extended lambda calculus. Technical Report AI Memo No. 349, Massachusetts Institute of Technology, Cambridge, UK, December 1975.

[22] D. Ungar and R. B. Smith. Self: The power of simplicity. *SIGPLAN Not.*, 22(12):227–242, 1987.

[23] M. Weiser. The computer for the twenty-first century. *Scientific American*, pages 94–100, Septiembre 1991. (inglés).

[24] Wikipedia: Sun spot. (inglés) `http://en.wikipedia.org/wiki/Sun_SPOT`, 2 de Julio 2009.

[25] A. Yonezawa, J.-P. Briot, and E. Shibayama. Object-oriented concurrent programming in ABCL/1. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pages 258–268. ACM Press, 1986.