An OSGi-based Service Oriented Architecture for Android Software Development Platforms

Aghiles Adjaz*, Samia Bouzefrane*, Dijiang Huang**and Pierre Paradinas*
* CEDRIC Laboratory,
Conservatoire National des Arts et Métiers
75141, Paris Cedex 03, France
e-mail: {First_name.Last_name}@cnam.fr

**Arizona State University
Brickyard Suite 470, 699 S. Mill Avenue, Tempe, AZ 85281-8809, U.S.A.
e-mail: dijiang@asu.edu

Abstract
During the past few years, service oriented approaches have been appeared as a new research paradigm providing better control, re-usability, and reliability for the software developments. With the growing complexity of embedded systems, new methodologies are needed to facilitate design, implementation, and maintenance of such systems, while providing means to capitalize software developments. Although embedded mobile devices are usually considered as resource restricted, there is a great demand to incorporate service-oriented approaches to achieve more dynamicity and robustness. To this end, this article aims at the establishment of a service oriented approach to integrate OSGi into the mobile Android platform to make the software development adaptive and dynamic. The main contribution of this paper is the establishment of a middleware solution incorporating OSGi into Android software development platform, which provides a service-architecture for Android-based applications. The presented solution is illustrated through an example. Additionally, performance evaluations are presented to demonstrate the effectiveness of our approach.

Key words: OSGi, Service Oriented Architecture, Android platform.

1. Introduction

The increasing complexity of the applications in embedded systems and the integration of high-level services within these systems (such as the integration of a Web service in Java Card 3.0 [2] or the proposition of OSGi (*Open Service Gateway Initiative*) ME [1] for embedding OSGi in sensors) led to the development of new methodologies to reduce the complexity of the software and to supply a support facilitating its re-use.

Recent development of mobile cloud computing [10] constructed a new service oriented framework that recruits mobile devices as service providers to build a sensing-based new application platform. In such a framework, each mobile device (usually an embedded device) is a service provider. First, an embedded device senses its surrounding information, such as wireless communication channel status, neighboring nodes information, environmental information (e.g., CO_2 and pollution levels, etc.), personal information (e.g., medical and health information using bio sensors), etc. Second, the mobile cloud creates a dual computing model in that an embedded device can outsource its computing-intensive computing tasks to the cloud. In the above described mobile cloud framework, service oriented approach is a natural choice to support mobile cloud software development and service provisioning.

Service oriented approach (SOA[1]) is viewed as a new paradigm to guarantee more control, re-use and reliability of the software. Mobile cloud demonstrated in this paper is based on mobile Android software development platform. This article focuses on how to convert the mobile Android platforms to a service oriented framework based on OSGi [3] framework. The notable feature of SOA is the service dynamicity, which is achieved by our solutions involving dynamic class loading, versioning management, and dynamic bundle reconfiguration without restarting Android application.

[1] Service Oriented Architecture

Many SOA approaches have been presented for mobile phones such as [4, 5 and 6]. However, in previous work, the service notion is synonym of Web services based on SOAP protocol with the objective of securing mobile communications. OSGi was originally designed for machine configurations of high resource footprint [3]. Recently, OSGi ME [1] was proposed for constrained embedded systems. OSGi ME has a footprint of 40KB with 32 bit-processors while frequencies vary from 8 to 72MHz. Another work aiming the use of OSGi technology in Java Card platforms has been addressed in [8]. The most relevant solutions related to our approaches are EZdroid project [7] and ProSyst'smBS Mobile SDK [9]. EZdroid is an open source project founded by Luminis BV and Akquinet AG. This platform is based on Apache Felix OSGi implementation running on Android, which is initially supplemented with components developed by the founders. The main drawback of this solution is that bundle updates need the recompilation of the application. Moreover, the bundles are duplicated within the Android applications, and there is no means to provide administrative functions using the OSGi platform.

The ProSyst's mBS Mobile for Android product is a carrier grade OSGi framework solution for the Android platform 1.5 and higher. A new type of bundle has been defined, making ProSyst bundles slightly different from traditional OSGi bundles. The consequence is that existing bundles have to be modified in order to be used by mBS Mobile. Another drawback is due to the installation of any Android application, which does not check the availability of the services, on which the application depends.

To address the drawbacks of existing work, in this paper, we present a new OSGi-based SOA framework to design Android applications based on OSGi bundles. Our solutions target to develop a component-based software package that implements the service-component model with modularity and reusability capabilities. The presented solutions make Android platforms more dynamic by providing SOA features such as dynamic class-loading, versioning management, and dynamic bundle configuration avoiding the Android platform restart. In fact, our methodology defines a middleware to launch Felix, an OSGi implementation, on Android terminals, and to manage the life cycle of the bundles that are used by Android applications. An application example is developed to illustrate the use of the proposed middleware, and experiments are conducted to measure the middleware overhead.

In the rest of this paper, we describe the principles of OSGi architecture and present the programming basics of Android platforms before emphasizing on the differences of OSGi and Android technologies in section 2. In section 3, we detail the solution we propose to make Android platform more robust by using dynamic services. Section 4 illustrates the use of our middleware by developing an application example. Section 5 gives the first experiments that have been carried out to measure the overhead due to our added middle-ware. Section 6 concludes the paper.

2. Description of Android and OSGi Platforms

In this section, we describe the principal features of OSGi and Android platforms in order to highlight the differences and to show the contribution of OSGi for Android platforms.

2.1 OSGi Architecture

OSGi technologies allow composing services dynamically for Java-based applications. OSGi is used in various domains such as in automotive, telecommunications or energy. OSGi is based on a simple and efficient model that allows dynamic programming with Java components that concentrate on the business logic. The deployment of these components is called bundles that cover all the bundle-related operations such as, loading, installing, activating, updating, or uninstalling. One major benefit of using OSGi is that these operations are carried out dynamically without restarting the OSGi platform. The platform checks the bundle dependencies (and their versions) before authorizing the activation of an application on which it depends.

OSGi platform consists of two parts: the OSGi framework and the standard services. The OSGi framework offers a service facility, where a service can be removed at any time. The OSGi framework

allows adding bundles in the container and managing their dependences and their versions using a class loader associated to each bundle. A bundle is a Jar file that contains a compiled code, resources as well as meta-data. The meta-data is stored in a configuration file called *Manifest.mf*. This file is necessary to the deployment of the bundles in the OSGi framework. It contains all the information related to the bundle such as its name, its version, its provider, the required dependencies, the exported and imported packages, and the information used during its life cycle. Packages are the exchanged units between the bundles. Indeed, a bundle sharing its service with other bundles has to mention it in the Manifest file by using the entry Export-Package. If a bundle requires another bundle, it has to import the corresponding package by using the entry Import-Package of the Manifest file.

2.2. The Android platform

Android is an open source operating system based on Linux and dedicated to mobiles phones, initially designed by the Android start-up and then acquired by Google. The Android platform is organized around various layers. An application layer supplies standard applications such as SMS application, a browser, a calendar, etc. Every application runs on a distinct virtual machine called Dalvik Virtual Machine (DVM) to avoid altering the functioning of the other applications.

The executable files .dex (Dalvik Executable) are generated from class files. Android contains a framework layer that facilitates the application development based on several components such as GUI management (Views); data sharing (Content Provider); resource access (Resource Manager), life-cycle management (Activity Manager), etc. This framework is based on C/C++ libraries and on Android runtime to provide Java basic functionalities.
In Android platform, a service is a component that runs in background. It is defined by an ".aidl" specification called Android Interface Definition Language (AIDL), from which a Java interface is automatically generated with an abstract Stub class. A service class contains an internal class extending Stub classes. To make the service available to other applications, the service-class name must be presented in the Service entry of the Manifest file. Additionally, Android introduces the activity notion to define a treatment associated with a physical view to interact with the user. The activity is the entry point of the application through a GUI. And the *Intent* messaging is a facility for late run-time binding between activities in the same or different applications.

2.3 OSGi versus Android

Compared to executing each application on a distinct DVM based on Android platform, all OSGi applications run in one JVM (see Figure 1). Android platform has drawbacks such as the duplication of components in DVMs. With a single VM in OSGi, the classes are shared and for each bundle is associated a distinct class loader that allows selective export of internal packages and selective import of required dependencies.

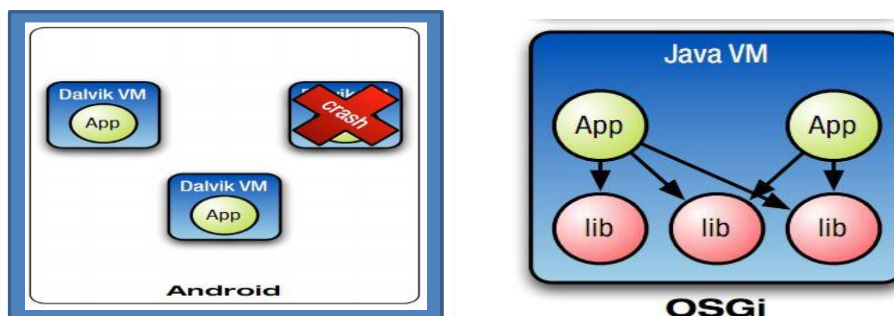

Figure 1.OSGi versus Android

In summary, using OSGi with Android will bring benefits such as:

- Designing applications by assembling components,
- Re-using existing OSGi components (bundles),

- Loading classes dynamically,
- Updating easily and managing bundles without restarting the Android application,
- A versioning support for bundles.

3. Description of the Presented Solution

This section describes a step-by-step methodology to integrate OSGi within the Android platform in order to take advantage of OSGi framework, i.e., the dynamic class-loading, service reusability, resolution of dependencies before executing an application, and update of services without restarting the application.

3.1 Integrating Felix platform in Android

In our implementation, we choose the Apache Felix platform as an implementation of OSGi due to its small size compared to other OSGi implementations. In the implementation, we use Apache Felix version 3.0.6, which has been incorporated in the release 4 of OSGi specification.

To integrate Felix within an Android platform, two approaches exist: (1) implementing Felix as an activity, or (2) implementing Felix as an Android remote service. We choose to implement Felix as an Android remote service. This is because Felix has to execute as a background task and does not need any graphical resources. Our implementation details are explained in the following sub-sections.

3.1.1 Android Services

Unlike other platforms in the market, Android provides an environment for applications that do not require user interfaces to operate. In Android system, a ***Service*** is an application extended with ***android.app.Service*** class, and it runs in the background. This service class can inform users that an event requires attention. In contrast to Android ***Activity*** that provides a user interface, a *Service* is completely invisible to the user, and can be controlled from other applications (activities).

An application that runs without requiring constant interaction with the user may be implemented as a ***Service***. Thus, we integrate the Felix Framework as a Service into the Android platform, called ***AndroLix*** that extends the **android.app.Service** class while providing different methods as presented in the following subsection.

3.1.2 AndroLix methods

The access to Felix being made via a RPC communication mechanism, *AndroLix* provides to clients an AIDL description containing different methods that are implemented by this service. The proposed *AndroLix* middleware allows the management of bundles' life-cycle, and calls bundles through the *AndroLix* AIDL, which is shown Figure 2. The *AndroLix* methods are presented as follows:

- **install():** installs a bundle from a specified location and returns a unique bundle ID
- **uninstall ():** uninstalls a bundle with a specific ID. If the Bundle is in a ***Resolved*** or ***Installed*** state, it is directly uninstalled. If it is in ***Active*** state, it is stopped before being uninstalled.
- **getBundleId():** returns the bundle ID.
- **startBundle():** starts a bundle after checking the availability of the bundle ID and the bundle status. The bundle is started if it is in ***Installed*** or ***Resolved***[2] state.
- **stopBundle():** stops a bundle after checking the availability of the bundle ID and the bundle status. The bundle is stopped if it is in ***Active*** state.
- **getBundlesContainer():** retrieves the symbolic name, the bundle ID and the status (*Installed, Resolved, Active*) of all the bundles present in the bundle container.

---

[2] the bundle's code dependencies are resolved

- **startFelixFramework():** defines a persistent set of properties (Felix Framework cache directory, Felix Framework and Android packages to be loaded, etc.) and starts the Felix Framework. This method is called when creating ***AndroLix Service***.
- **install_uninstall():** checks a package dependency of a bundle at a start-up or a stop of the bundle.
- **Call**() : allows a dynamic class-loading of classes to deal with different services wassociated to distinct contracts. The entry parameters are: the ID of the bundle that will load the service class, the complete name of the class including the package name, the name of the method to call, and the method arguments. The Call() method returns the result and the state of the operation.

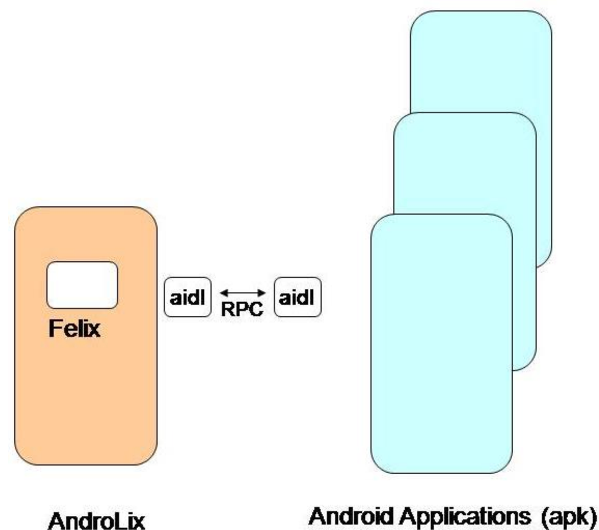

Figure 2. Overview of our architecture

3.2. Android applications that use bundles

By calling the *AndroLix* service, any Android application is able to interact with Felix platform. However, the OSGi bundles cannot use the mechanisms provided by the application framework of Android since the bundles are executed in background and cannot interact with Android applications. For this reason, we define an Android application as a pair of a bundle and an apk application. Both the application and bundle have the same name and are stored in the same package. The bundle specifies in Import-package entry not only the packages it requires but also the packages needed by the corresponding apk. The role of the associated bundle is to represent the application within the Felix container, i.e., the bundle guarantees the dependencies resolution of the apk application before its installation, and could access to other Felix bundles if they are required by the apk application. For example in Figure 3, the application B is composed of the apk and of an associated bundle contained in Felix and used to access to other Felix bundles. Moreover, the application is installed and uninstalled automatically according to the state of the associated bundle. In other words, if the bundle is active the corresponding application is installed, otherwise the application is uninstalled.

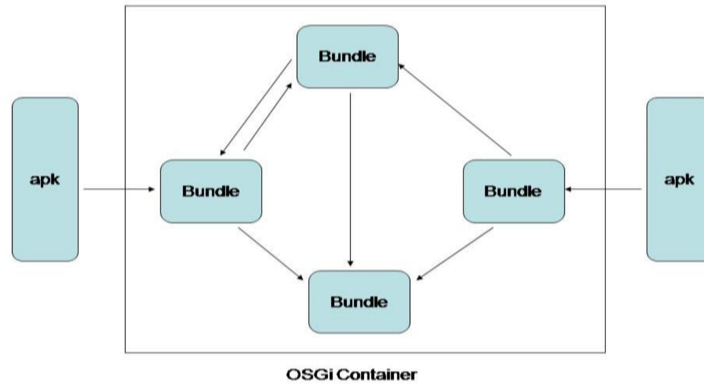

Figure 3. A typical application using bundles

3.3. The features of our proposed solution

By using Felix bundles to design Android applications with respect to a component approach, we provide more facilities for Android programmers. In fact, thanks to a single Felix container available to all apk applications, bundle sharing hence reusability is offered while versioning management is assured. The installation of the application is conditioned by the earlier resolution of the dependencies. Whenever a bundle has been updated, the involved applications are not required to be restarted. Contrary to the existing approaches, our solution does not change the structure of the bundle, i.e., classic bundles may be used by the android application without any modification. Furthermore, as described in Figure 4, a friendly graphical user interface to manage Felix container has been developed as an Activity to handle different bundles in real time. This interface allows the user to start, stop and remove a bundle, and to check the status of the bundles in the container.

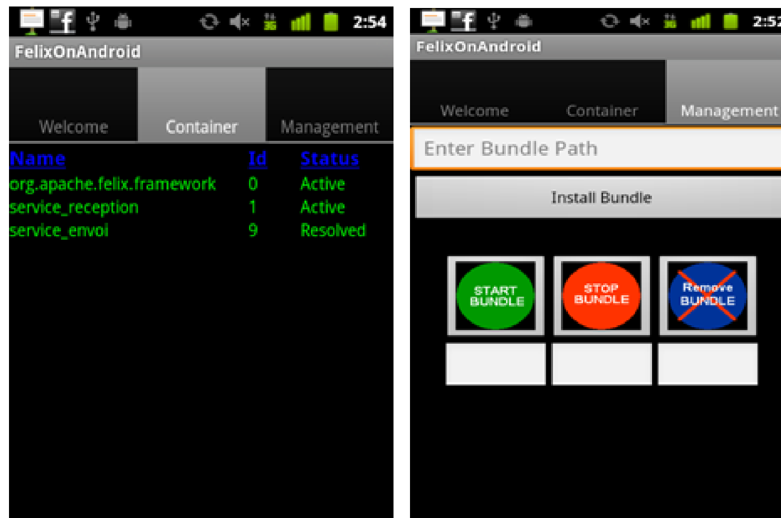

Figure 4. A GUI management of bundles

4. An Illustrative Example

To illustrate the use of the implemented middleware, we develop an application that uses OSGi services. The application offers geo-localization and tracking services in order to localize an object on a map based on geographical coordinates. The application is composed of two entities; each one is handled by a distinct Android platform. The first entity sends its geographical position during a movement, and the second entity receives the coordinates and finds the position on a map.

The information exchanged by the two entities is done based on SMS service in a transparent manner. As in Figure 5, the tracker entity starts by sending a localization request, and the tracked entity answers by sending its position each time a new position is detected. When the tracker entity does not

want to receive information any more, it sends to the tracked entity a stop request inviting this latter to stop its application.

Figure 5. Application example

To respect the requirements of the application, we designed three bundles as in Figure 6:

- ***SendService*** Bundle: this bundle provides three services: *Send* service to send an SMS message, *Crypt* service to encrypt a message and *Decrypt* service to decrypt an encrypted message.

- ***ReceiveService*** Bundle: this bundle provides two types of services: *isSmsLocation* service returns the type of the message passed as a parameter, and *getLocation* service that extracts the geographical coordinates from the message.

- ***Tracker*** Bundle: this bundle is associated to the apk application. It represents the application in the Felix container and accesses to the services provided by *SendService* and *ReceiveService* bundles.

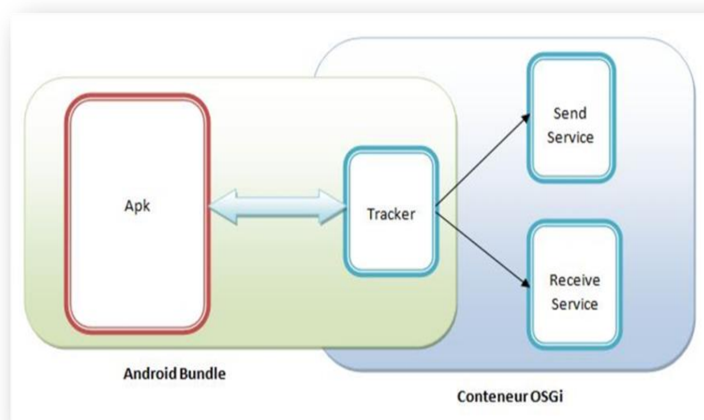

Figure 6. The service architecture of the application

In the developed application, if we update an OSGi service such as changing the implemented encryption algorithm, the update is handled automatically and no application using this service will be recompiled. This behavior increases the robustness of the application.

5. Some experiments

In the preceding sections, we proposed a new model to implement OSGi on Android mobile platforms. We illustrate the use of this model to highlight the benefit of our proposed solution. In this section, we present the performance of our solution by measuring the overhead generated when using the Felix middleware. For this purpose, we choose execution time as a measurement criterion and we use DDMS (Dalvik Debug Monitor Server) tool of Android SDK that runs in both emulators and real terminals. DDMS provides a powerful option called TraceView to measure execution time.

In these experiments, based on a TraceView tool, we measured the execution times of all the developed methods that interact with Felix. First, we computed the necessary time to start Felix, and then we measured, according to the number of bundles and their dependences, the execution time of these methods. The experiments have been undertaken on a Samsung Galaxy Tab mobile phone that uses a 2.2 version of Android, with Cortex 1.0 GHz processor. In the following figures, the execution time is given in milliseconds.

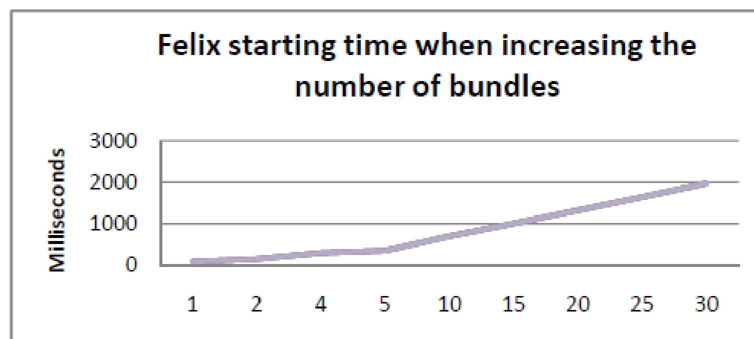

Figure 7.Felix starting time when varying the number of bundles

First, we start Felix and we measure the starting time when increasing the number of bundles as depicted in Figure 7. We notice that this time may reach two seconds if Felix handles up to 30 bundles. Even if this time seems to be important, Felix is started once when starting the Android platform.

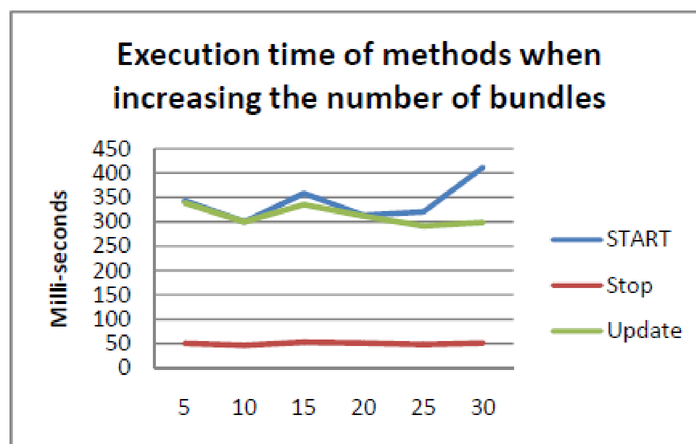

Figure 8. Execution times of different methods when varying the number of bundles

We also measure the execution time of three methods: *Start*(), *Stop*() and *Update*() to respectively start, stop and update bundles as in Figure 8. When increasing the number of bundles, the execution time of the three methods seems to vary independently of the number of bundles. However, when increasing the level of dependencies, the execution time of the methods Start() and Stop() increases

linearly with the number of dependency levels (see Figure 9). A dependency level corresponds to a bundle that depends on another. In fact, when a bundle is started, its dependencies are resolved first. In the same manner, when a bundle is stopped, its dependencies are freed.

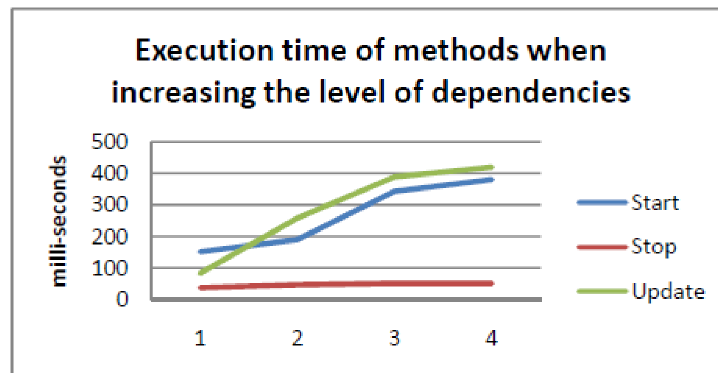

Figure 9. Execution times of different methods when varying the number of dependencies

Based on the above presented performance evaluations, we can see that the methods proposed to manage the bundles depend only on the number of the bundle dependencies for each bundle.

6. Conclusion

In this paper, we aimed at using facilities offered by OSGi in Android platforms. For this purpose, we proposed a solution to integrate Felix, a lightweight implementation of OSGi specification. We first define a set of methods, called *AndroLix*, in order to implement Felix as an Android remote service. Then, we present an application example to illustrate the use of the *AndroLix* middleware. Finally, we present performance measurements to evaluate the time performances of *AndroLix*. Our solutions allow android applications to take advantage of the facilities offered by OSGi such as to update bundles dynamically without restarting the involved applications, to manage the versioning, to share services between distinct applications, and to check the dependencies before the application launching avoiding errors during execution. For our future work, we plan to extend our solution to communicate with remote OSGi bundles in a mobile cloud computing framework.

7. References

[1] :Andre Bottaro and Fred Rivard, “OSGi ME - An OSGi Profile for Embedded Devices”, Eclipse Summit, Germany, 2009.
[2] : Java Card, 2011: http ://java.sun.com/javacard/3.0/
[3] :OSGi, 2011: http ://www.osgi.org.
[4] : Johnneth Fonseca, Zair Abdelouahab, Denivaldo Lopes and Sofiane Labidi, A security framework for SOA applications in mobile environment, *International Journal of Network Security and Its Applications* (IJNSA), Vol.1, No.3, October 2009.
[5] : Decker, M, and Bulander, R. "A Platform for Mobile Service Provisioning Based on SOA Integration", In *Communications in Computer and Information Science*, Vol 23, 2009, Springer Berlin Heidelberg, pp 72-84.
[6] :Natchetoi, Y. Kaufman, V. and Shapiro, A. "Service-oriented architecture for mobile applications", Proceedings of the *1st international workshop on Software architectures and mobility / International Conference on Software Engineering*, pp 27-32, 2008.
[7] :EZdroit project, 2011 : http://www.ezdroid.com/
[8] : Agnes C. Noubissi, Julien Iguchi-Cartigny and Jean-Louis Lanet, Convergence OSGi-Java Card : Fine-grained dynamic update, *E-smart10*, 21-24 September 2010, Sophia Antipolis France.
[9] : ProSyst, 2011 : http://www.prosyst.com/index.php/de/html/content/49/mBS-Mobile-for-Android/
[10] :Dijiang Huang, Xinwen Zhang, Myong Kang, and Jim Luo, MobiCloud: Building Secure Mobile Cloud Framework for Mobile Computing and Communication.

In *Proceedings of the 5th IEEE International Symposium on Service-Oriented System Engineering (SOSE)*, pages 27-34, 2010.