

Android/OSGi-based Vehicular Network Management System

Teng-Wen Chang

Department of Electrical Engineering, National Taiwan University, Taipei, Taiwan

Abstract- With the enormous market potential of the telematics industry and the rapid development of information technology, automotive telematics has attracted considerable attention for mobile computing and intelligent transport systems (ITSs). This study integrates the Open Gateway Service Initiative Vehicle Expert Group (OSGi/VEG) into an Android platform to generate a vehicular Android/OSGi platform that has the advantages of both original platforms. These features include remote management, rich class sharing, proprietary vehicular applications, security policies, easy management of application programming interface (APIs), and an open environment. This study also integrates a cloud computing mechanism into the Android/OSGi platform, allowing service providers to upload their telematics bundles onto storage clouds using a provisioning server. A management agent in the Android/OSGi platform can simultaneously update its application service modules using remote storage clouds and use visual intelligence to continually change the distinguishing features of applications based on context awareness without user intervention. To assess the feasibility of the proposed Android/OSGi platform, this study presents a vehicular testbed to determine the functionalities of different telematics applications. Android/OSGi platform applications require less memory and system resources than those on the original Android platform when performing complicated operations. Additionally, the Android/OSGi platform launches telematics services more quickly than the original Android platform. The proposed platform overcomes the problem of frequent non-responsive exceptions in the original Android platform.

Keywords- Intelligent Transport Systems, Vehicular Platform, OSGi, Android, Cloud Computing, Telematics Applications

I. INTRODUCTION

In recent years, sophisticated information technology has allowed the automotive industry to mature in the field of telematics. Telematics in automobiles involves the automated convergence of telecommunications and information technology. So far, automakers and third-party service providers have developed quite a number of telematics services, including monitoring, emergency roadside assistance, navigation, driver aids, remote diagnostics, entertainment, and web browsing. The telematics market is currently quite young, and most telematics technologies are independent of each other due to multiple platform standards. This situation also makes it difficult for service providers to create value-added telematics services. Consequently, numerous vehicle manufacturers have established and developed open/standard embedded platforms for vehicles. These platforms include OSGi/VEG, AUTOSAR, AMI-C, CVIS, OSEK/VDX, and Android. Fig. 1 illustrates an open Linux operating system ported into an embedded on-board terminal. This system not only provides a variety of device drivers, including CAN/LIN/FlexRay buses and out-network

connection modules, but also offers resources management. The open/standard telematics platforms in the telematics middleware layer mainly standardize API's and graphic/vocal HMI (human-machine interface) so that both service providers and car manufacturers can quickly deliver solutions to potential markets and simplify development. If service providers want to remotely deploy telematics services to an on-board terminal, or if road-side centers need to diagnose/configure vehicular devices or manage modules of telematics application, they should use remote management services that rely on open/standard telematics platforms.

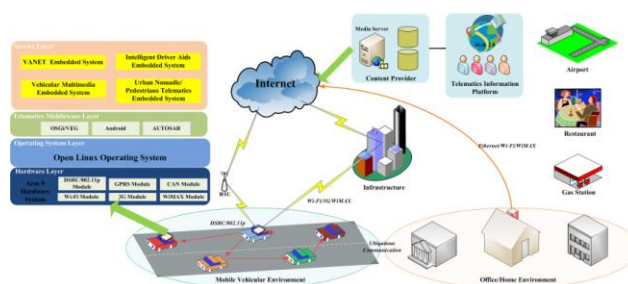


Fig. 1 Telematics Architecture

This study discusses how to combine OSGi/VEG into an Android platform to form a new vehicular Android/OSGi platform that has the advantages of both original platforms, including remote management/deployment, rich class-sharing, proprietary vehicular applications, security policies, and more.

II. RELATED WORKS

A. Adroid Platform

The Android™ platform delivers a complete set of software for mobile devices: an operating system, middleware, and key mobile applications [1]. Windows Mobile and Apple's iPhone provide a richer, simplified development environment for mobile applications. However, unlike Android, they're built on proprietary operating systems that often prioritize native applications over those created by third parties and restrict communication between applications and native data. Android offers new possibilities for mobile applications by offering an open development environment built on an open source Linux kernel. Real hardware can be accessed through a series of standard API libraries, allowing the user to manage Wi-Fi, Bluetooth, and GPS devices. The Open Mobile Alliance (OHA) [2] and Google support the Android platform and hope to reach the goal of ensuring global mobile services that operate across devices, geographies, service providers, operators, and networks.

Fig. 2 illustrates the Android software stack [3], which consists of a Linux kernel, a collection of Android libraries, an application framework that manages Android applications in runtime, and native or third-party applications in the application layer. The following list specifies these Android software stack components:

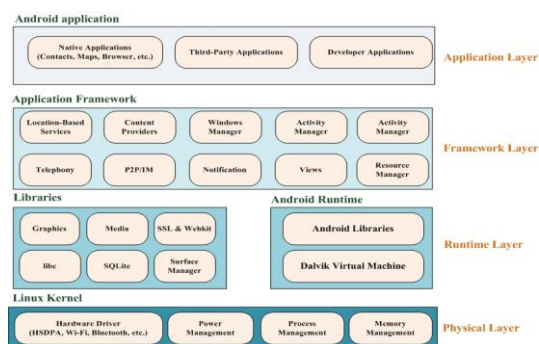


Fig. 2 Architecture of Android Software Stack

B. Open Services Gateway Initiative (OSGi) Service Platform

The OSGi™ platform standard [4] define a standardized, components-oriented computing environment for networked services that is the foundation of an enhanced service-oriented architecture (SOA). The OSGi specifications were initially targeted at residential internet gateways for home automation applications [6, 7]. However, the standard and extensible features of OSGi make enterprises regard OSGi technology as key solutions. For example, Nokia and Motorola adopted the OSGi technology standard for next-generation smart phones, and many car manufacturers have embedded OSGi specifications into their global system for telematics (GST) specifications [5]. The OSGi framework provides general-purpose, secure support for deploying extensible and downloadable java-based service applications, called bundles. OSGi bundles are applications packaged in a standard Java Archive (JAR) file, and contain a manifest file that describes the relationships among bundles and the OSGi framework, a series of compiled Java classes, and native code. Fig. 3 illustrates the OSGi framework running on a Java virtual machine. The OSGi framework provides a shared execution environment that dynamically installs, updates, and uninstall bundles without restarting the system.

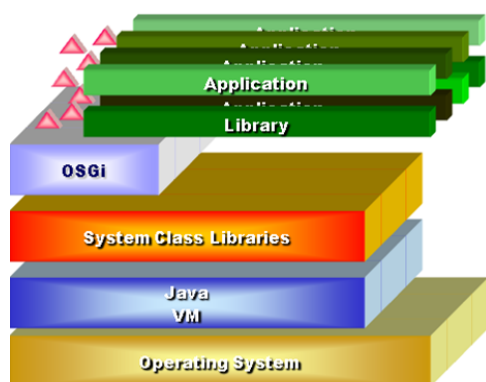


Fig. 3 Architecture of OSGi Framework

Automotive operating systems are also very significant in vehicles [8]. By supporting specific device drivers such as CAN/LIN buses, which communicate with other electronic control units (ECU) nodes for diagnostic purpose, previous studies have embedded an OSGi platform in a vASOS automotive operating system [9,10]. Fig. 4 illustrates OSGi-based middleware run on a K virtual machine (KVM), which is a compact, portable Java virtual machine intended for small, resource-constrained devices [5]. In addition, previous studies use an OSGi R3 implementation of Oscar to develop application bundles, including navigation and CAN/LIN/MOST access bundles. This study takes advantage of OSGi R4 implementation of Apache Felix to integrate with the Google Android platform, turning a variety of Google APIs into OSGi bundles, such as location-based, peer-to-peer communication, and network manager bundles. This approach not only allows service providers to deploy telematics services, but also enhances Android runtime layer performance and remote management functionality.

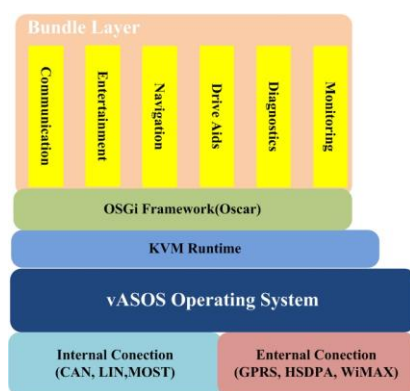


Fig. 4 Architecture of vASOS and OSGi-based middleware for vehicles

III. OPEN EMBEDDED ANDROID/OSGi PLATFORM FOR TELEMATICS

Fig. 5 illustrates the proposed architecture for a telematics system. Here, the Android platform adopts the latest Android 1.1 r1 kernel, allowing the proposed Android/OSGi platform to be integrated into real Android-based hardware devices. This design also allows the OSGi R4 implementation to be operated normally in an Android Dalvik virtual machine (VM), developing the communication interface between users and OSGi framework is called Android/OSGi applications, and embedded Android/OSGi applications. This study implements an advanced technique for embedding an OSGi instance into Android applications, to make them more adaptive and flexible. Any time-consuming operations, such as continually loading the main class of OSGi framework and updating the Android GUI interfaces, will be performed by Android background services. We also installed indispensable OSGi bundles for developing some valuable telematics applications, such as the Web management console, remote deployment interface, navigation, diagnostics, and so on. Finally, this study implements a provisioning server to remotely deploying packages or OSGi bundles to the client's Android/OSGi platforms.



Fig. 5 System Architecture

A. Location of OSGi Framework in the Android Platform

The Dalvik VM use the device's underlying Linux kernel to handle low-level functions, including security, threading, process, and memory management. It's also possible to read and write C/C++ applications that run directly on the underlying Linux OS. However, Dalvik VM can only execute Dalvik executable files (e.g., **classes.dex**), a format optimized to ensure minimal memory footprint. Consequently it cannot perform general Java packages and OSGi bundles, causing OSGi framework operations to fail when Android encounters OSGi functionalities in runtime. Besides, as Fig. 6 shows, if the OSGi framework successfully runs in the Android Runtime layer, it cannot directly interact with users, because users can only view Android Activities (or applications) in the application layer. For this reason, the Android/OSGi Activity must grab a report or log message from the OSGi framework, and use the Android UI to present information to the users. The proposed design uses the OSGi R4 implementation of Apache Felix to integrate with the Android platform because this container is more portable and light-weight, and better-suited to embedded hardware devices such as telematics systems.

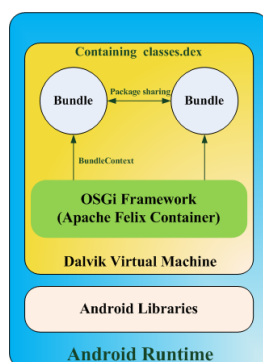


Fig. 6 Location of OSGi Framework

B. Android/OSGi Conversion Mechanism

The Android/OSGi conversion mechanism uses the OSGi

framework to dynamically load bundles through the Android Davik VM. Fig. 7 shows the Android/OSGi conversion mechanism, which makes the OSGi framework perform a series of compiled Java classes from OSGi bundles in the Android runtime layer using a **classes.dex** file to reference the OSGi bundle's general Java classes. The proposed design implements Android/OSGi conversion code through **Dalvik.system.dexFile** API, and imports two parameters, the package name (bundle API packages) and **ClassLoader** (reading the Java classes) objects. The conversion code then returns the collection of Java classes, allowing the OSGi framework to indirectly perform the functionalities provided by the OSGi bundles. With the Android conversion mechanism and dexified OSGi bundles, it is possible to port the OSGi framework to the Android underlying layer, along with other functional bundles such as telnet, deployment admin, http, and remote manager bundles etc, so that the Android platform can be operated and managed remotely.

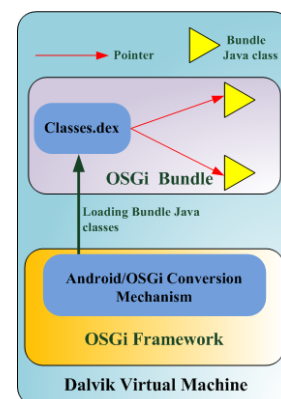


Fig. 7 Android/OSGi Conversion Mechanism

C. Development of Android/OSGi Application

We created an Android/OSGi application in the Android application layer, which can fetch the corresponding OSGi information from the Android runtime layer and provide an interactive environment between users and the OSGi framework. In the first integration of this procedure, the proposed Android/OSGi application had Android Activity properties. Unlike most traditional environments, Android applications have no ability of control their own life cycles. Instead, the Android application framework must to be in charge of the Android applications and react accordingly, taking particular care to be prepared for untimely termination. Fig. 8 illustrates the cross-communication operation of an Android/OSGi Activity launching the **FelixService** (Android background service), which represents a GUI interface of the OSGi console, and loading the main class of the OSGi framework after Android/OSGi Activity onCreate method has already been initialized. Android services run without a dedicated GUI, and usually perform background works silently so that users cannot perceive it. Before FelixService can run, it must to be attached to an Android/OSGi Activity. From a different point of view, the Android/OSGi Activity provides the Android **Context** object, which allows the Android application framework to manage its life states and

monitor communication between the Android/OSGi application and other native or third-party applications.

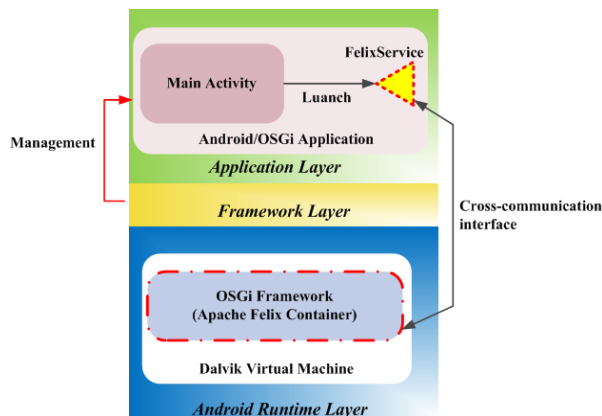


Fig. 8 Cross-communication Operation

To ensure that the proposed Android/OSGi application remains responsive, the proposed design moves all slow and time-consuming operations off the main Activity thread and onto a child thread. Examples of slow operations include continually reading the main class of the OSGi framework, and updating the GUI interface when the OSGi console view has been changed or GUI bundles provide new screen services. Consequently, this study implements the Felixservice to perform the above-mentioned operations. Fig. 9 shows that Felixservice can directly determine the location of the OSGi framework in the Android runtime layer and extract the main jar file of the OSGi framework, which is Apache Felix of R4 OSGi implementation. Using the Dalvik ClassLoader object to read the main class continually in child thread, the Android/OSGi platform can start the OSGi framework in the Android application layer. In addition, this study also implements a OSGi console view that can fetch the corresponding OSGi message and display them to users. The OSGi console view has two input/output interfaces that allow users to immediately interact with the OSGi framework.

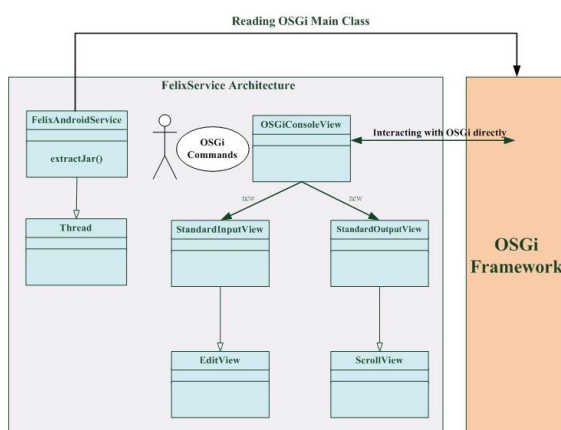


Fig. 9 Inter-architecture of the Felixservice

D. Embedding OSGi framework to Android application

In traditional devices, the OSGi framework is apart from application platform, and it utilizes the OSGi services and service registry as a serviced-based mechanism. This mechanism allows any application to run completely on top of the OSGi framework, but this way can't make OSGi integrating with Android tightly. Besides, the system components spend extra-time to communicate with OSGi framework in runtime layer. Therefore, if any Android application wants to use either the OSGi services or the APIs provided by bundles, it needs a complicated Android/OSGi communication mechanism. We created a hosted framework mechanism that allows the OSGi framework to tightly embed into Android applications in the application layer. In this way, any Android application can host the instance of OSGi framework by utilizing reflection approach, and application can externally load the services which OSGi bundles provided. Nevertheless, the extender mechanism has some drawbacks in that it lack dynamic changes in OSGi bundles/services and cannot configure OSGi instances. Therefore, this study combines two mechanisms with the Android platform to create a more integrated Android/OSGi environment.

IV. System Design and Implementation of Android/OSGi Platform

Testing the proposed telematics applications in real vehicles is inconvenient and expensive. Consequently as Fig. 10 illustrates, a vehicular testbed was built to test and verify the functionalities and security of the proposed Android/OSGi applications. This study regards sensor-based LEGO robots as simulated vehicles, and makes the vehicular Android/OSGi platform act as an on-board terminal for simulated vehicles. Instead of utilizing car buses, this on-board terminal mainly controls simulated vehicles using Bluetooth, like IBM iDrive systems. Communications between different vehicles is conducted via Wi-Fi. Finally, this study designs a provisioning server responsible for storing telematics applications and deploying an admin bundle called the management agent, which assign the location of remote provision server. When this agent has been started, it installs and starts remotely downloaded applications, and also manages their packages and resources.

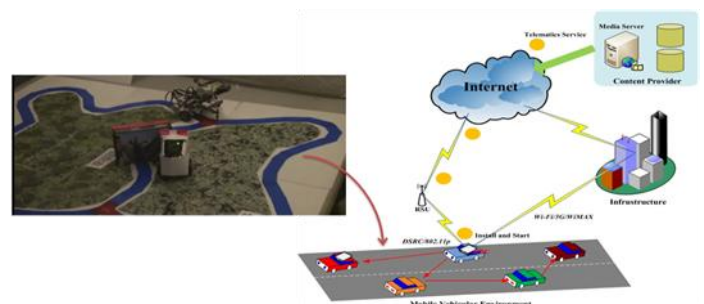


Fig. 10 Vehicular Testbed

A. Vehicular Android/OSGi Platform

Fig. 11 illustrates inter-architecture of Vehicular embedded Android/OSGi platform, in first, because Android/OSGi platform can only update its foreground screen after calling setContentView method, Android/OSGi application must to provide **ViewFactory** objects make other Android/OSGi bundles being able to provide or update their Android GUI interfaces into main Activity. Additionally, the Android/OSGi application should register its Activity instance service, so that host bundle can fetch this main Activity, and push registered GUI interfaces into it. The host bundle implements a ViewFactory interface with the primary responsibility of providing **DesktopApplication** service to other consumer bundles. The second service of DesktopApplication acts as a container for any Android View in the bundles. Consequently, if any bundles want to provide some Views for the main Activity, they must implement the DesktopApplication interface, and register their DesktopApplication services with the OSGi service registry. Besides, the host bundle implements **ServiceTrackerCustomizer** interface, which makes host bundle being able to track primary Activity service and other DesktopApplication services simultaneously in customized way.

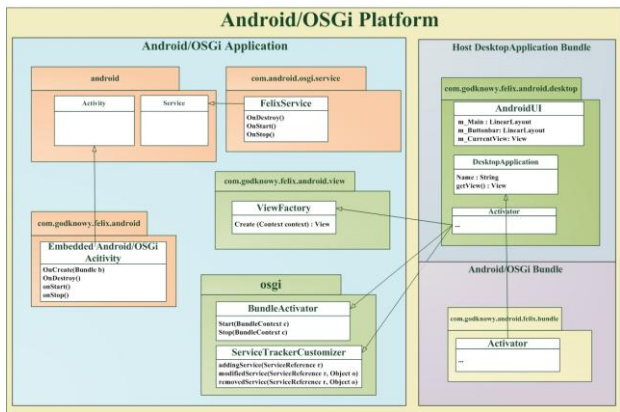


Fig. 11 Inter-architecture of Android/OSGi platform

B. Android/OSGi Bundles

(A) GUI-based Bundles

Figure 12 illustrates vehicular Android/OSGi platform has feature of lightweight bundles such as navigation bundle, which bundle size is less 11KB than original Android application. But vehicular Android/OSGi platform must to spend about 7.5 seconds to start OSGi framework in first time, and additional memory size of OSGi framework. In tested android hardware, it has total 990440KB memory size, and Android/OSGi platform need to spend about 1.91MB memory size for initiating and starting OSGi framework, and 0.25MB memory size for starting application, however in pure Android platform same application only needs to spend about 1.21MB memory size.



Fig. 12 Lightweight navigation application for Android/OSGi platform

Figure 13 illustrates Android/OSGi platform spends approximately 2MB memory size to launch OSGi framework, and when OSGi framework had been started, Android/OSGi platform can consume less memory size to start other applications. Total occupied memory size of probably lies in between 2MB and 3MB in Android/OSGi platform, however, pure Android platform has increased memory size, and grows up 5MB. Consequently, proposed Android/OSGi platform can spend less memory size than pure Android platform when OSGi framework had been started.

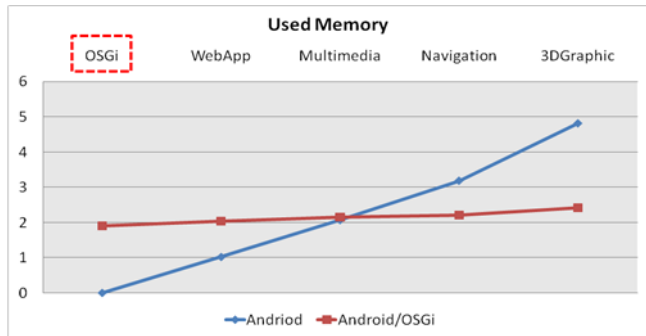


Figure 13: Used Memory Analysis

(B) Telematics Services

Figure 14 illustrates this study presents various telematics services to establish intelligent vehicles in mobile environment which mainly consist of line follow, object detection, keep distance, and so on. The first part of service helps vehicles follows guidelines to ensure driver security and reduce route-finding complexity to lighten the use of power. The second part of this service provides vehicles with visual intelligence to identify objects like traffic signals. The third part of the service helps vehicles maintain a set distance from vehicles in front of back. The last component of this service allows road-side centers to survey the condition of vehicles. If some of services access the significant components of vehicle, the vehicular Android/OSGi platform can enforce security policies to avoid accidents using a AOP-based OSGi weaving mechanism. For example if one service makes the vehicle travel faster than 70km/h, this service will be rejected.



Fig. 14 Android/OSGi telematics service

(C) Remote Management

Fig. 15 illustrates vehicular Android/OSGi platform mainly provide three management consoles, which include telnet, web management, and remote deployment console. The first part of the telnet console allows a remote manager to diagnose and manage Android/OSGi applications through a telnet interface, which extends the telnet bundle provided by the Oscar R3 implementation. This web management console relies on underlying OSGi bundles, including http service, declarative service, log service, configadmin, deployadmin, metadata, and so on. The web management console can make remote managers can log in local Android/OSGi platform, and manage system. The rest of the remote deployment console enables applications to be deployed onto local vehicular Android/OSGi platform by utilizing of provisioning server. This study develops management agent to communicate with remote cloud called Amazon EC2 through Restful API. Android/OSGi bundles are stored in Amazon S3, and Android/OSGi platform uses management agent to automatically install/update bundles with remote cloud same as active sync. way. Consequently, the Android/OSGi platform can dynamically reconfigure its application services by communicating with a provisioning server in runtime based on context-awareness. Besides, the Android/OSGi platform supports partial renewal of applications, reducing system delay time and enhancing system performance.

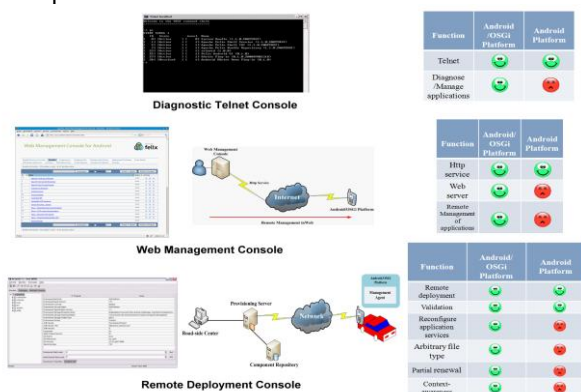


Fig. 15 Remote Management of vehicular Android/OSGi platform

V. Conclusion

Using the proposed vehicular Android/OSGi platform,

road-side centers can diagnose or manage the system status of vehicular platform remotely, and use visual intelligence to continually update their application services based on context-awareness without user intervention. This study also establishes a vehicular testbed to verify and analyze the functionalities of the Android/OSGi and original Android platform. Finally, this study proves that the proposed Android/OSGi platform has lighting applications and higher performance than a pure Android platform when performing complicated operations.

REFERENCES

- [1] Android Platform Official Site, <http://www.android.com/>
- [2] Open Mobile Alliance Official Site, <http://www.openmobilealliance.org/>
- [3] Google Android software stack in brief, <http://fuyichin.blogspot.com/2008/06/google-android-software-stack-in-brief.html>
- [4] OSGi Alliance. OSGi service platform, core specification release 4. Draft, July 2005.
- [5] GST "Global System for Telematics" Official Site, <http://www.gstforum.org/>
- [6] C. Lee, D. Nordstedt and S. Helal, "Enabling Smart Spaces with OSGi," IEEE Pervasive Computing, Vol.2, No.3, pp. 89-94, July-Sept. 2003.
- [7] K. Myoung, J. Heo, W.H. Kwon and D.S. Kim, "Design and Implementation of Home Network Control Protocol on OSGi for Home Automation," *Proceedings of the IEEE International Conference on Advanced Communication Technology*, Vol.2, pp. 1163-1168, July 2005.
- [8] S.V. Alberto and M.D. Natale, "Embedded System Design for Automotive Applications", *Proceedings of IEEE Computer Society on Computer Engineering*, Vol.40, pp.42-51, Oct. 2007.
- [9] Y. Ai, Y. Sun, W. Huang and X. Qiao, "OSGi based Integrated Service Platform for Automotive Telematics," *Proceedings of the IEEE International Conference on Vehicular Electronic and Safety*, pp.1-6, Dec. 2007.
- [10] Y. Sun, W.L. Huang, S.M. Tang, X. Qiao and F.Y. Wang, "Design of an OSEK/VDX and OSGi-based Embedded Software Platform for Vehicular Applications," *Proceedings of the IEEE International Conference on Vehicular Electronic and Safety*, pp.1-6, Dec.